eXcursion for Windows: Integrating Two Windowing Systems

By Dennis G. Giokas and Andrew T. Leskowitz

1 Abstract

Digital's eXcursion for Windows display server is an application for Microsoft Windows. The eXcursion for Windows product is based on the X Window System and allows X client applications to display output, receive input, and exchange data in the Microsoft Windows environment. The eXcursion software visually integrates the X and Microsoft Windows environments-applications from both environments display on a single screen and have the same appearance. A key component of Network Applications Support (NAS) and Digital's PC integration program, the eXcursion for Windows display server enables information exchange among PC users and non-PC users linked throughout a network.

2 Introduction

The eXcursion for Windows software is a display server based on the X Window System version 11, release 4 protocol and implemented as an application for Microsoft Windows software. eXcursion allows X11 client applications based on any X11 toolkit to display output and receive input in the Microsoft Windows environment. The two window environments are seamlessly integrated. Microsoft Windows software provides the window management for X Window System applications. The eXcursion display server smoothly handles the display and user input for the X applications along with data exchange between the applications.

This paper first establishes the relationship of the eXcursion display server to the X Window System and Microsoft Windows environments. It then presents the personal computing integration philosophy behind the development of the eXcursion product. This paper then relates the design philosophy and implementation architecture of the server. It concludes with a discussion of resource usage. <column>

The DECwindows architecture integrates the user and graphical interfaces of the VMS, ULTRIX, and DOS operating system and desktop environments. The X Window System client-server architecture, on which the DECwindows system is based, provides the means to achieve this integration. The X architecture, as implemented by Digital's DECwindows Motif program, is shown in Figures 1 and 2. This architecture is hardware and software system independent. It allows X applications, or clients, to execute on any processor and display on any device in a distributed network.

X applications are linked with toolkits and libraries that include windowing controls, user interface objects, and graphics capabilities. The X toolkits also include interprocess communications capabilities that provide data interchange between the application clients. Figure 2 presents some of the libraries in the DECwindows environment.

These applications communicate with an X Window System display server over a network through the X protocol. The X protocol is independent of operating system, network transport, and network wiring technology and topology. The display server provides basic windowing, graphics rendering, and user input services for X applications.

eXcursion Implementation

The eXcursion application implements the X Window System display server on Microsoft Windows. The eXcursion software allows the windows of the X applications, running on a remote host, to display on a personal computer. The two environments are visually integrated-applications from both environments display on a single screen and have the same visual appearance. The two environments use the same mechanisms to manage windows and thus present a consistent user interface. In addition, eXcursion uses metaphors and mechanisms familiar to the user of Windows. A control panel is employed to handle configuration and customization of the eXcursion application. The Windows Program Manager is employed to transparently invoke applications on remote hosts.

Figure 3 shows the eXcursion control panel, the Windows calendar, and the DECwindows Motif calendar as viewed on a desktop device. The Windows Program Manager is also displayed to show the eXcursion program group with icons installed. Users can simply double click the icons in the program group to start applications on a remote host.

figure 3 (Windows Display with eXcursion)-PHOTO, not available online.

eXcursion-A Component of PC Integration

One of the goals of Digital's PC integration program is to integrate PCs throughout a network so they may share resources. In a local area network (LAN) or a wide area network (WAN), PCs share files and printers through a file server. Traditionally, Digital has provided terminal emulation software for interaction with a time-sharing system on the network. The X Window System distributes another resource load throughout the network, namely application services. X applications can be run on a special-purpose host, such as a Cray system, or on a general-purpose host such as a VAX system. The applications share the CPU, memory, disk, and print resources of that host. Thus, the optimal or appropriate device can provide the application services. The eXcursion product is an X display server through which the PC user can access the X Window System class of application.

Because it enables information exchange among PC users and non-PC users throughout a network, the eXcursion software is a key component of Digital's Network Applications Support (NAS) and Digital's PC integration program in the Personal Computing Systems Group.

3 Design Philosophy

As in any software development project, a number of very important design goals and decisions were established for the eXcursion for Windows product which affected the implementation. The eXcursion application had to be extremely compatible with the Microsoft Windows environment. There were important reasons for this decision.

First, it was critical that eXcursion run on any PC, with any combination of devices that the standard Microsoft Windows environment supports. Typically, the manufacturer that builds the hardware is responsible for writing the Windows-compatible drivers. The devices that most affect eXcursion are keyboard, pointing device, and display.

Second, a tremendous amount of development effort has been invested in the functionality and performance of the Windows product. We wanted to apply that functionality and not duplicate it in the X server. For example, Windows software has a bit block transfer (BitBlt) routine that can more effectively handle that operation than eXcursion. It is one of the operations that Microsoft has optimized. In addition, it is one of the operations that can be customized and optimized for the PC's graphics adapter. If the graphics adapter can handle BitBlt operations with built-in hardware, it is more likely that the operation can be performed faster with that hardware than with the CPU. Therefore, eXcursion is completely insulated from the hardware and benefits from functions that have been optimized for specialized hardware.

The third reason for developing eXcursion as a well-behaved Windows application is independence from the internals of the underlying windowing system. We might have been able to do a slightly better job of integration of the Microsoft Windows and X Window System environments if we had obtained a source code license from Microsoft and truly blended the two environments into one. However, the cost, development resources, and time needed to implement this type of integration were prohibitive.

Fourth, the eXcursion application had to share the PC system resources of display, pointing device (mouse), keyboard, sound subsystem, memory, and network with another windowing system and its applications. The first five resources were all owned and managed by Microsoft Windows. We had to use its application programming interfaces (APIs) to correctly share those resources. The network resource was shared among many networked applications through its APIs as well.

Use of Windows Resources

A substantial portion of the design debate centered on the way eXcursion would use the Microsoft Windows resources. We needed to determine how to map the windows, graphics contexts, fonts, and color maps of the X environment to the windows, device contexts, fonts, and color maps of the Microsoft Windows environment.

The major dilemma was: Should each X window be created as a Microsoft Windows window and thus be known to both environments? Or should only the top-level X windows-those which were parented by the Windows desktop or root window-be created as windows in the Microsoft Windows environment, with all other windows created strictly as X windows and known only to eXcursion?

The first proposal was certainly easy to implement and led to consistency throughout the X server. The Windows environment had an API rich enough to make this plan feasible. In addition, Windows would handle all the window stacking and clipping for eXcursion fairly transparently. Despite these reasons, the alternative plan was proven more workable during our prototyping phase.

The X Window System was designed to employ many windows since they are considered to be inexpensive resources.[1] Servers use little memory for each window. X windows are fast to create, map, unmap, and destroy; and they can navigate quickly through the window tree. Thus, X-based toolkits, such as Motif, employ many windows. When we tested our initial proposal, we discovered that both windowing systems maintained window trees, which resulted in a performance problem. For example, when certain operations such as graphics were performed, some of the clipping was done twice, once by eXcursion and once by Microsoft Windows. In addition, Microsoft Windows limited the number of windows that could be created, by the 64 kilobyte (KB) memory it reserved for these and other system resources.

Functionally, the X Window System graphics contexts (GCs) mapped fairly well to the Microsoft Windows device contexts (DCs). However, the way X applications employ GCs is significantly different from the way Microsoft Windows employs DCs. X applications store many GCs; each is set up uniquely with different values for the drawing state variables. Sometimes many GCs are used for one window and often a different GC is used for each window. The use of many GCs can significantly reduce the communication between the X server and application, since graphics state is communicated only once. Microsoft Windows applications use one DC for all window painting, modifying it as needed. Some innovative caching algorithms in the eXcursion product were used to address this mismatch in usage style. <column>

Font resources were also efficiently mapped between the two windowing environments. A substantial portion of the graphics done by an application in a windowing environment is text. Microsoft recognizes this and optimized the text output routines in Windows. Thus, the optimal way of drawing text was through Windows. Therefore, the X server's font resources were compiled into Windows-compatible font file resources so Windows could do all the text drawing. For each X font resource, we included a second file for the font and glyph metrics that did not map to the Windows font file resource. Some of the eXcursion font file resources were modified to resolve inconsistencies between the two environments and make eXcursion compatible with Windows. For example, unlike X, Windows does not allow text drawing outside the characters' bounding box.

Color maps are another resource Windows shares with eXcursion. Microsoft Windows version 3.0 with standard video graphics array (VGA) hardware (a 640 by 480 resolution device with 16 colors supported) pre-allocates all 16 colors in the color table for the Windows environment. For eXcursion, this is effectively the X Window System static color visual, where the color map is read-only. With enhanced VGA cards that support 256 simultaneous colors, Windows pre-allocates 20 entries in the color table. For eXcursion, the X Window System's pseudocolor visual can be supported with only 236 entries for allocation in the color table. Again, it was important that eXcursion was well behaved with respect to color-map allocation and use within the Windows environment.

Performance Considerations

Performance of the eXcursion product is a continuing area of concern, investigation, and development. Many performance concerns were remedied by efficient code paths and innovative algorithms; others need to be addressed by the user in the form of trade-offs. In this section we discuss some major architectural differences between Microsoft Windows and the X Window System that leave X performance at a disadvantage when it is layered on another windowing system.

First and foremost, eXcursion has to translate X requests into Windows APIs as well as translate Windows events, API return values, and API errors into X events, X request replies, and X request error events, respectively. The disadvantage, of course, is the increased processing time eXcursion needs to complete these translation tasks. Since our design goal was to layer a foreign window system on the desktop device's native windowing system, we had to accept this performance penalty.

Second, X employs a client-server model. All X protocol requests of the X client, (X application) to the X display server have to be encoded into the X protocol and transmitted to the server through an interprocess communication mechanism. For the eXcursion product, this mechanism is a network because the client and server are always on different systems. Operations in X, e.g., menu sweeping and resizing of objects, always involve both the client and the server. These operations in particular have to be fast because they affect the user's perception of the windowing system's performance. Thus these code paths had to be efficient.

Third, X has strict pixelization rules. These rules determine which pixels must be included in the rendering of a graphics object. In general, all the interior points of an object are rendered, but only certain points on the outer boundary of the object are rendered. If the area of the pixel below and to the right of the center point is touched, then the pixel is included; otherwise it is not.[2] Thus, a rectangle has its top and left edges included, but not its right and bottom edges. The pixelization rules for the X protocol were strictly specified to satisfy the technical market's graphics requirements, such as CAD/CAM. If one were to tessellate polygons in the X environment, one would be guaranteed that each pixel is included once and only once.

The Microsoft Windows environment was designed with a business graphics presentation model. The pixelization rules are not widely known and may change.

Based on these facts, we chose to adhere to the X protocol and its pixelization rules. We believed most users would run office productivity applications. For these applications, pixelization rules do not affect the operation or functionality of the application. In a majority of cases, the user is never able to see the subtle differences in the rendering of a graphics object. As part of eXcursion's customization, we allow the user to select the way graphics are rendered-optimized for performance or optimized for correctness. This choice is analogous to printing draft (fast) mode for proof copies or letter-quality, high-resolution mode (high quality but slow speed) for final copy. The user can change this parameter at any time in eXcursion and force a redraw by the X application, e.g., through an iconify /deiconify procedure, to render the graphics in the other mode.

Seamless Integration

One of our design goals was the seamless integration of eXcursion into the Microsoft Windows environment to the greatest extent possible. Two important areas to integrate were window management and data exchange.

Window Management. We believed that Microsoft Windows should provide window management. Top-level windows in the two environments are peers and should be visually and functionally identical. With this capability the user does not have to run a remote window manager or learn and remember a second user interface.

We wanted the outer frame of the windows in X to look like the windows in Microsoft Windows. Furthermore, we wanted Windows to provide all of the end-user window management functionality- move, resize, iconify, deiconify, stacking, and focus. The windows for these operations had to contain the same user interface objects found in the Microsoft Windows environment.
We did violate this design principle in one case. In place of the standard Microsoft Windows system menu icon in the upper left corner of the window frame, we placed an "X" (see Figure 3). This object visually cued the user that the window represented an X Window System application running remotely but displaying within the Microsoft Windows environment.

On the other hand, X servers are not aware if the graphics object being rendered is a component of a scroll bar, command button, radio button, check box, text entry field, etc. For this reason, eXcursion cannot make graphics objects look like and function as the equivalent objects in the Microsoft Windows environment. Unfortunately, the user has to deal with these inconsistencies between the two windowing environments.

The eXcursion product had to conform to the X Consortium's Inter-Client Communications Conventions Manual (ICCCM) specification for window management within the Windows environment. Window properties such as name, icon name, size, and position on a top-level window must be recognized by eXcursion and must be set using the appropriate Microsoft Windows APIs.[3]

Data Exchange. We believed users should be able to seamlessly exchange text and bit-map data between the Microsoft Windows and X Window System environments. For example, the user should be able to use the standard application mechanisms to select data and cut or copy it from one environment, move to an application in the other environment, and use the standard application mechanisms to paste the data. No special user intervention between these two operations would be acceptable.

To enhance the data integration capabilities of eXcursion, we did implement a special feature to capture any part of an X window as bit-map data and save it in the Microsoft Windows clipboard. Microsoft Windows applications could then paste that data.

Cross-cultural Compatibility

eXcursion functions as any other Microsoft Windows application and conforms to its style guide in three areas-installation, configuration, and help.

The installation design principles are quite simple. Installation has to be performed through a Microsoft Windows application and has to allow the user to run the initial application without further configuration. Only two configuration parameters, fonts and keyboard, must be specified by the user. In addition, a user in the VMS, ULTRIX, or Sun OpenWindows environment has easy access to the standard applications of the operating system. The installation procedure installs icons that represent all of the standard DECwindows applications for the VMS and ULTRIX systems and standard Sun OpenWindows applications in the Microsoft Windows Program Manager. A user can invoke the application on the remote host using the standard Program Manager mechanisms, such as a double click of the program icon with the pointing device.

We devoted significant engineering resources to the configuration for eXcursion. Since the configuration was for a windowing environment, we decided to use the control panel metaphor that is common to other windowing environments, such as the Macintosh and Microsoft Windows. The eXcursion control panel (partially shown in Figure 3), provides access to all the user preference features and configuration parameters. Another important design principle was the immediate activation of configuration parameters or user preference features whenever it was technically feasible. We did not want the user to exit all the X applications or restart the X server to activate configuration parameters.

The eXcursion control panel also allows users to customize their X application environments. The eXcursion control panel provides a mechanism to build an applications menu within the control panel and install application start-up commands in the Microsoft Windows Program Manager as icons for easy invocation of remote applications.

On-line help also conforms to the Windows style guide. Our design goal
was to supply a concise Quick Start card with all the information a user
needed to determine the prerequisites for install, install the product, and
invoke the first application. All of the remaining end-user documentation

is available on line. The only other printed documentation is the reference manual.

For install, configuration, and help, human factors engineers provided usability evaluations, and a graphics designer assisted in the final design of the user interface.

X Server Internal Architecture

The X11 release 4 MIT sample server implementation provided the baseline for our development effort. This architecture is depicted in Figure 4. The sample server architecture has three distinct layers: device-independent X (DIX), operating system (OS), and device-dependent X (DDX). The DIX layer is primarily concerned with high-level decision making. The OS layer connects the X server to its underlying network transport. The DDX layer translates a client's request into a pixel display. To conform to the Windows application model, our implementation adds a fourth layer, the Windows message processing layer.

Device-independent X

The DIX layer consists of modules that provide high-level server data structure manipulation, X request vectoring, and server task scheduling. Every attempt was made during the development process to change as little as possible in this layer, and to maintain the firewall between the DIX layer and the underlying DDX layer. The DIX layer's most important task is the dispatch loop, the scheduler for eXcursion processing of all asynchronous client requests. Requests fall into three categories:

1. Edits to internal data structures such as the current procedure vector for drawing wide, dashed lines

2. Queries on internal resources such as available fonts and their metrics

3. Drawing requests such as rendering of text and lines

The DIX layer maintains the current state of the window tree and all its components, as well as the graphics contexts and all of their associated data. DIX code dynamically alters the processing paths chosen for x request completion based on the current states of these data structures. For example, suppose that a GC is being used to draw a series of single-width, solid lines in a window. now the X client wishes to begin drawing with 10-pixel-wide, tile-filled lines. Dix then reads the client requests dealing with the GC state changes, and updates its data to reflect the new drawing conditions for lines. DIX changes the drawing vector and updates the gc data structure. (Device-specific drawing operations are performed in the DDX layer.)

Windows Message Processing

The Windows message processing layer is the interface to the user's input devices, the mouse and keyboard. Actions taken by a user result in Windows messages containing information on the message type, conditions, and parameters being sent to the application's Windows message procedure. Here the data must be modified and translated into something that an X client can understand, an X event. Event processing is done by the DIX layer, and the event data is then shipped to the client by the OS layer.

Operating System

Data transferred on the X wire is arbitrated in the OS layer. When an X client application makes a server request, the underlying network code receives it, packages it, and makes it available to the OS layer. The eXcursion product runs layered above one of two entirely distinct network transports (either the DECnet or the TCP/IP protocol) and must provide some mechanism for passing data back and forth between the real mode of the network interface and the protected mode of a Windows application. For this reason, we chose to interface the server to the network by means of a generic OS module. Since all server-generated calls are now network-independent, the server is freed from any network-specific decisions.

Data conversions from real mode to protected mode are provided by a group of Windows dynamic link libraries (DLLs). Functions in DLLs are called directly from a Windows application (in this case, eXcursion). The DLLs in turn use Windows' extended memory manager to make DOS protected mode interface (DPMI) calls to pass the data to the network stack which runs in real mode. For example, assume eXcursion is running the TCP/IP protocol, and the user presses a mouse button in an eXcursion window. The data comprising the X event is assembled, packaged, and presented to the OS layer for shipment to the remote X client. The server makes its "send data" call into the generic OS module. This module makes a call into a common, shared DLL, and passes the data unchanged. The generic DLL acts as the network arbitrator. It knows about the underlying network transport and vendor since it performed a network installation check at start-up. Therefore, the generic DLL calls into the vendor-specific eXcursion DLL to modify the data, pack it into the format required by the network stack, and ship it to the real mode stack.

This implementation strategy requires several DLLs, but it completely shields the server, and more importantly the user, from the underlying network. The DLLs are simply copied once into the eXcursion execution path and forgotten. There is no need to reconfigure eXcursion if the underlying network changes.

Device-dependent X

All the visually recognizable work takes place in the DDX layer. DDX translates a client's X request into pixel manipulation on the screen. The sample server implementation that provided our starting point came with a DDX layer designed for monochrome frame buffer (MFB) devices. We replaced the MFB device-specific code in the DDX layer with implementation-specific code for Windows.

Our baseline sample server implementation also provided a machine-independent DDX mechanism (MI). The MI modules manipulate the video terminal as a virtual device: video memory is emulated and all drawing operations take place into this virtual space until the final output renders the bits onto the screen. The MI manipulates bits and performs logical operations until it achieves a final representation of the requested operation. This final drawing requires two distinct functions:

fill spans and push pixels. The fill spans function renders drawing output in single scan lines, making repeated calls to Windows BitBlt. The push pixels function does much the same thing, but at a more complex level-it pushes bits through a mask or filter before they appear on the screen. These mechanisms are required for proper text rendition when tile or stippled filled text characters are requested with unaltered character outlines and backgrounds. These mechanisms are, by definition, clumsy and inefficient, but they provide pixel perfect renditions. eXcursion uses these MI functions when any of the following conditions must be met.

1. Drawings are complex filled areas.

2. Tile and stipples used are not 8 by 8 pixels in size. (Windows is optimized to handle this one case, and breaks down easily for all other sizes.)

3. All operations require pixel perfection, such as display of a CAD application.

4 Using Windows APIs

We designed a set of Windows-specific modules that filled the hardware-dependent space provided by MFB. These functions are called by the DIX layer's request dispatcher through the request vectors set up in the server's main data structures (screen, window, GC; see Figure 5 for examples). All X relative drawing requests are translated here into Windows operations, and Windows APIs are called to satisfy them.

As described previously, we decided to match window trees by creating a Windows window for each top-level X window only. X child windows are handled as if they are rectangular areas of their parents, thereby saving room in the finite (64KB total size) pool of Windows resources available for other objects. This decision led to a difficult problem that needed a solution: how do we handle window clipping?

Window Clipping

Clipping is accomplished in X by maintaining a list, for each window in the system, of the rectangles into which drawing is allowed. Clipping in Windows is accomplished essentially the same way, but it requires allocation of another resource, a region. We implemented clipping by adhering to the X model, letting the server code do as much of the work as possible.

The DIX code manipulates and maintains a "clip list" for each X window. When a Windows window is created and used, Windows expects this clipping information to reside in the window's DC if something is to be drawn in the window. To get the X clip list into the Windows DC, we allocated a small pool of cached Windows regions. A DC (and X parallel GC) used for a drawing operation must be validated to ensure that all components are up-to-date. If the DC does not have a copy of the clip list, a Windows region is built from the rectangles in the X clip list and installed as the clipping region of the DC. When the drawing takes place, the clip list is installed. As long as the window is not moved, resized, or obscured, the region remains unchanged and further region validation is unnecessary. When the number of visible windows exceeds the cache limits, the least recently used DC is "thrown out" of the cache, and must be revalidated if it is used again. This mechanism allows smooth, efficient output to multiple windows without extensive use of Windows precious region resources.

Windows places a further restriction on resource usage. In addition to being created, a resource must be selected into a DC before it can be used. Deselected, old resources are deleted to save space. If a request asks for one of the deleted resources, it must be re-created and selected again. The caching and updating of DCs in Windows is handled by the same function that validates and refreshes GCs in X. When an X request results in a GC change, it may also result in a DC change. For example, if the line drawing mode changes from single-pixel-wide, solid fill to multiple-pixel-wide, tile fill, the GC is updated with new procedure vectors and data fields. At the same time, the DC must be updated so the next line drawing request results in a wide, tile-filled line. A Windows bit map is created for the X tile, and it is selected into the DC as the pattern. Any line then drawn using the DC results in a wide, tile fill. This method is used to update the DC whenever any GC object with a parallel Windows object is changed. The cache ensures that Windows objects can be allocated.

Drawing APIs

The Windows environment contains a rich collection of APIs designed to accomplish many types of drawing. The eXcursion application takes full advantage of these drawing APIs. Wherever X and Windows share drawing rules and conditions, the appropriate Windows API is called quickly to maximize performance. This mechanism is utilized when the user selects the "optimized for performance" drawing mode. When the rules between X and Windows differ, eXcursion calls the most appropriate API for the more common variants, again, to maximize performance. For example, since a wide, solid, horizontal line is rectangular, eXcursion calls the Windows FillRect API to draw it. Only rarely is the MI code path required.

Pixmap Manipulation

The X pixmap presented us with a major challenge. Since it is a bitwise representation of a visual object, its bit values must be maintained regardless of its use. Pixmaps can be used in a variety of ways by complex X client applications. Pixmaps can hold off-screen copies of window contents, or they can hold a pattern for a window background. They can provide a mask through which a color or pattern can be squeezed to give a stencil-like filling effect. They can also contain text characters prior to output.

The real challenge, however, lies in how pixmaps are manipulated. There are monochrome pixmaps, color pixmaps, pixmaps presented as an array of bits one color plane at a time, or packed to present each color plane for one pixel in succession. For these myriad forms and presentations we created a set of pixmap manipulation routines that translate back and forth between X and Windows. Since Windows provides a set of APIs for manipulating device-independent bit maps (DIBs), we stored the bit map internally in one, generic form regardless of its X representation. eXcursion extracts the bits, modifies them, and sends them to the client when it requests them in another format. One of the biggest performance bottlenecks in eXcursion lies in the pixmap format conversions which are constantly taking place under the surface. Since we have stored all pixmaps in device-independent format, the performance penalty is low.

Font Compiler

The X and Windows environments include a section dedicated to information about the font metrics and a section for the character bit maps. However, their font storage methods are different. Furthermore, since eXcursion is a compatible Windows application, it uses Windows fonts to draw text.

We designed a font compiler to create Windows-usable fonts from an X font file input. The font compiler takes a bit-map distribution format (.BDF)

(X Window System font files are supplied in this ASCII readable format) and produces two output files. One, called the X font file (.XFN), contains the X metrics readable by the server without having to load the character bit maps themselves. The other, a Windows font file (.FON), contains the character glyphs used by the Windows APIs. eXcursion's X-specific code uses the .XFN file to match available fonts with those requested, and to calculate string sizes, positions, character offsets, ascents, descents, and anything else related to the location and position of the characters. The .FON file is loaded as a Windows resource, selected into a DC as described above, and used for any drawing operations since it contains the actual character representations. The font compiler can generate custom fonts; any font compiled with it produces a Windows font file suitable for use in any other, non-X, Windows application. For example, any of the supplied eXcursion fonts could be used with Word for Windows.

5 Handling Input Devices

In the section Seamless Integration, we described our design strategy for eXcursion to handle drawing requests from X clients. In this section we discuss requests from the user.

When a user clicks a mouse button, or moves the mouse, or types on the keyboard, Windows generates messages which are shipped to eXcursion's Window message processing function. Interrupt processing is not needed since Windows shields eXcursion from the underlying hardware. In fact, eXcursion has generic input handlers that work with any hardware configuration supported by Windows.

The message processor translates the data into a format understood by X, then packages and transmits it over the X wire as an X event. Since these user-initiated actions are asynchronous events, eXcursion calls the Windows PeekMessage() function when it has finished processing an X request, or when it is in the idle loop.

Windows and X share the same coordinate mapping conventions. When eXcursion receives a mouse move message, it does not perform translations on the x and y coordinates; it merely reports in which window the pointer resides. Furthermore, when eXcursion creates a window in Windows, it stores the corresponding X window's handle in the extra data area of the Windows window structure. It can retrieve the handle of a matching X window at any time with the Windows API GetWindowLong(). Since eXcursion always matches a Windows window to a top-level X window, the combination of the top-level window handle and the x and y coordinates of the pointer allows eXcursion to scan the X window tree and determine which child window holds the pointer.

When a user presses a mouse button, the same kind of activity is used to determine which window contains the pointer. The X event data structure is filled in and shipped to the client for further action.

When a user presses a key on the keyboard, much the same processing takes place. Windows sends eXcursion all the information needed to build an event data structure containing the key state, the scan code of the key, and the key modifier state (whether Alt, Ctrl, or Shift are depressed). eXcursion then packages and ships the data structure to the client application.

eXcursion loads a keysym file at start-up. The file contains the keyboard mapping of hardware scan codes to keysym definitions for the user's keyboard. It permits custom configuration for a user's keyboard. The keysym compiler in eXcursion takes an ASCII text, keyboard mapping file as its input, and produces a binary keysym file as its output. As long as the user follows the layout of the input ASCII file, any key can be remapped in any way desired.

Manipulating Application Windows

As stated previously, eXcursion uses the Microsoft Windows window manager to manage and manipulate windows. Whenever the user moves, resizes, iconifies, maximizes, or closes a window, either by the Windows system menu or the mouse, Windows sends the eXcursion window procedure a message with specific parameters. For example, a message sent when a window is resized contains the old and new sizes and origins of the window. eXcursion translates every Windows input message into an X event and sends it to the X client.

Individual messages from Windows generally correspond to X event types that provide data to clients. However, complications arise when Windows generates multiple messages for a single action. For example, when a user presses a button to select an item from a menu, a new window is created, mapped, sized, placed on the screen, activated, and given the input focus-all as a result of the single user action. Windows messages are generated for each of these operations, yet the user has provided no further action.

To handle this extremely complex web, we benefited from our initial design decision to create only top-level Windows. We eliminated literally hundreds of Windows messages for each child window, simply by not creating them. Messages are sent only to the top-level window, and eXcursion can quickly determine which child (if any) needs attention. On the other hand, we had to observe and study window stacking, configuration, reparenting, activation, and window focus before we arrived at the final implementation. Only through extensive prototyping and empirical testing were we able to eliminate poor design choices and arrive at the best ones. As a result, every possible window manipulation action, whether initiated by the user or directed by a client, requires a translation from Windows to X and a careful selection of Windows function calls to keep the delicate balance between X and Windows.

Cutting and Pasting Data

To cut and paste data between X and Windows applications, we merged the Windows clipboard mechanism with the X selection mechanism by incorporating the cut/paste "pseudo-client" into eXcursion. This module watches for data cut-and-paste requests from X clients, as well as those from any Windows applications running on the PC. When it notices an X client gaining control of a selection, it asks the controlling client for the selected data, which it then puts into the Windows clipboard. The data thus becomes available to any Windows application with access to the clipboard. When a Windows application cuts or copies data into the Windows clipboard, the pseudo-client is notified, at which point it informs all X clients that it now owns the clipboard selection. X clients can then request the data from the pseudo-client by selecting paste from their edit menus.

Accessing Remote Applications

The user initiates remote X client applications through an application launching mechanism that provides several starting options.

1. Selection of an application from the eXcursion control panel's application pull-down menu

2. Selection from a dialog box of defined applications

3. Selection of the "Start X Application" dialog box

4. Double clicking on an icon installed for the application in the Windows Program Manager

The most interesting option, double clicking on an installed icon in the Windows Program Manager, allows the user to start up an X application without any knowledge of the current state of eXcursion. The double click activates XREMOTE.EXE, the remote application launcher. XREMOTE sends out a Windows message, with an identification known only to eXcursion. If eXcursion responds, XREMOTE passes it the command line for application start-up. If eXcursion does not respond within a short timeout period, XREMOTE issues a WinExec call, requesting start-up of eXcursion itself. Windows starts up eXcursion, passing it the command line string for the selected application start-up sequence. XREMOTE then terminates until the next start-up request.

Obviously, security is a major concern for any system that requires and handles account passwords; eXcursion application activation is no exception. Users log into their accounts by activating an X application such as DECterm. Two distinct passwords are required: (1) the eXcursion global, session password and (2) individual, application account password.

The eXcursion session password is optionally selected and set by the user from a control panel dialog box. It is stored as an encrypted string in the initialization file, and is used as the decryption key for the individual application account passwords, also stored in the initialization file. This design prevents an unauthorized person from using someone's .INI file to obtain access to an account. The user is prompted for the session password when eXcursion starts up. If an incorrect value is entered, the server terminates and application activation is impossible. A further level of security is provided by the "Prompt for Password" option, which the user can select for any application start-up.

6 Summary

The eXcursion for Windows display server seamlessly integrates the Microsoft Windows and X Window System environments. It provides a desktop integration tool that allows the user to display and interact with applications designed for both windowing systems at the same time. Data can be exchanged between them and desktop resources shared. A user is no longer required to work with two incompatible desktop devices in order to complete work assignments.

7 Acknowledgements

The authors would like to thank everyone who worked on the product during its development. In particular we would like to thank the other full- or part-time members of the software development team: Ray Shapiro, John Freitas, Mike Pfeffer, Lee Karge, Alice Chen, Mary VanLeeuwen, and Andy Nourse. Two other members of the PC DECwindows Group who work on the DOS-based X server, John Wasser and Jim Peterson, provided some valuable assistance. We are also indebted to the following for their support and contributions: Emilie Schmidt, Carnel Hoover, Kathy Maxham, Andre Fontaine, Alice Chen, and Tracey Wemett. This great group of people made this project a joy to work on and a success.

8 References

1. R. Scheifler, J. Gettys, and R. Newman, X Window System C Library and Protocol Reference (Bedford, MA: Digital Press, 1988): xvii.

2. R. Scheifler, X Window System Protocol (Cambridge: MIT Laboratory for Computer Science, 1989): 37.

3. D. Rosenthal, Inter-Client Communication Conventions Manual (Cambridge: MIT Laboratory for Computer Science, 1989): 18-36.

9 General References

Digital Technical Journal, vol. 2, no. 3 (DECwindows Program, Summer 1990).

R. Scheifler, J. Gettys, and R. Newman, X Window System C Library and Protocol Reference (Bedford, MA: Digital Press, 1988).

Microsoft Windows Software Development Kit Reference, vols. 1 and 2 (Redmond, WA: Microsoft Corporation, 1990).

Microsoft Windows Software Development Kit Guide to Programming (Redmond, WA: Microsoft Corporation, 1990).

10 Trademarks

11 Author Biographies

Dennis G. Giokas Dennis Giokas is the group technical lead for PCSG's Network Client Engineering and the engineering manager for its New User Interface Group. His primary responsibility is technical lead for the next generation of the PATHWORKS for DOS and OS/2 products. Prior to this work, Dennis contributed to PC DECwindows development. Before joining Digital in 1984, he was employed by Arco Oil & Gas and The Foxboro Company. Dennis holds a B.M. (1974) from the University of Massachusetts at Lowell, a M.M. (1976) from the New England Conservatory, and a M.S.C.S. (1989) from Boston University. He has two patents pending.

Andrew T. Leskowitz A principal software engineer in the PCSG X Server Development Group, Andy Leskowitz is the project leader for the eXcursion display server. Since coming to Digital in 1987, he has contributed to various X development projects and designed the PATHWORKS LANSESS component. Andy's prior experience includes engineering positions at Datatrol, The Foxboro Company, and Raytheon Company. He has a B.S. (1976) in biology from Swarthmore College. Andy has applied for a patent related to his X server development work.