Design of eXcursion Version 2 for Windows, Windows NT, and Windows 95

John T. Freitas
James G. Peterson
Scot A. Aurenz
Charles P. Guldenschuh
Paul J. Ranauro

Version 2 of the eXcursion product is a complete rewrite of the successful Windows-based X server software package. Based on release 6 of the X Window System version 11 protocol, the new product runs on Microsoft's Windows, Windows NT, and Windows 95 operating systems. The X server is one of several components that compose this package. The other components are X Image Extension, the control panel (which constitutes the user interface for product configuration), the error logger, the application launcher, and the setup program. An interprocess communication facility enables the eXcursion components to communicate in a uniform fashion under all three operating systems. A unique server design using object-oriented programming techniques integrates the X graphics context with the Windows device context into a combined state management facility. The resulting implementation maximized graphics performance while conserving Windows resources, which are in limited supply under the 16-bit version of the Windows operating system. The control panel was coded completely in the C++ programming language, thus making full use of the Microsoft Foundation Class library to minimize development time and to ensure consistency with the Windows user interface paradigm.

Digital developed the eXcursion family of display server products to provide interoperability between desktop personal computers (PCs) running the Microsoft Windows operating system and remote hosts running the X Window System operating system under the UNIX or OpenVMS operating systems. The first version of the eXcursion X server was a 16-bit application written specifically for Microsoft Windows versions 3.0 and 3.1. As the popularity of Windows increased and desktop systems were connected to corporate networks, the market for X interoperability grew quickly. The 16-bit eXcursion code, much of which had been ported from 32-bit UNIX code, was again ported—this time to Microsoft's Win32 application programming interface (API) to support the Windows NT operating system. When release 6 of the X Window System version 11 protocol (X11R6) appeared and a new sample implementation source kit became available from the X Consortium, the eXcursion team decided that it was time for a complete rewrite of the eXcursion software. Microsoft had established the Win32 API as a uniform coding interface for all its Windows-based operating systems. Since development tools such as 32-bit compilers and debuggers of sufficient quality and robustness had become available, it was now possible to implement a high-quality, 32-bit product. This product would support the entire range of Windows-based platforms, from notebook PCs running the Windows operating system to high-end Alpha systems running the Windows NT operating system.

Terminology

This paper incorporates certain conventions to clarify the distinction between the two window systems under consideration. *X window* refers to the collection of data structures, concepts, and operations that constitute a window, as defined in the X Window System environment. *Win32 window* refers to a window as defined in Microsoft's Win32 API.

When referring to a window system as opposed to a particular window instance, *X Window System* is sometimes abbreviated to *X. Windows* denotes the Microsoft Windows operating system.

Note that the word *bitmap* has more than one meaning. In the X environment, a bitmap is a two-dimensional array of bits, and a *pixmap* is a two-dimensional array of pixels, where each pixel may consist of one or more bits. Under the Win32 API, the term bitmap is used exclusively; that is, no distinction is made between an array of depth 1 and an array of depth *n*. In this paper, the term pixmap is used in its general sense to refer to X pixel arrays, and the term bitmap refers to the Win32 concept.

Another common point of confusion when discussing the X Window System environment is the use of the terms *server* and *client*. To one familiar with file and print servers, the meanings of these two terms in the X environment may seem to be reversed. In the X environment, the server is a display server, and the clients are the applications requesting display services. The X server and the X client applications may reside on the same PC, but the power of the eXcursion software is in its ability to bridge the gap between the Windows desktop and the traditional X11 UNIX and OpenVMS workstations.

eXcursion Version 2 Product Goals

The design of eXcursion version 2 was driven primarily by the following product goals:

- Support X Window System version 11, release 6.
- Support the Microsoft Windows, Windows NT, and Windows 95 operating systems.
- Code the single source pool to Microsoft's Win32 API.
- Exceed graphics performance of eXcursion version 1 as measured with the standard benchmark tests X11perf and Xbench.
- Preserve maintainability by using modular coding and limiting changes of the sample implementation from the X Consortium.
- Maximize reliability by performing extended error checking and resource management.
- Correct known protocol conformance deficiencies in version 1. For example, in version 1, plane mask support was implemented for only a few graphics operations. Version 2 would provide plane mask support for all graphics operations.

Components of eXcursion Version 2

In eXcursion version 1, most of the functions provided by the product were combined in a single executable. To conserve resources and to partition the code for easier maintenance, version 2 is divided into several separate components or modules. Some of these run as individual processes, and some are built as dynamic link libraries (DLLs). A DLL is a shared memory library module that is linked to the calling program at run time.

eXcursion version 2 is partitioned into the following major components:

- X server. The X server is the primary component of eXcursion version 2. The X server process is responsible for displaying windows and graphics on the Windows desktop and for sending keyboard, mouse, and other events to the client application.
- X Image Extension. X extensions are additions to the server that support functionality not addressed by the core X11 protocol, such as displaying shaped (nonrectangular) windows, handling large requests, testing/recording, and imaging. All extensions except the X Image Extension (XIE) are implemented internally in the X server. Because of its size, XIE is implemented as a pair of DLLs, one for XIE version 3 and one for XIE version 5.
- Control panel. As the primary user interface, the control panel provides the user with access to the many configuration settings. It is an independent Win32 application implemented using Microsoft Visual C++ and the Microsoft Foundation Class (MFC) library.
- Interprocess communication library. The interprocess communication (IPC) library is an operating system–independent library used by cooperating processes or tasks to communicate configuration and status information.
- Error logger. The error logger is a simple Win32 application that records error and status information from other eXcursion components in a window, a file, or the Windows NT event log.
- Application launcher. The application launcher is a Win32 application that starts X client applications at the request of the X server or the control panel. The application launcher is invisible to the user.
- Registry interface. The registry interface is an operating system–independent interface to the eXcursion configuration profile. The registry interface is implemented as a Win32 DLL.

X Server

The core of the eXcursion product is the X server, a Win32 application that accepts X requests from client applications and transforms them into graphics on the Windows desktop. The device-independent portion of the server code is ported from the sample implementation provided by the X Consortium. The device-dependent portion treats the Win32 API as the device interface through which client requests are materialized on the screen. The eXcursion X server is illustrated in Figure 1.

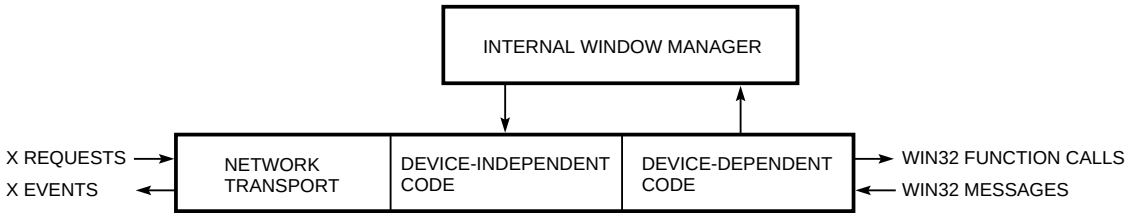


Figure 1
The eXcursion X Server

The server can operate in one of two modes: single-window mode or multiwindow mode. In single-window mode, the server creates one Win32 window, which represents the X root window. All descendant windows and their contents are drawn into the root window using Win32 function calls. In multiwindow mode, the root window is a virtual window; that is, it is never drawn on the screen. Each top-level child of the root window has a corresponding Win32 window, which is created when the X window is mapped. All descendants of a top-level window are drawn inside the Win32 window with Win32 calls. Multiwindow mode thereby creates a desktop environment in which X applications are peers of other Win32 applications.

Single-window mode is useful for emulating a complete workstation environment including the window manager and the session or desktop manager. In multiwindow mode, drawing to and getting input from the root window is restricted by the X server to prevent conflicts with the Microsoft Windows system's use of the desktop window. Despite this restriction, the multiwindow mode, when used with the native window manager, provides the cleanest integration of the X and Windows environments.

Resource Management and Performance

Both the X and Win32 systems have built-in notions of graphics state and resource allocation. The semantics and usage of the concept, however, are quite different in the two window systems.

In X, graphics state is maintained in a data structure known as a graphics context (GC). A GC has an independent existence and may be created, destroyed, updated, queried, and copied at will by the X application. During graphics operations, a GC is associated with the X "drawable" (window or pixmap) being drawn into, and information in the GC is used to fully define the operation. For example, the GC may specify foreground or background colors, line styles, or font information.

The Win32 API has a concept called a device context (DC), which also contains state information but whose purpose is more closely related to providing device independence. Consequently, two different types of DCs are required under the Win32 API, depending on whether the graphics operation is drawing to a window or to a bitmap. Furthermore, a window DC may be allocated either permanently or from a cache, depending on its expected lifetime. Any drawing operation therefore requires that both the GC used in the X graphics request and the DC used in the ultimate Win32 call be properly set up and synchronized. The manner in which this is done has a significant effect on the graphics performance of the server.

Before an X graphics operation can be started, the GC must be validated. Validation is a process of preparing the output device to render the graphics properly. In the case of the eXcursion server, the output device is a Win32 DC. For every graphics command, the GC must be checked for changes and the appropriate Win32 objects and state values must be selected into the DC. This process can be very time-consuming. The key to maximizing performance is to recognize that most operations are repetitive. A typical stream of X requests tends to contain many commands directed at the same window with the same GC. Therefore, the way to reduce GC/DC validation time is to cache the most recent GC/DC pair so that subsequent commands that use the same combination need not trigger a validation step. In some cases, graphics operations will toggle between two or more GCs. (For example, the CopyArea operation takes a source and a destination.) The performance in these cases can be improved by simply caching more than one recent GC/DC pair. Tuning experiments on the server revealed that a cache size between 2 and 4 was sufficient to maximize performance. Under the Windows and Windows 95 operating systems, where resources are limited, a cache size of 2 is used. Under the Windows NT operating system, the cache size is 4.

In the eXcursion server, the notion of a cached GC/DC pair is encapsulated in a C++ class called a WXDC. The WXDC remembers the Win32 objects that have been selected into the DC and the last GC that was used with it. As long as these elements do not change from one graphics operation to the next, no validation is necessary. If the client application changes the contents of the GC, any affected objects in the DC are tagged and the next graphics operation on that WXDC will require new objects to be selected into the DC.

Events in the window system can also cause WXDC elements to become invalid. For example, if the window is moved on the screen by the window manager, its clip list may have changed. This causes the WXDC to invalidate the clip region in its DC. (Clip list and region are defined in the following section.) The next graphics operation on that window will require the clip region to be recalculated and reloaded.

Clipping in Single-window Mode

In the X Window System environment, all descendants of the root window have a clip list, which is a list of rectangles that defines the visible area of the window. The clip list is equal to the area of the child window minus any areas that are occluded by other X windows. Before drawing into a descendant window, the server must convert the clip list into a Win32 region. In the Win32 API, a region is a polygonal area, not necessarily rectangular, that can be selected into a DC for clipping. Before initiating a graphics output operation, the target WXDC checks to see if the current region for the window is valid. If it is not, the X clip list is converted to a Win32 region and combined with the client-supplied clip list in the GC, if any. The result is selected into the output DC.

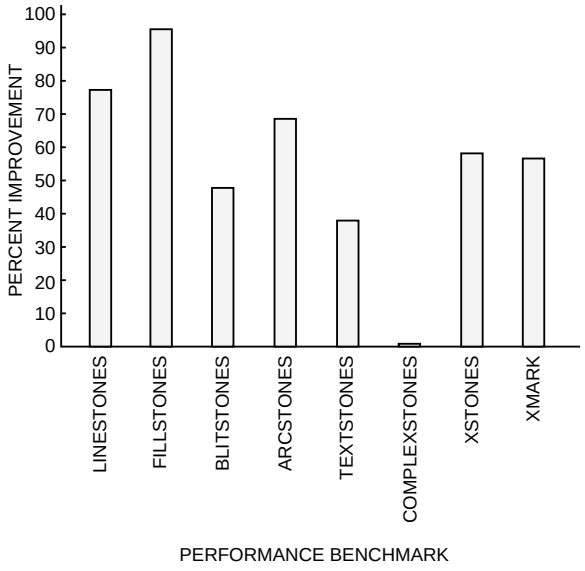
Clipping in Multiwindow Mode

In multiwindow mode, the root window is invisible. Each top-level X window (first-generation child of the root) corresponds to a Win32 window on the desktop. No clipping is necessary for these windows, because Win32 does this automatically. For windows below the first generation, clipping is accomplished in a manner similar to that used in single-window mode, except that the offset of the clip region must be adjusted to be relative to the top-level window instead of relative to the root window.

Graphics Rendering

Graphics rendering is at the heart of the X server. Two of the core goals for the eXcursion version 2 project were to significantly improve server performance over that of the eXcursion version 1 server and to improve server compliance to the X protocol specification. Figure 2 compares the performance of the eXcursion version 2 server with that of the version 1 server. The standard benchmark tests X11perf and Xbench were run over a local area network to eXcursion running on a 66-megahertz Pentium processor with an S3 video card.

The sample X server upon which the eXcursion X server is based provides a machine-independent layer that is capable of rendering all X graphics through a small set of device-dependent functions. In the eXcursion X server, the Win32 functions provide the virtual hardware interface. For maximum performance, X graphics requests are passed to the Win32 API as early as possible without compromising the requested rendering. Many X graphics requests map neatly into Win32 calls with little or no data manipulation. Some complex graphics requests, however, cannot be practically mapped into high-level Win32 calls and achieve proper pixelization. In such cases, the machine-independent functions are called as helper functions to break the request down into simpler graphics requests.



Performance Benchmark	eXcursion Version 1	eXcursion Version 2	Improvement
XBench			
lineStones	135,735	239,740	76.6%
fillStones	38,083	74,331	95.2%
blitStones	59,743	88,320	47.8%
arcStones	2,172,720	3,662,770	68.6%
textStones	156,190	214,762	37.5%
complexStones	71,633	71,699	0.1%
XStones	80,057	126,408	57.9%
X11perf			
Xmark	**1.6495**	**2.5805**	**56.4%**

Notes:
The test machine was a DECpc XL 566.

Since eXcursion version 1 did not support 16-bit fonts, the version 2 numbers were substituted to obtain the Xmark number.

Figure 2
Comparison of eXcursion Version 1 and Version 2 Performance

GDI Context Switching To reduce context switching, Windows batches graphics device interface (GDI) calls. The default GDI batch size is 20, but this limit can be adjusted per thread. Testing with a mix of all X requests showed that an overall performance increase of about 9 percent could be achieved by increasing the GDI batch limit to 30. At this level, there is no measurable latency, and, furthermore, increasing the batch size beyond this point had no measurable benefit.

Some competing X server products set the batch size very high (100) at the beginning of every request and flush the queue at the end. This approach has no measurable benefit over our simpler method, probably because the Windows operating system already performs timer-based flushing to prevent drawing latency.

Similarly, whenever possible, Win32 graphics calls are combined to reduce the overhead of context switching. For example, an X PolyLine request could be rendered with a series of Win32 LineTo calls, but it is much more efficient to render the PolyLine request with a single Win32 PolyLine call. Similarly, a PolyRectangle X request is best rendered with a single PolyPolyLine call.

Solid Fills Many different Win32 resources such as pens, brushes, fonts, and clip regions may be required for any given graphics request. The resources needed are determined by the graphics operation itself and the state of the X GC. As noted earlier, these resources are created as needed and managed by theWXDC objects, removing significant complexity and nearly redundant code from the actual graphics drawing routines.

Windows Pen structures provide color and dash pattern when drawing line objects. For drawing lines, segments, and arcs, the X server creates and uses Pens that correspond to the GC state. In some cases, however, exact pixelization cannot be achieved when using Windows Pens. Examples of this are drawing wide lines with raster operations other than GXcopy or with long, dash patterns. In these cases, machine-independent functions are used to reduce the request to a set of spans (single-width horizontal lines) to be filled. The use of Pens is also abandoned in special cases when the highly optimized GDI pattern block transfer (PatBlt) function can be used. PatBlt fills rectangular regions with specified colors or patterns. It is faster, for example, to use the PatBlt function to draw vertical or horizontal lines than to use the Windows traditional line-drawing functions.

Windows Brush structures provide color and pattern when drawing filled rectangles, filled polygons, and filled arcs. Again, for performance reasons, the PatBlt function is often used even when there is a higher-level function that seems to be a closer match. For example, PatBlt can perform the X PolyPoint request about 10 percent faster than SetPixelV, the Windows standard call for setting single pixel values. Similarly, PatBlt can perform the X PolyFillRect request about 14 percent faster than the Windows FillRectangle call.

Tile and Stipple Fills An X pixmap can be specified as a pattern to be used when performing fill operations. When the pixmap is created, it is realized as a Win32 bitmap. When the pixmap has a depth greater than 1, it is used as a color tile that will be used for the fill. If the pixmap has a depth of 1, it can be used as either a transparent or an opaque stipple. An opaque stipple draws both the GC's foreground and background colors, where the stipple is 1 and 0 respectively. A transparent stipple is similar except that it leaves the destination untouched where the stipple is 0.

When the tile or opaque stipple is 8 by 8 or smaller, a Win32 color brush is created and cached for the drawing. On the Windows NT system, brushes larger than 8 by 8 can be created, but our experience has shown it to be slower to draw with them than it is to perform a series of bit block transfer (BitBlt) operations from the tile/stipple bitmap to the destination.

Transparent Stipple Fills There is a Win32 function, MaskBlt, that seems ideally suited for performing transparent stipple fills. This function, however, was not fully implemented on all platforms at the time we designed the eXcursion version 2 software product. Without this function, there is no easy way in the Win32 environment to perform the transparent stipple operations. When the foreground color is either 0 or 0xFFFF, the raster operation can be remapped to get the proper effect. General rectangular fills that do not meet the requirements of the special case previously mentioned must be accomplished by first converting the stipple bitmap to the depth of the destination and then remapping the raster operation. In general cases that are not rectangular fills, machine-independent functions are called to break down the request into spans.

Image Requests The GetImage and PutImage requests are other examples of X graphics requests that do not map well into the Win32 API. The only way in the Win32 environment to put image data on the screen is to first create a Win32 bitmap and initialize it with the image data, and then call the BitBlt function to copy the bitmap to the screen. X image data always lists the top scan lines first, whereas the bottom scan lines are listed first in Windows bitmap data. Therefore, before the bitmap is initialized, the X image data must be scan-line flipped. Similarly, the X GetImage request requires the use of an intermediate bitmap and also requires the scan-line flip.

Plane Mask Support Any graphics operation in X can be modified by setting a plane mask in the GC. The plane mask specifies which bits of the destination pixel are allowed to be changed. Without a plane mask, an X graphics operation may be defined as

$$\text{dst} \leftarrow \text{src} \otimes \text{dst},$$

where $\otimes$ is one of the 16 binary raster operations (e.g., OR, AND, and XOR). When a plane mask is given, the following assignment defines the destination pixel:

$$dst \leftarrow ((src \otimes dst) \& pm) \mid (dst \& \sim pm)$$

Most video hardware devices support plane masking, and those that do not support it generally provide fast access to video random-access memory (RAM). The Win32 API, however, provides neither plane masking nor direct video RAM access. To understand why, you must realize that Windows has virtualized the color handling in an attempt to mediate conflicts between applications that would otherwise want to modify the colormap (the pixel–to–color mapping table). In this virtual color environment, the concept of plane masks has no meaning because Win32 applications need not know the pixel value that corresponds to a particular color. See the section Color Resource Management for an explanation of how the eXcursion software manages to assign specific pixel values to colors.

In the general plane mask case, it is necessary for the X server to first save the contents of the destination in a bitmap. The graphics can then be temporarily drawn without regard to the plane mask. Those bits in the destination that are specified by the plane mask as being unaffected can then be restored from the saved bitmap. This process will work in every case but is inefficient since it involves several graphics operations before achieving the final result. Many special cases can be reduced to one or two simple steps by modifying the source color and raster operation. Table 1 shows how the source color and raster operation can be set to achieve the plane mask effect. The eXcursion X server uses these optimizations for many graphics operations when the source fill is a solid color.

Internal Window Manager

In the absence of a window manager, the eXcursion server creates all windows as pop-up windows. All windows, including top-level windows in multiwindow mode, are undecorated. They have no Win32 borders, title bars, or system menus. To move, size, minimize, maximize, or close windows, the user must run a window manager.

An eXcursion user always has the option of using one of the many X-based window managers available, such as the Motif Window Manager. However, many users will want a window manager paradigm that is consistent with Windows so that all windows on the desktop have the same user interface. To accomplish this, a built-in window manager is provided as part of the eXcursion server. This internal window manager is operative only in multiwindow mode.

The internal window manager, although linked with the server, is functionally isolated from the rest of the code so that it can easily be disabled. This allows external window managers to be used and also facilitates debugging by allowing problems to be isolated. The window manager creates a "hook" into the server's window procedure, so that all Win32 messages are first examined by the window manager. This gives the window manager the opportunity to act on window management–related messages such as those that indicate a change in the window's configuration or state. If the window manager decides to handle a message, it is removed from the queue, and the server never sees it. If the window manager is not interested, the message is passed on to the normal window procedure.

The purpose of the internal window manager is to give X windows the same appearance and behavior as Win32 windows that are created by typical desktop applications, such as word processors and spreadsheets. When an X window is mapped for the first time, the internal window manager receives a Win32 WM_CREATE message. Before the window becomes visible on the screen, the window manager alters the style of the Win32 window to WS_OVERLAPPEDWINDOW. Win32 windows with this style are automatically managed by Windows, which handles moving, resizing, iconifying, maximizing, and closing the windows. Each of these actions causes a corresponding message to be sent to the server's window procedure. The internal window manager intercepts the messages and dispatches them to the appropriate internal function.

The role of the internal window manager complements the role of the server. The server processes client requests on X windows and translates them into operations on Win32 windows. The internal window manager handles Windows messages that indicate changes to a Win32 window and translates them into corresponding changes to the underlying X window. For example, the most important message that the window manager handles is WM_WINDOWPOSCHANGING. This message is sent just before any change in the window's position, size, stacking order, or visibility. If this message indicates that the window size changed, the window manager changes the size of the corresponding X window and sends a ConfigureNotify event to the client. Similarly, the window manager translates other user-directed events such as focus change, window stacking, and iconification into changes to the underlying X data structures. In most cases, the window manager does this by calling into the device-independent layer, thus simulating an X request that would occur from an external window manager.

Mouse, Keyboard, and Input Focus

Mouse actions and keystrokes are received by the eXcursion server as Win32 messages. Each message contains information about the window that received the input and the time of the input. For mouse moves and clicks, the server uses the window information to locate the corresponding X window and forwards an X event to that window. Keyboard input is forwarded to the window that currently has X focus.

Table 1
Plane Mask Optimizations

Requested X Raster Operation	src 0 dst 0	0 1	1 0	1 1	Notes	Modified Source Color and Raster Operations
GXclear	0	0	0	0	4	src ← ~pm, rop ← and
GXand	0	0	0	1	1	src ← src | ~pm
GXandReverse	0	0	1	0	6	src ← src | ~pm src ← ~pm, rop ← xor
GXcopy	0	0	1	1	8	src ← ~pm, rop ← and src ← src & pm, rop ← or
GXcopy (src & pm) = pm	0	0	1	1	8	src ← pm, rop ← or
GXcopy (src & pm) = 0	0	0	1	1	8	src ← src | ~pm, rop ← and
GXandInverted	0	1	0	0	2	src ← src & pm
GXnoop	0	1	0	1	10	—
GXxor	0	1	1	0	2	src ← src & pm
GXor	0	1	1	1	2	src ← src & pm
GXnor	1	0	0	0	7	src ← src & pm src ← ~pm, rop ← xor
GXequiv	1	0	0	1	1	src ← src | ~pm
GXinvert	1	0	1	0	5	src ← pm, rop ← xor
GXorReverse	1	0	1	1	7	src ← src & pm src ← ~pm, rop ← xor
GXcopyInverted	1	1	0	0	9	src ← ~pm, rop ← and src ← ~src & pm, rop ← or
GXorInverted	1	1	0	1	1	src ← src | ~pm
GXnand	1	1	1	0	6	src ← src | ~pm src ← ~pm, rop ← xor
GXset	1	1	1	1	3	src ← pm, rop ← or

Notes:

1. dst is unchanged when src equals 1 for these raster operations. Therefore, to preserve the value of dst when pm equals 0, set src equal to 1.
2. dst is unchanged when src equals 0 for these raster operations. Therefore, to preserve the value of dst when pm equals 0, set src equal to 0.
3. This operation sets all dst bits to 1 except where the plane mask equals 0. This can be done simply by ORing pm into dst.
4. This operation clears all dst bits except where the plane mask equals 0. This can be done simply by ANDing pm into dst.
5. XORing with 1 has the effect of inverting. To invert only where pm equals 1, XOR pm with dst.
6. These operations are performed in two steps. Note that dst is inverted when src equals 1. First perform the operation with src set to 1 where pm equals 0. dst is now correct except that it is inverted where pm equals 0. The second operation of XORing with the invert of pm corrects this.
7. These operations are performed in two steps. Note that dst is inverted when src equals 0. First perform the operation with src set to 0 where pm equals 0. dst is now correct except that it is inverted where pm equals 0. The second operation of XORing with the invert of pm corrects this.
8. This operation is performed in two steps. First dst is set to 0 whenever pm equals 1. Then dst is set to 1 whenever both pm and src equal 1. The two special cases can be reduced to operations that use GXset and GXclear.
9. This operation is performed in two steps. First dst is set to 0 whenever pm equals 1. Then dst is set to 1 whenever pm equals 1 and src equals 0.
10. dst is unchanged; therefore, no operation is required.

The X server is a single application in the Win32 environment that "owns" all the X windows it creates. From the user's perspective, though, there may appear to be more than one X application running, each with its own collection of windows. The user expects to be able to shift the keyboard focus from one window to another in the same fashion that focus is shifted between other applications. When an external window manager is in use, focus control is straightforward. The window manager, using whatever semantic it was designed for, monitors mouse events and shifts focus accordingly. However, the semantic model for this may or may not be consistent with the Win32 model. In either case, the window decorations, e.g., borders, title bars, and menus, are almost guaranteed to be different. A user who wants a consistent user interface model across all applications must employ the internal window manager.

At any given time, one window on the screen has Win32 focus and one X window has X focus. The two windows are not necessarily the same. Since the X server creates and owns all the X windows in use, the server receives keyboard input when any one of its windows has Win32 focus. The keystrokes are not necessarily sent to the underlying X window, however. They are sent to the window that has X focus. The internal window manager assigns X focus to the X window that receives Win32 focus. The client receives notification of this event and may decide to assign X focus to some other window, perhaps a child window.

The server must therefore keep track of both the X window that currently has focus and the state of Win32 focus. When the server loses Win32 focus, the X focus is assigned to the root window. When the server receives Win32 focus, X focus is assigned to the X window that previously had it. Whenever X focus is changed by an application or by the window manager, the current X focus state is cached so that it can be restored later, if necessary.

Font Management

Fonts and text functionality make up a significant portion of any graphics architecture. Both the X and the Win32 systems define a rich set of text-rendering operations and can process several font formats.

X and Win32 Fonts The X font management library is a modular architecture that defines an API for reading and writing individual font formats. The module that implements the API for a given font format is called a renderer. This approach allows X to support several font formats: the library's renderer modules convert external formats to a single, internal bitmap format, which is used for all drawing operations. The term *X font* refers to font data in this internal format.

The font management library supports both bitmap and scalable outline fonts. Bitmap font glyphs are simply reformatted and used. Scalable formats, such as Adobe Type1, are rasterized on demand into the X font format.

For maximum performance, the server draws text with native Win32 fonts using the Win32 API. Win32 fonts are bitmap fonts in the FON format. Win32 functionality covers the great majority of text-drawing operations, but there are a few cases in which it is either not possible or not efficient to use Win32 fonts.

The server can also draw directly with the X fonts to provide full X font support and complete text-drawing functionality. This method uses Win32 BitBlt( ) operations to copy the character glyphs to the display as bitmaps. Drawing speed with this method is acceptable but not maximum.

Therefore, both X and Win32 fonts are used. The Win32 fonts may be thought of as optional accelerators: the server uses them whenever possible and falls back to the X fonts when necessary. The decision to fall back can be made on a variety of conditions. This technique has also proved useful in working around problems such as text-drawing bugs in individual video drivers.

Since scalable font outlines are rasterized into bitmaps at run time, they are generally drawn directly with the internal X font format. The extra work of compiling a companion Win32 font at run time generally outweighs its value as an accelerator.

X bitmap fonts are most commonly distributed in the Bitmap Distribution Format (BDF), an ASCII text source file. The eXcursion team wrote a font compiler tool that generates native Win32 (FON format) fonts from the BDF sources. The fonts created can be used by any Win32 application.

The compiler can generate either the commonly used version 2 format or the extended version 3 format, which is necessary for large fonts that require more than 64 kilobytes (KB) of glyph storage. Figure 3 illustrates the process of generating equivalent X and Win32 fonts from a common source.

The X font format contains extra information (e.g., metrics and properties) that cannot be derived from

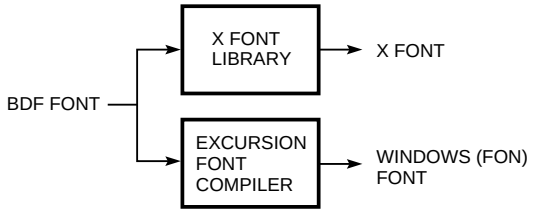


Figure 3
Font Conversion

the Win32 font. Therefore, the X and Win32 fonts are used together; the X information comes from the X font and the Win32 font is used by the Win32 API.

Realizing Win32 and X Fonts When the X server first opens a font, it invokes the function RealizeFont(). This function gives the server an opportunity to initialize data structures and perform any format-specific operations necessary to make the font available.

To make a Win32 font available for drawing, the server retrieves the filename of the font from the server's look-up table and registers it with the Win32 API using the function AddFontResource(). A handle to the font is obtained from CreateFontIndirect(), and thereafter the handle is selected into the desired DC for drawing operations. If the Win32 realization fails for any reason, the code simply realizes the X font instead. Failing to realize a Win32 font does not necessarily imply an error condition. Such failure happens in any case in which the server decides that it is best to use the X font directly.

The internal X font format is a set of data structures. The glyphs are stored in conventional arrays in user memory. To improve performance, the server realizes an X font by writing all glyphs to a Win32 bitmap in off-screen memory. CreateBitmap() returns a handle for later reference, and the glyphs in the bitmap are indexed for use in drawing operations.

Drawing with Win32 and X Fonts The glyphs in X text strings are often kerned, that is, overlapped for best typographic appearance. To draw with Win32 fonts, the server emulates the way X draws text by using ExtTextOut(), which uses an intercharacter spacing vector to place the individual glyphs. The font's X metrics are used directly to calculate this vector.

Glyphs from X fonts are drawn by performing BitBlts from the Win32 bitmap to the target window or bitmap. The server places the glyphs using the font's X metrics as described in the previous paragraph.

Color Resource Management

Although some X Window System concepts and structures map fairly closely to those in the Win32 system, color resource management is handled very differently. The difference is most evident when dealing with pseudocolor video systems. Consequently, this paper describes only this case.

The X Window System environment shares 256 colormap cells among all applications that use the default colormap (i.e., those that do not have a private colormap). Applications can allocate cells in the default colormap to protect them from modification by other applications. In contrast, the Win32 system allows each application complete access to the system palette while the application has focus and maps the palettes of the windows without focus as best it can.

In the X Window System environment, when an application reserves a colormap cell, it references the cell with a pixel value. This value is an index into the colormap and is used to look up the value that will actually be stored in screen memory when that pixel value is used in a drawing operation.

In the Win32 system, color management is handled by the palette manager through a palette structure. Each application has a logical palette, and a single system palette contains the colors currently mapped to the hardware colormap. Applications reference colors relative to their logical palette, and the palette manager handles the mapping between the logical palette and the system palette. When an application is given focus, the palette manager maps all the colors from the logical palette into the system palette. If the system palette does not have enough empty cells, the palette manager frees cells allocated to other applications. If this occurs, the palette manager will attempt to remap the other applications' colors into any remaining free cells in the system colormap. If not enough cells are free, any remaining unmapped colors are mapped to the system palette colors that most closely match.

Because of this way of handling color resource management, an application does not know what value is being stored in screen memory for any particular color and the value stored for any color can change over the lifetime of the application. This situation presents significant difficulties for X operations that require exact knowledge of the pixel values in screen memory, such as the GetImage operation and operations involving plane masks. The server works around the difficulties by creating two Win32 logical palettes.

The first palette, i.e., the working palette, corresponds exactly to the X default colormap and does not allow sharing of the palette by Win32 applications. Whenever an X window has focus, the working palette is in use. This causes the Win32 palette manager to set up the system palette such that it directly corresponds to the X colormap, and operations that are pixel based work properly.

The other palette, i.e., the identity palette, is set up to correspond exactly to the system palette. The identity palette is used whenever no X window has focus. Because of the correspondence, no translation is involved between the identity palette and the system palette, which allows the X server to know what pixel value is stored in screen memory.

The X Window System environment allows for private colormaps, which are created and used by a single application. The server creates a working palette for every colormap created. When the colormap is installed (normally by the window manager when the X application is given focus), the eXcursion software installs the working palette associated with the private colormap.

The eXcursion X server currently supports the PseudoColor visual class and the StaticGray depth 1

visual class, which is mainly used for bitmaps. eXcursion version 1 also supported a StaticColor visual class for 16-color video graphics array (VGA) displays. eXcursion version 2 treats VGA devices identically to PseudoColor devices and allows the Windows palette manager to generate dithering patterns for the unavailable colors.

Network Interface

With the release of X11R6, the X Consortium combined all transport-specific code into a single place in the source tree, the X transport interface. The eXcursion team extended the X transport interface to include Network Computing Device's (NCD's) Xremote serial line transport. Combined with the transmission control protocol/internet protocol (TCP/IP) and DECnet transports, the eXcursion product can now execute X sessions over any of these transports simultaneously. The eXcursion product supports any TCP/IP stack that complies with the Winsock version 1.1 implementation, PATHWORKS DECnet protocol, and NCD's Xremote protocol for serial line.

The X transport interface provides functions that are common to all transports, such as parsing an address into a host and port number. The interface does not provide an abstraction for the select() call, because it assumes that this call is transport independent. Unfortunately, the Xremote protocol requires an independent select() mechanism, and, thus, it was necessary to implement a select() abstraction to combine the transport-independent select() with the Xremote select(). Although somewhat compromised by this addition, performance was a problem only when the Xremote protocol was used in combination with either the TCP/IP or the DECnet protocol.

X Image Extension

eXcursion version 2 provides versions 3 and 5 of the X Image Extension to support a wide range of imaging applications. Because it is a large body of code, XIE is implemented as a pair of Win32 DLLs to conserve memory on systems that will not be running applications that use XIE.

Normally, access to a DLL is one-way. Applications can load and make function calls into a DLL, but because it is linked dynamically at run time, the DLL code cannot make function calls back into the calling application. XIE, however, must call into the device-dependent layer of the server to perform any required drawing after processing its imaging requests. To permit this, an addition to the interface was designed. When the XIE DLL is initialized, the caller supplies a list of pointers to the functions needed by the XIE. The DLL fills an array with these pointers and then calls back indirectly through the array. On the Windows operating system, this design could create a problem because under Win32 APIs, global data in a DLL is not instanced; that is, the code is not reentrant. The approach works in this case because there is only one copy of the DLL loaded. If another application was sharing the DLL, the pointers would be overwritten by the second initialization.

Control Panel

The eXcursion control panel is the primary interface through which the user configures and controls the product. Some other components create simple windows or icons, but these functions are limited. The control panel constitutes 90 percent of the user interface for the eXcursion application. This fact makes the control panel an ideal candidate for the rapid application development features of the Microsoft Visual C++ environment. The control panel is a Win32 application coded almost entirely in C++ and linked with the Microsoft Foundation Class library.

The main purpose of the control panel is to present a manageable interface through which the user can view and modify the eXcursion configuration profile. To do this in a manner consistent with the new Windows 95 shell, the Property Sheet MFC object was chosen. Property Sheets are tabbed dialog boxes that have the advantage of organizing large amounts of data settings in a compact space. They are used extensively by the Windows 95 operating system and by the most recent versions of Microsoft applications.

The Property Sheet object is a subclass of the Windows object and is essentially a container for the tabbed pages. Each tab, when clicked by the user, displays a dialog box that is subclassed from the MFC Property Page object. The individual pages can be visually configured and revised using the class wizard feature of Microsoft Visual C++. The designer simply selects dialog box controls such as buttons, drop lists, or edit fields and positions them on the dialog box. The code to handle user actions is then filled in.

The eXcursion control panel is shown in Figure 4. We constructed an initial prototype of the control panel application with about 60 percent of the final functionality in less than one month.

Interprocess Communication Library

eXcursion version 2 consists of several cooperating processes that must communicate and synchronize with one another. When a remote X application is started by the server or the control panel, the application launcher signals when the operation is complete.

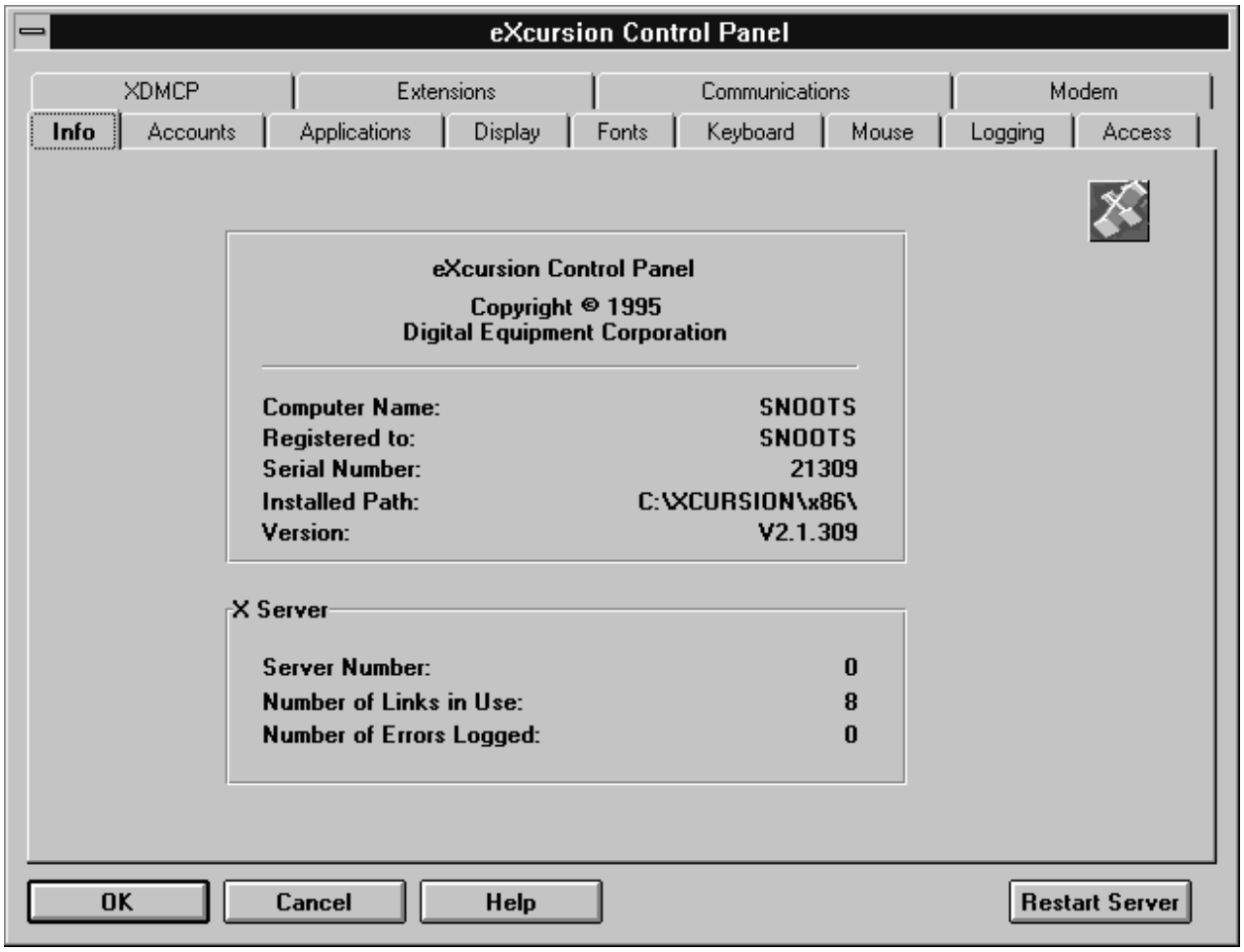


Figure 4
The eXcursion Control Panel

Error and status information is sent to the error logger by the other components. When the user changes a configuration setting through the control panel, the change must be communicated to the X server, if it is running. In some cases, the change can take effect immediately; in other cases, the server cannot implement the change without restarting. The control panel and the server must engage in a dialog so that the user can be informed as to what action must be taken, if any. The IPC library is an operating system–independent API that permits eXcursion components to determine which other components are present and to exchange commands and configuration information.

The Windows NT operating system provides several built-in IPC mechanisms, but most are not available on the Windows or Windows 95 systems. The only mechanism that is universal to the three operating systems is the message-passing interface in the Win32 API. This mechanism, while not the most efficient, is relatively straightforward to implement. Since the performance demands on the IPC library were determined to be very light, this mechanism was chosen.

The disadvantage of the Win32 message-passing interface is that it is window based, not process based. Messages are received by a callback procedure that must be associated with a window before any communication can take place. If an application has not yet created a window, or never creates a window, as is the case with the application launcher, no communication is possible. To remedy this, the IPC library creates its own window when the calling process initializes. The IPC window is never mapped to the screen, so it is not visible to the user. All interprocess communication passes through the IPC window.

The IPC library consists of a collection of unique messages and an API. The messages are registered with the Win32 function RegisterWindowMessage. This ensures that the messages used by the eXcursion application do not conflict with system messages or messages used by other applications. The eXcursion IPC messages are

- ipcComponentStartedMsg, which the IPC posts to all components when a component initializes.

- ipcRestartServerMsg, which the IPC sends to the server to tell it to restart.
- ipcRestartServerStatusMsg, which the IPC posts with the status of the restart request.
- ipcInquireMsg, which the IPC sends to retrieve a data item from a component.
- ipcProfileChangedMsg, which the control panel sends when the registry profile changes.
- ipcLaunchOneCompleteMsg, which the application launcher sends to notify the server of launch completion.
- ipcLaunchAllCompleteMsg, which the application launcher sends to notify the server of launch completion.
- ipcHideAllWindowsMsg, which the server sends to all components to tell them to hide all their windows. The eXcursion application uses this message to execute the pause/resume feature.
- ipcShowAllWindowsMsg, which the server sends to all components to tell them to show all their windows. The eXcursion application uses this message to execute the pause/resume feature.

In addition to sending and receiving messages, eXcursion processes can use the IPC library to determine which other components are running. The IPC initialization procedure creates a window with a unique name that identifies the calling component. To determine whether a specific component is present in the system, the IPC searches all windows on the system until it finds one with the correct name.

Error Logger

The error logger is a Win32 application that receives error and informational messages from other components and either displays them in a window or logs them to a file. On the Windows NT operating system, information that may help system managers or users diagnose problems may additionally be recorded in the Windows NT event log.

Application Launcher

The application launcher is a Win32 application that handles requests from the control panel or server to start X client applications. The client may reside on a remote host or on the same machine.

When the user requests the server or control panel to start an X client application, it starts the eXcursion application launcher in a separate process. The application command, host name, account information, network transport, and command shell are passed to the launcher in its command line arguments. The launcher makes the connection to the remote system, initiates the command using the selected protocol (rexec, rsh, DECnet object, or local command), and sends an IPC message to the server indicating that a new application is starting.

Registry Interface

The Windows NT operating system introduced a new concept called the registry. This is a protected database maintained by the operating system, wherein Win32 applications may store configuration and state information. The registry has a well-defined API and a maintenance utility program that is shipped with the Windows NT operating system. Under the Windows operating system, configuration information is kept in simple text files, which are vulnerable to accidental or malicious tampering. At the time the design of eXcursion version 2 was under way, it was unknown which, if either, of these two methods would be available under the Windows 95 operating system. Nevertheless, all three of these operating systems had to be supported.

We designed an API for accessing the configuration information in a manner independent of the operating system. Knowledge of the operating system and its registry access method is encapsulated in the library. Since several independent processes must access the information, the library is built as a DLL to conserve memory. The interface basically resembles that of the Windows NT registry API but eliminates some of the complexity.

If the eXcursion software has not been configured when the registry interface first accesses the profile, default values for all settings are selected to allow the software to function normally.

Summary

With computer systems based on the Microsoft Windows operating system increasing in power and decreasing in price, Windows-based systems are appearing on desktops that once held workstations running the UNIX or OpenVMS operating systems. Windows systems must be able to access applications on remote file and compute servers running in the X Window System environment. Version 2 of the eXcursion product provides desktop integration of X client applications with native Win32 applications. Modular coding techniques, object-oriented programming, and selective use of the Microsoft Foundation Class library helped reduce development time, and improve performance, maintainability, and reliability.

General References

D. Giokas and A. Leskowitz, "eXcursion for Windows: Integrating Two Windowing Systems," *Digital Technical Journal,* vol. 4, no. 1 (Winter 1992): 56–67.

X Window System

S. Angebranndt et al., *Definition of the Porting Layer for the X v11 Sample Server* (Cambridge, Mass.: X Consortium, Inc., 1994).

J. Fulton, *The X Font Service Protocol, Version 2.0, X Version 11, Release 6* (Cambridge, Mass.: X Consortium, Inc., 1994).

E. Israel and E. Fortune, *The X Window System Server, X Version 11, Release 5* (Woburn, Mass.: Digital Press, 1993).

O. Jones, *Introduction to the X Window System* (Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1989).

K. Packard and D. Lemke, *The X Font Library* (Cambridge, Mass.: X Consortium, Inc., 1995).

D. Rosenthal, *Inter-Client Communication Conventions Manual, Version 2.0* (Cambridge, Mass.: X Consortium, Inc., 1994).

R. Scheifler, *X Window System Protocol, X Version 11, Release 6* (Cambridge, Mass.: X Consortium, Inc., 1994).

R. Scheifler and J. Gettys, *X Window System* (Bedford, Mass.: Digital Press, 1992).

Networks

M. Hall et al., "Windows Sockets: An Open Interface for Network Programming under Microsoft Windows, Version 1.1" (1993).

K. Packard, *X Display Manager Control Panel, Version 1.0, X Version 11, Release 5* (Cambridge, Mass.: MIT X Consortium, 1989).

W. Stevens, *UNIX Network Programming* (Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1990).

X *Transport Interface* (Dayton, Ohio: NCR Corporation, 1993).

Windows Operating Systems

R. Blake, *Optimizing Windows NT, Windows NT Resource Kit,* vol. 3 (Redmond, Wash.: Microsoft Press, 1993).

H. Custer, *Inside Windows NT* (Redmond, Wash.: Microsoft Press, 1993).

A. King, *Inside Windows 95* (Redmond, Wash.: Microsoft Press, 1994).

Win32 Programmer's Reference, vols. 1–5 (Redmond, Wash.: Microsoft Press, 1995).

Windows Programming

K. Christian, *The Microsoft Guide to C++ Programming* (Redmond, Wash.: Microsoft Press, 1992).

P. DiLascia, *Windows++: Writing Reusable Windows Code in C++* (Reading, Mass.: Addison-Wesley Publishing Company, 1992).

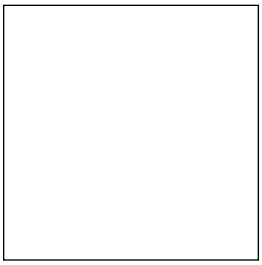
The GUI Guide, International Terminology for the Windows Interface (Redmond, Wash.: Microsoft Press, 1993).

S. McConnell, *Code Complete: A Practical Handbook of Software Construction* (Redmond, Wash.: Microsoft Press, 1993).

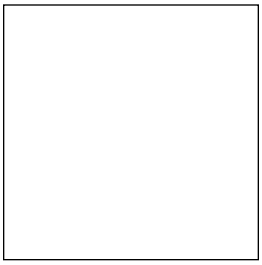
C. Petzold, *Programming Windows,* 2d ed. (Redmond, Wash.: Microsoft Press, 1990).

B. Stroustrup, *The C++ Programming Language* (Reading, Mass.: Addison-Wesley Publishing Company, 1986).

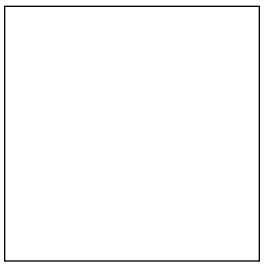
Biographies



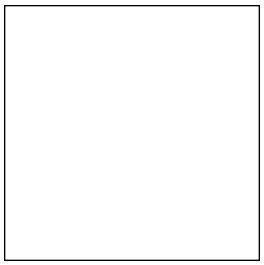
John T. Freitas
Presently a software engineer at Atria Software, John Freitas worked at Digital for 15 years. For the last few years, he was associated with Digital's eXcursion product as an individual contributor, an architect, and a designer. Previously, he was in the Workstation group. John received a B.S.E.E. from Northeastern University in 1967. While in college, he worked as a co-op student on the Apollo Project at MIT's Draper Laboratory. During the 1970s, he worked for Harvard University developing and maintaining medical computing facilities at Massachusetts General Hospital.



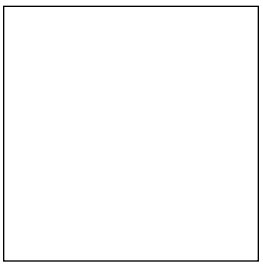
James G. Peterson
James Peterson is currently a software engineer at DeLorme Mapping. As a member of Digital's Windows NT group, James led the releases of the eXcursion software from version 1.1 through version 2.1. In addition, he worked as architect and individual contributor on the eXcursion project, concentrating on graphics and performance. Earlier, he worked in the PATHWORKS and Rainbow groups. James was employed by Compion Corporation before joining Digital in 1984. He received a B.A. (1979) in mathematics from Indiana University and an M.S. (1981) in mathematics and an M.S. (1984) in computer science, both from the University of Illinois.



Scot A. Aurenz
Scot Aurenz is a principal software engineer in the Windows NT group where he works on the development of the eXcursion PC X server. Scot has contributed to many projects at Digital, including the Language Sensitive Editor (DECset LSE) and the SUVAX workstation. Scot came to Digital in 1979 as a Purdue University co-op student and became a full-time employee after receiving his B.S.E.E. in 1982. He received an M.S.E.E. from the University of Illinois in 1986.



Charles P. Guldenschuh
Charles Guldenschuh is a principal software engineer in Digital's Windows NT group. He is responsible for color support and software installation of the eXcursion product. Previously, he worked in the Real-Time Software, Professional 300 Software Engineering, and RT-11 Engineering groups. Charles joined Digital after receiving his B.S. in information and computer science from the Georgia Institute of Technology in 1976.



Paul J. Ranauro
Paul Ranauro joined Digital in 1987 and is a principal software engineer in the Windows NT group. He is responsible for application failover for the Digital Clusters for Windows NT product. In earlier work, he participated in the development of the eXcursion software and the ACMSxp transaction processing monitor, specifically, in the implementation of the RTI protocol. He also participated in the implementation of the Manufacturing Messaging Service OSI application layer protocol for the DEComni product and a network performance analyzer. Prior to coming to Digital, he was a consultant at Index Systems and a senior software engineer at Micom-Interlan. Paul holds a B.A. in history from the University of Massachusetts at Boston.