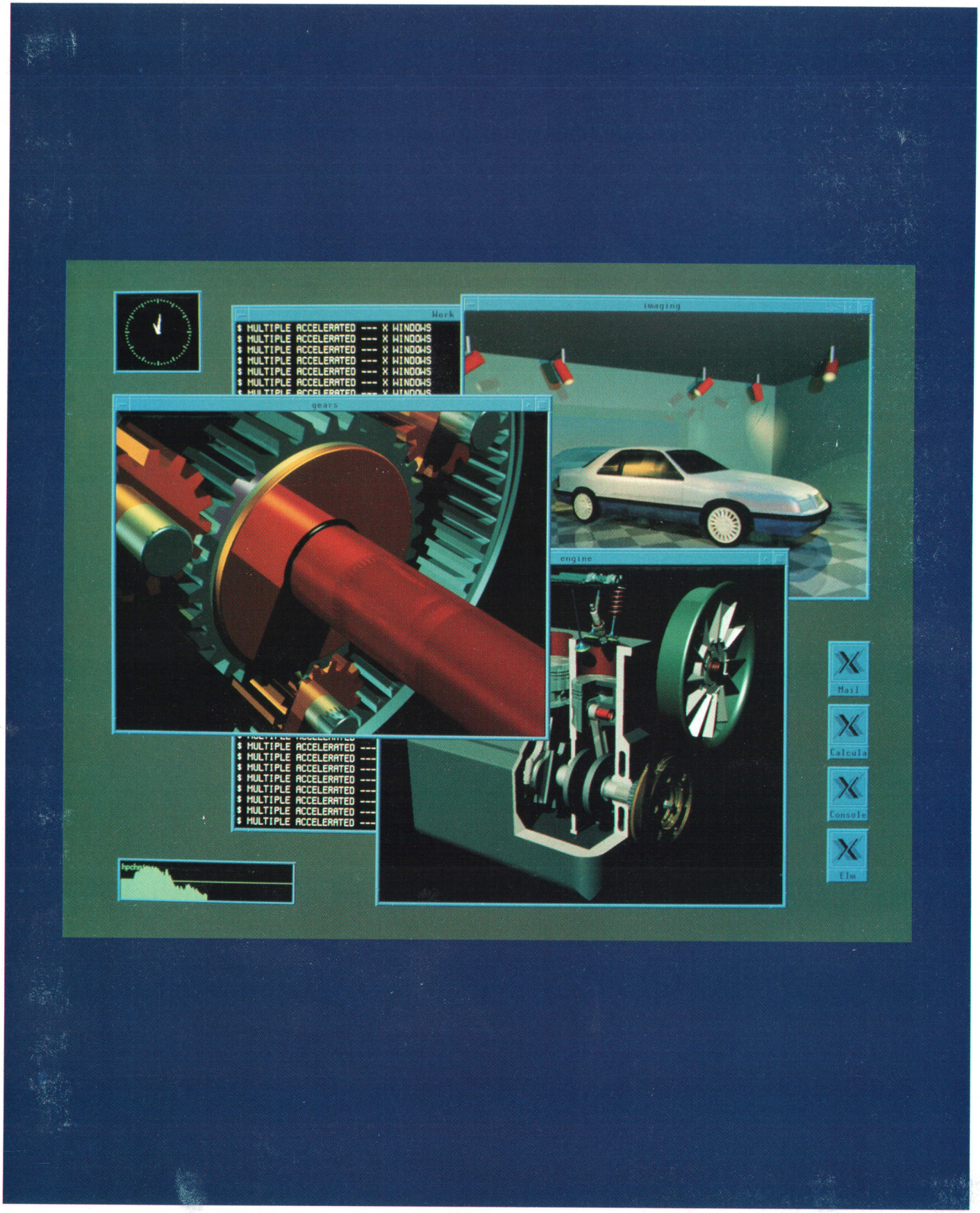
HEWLETT-PACKARD
JOURNAL

DECEMBER 1989



HEWLETT
PACKARD

HEWLETT-PACKARD JOURNAL

December 1989 Volume 40 • Number 6

Articles

Departments

The **Hewlett-Packard Journal** is published bimonthly by the Hewlett-Packard Company to recognize technical contributions made by Hewlett-Packard (HP) personnel. While the information found in this publication is believed to be accurate, the Hewlett-Packard Company makes no warranties, express or implied, as to the accuracy or reliability of such information. The Hewlett-Packard Company disclaims all warranties of merchantability and fitness for a particular purpose and all obligations and liabilities for damages, including but not limited to indirect, special, or consequential damages, attorney's and expert's fees, and court costs, arising out of or in connection with this publication.

Subscriptions: The Hewlett-Packard Journal is distributed free of charge to HP research, design, and manufacturing engineering personnel, as well as to qualified non-HP individuals, libraries, and educational institutions. Please address subscription or change of address requests on printed letterhead (or include a business card) to the HP address on the back cover that is closest to you. When submitting a change of address, please include your zip or postal code and a copy of your old label.

Submissions: Although articles in the Hewlett-Packard Journal are primarily authored by HP employees, articles from non-HP authors dealing with HP-related research or solutions to technical problems made possible by using HP equipment are also considered for publication. Please contact the Editor before submitting such articles. Also, the Hewlett-Packard Journal encourages technical discussions of the topics presented in recent articles and may publish letters expected to be of interest to readers. Letters should be brief, and are subject to editing by HP.

Please address inquiries, submissions, and requests to: Editor, Hewlett-Packard Journal, 3200 Hillview Avenue, Palo Alto, CA 94304, U.S.A.

In this Issue

The Massachusetts Institute of Technology's X Window System, Version 11, has become an industry standard window system for supporting user interfaces in networks of workstations running under AT&T's UNIX operating system. In Hewlett-Packard terms, this means HP 9000 Series 300 and 800 workstations running under the HP-UX operating system. The X Window System lets an application program running on one workstation display information to a user sitting at any workstation in the network. HP 9000 Series 300/800 workstations also offer a high-performance 2D and 3D graphics library called Starbase. Naturally, users wanted their application programs to be able to use the Starbase graphics library and run under the X Window System. Unfortunately, they couldn't do both at once. The two systems had been designed independently, and both assumed exclusive ownership of the display and input devices. Furthermore, while many X applications could be active in the network simultaneously, only one Starbase application could run on a workstation. As a result of these differences, the two systems couldn't coexist. Working out a solution to this problem required a joint effort of engineers at two HP Divisions, dubbed the Starbase/X11 Merge project. Merging the two systems was a nontrivial technical challenge. It had to be done without sacrificing the performance of Starbase applications or requiring that they be rewritten. As related in the article on page 6, it required changes to the architecture of both systems, development of cooperating display drivers for the two systems, restructuring the interface between the drivers and the X server process, and development of a facility to handle communication between the two systems. In other articles, you'll find details of the changes as they relate to the management of graphics resources (page 12), access to display hardware (page 20), use of display memory (page 33), sharing of input devices (page 38), and modification of existing test suites (page 42).

The capabilities of the Starbase graphics library include high-performance color rendering and 3D solids modeling. For determining the intensity of light reflected to the observer's eye from any object, the library offers three illumination models—one local and two global. A local model considers only the orientation of an object and light from light sources. A global model also considers light reflected from or transmitted through other objects in the scene. The two Starbase global illumination models are based on methods called ray tracing and radiosity. In the paper on page 78, David Burgoon presents the mathematical foundations of the radiosity method and compares its capabilities and limitations with those of the ray tracing method.

The Starbase graphics library runs on HP 9000 Computers equipped with the SRX or TurboSRX graphics subsystems. The TurboSRX is an enhanced-performance version of the SRX design. On page 74, Larry Thayer explains how analysis of the data-flow pipeline of the SRX revealed where custom VLSI chips could be used to improve the performance. He then describes three chips that were designed to take advantage of these opportunities for the TurboSRX version.

For HP's commercial computer systems based on the HP 3000 Computer, the last resort in troubleshooting usually involves analyzing a dump of the computer's memory. While powerful tools have evolved for on-line dump analysis, until recently no parallel progress had occurred that would allow efficient on-line examination of operating system source code. After finding clues in the memory dump, HP support engineers had to rely on a complex manual process to locate specific source code in a printed listing. Fortunately, this isn't true anymore. HP support facilities now have HP Source Reader, a system for accessing source code stored on compact disk read-only memory, or CD-ROM. The source code is stored on the CD-ROM in a proprietary format and is retrieved by an access program that runs on an HP Vectra Personal Computer and allows relevant information to be popped onto the screen in seconds. On page 50, three of the system's designers—support engineers themselves—describe HP Source Reader and present an example of its use.

As integrated circuit clock rates and signal transitions have become faster and faster, it has become necessary to treat even very short wires and printed circuit board traces as transmission lines. This means that impedance matching, reflections, and propagation delays are important considerations. In automatic testers for such high-speed devices, transmission line techniques must be applied to the tester-to-device interconnection if the device is to be tested at operating speeds and accurate results are required. The paper on page 58 describes how this interconnection is implemented in the HP 82000 IC Evaluation System to ensure high-precision measurements even for difficult-to-test CMOS devices. A resistive divider arrangement makes it possible to test low-output-current devices up to their maximum operating frequencies.

R.P. Dolan
Editor

Cover

This HP 9000 Series 300 display shows the results obtainable using a Starbase/X11 Merge system display mode called combined mode. This mode takes advantage of the sophisticated rendering capabilities of the TurboSRX 3D graphics accelerator, causing the two sets of display planes—image and overlay—to be treated as one screen. The complex 3D images were rendered in the image plane and the listing, the clock, the buttons, and the plot were rendered in the overlay plane.

What's Ahead

The HP OSI Express card provides on one HP 9000 Series 800 I/O card the capabilities of the network architecture defined by the ISO Open Systems Interconnection (OSI) Reference Model. In the February issue, ten articles will provide insight into the OSI Express card implementation of the model and will define what sets this implementation apart from other networking implementations. Also featured will be the HP 71400A Lightwave Signal Analyzer, which measures the characteristics of high-capacity lightwave systems and their components, including single-frequency or distributed feedback semiconductor lasers and broadband pin photodetectors. An accessory, the HP 11980A Fiber Optic Interferometer, helps characterize the spectral properties of single-frequency lasers.

System Design for Compatibility of a High-Performance Graphics Library and the X Window System

The Starbase/X11 Merge system provides an architecture that enables Starbase applications and X Window System applications to coexist in the same window environment.

by Kenneth H. Bronstein, David J. Sweetser, and William R. Yoder

HP'S HIGH-PERFORMANCE 2D and 3D GRAPHICS library called Starbase has proven very successful in engineering workstation applications. Similarly, The X Window System™ Version 11, or X11, has become the de facto industry standard window system for supporting user interfaces on workstations connected across a network.[1,2] Both of these systems run in the HP-UX environment on the HP 9000 Series 300 and 800 Computer systems (see boxes on pages 7 and 8).

Before the Starbase/X11 Merge project, the X Window System and Starbase graphics applications were not able to run on the same display. An application could use either the Starbase high-performance graphics or it could run in the X Window System, but not both simultaneously. These systems each make simple assumptions about ownership of the display and input devices, and this makes them unable to coexist. Since HP is one of the industry leaders in the X Window System technology and Starbase is a widely used graphics library, the Starbase/X11 Merge project was started to design and implement a scheme whereby X and Starbase applications could coexist on the same display.

There were three major challenges associated with merging Starbase and X11. The first challenge was to change the architecture of the Starbase graphics libraries and the X Window System so that a Starbase application could run within an X window with full functionality and with performance comparable to Starbase running on a dedicated (nonwindowed) display. The second important challenge was to enable existing Starbase applications to relink simply with the new Starbase drivers and run in an X Window System with no modifications to the application's source code. The final major challenge was to coordinate the design and development of this product over geographical and organizational boundaries. The Starbase/X11 Merge project was the joint effort of software engineers located at HP's Graphics Technology Division (GTD) in Ft. Collins, Colorado, and HP's Corvallis Information Systems Operation (CIS) located in Corvallis, Oregon. The team in Colorado was responsible for the Starbase portion of the project and the team in Corvallis was responsible for the X Window System portion of the project.

This article and the next five articles in this issue describe the design and implementation techniques used to handle

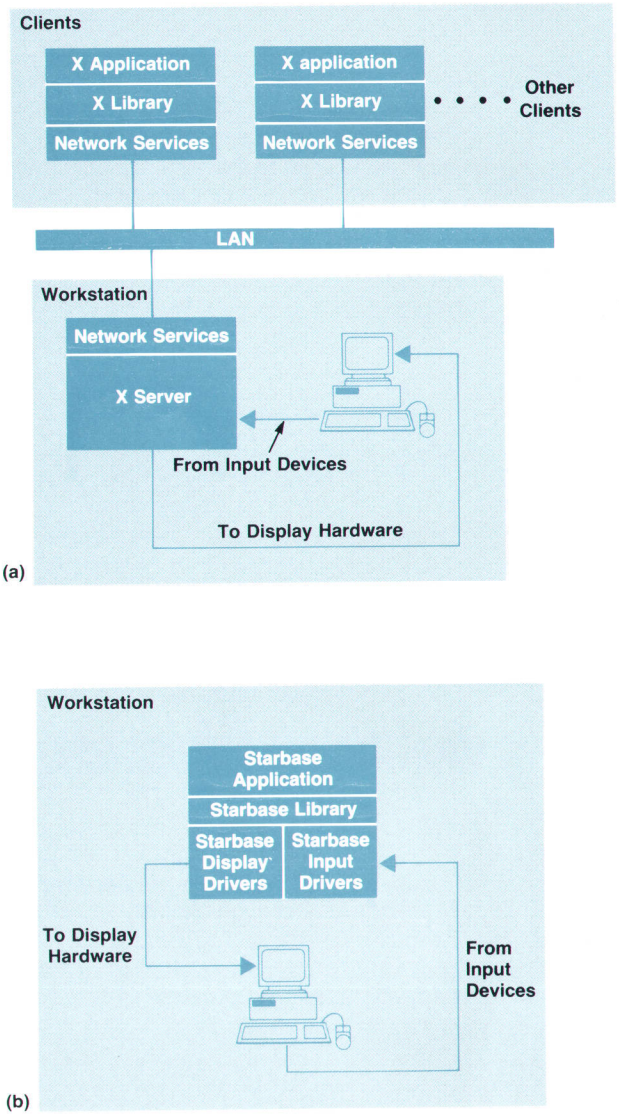


Fig. 1. *Incompatible architectures. (a) The architecture for an X application. (b) The architecture for a Starbase application. Both architectures assume complete ownership of the display.*

these challenges.

Design Alternatives

The architectures for a client (application) running in the X environment and an application using Starbase are shown in Fig. 1. The X Window System is network transparent, which means that an application running on one workstation can display itself to a user sitting at the same workstation or at another system across a network. Applications, or clients, running in the X Window System are allowed access to the display only through the X server, which is a separate process that arbitrates resource conflicts and provides display, keyboard, and mouse services to all applications accessing the display. Also, as shown in Fig. 1, many X applications can be served by the X server simultaneously. Starbase, on the other hand, is a collection of libraries and drivers for 2D and 3D graphics applications, and only one Starbase application can run on the workstation at a time.

In trying to merge Starbase and X,* we did not lack alternative solutions. During the investigation stage there was little doubt that we could change the architectures of Starbase and X to coexist, but how to merge the two was not clear. The design alternatives included:

- Following the existing HP Windows/9000 model of adding window management utilities to the Starbase libraries.
- Implementing the X server on top of the Starbase graphics libraries.
- Implementing the X server using an internal low-level Starbase interface.
- Implementing an X driver for Starbase, using X Window System Xlib calls.
- Writing an X extension that implements Starbase low-level semantics.
- Developing Starbase and X drivers that cooperate in accessing the display hardware.

The project team selected the last alternative. This approach resulted in creating low-level drivers to support the rendering requirements of both Starbase applications and the X server, the restructuring of the server interface between the low-level drivers and the device-independent portion of the X server, and the development of a facility to handle communication between X and Starbase.

Low-Level Driver Redesign

The Graphics Technology Division manufactures a variety of display types with the following characteristics:

- On-screen resolutions that range from 512 by 400 pixels to 1280 by 1024 pixels.
- Display planes that range from 1 (capable of displaying black and white) to 24 (capable of displaying any of 16 million colors, with every available pixel a different color).
- Advanced hardware features, such as 2D and 3D graphics accelerators. Graphics accelerators provide graphics operations such as polygon clipping, rotation, and other transformations implemented in high-speed hardware.

To put the responsibility where the expertise lay and to accommodate all these display types, the engineers at GTD implemented the new display drivers, and the engineers at CIS implemented the code to translate X server semantics into display driver formats. The interface between the display drivers and X was called the X driver interface, or XDI. XDI is discussed later in this article.

During the design investigation phases, we discovered that many requirements of the Starbase environment and the X server environment were similar and the basic algorithms that use the hardware were the same. This led to the concept of shared drivers between the X server and Starbase applications. Originally we hoped that the drivers could be shared at the object code level, that is, the drivers

The Starbase Graphics Package

Starbase is a library of utilities for drawing computer graphics. It was first released in 1985, based on a draft of the ANSI and ISO standard Computer Graphics Interface, or CGI. Since its first release, features have been added to Starbase that go beyond the CGI standard. The library includes functions that draw lines, polygons, text, splines, circles, and arcs. It includes routines that read locations or button and key presses from input devices, and routines that echo the position of an input device on an arbitrary display.

An important goal of the Starbase product is to provide a library of functions that can be used on a range of devices. Starbase conceals the details of device dependencies, allowing each program to be used with a growing list of devices without making changes to the program. The current Starbase products support over 20 different devices. They include workstation displays, plotters, terminals, mice, and data tablets. New devices can be used as they become available by linking a program with new device drivers. This device independence is also used to assist the development of other graphics libraries. Implementations of libraries for the ANSI standards Core Graphics System (CORE), Graphics Kernel System (GKS), and Programmers Hierarchical Interactive Graphics System (PHIGS) use the Starbase device drivers to support the same range of devices.

The device independence of Starbase coexists with access to the full features and maximum performance of each device that it works with. Common features, such as line and polygon drawing, are supported directly on capable devices and emulated on simpler devices. The more sophisticated features of advanced displays, such as shaded images, are available to programmers that require these features, but not emulated on simpler devices.

Starbase has features tuned to the needs of particular groups of customers. Some additions optimize strictly two-dimensional graphics, such as for printed circuit layout, electrical design, and drafting. Functions have been added to Starbase to support integer coordinates and transformations that allow faster, more cost-effective display systems for these applications. Other additions emphasize three-dimensional images such as used for advanced mechanical design. Starbase supports perspective views of objects with shading simulating light sources, and draws only those parts of an image that are not hidden behind solid objects. The most recent additions to Starbase provide photorealism, the appearance of near reality, through ray tracing and radiosity technologies. See the article on page 78 for more information about radiosity.

*In this and other articles X11 and X Window System will also simply be referred to as X.

The X Window System

The X Window System, commonly referred to as X, is an industry standard, network transparent window system. X presents to the user a hierarchy of resizable overlapping windows providing device independent graphics. A graphical user interface is commonly included as an integral part of the X window system. The X Window System definition is maintained by the Massachusetts Institute of Technology X Consortium.

The first implementations of X were developed jointly at MIT by Project Athena and the Laboratory for Computer Science. Project Athena was faced with the problem of writing software for hundreds of displays from different vendors on machines all connected by a local area network. They designed X, based on the W window system, which was the work of Paul Asente, Brian Reid, and Chris Kent of Stanford University and Digital Equipment Corp.

The 1986 MIT release of X, Version 10.4, was the first version with multivendor support. HP was among the first computer manufacturers worldwide to sell X as a product when in March, 1987, the company began shipping the X Window System for HP-UX. In January 1988 the MIT X Consortium was formed, with HP being one of the founding members. X Consortium members include Apple Computer Inc., Ardent Computer, American Telephone and Telegraph Inc., Calcomp Inc., Control Data Corporation, Digital Equipment Corporation, Data General Corporation, Fujitsu Microelectronics Inc., Hewlett-Packard, International Business Machines Corporation, Eastman Kodak Corporation, NCR Corporation, Nippon Electric Corporation, Prime Computer Inc., Silicon Graphics, Sun Microsystems Inc., Tektronix Inc., Texas Instruments Inc., Unisys, Wang Laboratories Inc., Xerox Corporation, and others.

The X Window System designers, Robert Scheiffler of MIT and Jim Gettys of Digital Equipment Corporation, adopted a set of critical design objectives, specifying that the window system must:

- Work on a wide variety of hardware platforms and displays
- Facilitate implementation of device independent applications
- Be network transparent
- Allow for application concurrency
- Support differing application and management interfaces
- Provide overlapping windows and output to obscured regions of windows
- Support a hierarchy of resizable windows
- Provide support for text, 2D graphics, and imaging
- Be extensible.

Their implementation of this design has gone through a number of revisions. The implementation has stabilized at X version 11, which has been adopted as an industry standard. The current standards bodies that have adopted some portion of X or are in the process of adopting X include ANSI, IEEE, ISO (International Standards Organization), NIST (National Institute of Standards and Technology), OSF (Open Software Foundation), and X/OPEN. MIT has facilitated the acceptance of X as a standard by distributing the standards definition documents and the source code of sample implementations for public use for a nominal fee.

The X Window System consists of the X server, the standard X library, various library toolkits, and a set of X client applications.

- The X server controls access to display hardware and input devices.
- The X library is the basic programmatic interface providing a standard method to manipulate windows, control input, handle window system events, provide text output, manipulate color maps, render 2D device coordinate graphics, and extend the client/server protocol.
- The X toolkits provide standard sets of widgets, menus, and other user interface objects. The toolkits facilitate the development of applications that have a consistent, easy to use, graphical user interface.
- A window manager is provided as a special X application. The functionality of the window manager has been separated from the lower-level X server and X library. This modular design has allowed different window managers and different user interface models to be incorporated in any user's X environment.

The X server and the X library communicate via an asynchronous stream-based interprocess communication protocol. This protocol separates the application interface from the X server implementation. The X server can then be ported to new display devices without the need to modify the application programs. Executable application code compatibility is maintained across displays. This network protocol also provides the basis of network transparency and interoperability. Network transparency means that an application running on one computer can perform all display and input operations for a user sitting either at the same system or at another computer across the network. Network transparency is provided at no cost to the application as part of the standard X implementation. Interoperability implies that network transparency is preserved across various computer vendors' products.

could be compiled once and linked into both the X server and Starbase programs. The data structure environments and some of the rendering semantics of the two environments were too different to allow this, so the less restrictive alternative of shared source code with conditional compilation was chosen. This scheme enabled us to avoid changing existing Starbase library code and duplicating low-level display control and rendering operations for different display types.

Restructuring the X Server

A sample implementation of the X server exists in the public domain and is available from the Massachusetts Institute of Technology (MIT). This sample implementation has contributed greatly to the success of the X Window System. The X server maintained by MIT provides X vendors with a source code template from which X server products can be developed. Starting with MIT's sample X server, vendors can develop a version of the X server that works on their hardware. The sample X server consists of three major sections:

- Device Independent X (DIX). High-level device independent code for handling cursors, events, extensions, fonts, and rendering requests.

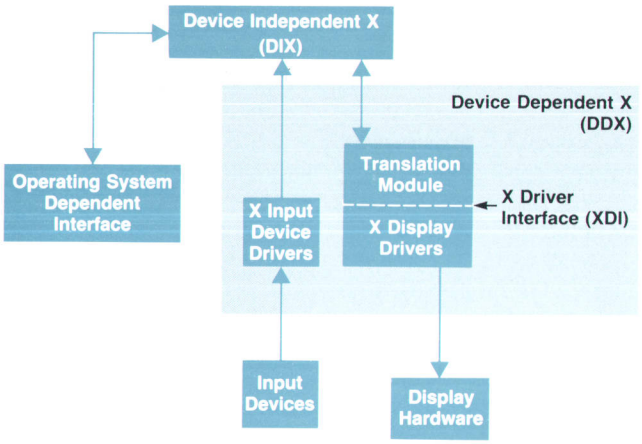


Fig. 2. *The modules in the X server. The device dependent module (DDX) shows the modifications made to accommodate the needs of the Starbase/X11 Merge system.*

- Operating System Dependent Interface. This section contains utilities used primarily by DIX to perform tasks specific to the host operating system. For example, DIX makes no assumptions about the structure of the host's file system or about how to open communication channels—these details are handled by the code in this section.
- Device Dependent X (DDX). DDX contains the code that performs device dependent I/O. For example, when a client asks the X server to draw a circle or to display text, DIX code interprets the request and passes it to the appropriate procedure in DDX for proper device dependent I/O. Conversely, when the user moves the mouse or types on the keyboard, DDX conveys this information to DIX for processing. DIX passes the information back to interested clients.

To handle our needs, the DDX layer was split into two more layers: a translation module and the X display drivers (see Fig. 2). The translation module, which was written by the engineers in Corvallis, translates the data formats and requests from DIX into a form suitable for the X display drivers. The X display drivers, which were written by the engineers at GTD in Colorado, do the rendering to a particular display. Between these two layers is the X driver interface (XDI). The X driver interface contains about four dozen driver entry points, the corresponding data structures, and a strict protocol for accessing the entry points.

This organization of DDX provided two benefits. First, it enabled us to carry on development at two separate locations and organizations, and second, it helped to eliminate redundant Starbase and X display driver code development. The functions provided by XDI include:

- Driver and device control
- Color map manipulations
- Accelerated graphics window support
- Cursor, raster, filling, vector, and text operations.

The translation module, which translates rendering requests from DIX into a format appropriate for the low-level X display drivers, can be very simple as in the following DDX-to-XDI routine which handles the DIX request to fill horizontal rows of pixels.

```
void
FillSpans(pDrawable, pGC, nInit, pptInit, pwidthInit, fSorted)
    DrawablePtr     pDrawable;      /* pointer to drawing surface */
    GCPtr           pGC;            /* pointer to the graphics context */
    int             nInit;          /* number of spans to fill */
    DDXPointPtr     pptInit;        /* pointer to list of start points */
    int             *pwidthInit;    /* pointer to list of n widths */
    int             fSorted;        /* ignored */
DECLARE_XDI_POINTERS                /*set up data pointers */
GET_XDI_INFO                        /* get information */
PREPARE_TO_RENDER                   /*set up display hardware */
```

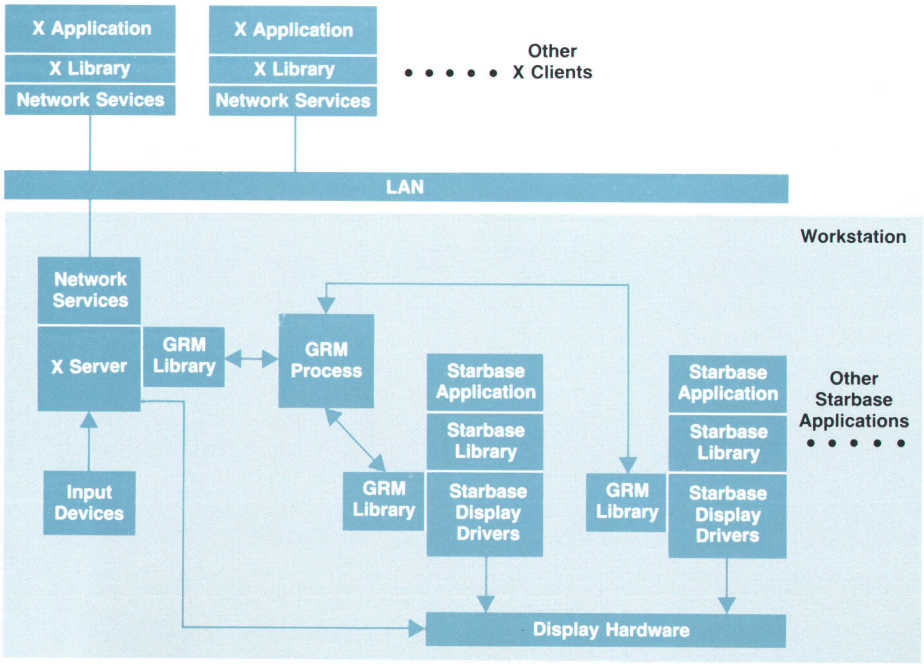


Fig. 3. *Starbase/X11 Merge software architecture.*

```
/* FillScanline is an XDI routine that accomplishes the
   fill request */
(*(pxdiGCJumpTable→FillScanline))(pxdiRender,
   (pxdiDrawable,pGc,
    nInit,                          /* number of spans to fill */
    (int16 *)(pptInit),             /* pointer to list of start points */
    (int32 *)(pwidthInit));         /* pointer to list of n widths */
FOLLOWUP_RENDERING                  /* restore state */
```

To allow processes to acquire specialized information from the X server and to make specialized requests to the X server, a small number of extensions were added to the X server so that Starbase applications could:

- Register Starbase windows with the server
- Retrieve the current list of rectangles that define windows visible on the screen
- Set up an error handler
- Note changes to the hardware color map.

Resource Sharing

To facilitate the exchange of information between Starbase and X, and to allow multiple processes to share off-screen memory and other display resources efficiently, the graphics resource manager (GRM) was developed. The GRM does not access the hardware directly because it is designed to function as a notepad on which Starbase and X can both write information regarding their use of display resources. The GRM also keeps track of shared resources so that both X and Starbase applications can coexist on the same display. See the article on page 12 for more information on the graphics resource manager.

Starbase/X11 Architecture

Fig. 3 depicts the basic software architecture for the Starbase/X11 Merge project. The figure implies that X and Starbase are both accessing the display at the same time. The design allows for any number of Starbase applications and any number of X clients to coexist on the same display.

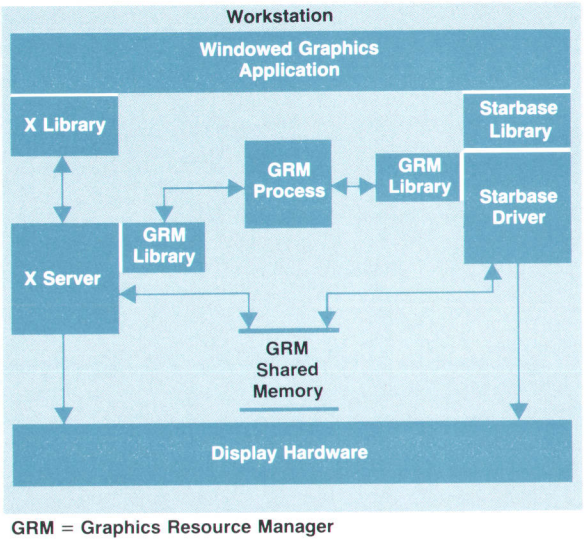


Fig. 4. *A "window smart" application that uses Starbase and X11 in the same application.*

The role of the GRM in this figure is to allocate resources among cooperating X server and Starbase processes.

Fig. 4 shows the architecture of a "window-smart" graphics application that makes programmatic use of both Starbase and X from within a single program. This facility allows Starbase programmers to use X rendering facilities to enhance the usability and appearance of their applications.

Conclusion

The Starbase/X11 merge project occurred in an era of increasing complexity in computer software. Software projects are getting larger and more geographically distributed. This complexity is also being faced during a time when a new tactical model has emerged in the computer industry. Diverse groups (sometimes involving a company's competitors) are forming alliances to achieve a greater goal than any entity could achieve alone. The Massachusetts Institute of Technology X Consortium is a successful example of this new model at work.

Acknowledgments

The successful completion of the Starbase/X11 Merge project was because of the dedicated effort of many people. The team at HP labs—Don Bennett, Dan Garfinkel, Steve Hoyle, and Bob Leichner—contributed heavily to the investigation. Jim Brokish and Steve Scheid at GTD also contributed during the investigation and helped with the early implementation of the graphics resource manager. Larry Rupp at GTD provided the Starbase/X11 Merge product with new graphics hard-copy capabilities for storing, retrieving, and printing images. Thanks are due Thaddeus Konar, Penny Telleria, Art Barstow, and Alesia Duncombe in the CIS quality and productivity department. Thanks also to Robert Casey at GTD and Mike Hatam at XTTC (UNIX® Test Technology Center) who provided invaluable assistance in system integration and testing. Special praise goes to Harry Phinney who, as lead engineer for HP's first release of X11, provided guidance in X server and driver issues.

References

1. F. Hall and J. Beyers, "X: A Window System Standard for Distributed Computing Environments," *Hewlett-Packard Journal*, Vol. 39, no. 5, October 1988, pp. 46-50.
2. R.W. Scheifler, *X Window System Protocol, X Version 11, Release 2*, Massachusetts Institute of Technology, September 1987.

Starbase/X11 Merge Glossary

Because some of the terminology used here and in the rest of the of the articles in this series may be new or specific to Starbase/X11 or they may be used before they are explained, the following terms are defined.

Backing Store. Locations in offscreen or virtual memory where the contents of a window are backed up if a window becomes obscured because of some window system or user action.

Bit Map. A pixmap having a depth of one. On monochrome displays the X server maintains all pixmaps as bit maps.

Clip List. A list of rectangles representing the obscured and/or unobscured areas of a window.

Clipstamp. An integer, associated with a window, that is used to determine the current validity of a list of clipping rectangles associated with that window.

Color Map. A set of hardware registers that maintain the red-green-blue components of individual pixels. Pixel values, which are commonly in the range of 0 to 255, serve as indexes into the color map.

Combined Mode. An X server operating mode on the TurboSRX display in which the overlay and image planes appear as a single, integrated set to the user.

Cursor. An indicator on the screen used to direct the user's attention. The X cursor (or input pointer) traverses the whole display, whereas Starbase cursors (commonly referred to as echoes) move within individual Starbase windows.

DDX. Device Dependent X. The portion of the X server devoted to handling device dependent I/O.

DHA. Direct Hardware Access. A method that allows a Starbase application to bypass the X server and render directly to the frame buffer.

Display Enable Register. A hardware register that controls which planes of the display are viewable. Starbase and X use the display enable register to implement double buffering.

DIX. Device Independent X. A section of the X server that contains a scheduler, a resource allocator, a high-level color map, and code for handling window functions, such as cursors, events, extensions, fonts, graphics context, and rendering.

Drawable. A logical raster (on the screen or in memory) upon which X and Starbase can draw. Windows and pixmaps are both types of drawables.

Double Buffering. A graphics technique to enhance the smoothness of motion. The technique works by using the display enable register to toggle between two buffers. While one buffer is being rendered into, the other is displayed. When rendering to the hidden buffer is complete, the display enable register is changed and the hidden buffer is displayed and the previously displayed buffer becomes the new hidden buffer.

Frame Buffer. The video memory of a display device in which each element represents one picture element, or pixel. The frame buffer is divided into two parts, on-screen memory (current image on the screen) and offscreen memory (graphics memory that is never visible).

Graphics Context. A self-consistent set of attributes such as foreground and background colors, line styles, and fill patterns which are used by X clients to specify how the X server should render the drawing requests it receives.

Gopen (Graphics Open). The Starbase action of opening a display device or window to create a virtual device that Starbase can render to.

GRM. Graphics Resource Manager. The GRM is a process that handles requests from the X server and Starbase applications for display resources such as offscreen memory and shared memory.

Image Planes. The primary display memory on HP's display systems, used for rendering complex images.

MOMA Windows. Multiple, obscurable, movable, and accelerated windows. Hardware logic in the graphics accelerator provides very fast drawing and clipping of multiple windows.

Naming Conventions. The following conventions apply to procedures mentioned in these articles:

- X<name> is a standard X library procedure (e.g., XGetWindowInfo).
- XHP<name> is an HP X-extension library procedure (e.g., XHPGetServerMode).
- xos<name> is a procedure inside the X server, located in the translation layer between DIX and the X display drivers.
- <name> without any prefix is typically an application-level procedure, but must be interpreted in context.

Offscreen Memory. A portion of the frame buffer that cannot be displayed on the monitor. In all other respects, offscreen memory behaves the same as on-screen (visible) memory. Starbase and X use offscreen memory to hold character, cursor, pixmap, and scratch information for rapid transfers to on-screen memory.

Optimized Font. A character set that has been placed into offscreen memory to increase its display output performance.

Overlay Planes. Planes of display memory that are visually on top of or in front of the image planes. These planes are disabled or set to a transparent color to view the image planes.

Pixel. The smallest addressable picture element of a display. Typical HP displays have between one and two megapixels.

Pixel Value. A numeric value, typically between 0 and 255, which determines the color of an individual pixel.

Pixmap. A hidden rectangle of raster data which is maintained in offscreen memory when there is room, and in virtual memory when there is no room in offscreen memory.

Raster Data. A data structure described by a two-dimensional array of pixel values.

Raw Mode. Running a Starbase application without any window system.

Rendering. Any form of drawing operation, including text, line, and raster output. Rendering may occur to on-screen memory, off-screen memory, or virtual memory.

Sample Server. The X11 server template source code made available to the general public by the X Consortium that enables X vendors to develop servers for their own products.

Scanline. A horizontal row of pixels.

Shared Memory. A contiguous area of process data space that is shared with another process. The X server and Starbase applications use shared memory for communication and sharing fonts, color maps, and other display resources.

Socket. A communications channel between two HP-UX processes. There are two types of sockets: internet sockets, which are communication channels between machines across a network, and HP-UX domain sockets, which provide faster communication within the same machine.

Stacked Screens Mode. X Server operation on overlay and image planes in which the two sets of planes are treated as separate display devices.

Stacking Order. An ordering imposed on a set of windows that represents the apparent visual ordering of the windows to the user. For a window to be at the top of the stacking order means that it cannot be occluded by any other window.

Tile. A pixmap replicated many times to form part of a larger pattern.

Transparent Color. A pixel value in the overlay planes that causes the information in the image planes to be displayed instead of the information in the overlay planes.

TurboSRX. A 3D graphics subsystem that includes a triple transform engine, a scan converter, a 16-bit z-buffer, four overlay planes, and up to 24 image planes. The TurboSRX also includes the microcode to provide interactive 3D solids rendering, photorealism, and window clipping capabilities.

Virtual Memory. Memory that the HP-UX operating system allocates to an executing process. It is called virtual because although the memory appears to be in physical memory to the process, the system may swap it to and from a disk. The X display drivers are capable of rendering graphics images to virtual memory as well as to on-screen memory.

Visual Type. The color map capabilities of a given display. Common visual types supported on HP displays include 1-bit static gray (or monochrome), 8-bit pseudo color (having 256 color map cells of RGB values), and 24-bit direct color (using 8 bits each for red, green, and blue values).

Window. An on-screen rectangle of raster data that can be mapped (displayed), unmapped (removed), and rendered to.

XDI. X driver interface. A set of entry points that exist in the device dependent section of the X server, which provide an interface between the server's translation module and the X display drivers.

X Client. A program that interacts with the X server through one of the X libraries using the X client/server protocol.

X Protocol. The specification from the MIT X Consortium that precisely defines the behavior of the X server in its treatment of clients, its handling of events and error conditions, and its rendering operations.

Managing and Sharing Display Objects in the Starbase/X11 Merge System

To allow Starbase and X to share graphics resources, a special process called the graphics resource manager was created to manage access to the shared resources. An object-oriented approach was taken to encapsulate these shared graphics resources.

by James R. Andreas, Robert C. Cline, and Courtney Loomis

ONE OF THE CHALLENGES for the Starbase/X11 Merge project was designing an architecture that supports sharing of resources among X and Starbase applications. These HP-UX processes can realize significant memory savings by sharing resources such as character sets or fonts. X and Starbase also compete for private use of display resources. The architecture we developed, called the graphics resource manager, or GRM, supports the allocation of shared resources and at the same time provides use of display resources by individual processes.

The GRM consists of an HP-UX process and a library. The GRM library is linked with the X server and Starbase applications and calls are made to the GRM library to communicate with the GRM process. Fig. 1 shows the GRM architecture discussed in this article. The GRM handles a request it receives from the library and returns a response to the library. The library unpacks the response and returns the information to the caller. The GRM supports three modes of operation:

- The X server operating alone
- A Starbase application operating alone without the support of any window system
- The X server with a Starbase application running in a window.

What Is Managed?

When we began investigating the GRM architecture, we assumed that we would be allocating two basic resources, shared memory and offscreen memory. Shared memory is a memory resource supported by HP-UX[1,2] which can be attached to the address space of multiple processes. Each process can access the shared memory space directly. By using shared memory in the GRM architecture, one process

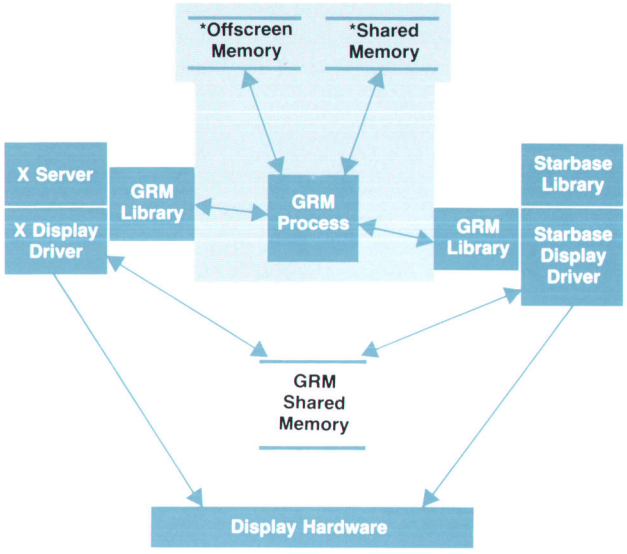


Fig. 1. *The architecture of the graphics resource manager.*

can load character font information into shared memory, and another process can later use the font.

Offscreen memory is a region of the display frame buffer that is not visible on the display screen. The frame buffer is the video memory of a display device dedicated to maintaining the value of the pixels. The X server and Starbase drivers use offscreen memory to optimize a variety of rendering operations. Many of HP's graphics hardware products provide offscreen memory in various shapes and sizes. Fig. 2 shows an example of the frame buffer memory available in the HP 98550A Color Graphics Board. The block mover hardware can be used to copy areas of the offscreen memory into visible memory. Font glyphs, which define the pixels to be turned on for a particular character font or set, are generally loaded into offscreen memory so that the block mover can be used to render the glyphs to a window at very high speeds. Pixmap patterns are also loaded into offscreen memory so that the block mover can be used to paint areas of the screen using the pixmap pattern (this is how a window background is painted). A pixmap is an array of pixel values (numerical values typically between 0 and 255) that determine the color of individual pixels.

Offscreen memory is limited to the size provided by the display hardware. Additional memory cannot be allocated by the system, and so the allocation of offscreen memory must be done carefully. Other processes can obtain offscreen memory for the storage of unique pixmaps. The pixmaps can subsequently be used for rendering operations, such as a tile to fill a polygon, a background pattern for a window, or an image used frequently in a program (e.g., a pushbutton outline).

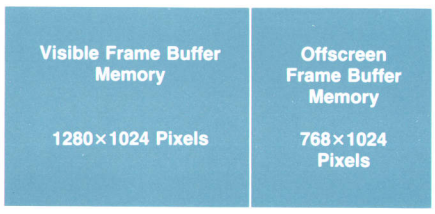


Fig. 2. *Frame buffer memory in the HP 98550A Color Graphics Card.*

Object-Oriented Approach

When it came to deciding how to implement the allocation of shared and private resources for clients, we decided to use an object-oriented approach and encapsulate the resources in objects.[3] The first thing we did was to identify the items we wanted to treat as objects. We identified three types of graphics resource manager objects and their attributes.

- Shared memory objects, which are used to share fonts or information about some aspect of the display or system state.
- Offscreen memory objects, which are used to reserve an area of the offscreen memory resource.
- Semaphore objects, which are used to share a system semaphore. The semaphore helps synchronize various processes.[4]

The attributes of GRM objects are divided into two groups, general attributes and specific attributes. The general attributes of a GRM object are a set of fields that define the object's name. These fields are consistent among all GRM objects. The following shows the name fields for a GRM object.

```
int     class;     /* class of object, client defined */
dev_t   screen;    /* screen device */
int     window;    /* X window id */
char    name[GRM_MAX_NAME_LENGTH]; /* string identifier
        of object */
dev_t   device;    /* disk device for fonts */
int     inode;     /* inode of a font */
int     key;       /* key of a font */
int     partition; /* partition of offscreen memory */
```

The name of a GRM object is a conjunction of all the fields. Two objects may differ by as little as one value in any one of the fields.

Object-specific information is added to the instance of an object. For example, a shared memory object includes the specific size of the object and its specific location in the GRM shared memory segment, and an offscreen mem-

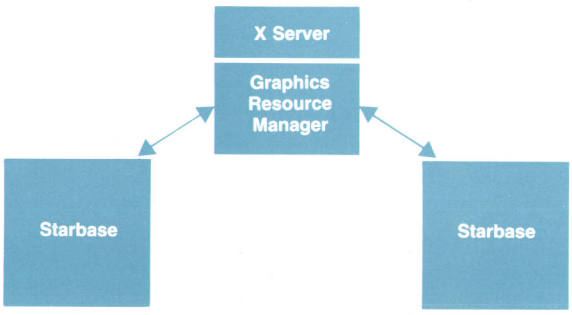


Fig. 3. *Architecture for building the GRM into the X server.*

ory object is described by its specific width, height, and depth, as well as its specific location in three dimensions in offscreen memory.

Operations on Objects

The GRM supports a set of operations that can be performed on the objects in a consistent way.

- GrmCreateObject. The GrmCreateObject function allocates an object of the requested class with the object instance, allocates the requested resource, and adds the client to the list of clients that are using the object. If the object already exists or cannot be created, the GRM returns an error. The client may then share the object, if it desires, by calling the GrmOpenObject function.
- GrmOpenObject. If the described object already exists (from calling GrmCreateObject), the client is added to the list of clients that are sharing the object. The GRM then passes the object's attributes back to the client. If the object doesn't exist, the GRM returns an error.
- GrmCloseObject. The GrmCloseObject function causes the GRM to delete the client from the list of clients that are sharing the object. When all clients have lost interest in an object, the object is destroyed, and the object's resources are freed.

Each function is an atomic operation because no other operation is allowed to be performed while one is in progress. As the project progressed it became necessary to group several of these operations into one large atomic operation. Functions were added to mark the beginning and end of these larger transactions.

The GRM also supports a function to find and list the objects it has created. To query the existence of sets of objects, the client can supply an object name with the fields set to match the value fields in other objects. This function is primarily used for debugging purposes.

Design and Implementation

The project teams investigated three main architectures to determine the best design:

- Build the GRM into the X server. One of the first architectures we examined was building the GRM functionality into the X server. In this architecture, the Starbase programs would communicate with the X server to allocate resources. Fig. 3 shows this architecture. We did not choose this architecture for several reasons. One reason is that the X server is used primarily as a rendering engine. The X server could be busy for many seconds performing a rendering request, causing the Starbase client to block until the X server could process a request. Also, GRM functionality would become dependent on a particular software technology in the X server, which may change as enhancements are made to X. Another problem occurs when a Starbase application is running alone in raw mode. The X server would have to be executed to support the Starbase client, even though the X Window System operation was not desired.
- Construct the GRM as a library. The second architecture the team examined implemented the GRM as a library, which could be linked into the X server and Starbase clients (see Fig. 4). Resource allocation would be performed by the library with multiprocess communication done through a single shared memory segment. With this scheme, allocation of objects could be done very quickly. The allocation operation would consist of directly manipulating data structures in the shared memory segment. This model was not chosen because of concerns about its ability to support future upgrades, and because it relies on consistent operation among all implementations that manipulate the shared memory information. We felt that we could achieve more robustness by choosing a protocol-based communications model. To support future version changes in this model, the data structures would have to be designed with built-in flexibility and version information. Proving that a newer version of the GRM library would work properly with older versions and vice versa would have been very difficult.
- Form the GRM as an independent process. After considering the previous two models, the project team settled on implementating the GRM as an independent process. The independent process model is shown in Fig. 5. The independent process model provides logical isolation between the GRM and its client processes (the X server and Starbase processes). The GRM process is free to define its data structures for allocating objects without worrying about access to these structures by the client processes. This architecture also enabled the designers of the GRM to be flexible in the algorithms used to allocate the objects without worrying about backwards compatibility with previous versions of the GRM. The protocol between the GRM and its clients is also typed with a version number, and the protocol data structures are padded to maximize the potential upgradability of the GRM services.

Interaction with X and Starbase

Communication with the GRM is originated by either

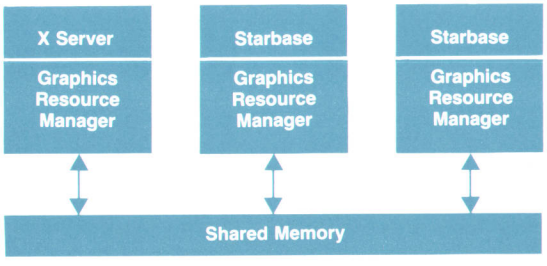


Fig. 4. *Architecture for constructing the GRM as a library.*

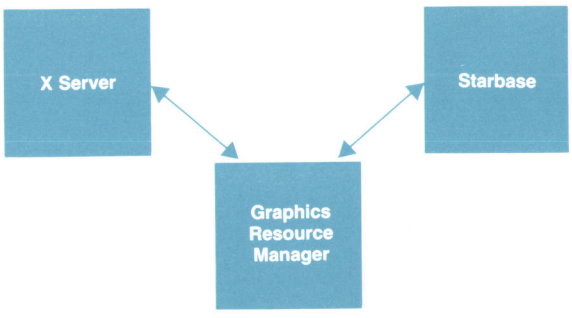


Fig. 5. *Architecture for constructing the GRM as an independent process.*

the X server or the Starbase display driver. The GRM process works by receiving a request, processing the request, and then returning a reply message to the requester. An application can perform both X library calls and Starbase library calls. This results in activity by both the X server and the Starbase driver. To get their work done, these GRM clients can call functions in the GRM library to create or open objects. The operation is synchronous because the client is blocked until the operation is completed by the GRM.[5] The GRM library packages the client request and sends the request to the GRM process. The GRM process processes the request, and if it is asked to create an object, allocates the resources for the object. The client is then added to the list of clients referencing the object. Finally, the GRM process returns a reply, which is received by the GRM library. The GRM library unpacks the reply and returns information describing the object to the caller.

The GRM process never directly modifies data in the GRM shared memory segment or in the display hardware. The GRM process instead acts upon an "abstract view" of these resources. The GRM maintains a data structure representing the available resources in the GRM shared memory and the display hardware offscreen memory (see these data structures in Fig. 1). When the GRM process allocates an object, it updates the associated data structure.

Allocation of Offscreen Memory

Currently, all HP display devices supported by the HP 9000 Series 300 and Series 800 product lines provide an offscreen memory resource. This memory is configured on the device as an extension to the memory used to hold viewable information on the display. Since display memory has a width, a height, and a depth, the offscreen memory also has these dimensions. This complicates the sharing of this memory because the GRM memory manager must allocate three-dimensional objects. Offscreen memory is relatively easy to manage if only one process wishes to display data on the screen at a time. However, in the Starbase/X11 Merge architecture, multiple applications share the display device, so managing the sharing of the offscreen memory resource is quite a challenge. On some HP display devices, pixmaps of varying depths can be allocated. Also, some display devices require that the pixmaps be aligned on pixel boundaries for efficient access. The challenge is to be able to allocate an arbitrarily sized and aligned three-dimensional box out of an arbitrarily sized three-dimensional box of free space. In addition, the algorithm must efficiently deal with the resulting free space for future allocations.

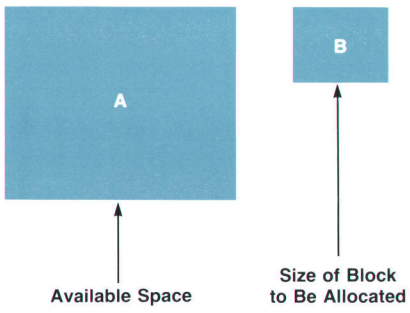


Fig. 6. *Two-dimensional allocation of offscreen memory. Box A is the available memory and box B is the space to be allocated from box A.*

Three-dimensional objects are typically perceived as spheres and polyhedra of various shapes and sizes. Pixmaps are represented as three-dimensional objects as six-sided blocks. A pixmap generally has a uniform width, a uniform height, and a uniform depth. The GRM algorithm addresses just such pixmaps.

The Two-Dimensional Case

Three-dimensional allocation is best explained as an extension of the two-dimensional case. The following discussion of the two-dimensional case will show that the addition of a third dimension is a fairly simple extension of the two-dimensional philosophy.

We start with the two boxes shown in Fig. 6. Box A represents the available memory resource and box B is the space to be allocated out of box A. If box B is placed inside box A, the rest of A can be divided into any of the configurations shown in Fig 7.

Configuration 1 produces a lot of fragmentation of the free space. This fragmentation alone is enough to discount it as a viable option. This leaves configurations 2 and 3. There is only one difference between these two configurations and that concerns how memory is globally allocated. With configuration 2, free space is cut into vertical strips which results in memory being allocated in vertical strips, and in configuration 3, free space is cut into horizontal strips which results in memory being allocated in horizontal strips. In general, it makes little difference which configuration is chosen. For software used on Hewlett-Packard workstations, there is a reason to use horizontal strips. Fonts are stored as horizontal strings of characters. Since caching fonts is a major use of offscreen memory, configuration 3 was chosen as the optimal solution.

Adding a Third Dimension

Adding a third dimension to this problem means taking the two-dimensional view and adding the concept of a

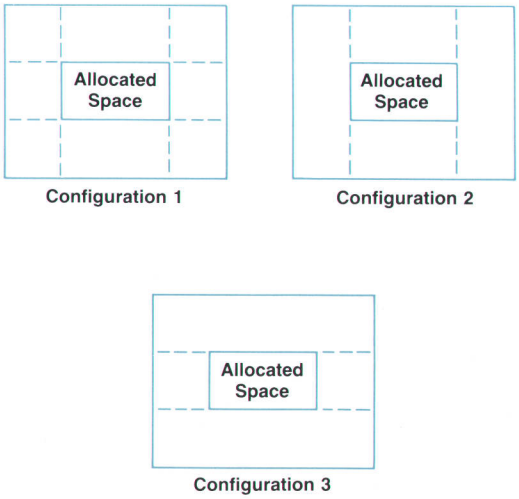


Fig. 7. *Different memory allocation configurations possible when space for a box is allocated from a larger box.*

front and a back to the object being allocated (see Fig. 8a). As with the two-dimensional model, there are a few ways to handle breaking off front and back pieces to make efficient use of the resulting space. Each method results in six free blocks and one allocated block out of the original block of memory. To coalesce the blocks when an allocated block is freed, the GRM associates the free blocks resulting from an allocation with the allocated block (see Fig. 8b). With this scheme, when the originally allocated block is freed, the blocks that can coalesce with it are easily found.

The allocated block forms a node of a tree, with the leaves of the tree initially being free blocks. New requests for offscreen memory cause one of the surrounding blocks to be allocated, with the result being that the new allocated block becomes a node with a new set of leaf blocks showing the free areas—that is, the tree grows (see Fig. 9). As blocks are freed the tree shrinks as leaves are coalesced with parent nodes. For efficient access, the GRM maintains a list of free blocks. This list optimizes the search for the best-sized free block to satisfy an allocation request.

The GRM Daemon

The purpose of the GRM process, or daemon, is to manage the allocation of graphics display hardware resources for all processes that want to use these resources. As such, it maintains a comprehensive list of the resources that have been allocated to these processes. The GRM daemon can only perform this task correctly if it can be certain that there is only one GRM daemon process that is allocating resources to all applications requesting them.

Typical daemon processes are started by an initialization script at system boot time. In this situation, uniqueness of a daemon process can be easily assured by avoiding multiple invocations of the script that starts the daemon process. However, the situation for the GRM daemon is different because the GRM daemon is not started at boot time.

Since the GRM daemon has a specialized purpose, it is preferable to have it executing only on an as-needed basis, rather than running continuously as would be the case if it was started at boot time. The GRM daemon is therefore designed to be spawned only by a process that requires access to the display hardware. Of course, it is only necessary to spawn a GRM daemon process if one has not already been put into service by another graphics application.

The design of the Starbase/X11 system dictates that the X server and all Starbase applications absolutely depend on the proper functioning of the GRM daemon. As such, the design of the GRM daemon required a foolproof method to ensure that for a particular host system, exactly one GRM daemon is given the task of mediating the use of all display hardware associated with that host, even when two or more

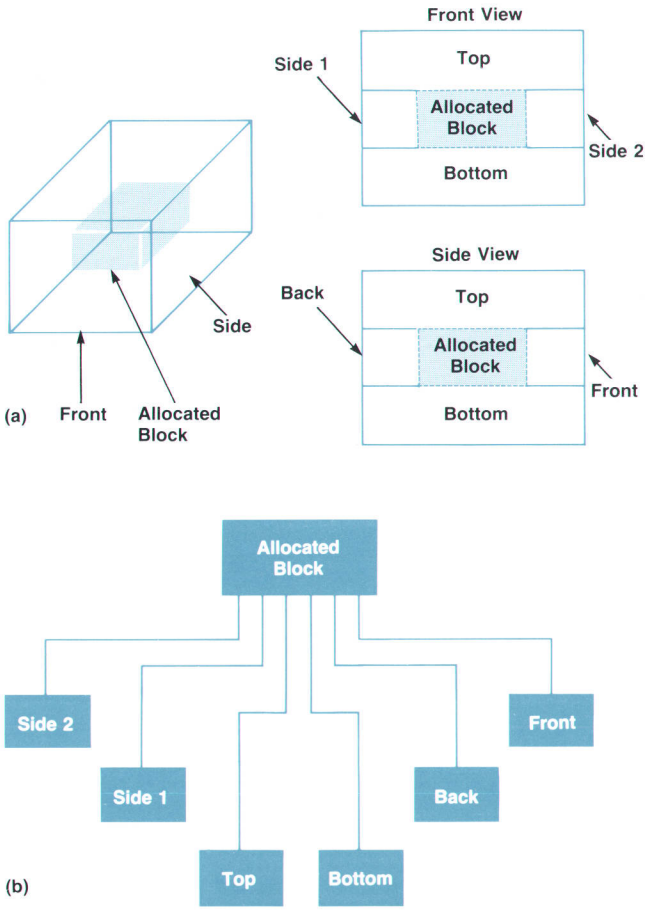


Fig. 8. *(a) Two-dimensional views of an allocated block with front and back added. (b) Data structure representation of an allocated block with the six free blocks from the original block of memory.*

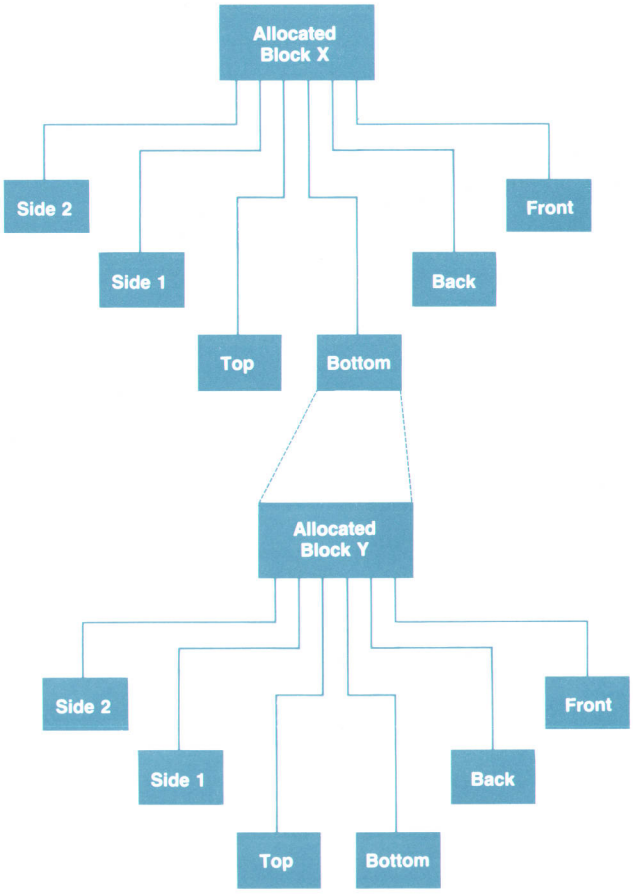


Fig. 9. *Allocation of a new block. The new allocated block becomes a node with six leaves, which represent free blocks.*

applications attempt to spawn a GRM daemon simultaneously.

The HP-UX Semaphore System

A simple solution to the problem of guaranteeing uniqueness for the GRM daemon process is to use a semaphore to ensure that only a single daemon has permission to continue as the resource manager. Potentially, several GRM daemon processes could be starting simultaneously, each trying to test and set the GRM daemon semaphore. The GRM semaphore mechanism ensures that only one of those processes actually succeeds in the test and set operation, with the remaining processes being obligated to recognize that another GRM daemon process is principal and to exit.*

Using a system semaphore to implement this scheme would have been trivial had it not been for a limitation in the behavior of the HP-UX system semaphore during the creation of a semaphore.** This limitation is that the value of a semaphore after its creation is not defined.

While the operating system does provide an atomic operation for creating a system semaphore exclusively (the operation succeeds only if the semaphore does not already exist), it does not guarantee the state of the newly created semaphore to be any particular value. Therefore, a process can know that it has created a previously nonexistent system semaphore and that it must initialize the value of the semaphore, but a separate process cannot know that a given semaphore has just been created and is not yet initialized. Since the creation of a semaphore and the initialization of its value is a two-step process, it is conceivable that another process might attempt a semaphore operation between the creation and initialization steps. For an application such as the GRM daemon, this limitation presented a severe problem that required a substantial workaround.

The problem with the system semaphore can be clarified with an example (see Fig. 10). Consider the situation where two GRM daemon processes (process A and process B) have been started and they are both attempting to create and then test and set the GRM semaphore. Suppose that process A successfully creates the GRM semaphore. Before process A has had a chance to initialize the value of the semaphore it is preempted by the kernel's scheduler. Process B then comes along, notices that the GRM semaphore already exists, and attempts a test and set operation on the semaphore which currently has an undefined value. The test and set operation may or may not succeed depending on the random value of the semaphore. However, if it does succeed, process B will think that it has been designated as the principal GRM daemon and carry on as such. Meanwhile, process A has regained the processor and proceeds to initialize the value of the GRM semaphore, overwriting the effect of the test and set operation of process B. Subsequently, process A will successfully execute a test and set operation on the GRM semaphore resulting in two GRM daemon processes running when there should only be one.

Various workarounds to the semaphore initialization problem were attempted, but none of them that exclusively used system semaphores would work because it could not be determined whether or not the value of a semaphore was valid. A colleague who had experienced similar problems with system semaphores suggested that a file lock*** could be used as the GRM semaphore. Besides being used to control access to a file, file locks can be used in an advisory capacity in much the same way as a system semaphore. File locks have the advantage that the test and set operation does not require the two-step (not atomic) "create and initialize" procedure used by system semaphores. However, file locks can be difficult to manage when the file being used as the subject of the lock, that is, the lock file, is not writable or is transitory. As such, the GRM daemon uses a file lock only as a means to control access to the system semaphore, and the semaphore is responsible for awarding a single GRM daemon process the guarantee of uniqueness.

The GRM Daemon Semaphore System

As mentioned earlier the purpose of the GRM daemon semaphore system is to ensure that exactly one GRM daemon process successfully claims responsibility for managing the allocation of the display hardware. The system must be reliable in the face of an arbitrary number of com-

*In the context of this article, a semaphore can have a value of zero or one. A semaphore is initialized to a value of zero. A test and set operation on a semaphore succeeds only when the value of the semaphore is initially zero. A successful test and set operation results in the semaphore's having a value of one. The process of testing and setting the value of the semaphore is said to be an atomic operation, meaning that the operation is indivisible.

**The HP-UX system semaphore conforms to the AT&T UNIX System V Definition.

***A file lock is a file system semaphore associated with a segment of a particular file, which is referred to here as the lock file.

Time Slice	T0	T1	T2	T3	T4	T5	T6
GRM Daemon Process A	Instantiated	Create GRM Semaphore (1)	Idle	Idle	Initialize Value of GRM Semaphore (3)	Test and Set GRM Semaphore	Begin GRM Daemon Operations (4)
GRM Daemon Process B		Instantiated	Finds Existing GRM Semaphore	Test and Set GRM Semaphore (2)			Begin GRM Daemon Operations (4)

(1) For this example, the GRM semaphore has an incidental value of zero following its creation. In general, an HP-UX semaphore has a random initial value.
(2) Since the GRM semaphore had an initial value of zero, the test and set operation succeeds.
(3) The effect of the test and set operation of GRM daemon process B at time T3 is nullified by the initialize operation.
(4) Each of two GRM daemon processes has successfully tested and set the GRM semaphore. The semaphore thus falls to allow only one process to continue as the principal GRM daemon.

Fig. 10. *Timing diagram of a semaphore failure at initialization of two processes.*

peting infant GRM daemon processes (processes that have not yet established their principal status). The system must not require user intervention even in the face of an ungraceful exit by a GRM daemon process.

Given these considerations the GRM semaphore system was designed to accommodate the following situations:

- Any process killed without opportunity for a graceful exit. This means that the design must be able to recover when the GRM system semaphore and/or lock file are left around after the process that created them is terminated by other than programmatic means.
- An arbitrary number of infant GRM daemon processes attempting to claim principal (unique) status simultaneously.
- An existing GRM daemon process holding the GRM semaphore while in the process of exiting.

A GRM daemon process has three phases. Its first phase is during the initialization of an application requiring the services of a GRM daemon. The second phase is during its attempt to set the GRM semaphore and claim principal status. The third phase is the operational phase, when it is assured uniqueness and carries out the tasks required of the display hardware resource manager.

The Application Phase. Starbase applications and the X server must have an executing GRM daemon to function. During initialization, a GRM library routine within these programs attempts to make a connection with the GRM daemon through its designated socket address.[6] If it fails to make the connection, the routine assumes that there is no GRM daemon process executing and it spawns a GRM daemon process. The spawned process "daemonizes" itself (detaches from any terminal or parent process), sets its user identification number, and then attempts to establish itself as the only GRM daemon process.

Claiming Principal Status. Immediately after an infant GRM daemon process is daemonized, it proceeds in its attempt to become the only GRM daemon process. Fig. 11 shows the timing diagram for two processes (processes B and C) trying to claim principal status and control of the GRM semaphore.

The first step is to test and set the file lock, thereby claiming exclusive access to the GRM semaphore. In this way, if the semaphore does not already exist then the GRM daemon can create the semaphore and initialize its value without fear that another GRM daemon process may be trying to access the semaphore at the same time. The lock file, which is used as the subject for the file lock, must be created if it does not already exist. If the file already exists, either another process is trying to access the GRM semaphore or a process was killed while attempting such access.

The next step is to see if the GRM semaphore exists. If the semaphore already exists, then it is known to have a valid value. This is true since any GRM daemon process that created the semaphore is by convention guaranteed to have initialized its value before releasing the file lock. If the GRM semaphore does not exist, then it is created and initialized with a valid value. Since the process is holding the file lock, it need not worry about another process attempting to test and set or initialize the value of the GRM

Time Slice	T0	T1	T2	T3	T4	T5	T6	T7	T8	T9
GRM Daemon Process A	Lose Last GRM Client	Remove Listening Socket	Remove Semaphore	Exit						
GRM Daemon Process B	Test and Set File Lock	Test and Fail to Set (1) Semaphore	Release File Lock	Sleep	Test and Fail to Set File Lock (3)	Retry (3)	Test and Set File Lock	Test and Fail to Set (5) Semaphore	Retry until Timeout (6)	Exit (7)
GRM Daemon Process C		Test and Fail to Set File Lock (2)	Retry (2)	Test and Set File Lock	Test and Set Semaphore (4)	Release File Lock	Open Listening Socket	Accept GRM Clients	Continue with GRM Daemon Operations	

(1) GRM daemon process A holds the GRM semaphore.
(2) GRM daemon process B holds the GRM file lock.
(3) GRM daemon process C holds the GRM file lock.
(4) The test and set semaphore operation includes the creation of the semaphore if it doesn't already exist. The creation, testing, and setting of a semaphore can be considered to be an atomic operation since all of these operations are executed while holding the file lock and only one process can be holding the file lock at a given time.
(5) GRM daemon process C holds the GRM semaphore.
(6) A retry cycle includes setting the file lock, testing the semaphore, releasing the file lock, and a short sleep.
(7) Concede and exit (i.e., time out) after enough time has elapsed during the retry cycle to allow an existing GRM daemon process to service a disconnect request from its last client, free various resources, remove its listening socket, and remove the GRM semaphore.

Fig. 11. *Timing diagram for the GRM semaphore system.*

semaphore.

Once the existence of the GRM semaphore is established and it is known to have a valid value, an attempt can be made to test and set the semaphore. If successful, the semaphore is then held by the process that set it. Once the semaphore is set, the lock file can be removed, allowing other processes to create a new lock file in order to access the semaphore. The life of the lock file is generally limited to the duration of the creation and initialization of the GRM semaphore.

If the test and set or any of the preceding operations is not successful, then the file lock must be released to provide other infant GRM daemon processes the opportunity to access the GRM semaphore. After the file lock has been released, the infant GRM daemon process will sleep for a short period of time and then retry the entire procedure. The sleep duration is short enough to expedite the GRM daemon startup procedure. However, the retry loop results in a delay that is long enough to ensure that there is enough time for an exiting GRM daemon to finish its exit and clear the GRM semaphore.

Operations Phase. Once established as the principal GRM daemon process, the GRM daemon goes about initializing its data structures and opening its listening socket to begin serving its purpose. One or more GRM clients will then make connections to the GRM daemon and request display hardware resources as needed. When a GRM client exits, its connection with the GRM daemon is closed and the resources allocated to it are freed and made available to other GRM clients.

When the GRM daemon detects the absence of its clients, it removes the listening socket, removes the semaphore, and then exits. Any GRM client that may be starting up at this time will fail to establish a connection, which includes verifying the connection with a full handshake, and it will start the process over again by spawning a new GRM daemon.

Conclusion

The GRM provides a means for allocating a system's display resources among various competing clients. The GRM also provides a means of sharing information among the clients through the encapsulation of the information in objects. One client can access an existing object if it knows the name of the object, even if the object was created by another client. The client can access the data by asking the GRM to open the object. The GRM also provides a sophisticated memory allocation mechanism for the scarce offscreen memory resource. The mechanism includes a means to coalesce freed fragments of offscreen memory for reuse. Finally, the design of the GRM interface ensures that only one GRM daemon process runs on a given system, even though several clients initiate access to the GRM simultaneously.

Acknowledgments

We'd like to acknowledge Dan Garfinkel and Steve Lowder of HP Laboratories for their conceptualization of the graphics resource manager facility and their aid in designing the architecture.

References

1. F.W. Clegg, et. al, "The HP-UX Operating System on HP Precision Architecture Computers," *Hewlett-Packard Journal*, Vol. 37 No. 12, December 1986, pp. 4-22.

2. M. Rochkind, *Advanced UNIX Programming*, Prentice-Hall, 1985.

3. G. Booch, "Object Oriented Development," *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 2, February 1986, pp. 211-221.

4. D. Comer, *Operating System Design, The XINU Approach*, Prentice-Hall, 1984, pp. 13-16.

5. R. Summers, "A Resource Sharing System for Personal Computers in a LAN: Concepts, Design, and Experience," *IEEE Transactions on Software Engineering*, Vol. SE-13, No. 8, August 1987, pp. 895-904.

6. S. Leffler, et. al, *Design and Implementation of the 4.3 BSD UNIX Operating System*, 1989, pp. 296-298.

Sharing Access to Display Resources in the Starbase/X11 Merge System

The Starbase/X11 Merge system provides features to allow Starbase applications direct access to the display hardware at the same time X server clients are running. There are also capabilities to allow sharing of cursors and the hardware color map.

by Jeff R. Boyton, Sankar L. Chakrabarti, Steven P. Hiebert, John J. Lang, Jens R. Owen, Keith A. Marchington, Peter R. Robinson, Michael H. Stroyan, and John A. Waitz

HP'S GRAPHICS DISPLAY HARDWARE provides many display resources that must be carefully managed to maintain order on the display when competing HP-UX processes, such as the X server and Starbase applications, are attempting to access the display hardware at the same time. The hardware resources that must be shared among these processes include the frame buffer (video RAM), cursors, fonts, and the color map. This article discusses methods used to allow Starbase applications and the X server to share access to this common pool of hardware resources, and a method called direct hardware access (DHA), which enables Starbase applications to achieve high performance when accessing the display, while maintaining the integrity of the X Window System.

Display Hardware

Fig. 1 is a block diagram of a typical graphics display. This is a generalized model and does not represent the implementation of any particular graphics product. Some elements are optional—for example, only 3D systems need a z-buffer and some low-end graphics systems have no graphics accelerator.

Graphics Accelerator. The graphics accelerator provides specialized hardware to perform graphics operations on commands and data from the display driver running on the host system. The fundamental job of the accelerator is

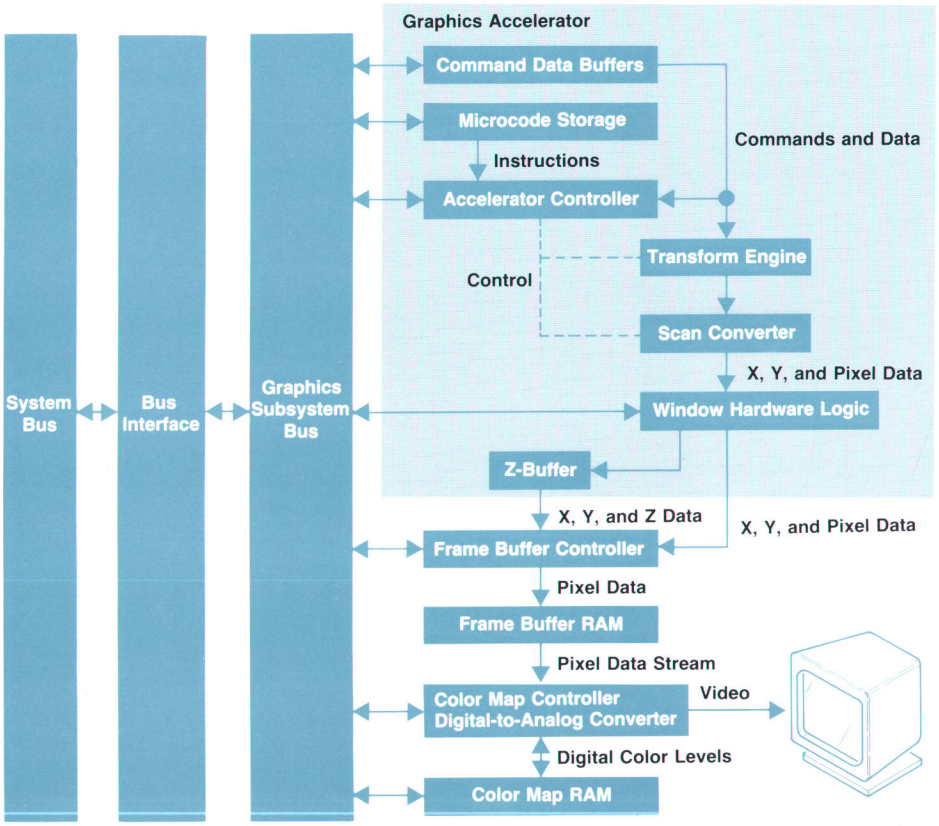


Fig. 1. *Block diagram of a typical HP hardware display system.*

to apply viewing and modeling transforms and light source models to the data to convert it into a format usable by the scan converter. The scan converter consists of hardware for the generation of pixel data that represents polyline and polygon primitives. Operations on more than one window are supported by the window control logic, and hidden surface removal is provided by the z-buffer. The accelerator also has responsibility for the control of most other hardware resources in the graphics processor, such as the configuration of the frame buffer and color map.

Frame Buffer. The frame buffer is a specialized (usually dual-ported) RAM. Each addressable location in the frame buffer represents one picture element, or pixel. Some portion of the frame buffer is displayable, so its contents represent the current image on the screen. Pixel values are read sequentially from the frame buffer and converted to a video signal by the color map and its associated circuitry. The entire frame buffer can be scanned as many as 60 times per second to keep a steady image on the monitor. The portion of the frame buffer that is not displayed is called offscreen memory. Special circuitry called a block mover, which is located in the frame buffer controller, is used to copy a rectangular region from one place in the frame buffer to another. Both the on-screen and offscreen portions of the frame buffer are accessible to the graphics accelerator.

The frame buffer is physically separate from system RAM* but it is mapped into the virtual address space of all processes that access it. Therefore, it is possible for several processes to have the same physical RAM of the frame buffer mapped into their virtual address space (see Fig. 2). This requires that processes must cooperate when making modifications to the frame buffer. The methods we use to share the frame buffer are discussed later.

Color Map. The color map is a very high-speed lookup table that maps the numbers stored in the frame buffer to the actual color values. The user specifies the mapping with commands like: the number 5 in the frame buffer represents the color a,b,c where a,b,c are the intensities of red, green, and blue that must be mixed to create the desired color. After looking these intensities up, the color map converts them to analog voltages and sends them to the monitor.

*System RAM and frame buffer RAM are both components of the physical address space. The physical system RAM is the 4M to 48M bytes of DRAM memory purchased with the machine. The physical frame buffer memory is video memory mounted on the display controller card.

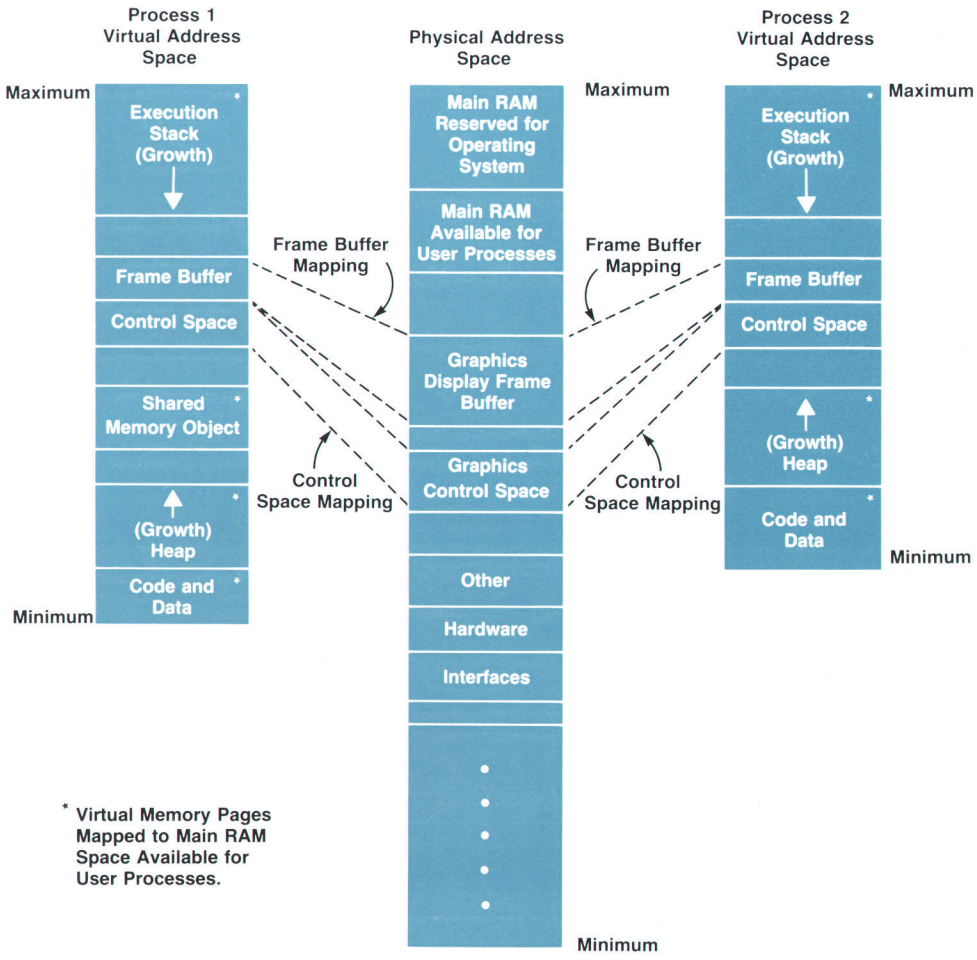


Fig. 2. *Two HP-UX processes mapping the display frame buffer and control space into the same physical address space.*

Direct Hardware Access

In the X Window System, user processes, or clients, do not render directly to the frame buffer. To gain access to the frame buffer, clients make drawing requests to the X server, which is the only process with access to the frame buffer. The server has control and knowledge of the state of the frame buffer. However, to achieve maximum performance and functionality, some clients, such as Starbase applications, require direct access to the frame buffer. To gain direct access to the display in an organized way, a client must cooperate with the server. The client must obtain information from the server about the areas of the frame buffer that represent the visible area of the client's window and all rendering by the client must be clipped to this area. This is done by requesting the server to register an existing window for direct hardware access (DHA). In response to this request the server sets up mechanisms to pass the clipping information to the client and to update it as necessary.

Two methods are used to pass information from the server to the DHA client: shared memory and HP X extension library calls. Graphics resource manager shared memory is used for information that does not change in size, such as the cursor state and fonts. Variable-size data such as the clip list is passed to the client via HP X extension library calls (routines with an XHP prefix). Using shared memory for variable information would create shared memory fragmentation problems, and the overhead of conversing with the graphics resource manager (GRM), which manages the shared memory area used by X server and Starbase processes, could cause performance problems. The communication links between a DHA client and the X server are shown in Fig. 3.

Data Structures

Fig. 4 shows the data structures in GRM shared memory and process private memory that allow direct hardware access by Starbase DHA applications. Pictured are the data structures that would exist for one window on one screen. Multiple windows, color maps, and screens are supported and many of the structures shown are replicated in such circumstances. The X server and the Starbase processes have pointers for accessing the data structures in shared memory. The data items shown in Fig. 4 will be referenced and explained in later sections of this article.

Opening a Window

To allow a Starbase DHA process to be ported to run in X with little or no source code changes, it is important that the normal gopen() procedure work the same way it does when the application is drawing directly to a graphics display.

The following activities occur during a Starbase open (gopen()) of an X window:

- If it is not already running, the graphics resource manager is started so that the Starbase process can access shared memory objects resulting from a DHA window registry.
- Tests are made to determine if the pathname parameter, which names the window, refers to an X window or one of the other supported objects of gopen().
- If the object being opened is an X window, the host name, the display identifier, and the screen number are obtained. If a driver-level socket connection to the server for that screen does not exist, one is opened.
- If the window is to be an accelerated window, an accelerator state identifier is generated.
- The XHPRegisterWindow() procedure is called. If it succeeds, then a data structure (DHA window object) will be created in shared memory that contains the registered window information.
- The frame buffer is opened and mapped into virtual space using the device pathname returned by the window registry call.
- The registered window object and other DHA shared memory objects, such as the DHA screen object, the display state, and the X server's cursor state, are opened. These data structures are shown in Fig. 4.
- The initial clip list for the window is obtained from the server.

Registering for DHA Access

An HP X library extension, XHPRegisterWindow(), was added to the server to allow a client to request DHA access to a window. The client passes the identification numbers of the desired window and the screen containing the window. Additionally, the client may request that the window be registered for use by a graphics accelerator. Upon receipt of the registration request, the server requests the graphics resource manager to create a structure in shared memory and fill it with information pertinent to the window. In Fig. 4 this structure is called the DHA window object. The information in the DHA object for each registered window includes:

- Clipstamp. An integer counter that is incremented whenever the clip list for the window changes. This is used as a trigger to the client that it needs to obtain a new clip list via the XHPGetClipList() library procedure.
- nUsers. An integer value representing the number of registrations active against the window.
- nAccelerated. An integer value representing the number of accelerated registrations active against the window.
- window_id. An integer value representing the server's identification number for the registered window.

After the DHA window object is created, the server passes its GRM shared memory identification back to the client.

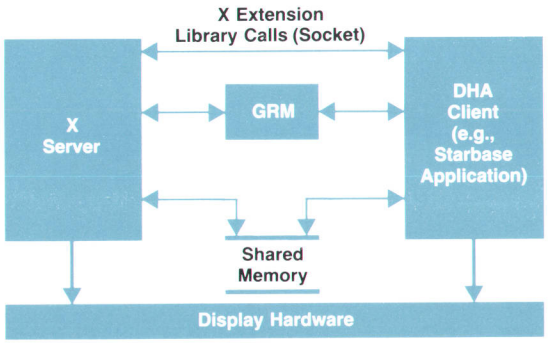


Fig. 3. *Communication paths between a DHA client and the X server. GRM = graphics resource manager.*

The client obtains access to the DHA window object in GRM shared memory and reads the information supplied by the server. As the state of the window changes, the information in the DHA window object is updated by the server.

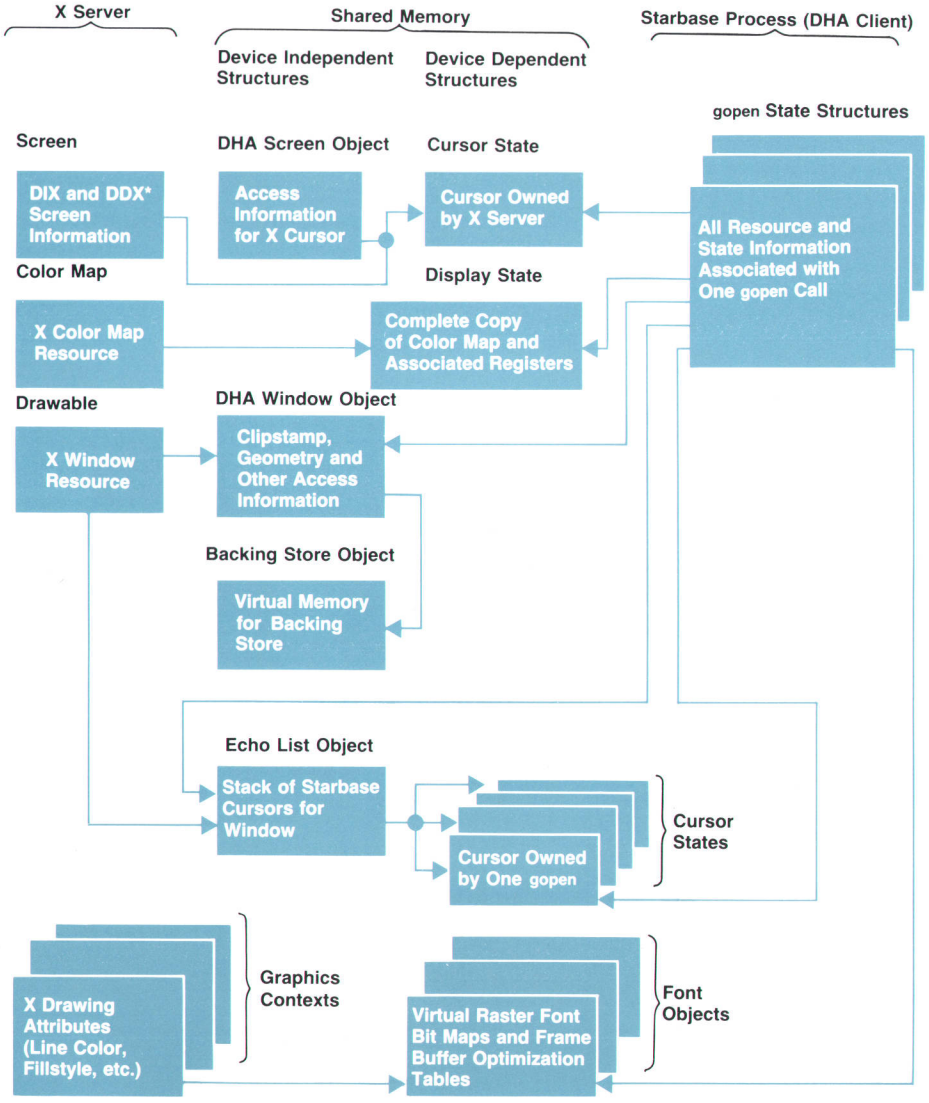
A window may be registered for DHA access multiple times by the same client or by multiple clients. All registrations use the same shared memory object (DHA window object). A count is kept of the the number of current registrations on a window. A client terminates a registration with the library procedure XHPUnRegisterWindow(). When the number of registrations drops to zero, the server requests the GRM to delete the shared memory object and the window is no longer directly accessible by clients.

The Clip List

The visibility and position of the registered window can change at any time. The user can partially obscure the registered window with another window, move it to another area, iconify it, and so on. The clip list is a list of rectangles describing the areas of the window that are visible or obscured. In Fig. 5a window A is partially obscuring window B. Window A is completely visible and its clip list consists of only one rectangle. The clip list for window B consists of three rectangles, two visible and one obscured.

The clip list is a dynamic list that can be as small as one rectangle (the window is fully visible) or as large as several hundred rectangles. Rather than pass this information through shared memory, it is the responsibility of the DHA client to request the list via a library procedure. The clipstamp, which is created when a DHA client registers a window, provides a fast mechanism to notify all interested DHA clients when the clip list changes and they need to obtain a new clip list.

Whenever the clip list for a window changes because of events such as a window move or stacking order change, the server increments the clipstamp field of the DHA window object. When the DHA client wishes to render, it compares



*DIX = Device Independent X
DDX = Device Dependent X

Fig. 4. *The data structures that contain the information that enables display resource sharing between Starbase applications and the X server.*

the clipstamp in the DHA window object against its local copy. If they differ, the client knows the clip list has changed since its last rendering operation and it must request a new clip list. After making the request, the client copies the shared memory value of the clipstamp into its local copy for the next time. This mechanism avoids synchronization problems because no client ever clears the clipstamp field. Multiple clients sharing the same window merely bring themselves into synchronization with the current clipstamp value.

To obtain a new clip list, the client uses the library procedure XHPGetClipList(). The client passes to the server the identification numbers of the registered window and the screen containing the window. The procedure returns to the client the following information:

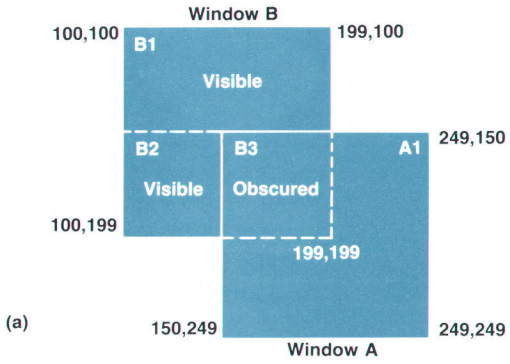
- x,y. Integer values representing the origin (upper left corner) of the window. This value is relative to the origin of the screen.
- Width. An integer value representing the width in pixels of the window.
- Height. An integer value representing the height in pixels of the window.
- Count. An integer value representing the number of rectangles in the clip list.
- Clip List Pointer. A pointer to a list of rectangles constituting the clip list.

The DHA client knows the size of the frame buffer and where the frame buffer's physical memory is mapped in its virtual memory space. By using this information in conjunction with the origin and size of the window, the client can index into the frame buffer and calculate the memory addresses it is allowed to access.

The union of the rectangles in the clip list covers the renderable area of the window. Each rectangle is specified by the x,y coordinates of its upper left and lower right corners. The values of these coordinates are relative to the origin of the window (see Fig. 5b). Each rectangle is marked as either visible or obscured. Visible rectangles are visible on the screen. Obscured rectangles are not visible because they are either obscured by another window or are partially off the screen. The client traverses this list, rendering into the visible rectangles. If the window has no backing store, which is a location in memory for backing up windows that become obscured, rendering to the obscured areas is discarded. If the window has backing store available and the client can render to it, then rendering to obscured rectangles is diverted to the backing store. Backing store is discussed in detail later in this article.

The client can request a clip list in one of three formats: YXBANDED, VISIBLE, or OBSCURED. In the YXBANDED format, both visible and obscured rectangles are present in the list (see Fig. 6). They are split and ordered so that all rectangles with the same y-origin will have the same height, thus creating bands across the window. Rectangles in the same band are sorted by increasing x-origin value. This type of ordering can enhance performance when rendering is done by filling horizontal scan lines. In the VISIBLE and OBSCURED formats, only rectangles of the respective type are present in the list. They are coalesced into the fewest possible number of rectangles and are not ordered. These formats are useful for displays that have hardware clipping capabilities.

A DHA client can use the X facility XSetClipRectangles() to restrict rendering to a subset of the window. If the graphics context containing the client clipping is specified to the



(a)

Clip List for Window A

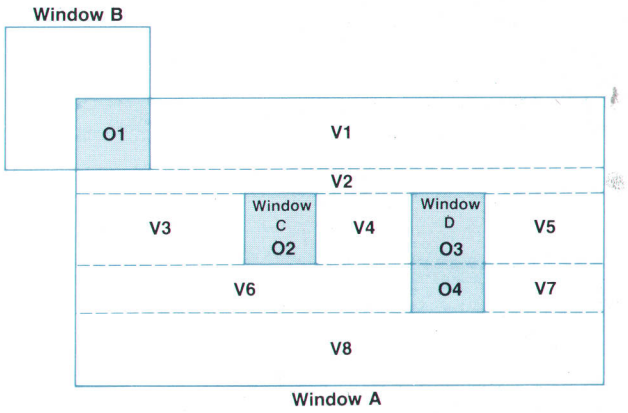
Rectangle	Obscured?	X1	Y1	X2	Y2
A1	False	0	0	100	100

Clip List for Window B

Rectangle	Obscured?	X1	Y1	X2	Y2
B1	False	0	0	100	50
B2	False	0	50	50	100
B3	True	50	50	100	100

(b)

Fig. 5. *(a) Two overlapping windows showing their positions in screen coordinates. (b) The clip lists for the overlapping windows in window-relative coordinates where X1,Y1 = upper left and X2,Y2 = lower right. X2 and Y2 are one pixel outside of the true boundary to make the mathematics easier.*



YXBANDED Clip List Rectangles for Window A:
O1, V1, V2, V3, O2, V4, O3, V5, V6, O4, V7, V8

Where:
O = Obscured
V = Visible

Fig. 6. *An illustration of a clip list for a* YXBANDED *window.* YXBANDED *window A is partially obscured by windows B, C, and D.*

XHPGetClipList() function, the resultant clip list will be restricted to that subarea.

MOMA Windows

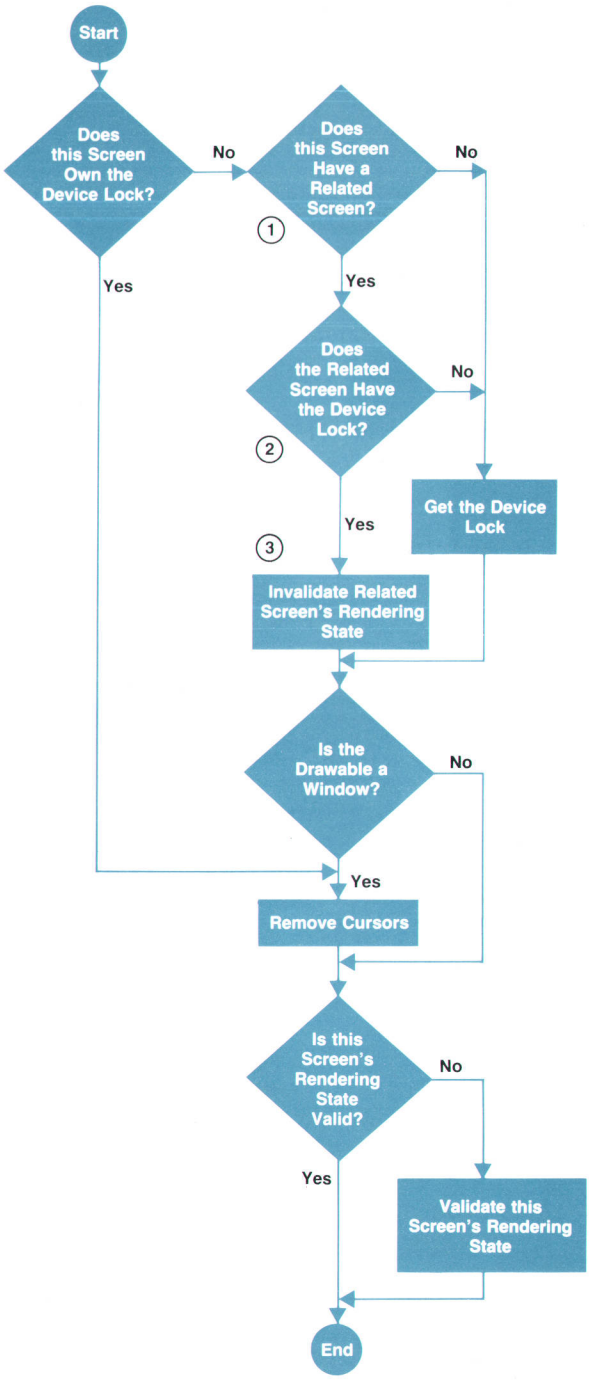
Multiple, obscurable, movable accelerated windows, or MOMA windows, refers to the hardware logic in the graphics accelerator that provides very fast drawing and clipping of multiple windows. The HP 98556A 2D Integer-Based Graphics Accelerator and the HP 98732A 3D Graphics Accelerator contain graphics accelerator engines that use hardware facilities for clipping. When a DHA client wishes to use an accelerator to render into a window, it registers the window as accelerated. For some devices, such as the HP 98556A, this also implies that the server will allocate a MOMA hardware clipping state on behalf of the client. For other devices, the DHA client allocates the clipping state.

When the clip list for an accelerated window changes, the server downloads the new clip list directly into the MOMA hardware on behalf of the client. However, there may be reasons why the DHA client must also be able to load the clip list directly into the accelerator. For example, on the HP 98732A 3D Graphics Accelerator, the clipping rectangles for only a single window are stored on the device. As graphics contexts are swapped into the accelerator, appropriate clip rectangles must be loaded into the MOMA hardware. When the server is able to maintain the clip list state in the accelerator, the accelerated DHA processes are able to achieve a steady throughput because they do not have to spend time downloading clip lists.

The server itself does not take advantage of graphics acceleration. There are two reasons for this. Currently no graphics accelerators render according to all the X specifications. More important, HP's accelerators are basically first-in, first-out queues—the rendering commands are processed in the order they arrive. Some operations that can be performed by HP's advanced graphics devices, such as the HP 98732A, can take a significant amount of time for X to perform. However, a critical factor in the usability of a window system like X is the response time for operations such as window moves and creations. If the X server operations must wait in line behind a long stream of complicated graphics primitives, the response time will not be acceptable.

Starbase/X11 Merge Locking Strategy

Graphics driver software is closely coupled to the graphics hardware it supports. The driver routines set hardware registers to certain values and then drawing operations or other actions are started. In a multitasking environment such as HP-UX, there may be more than one process that includes a graphics driver that needs to access the display hardware, and one process may be preempted or swapped out at any time, even during the execution of driver procedures. To prevent indeterminate results arising from multiple processes using the graphics hardware in an uncontrolled way, there must be some means of restricting access to one process at a time. The permission (or token) to use the display must be passed from one process to another.



1. In stacked screens mode, the other screen on the same device is referred to as this screen's "related screen."
2. Since stacked screens implies sharing one piece of hardware, only one lock exists in the HP-UX kernel so only one screen or the other can lock the device.
3. If one screen of a stacked screens mode server takes the lock from another, the screen losing the lock can make no more assumptions about the hardware. Therefore, the old screen's rendering state is invalid.

Fig. 7. *Flowchart for the routine* xosPrepareToRender *which is used to handle locking within the X server.*

One way this might be done is by implementing a token that the kernel controls. Only the process that has the token would be allowed to access the graphics hardware, and all other processes would actually be prevented from accessing the registers. This is not how the problem is solved in HP-UX. Instead, all processes are free to access the hardware, requiring that a convention be established and followed to ensure that only one process gains access to the graphics display at one time. The HP-UX kernel helps in this matter by providing a token in the form of a software semaphore, and by blocking processes that request the semaphore while another holds it. Processes that do not follow the protocol of waiting to gain access to the token are not prevented from changing the hardware registers. The special kernel semaphore in the Starbase/X11 Merge system is often called the display lock or kernel lock, and it locks access to the physical display.

X Server and DHA Processes

Since the display lock is a system resource that processes contend for, it is a prime candidate for creating the classic deadlock problem. A typical deadlock problem was encountered and solved for a situation involving a Starbase DHA process and the X server. A Starbase process might gain access to the display lock not only to operate on the display hardware, but also to operate on shared memory structures associated with the display. In the course of its operations, it may need to call one of the standard HP extension X procedures to communicate with the server. When the server wakes up to service this request, as well as any other input it has received, it attempts to get the display lock. A deadlock occurs because the Starbase process is waiting for the server to respond, but the server is waiting for the display lock.

To solve this and similar problems in the Starbase drivers, the calls to X procedures are strategically placed outside of code regions where the lock is held. An interesting example of this is the code to fetch a new window clip list. As long as a Starbase process running in a window does not hold the lock, the X server can process a request to change the clip list for the window. However, if the Starbase process gets the lock, then it cannot ask the server for the current clip list because of the deadlock that would result. The code to solve this problem incorporates the following algorithm:

```
while the clip list is out-of-date
        request a new clip list from the server
        get the display lock
        if the clip list that was fetched is still up-to-date
                then exit the loop—go on to render
                else release the lock—go back around the
                        loop again
endwhile
```

Locking within the X Server

The X server typically processes requests from several clients for one or more windows each time it detects that there is input to be processed (a wakeup). At some point during this processing, before the graphics hardware is accessed, the server process must obtain the display lock. All access to the hardware in the Starbase/X11 server is governed by the routine xosPrepareToRender() and its greatly simplified cousin xosLockDevice(). The duties of xosPrepareToRender() are to verify ownership of or claim the display lock, remove cursors (Starbase or X) from the area to which the server is about to render, and ensure that the X server's and X display driver's concept of the current rendering state are the same. Fig. 7 summarizes the actions of xosPrepareToRender. xosLockDevice(), as its name implies, only performs the locking portion of xosPrepareToRender(). It is used when it is desired to lock the hardware but not change the display.

In some places it is difficult for the X server software to determine whether the lock is already held. To handle the possibility of nested attempts to gain the display lock, each X display driver maintains a lock count. When the lock count (nesting level) reaches zero, the X display driver issues an unlock call to the graphics driver in the HP-UX kernel that maintains the semaphore for the locked device. Immediately before unlocking the device, the X display driver resets the hardware and any software registers it might be maintaining to a state consistent with the expectations of other processes that might access the display. Under normal circumstances, this reset is valuable. However, in stacked screens mode the reset is disastrous.

In stacked screens mode one physical display device is made into two screens and is opened as two separate devices. This causes the display driver to maintain a separate lock count for each open. If either count goes to zero, the physical device will be reset and unlocked. A busy server is likely to render to both screens in a single wakeup, so locking one half of a stacked screens mode server must imply locking the other half. Although the display lock is shared, rendering to one half of a stacked screens device invalidates whatever is known about the hardware state in the other half. Stacked screens mode is described in the article on page 33.

Since claiming the lock on a device excludes other processes from access to that device, sharing the hardware requires that the lock be claimed at the last minute. The deferred lock claim avoids holding off direct hardware access clients any more than necessary. This requirement is especially critical when running with multiple physical screens. There is obviously no need to hold off direct hardware access to one screen while the X server is writing to another.

Each X display driver provides an entry point called ValidateRenderingState(). This routine ensures that the hardware, display driver, and server are consistent and set up for rendering. Calls to ValidateRenderingState() can be very expensive, so care is taken to use it as little as possible. The usual reason for calling ValidateRenderingState() is that the hardware state is unknown or known to be invalid. For example, when the display driver releases the lock, the hardware is returned to its base state, so revalidation of the rendering state is necessary upon claiming the lock.

To minimize the number of times ValidateRenderingState() is called, the server keeps a pointer to the the last rendering state structure used for each screen. This pointer is set to null whenever the lock is surrendered, the cursor changes shape, color, or position, or the attributes of the window

change. Any of these changes means that the contents of the rendering structure itself have changed. When xosPrepareToRender() is invoked, if the new rendering structure is the same as the current one, the call to ValidateRenderingState() may be skipped.

Sharing Cursors

In the effort to ensure that Starbase applications running in the X Window System have full functionality so that user programs can be used in the new environment without source code changes, one particular area of Starbase functionality that proved especially difficult was the implementation of cursors in windows. Starbase implements many different kinds of cursors, including crosshairs, rubberband boxes, and raster cursors. For a Starbase process to draw, it must remove the cursors that interfere with the window to be accessed, perform the rendering operations, and replace the cursors. The same is true of the X server when it needs to render somewhere on the screen. The shared drivers effort described on page 7 allows much of the code that draws and undraws the cursor to be shared, but there is still a lot of logic that had to be carefully designed to ensure that the server and Starbase behave correctly in all situations. In Fig. 4 the data structure labeled "Cursor State" contains a data block for each cursor.

Cursor removal is complicated by the existence of both Starbase and X cursors. These two types of cursors have significant differences. Starbase cursors can have multiple instantiations—one window can contain more than one Starbase cursor. In the X environment only one X cursor can exist on the screen. Starbase cursors also differ from X cursors in that Starbase cursors are clipped to the windows containing them. Starbase cursors cannot extend into the borders of their containing windows. The X cursor is a global entity in that it is never clipped and can extend through multiple windows and their borders. To ensure that cursor operations are consistent and predictable, all the cursors in a window have a stacking order, and no cursor can be moved or operated on unless all the cursors on top of it have been removed. The X cursor is always on top.

Because Starbase is allowed to gopen (open) a single window many times it is possible for an X window to have multiple Starbase cursors in it. A mechanism was added to the Starbase display drivers to maintain a list of active cursors for a particular window. This list, which is labeled "Echo List" in Fig. 4, is located in GRM shared memory. The list is traversed before the Starbase drivers do any rendering, and in the procedures associated with the XDI entry calls RemoveCursor() and ReplaceCursor(), each active cursor in the list is removed in the order it is found. When a Starbase cursor is activated, the Starbase driver adds it to the list, and when the Starbase cursor is deactivated or the program dies, the cursor is removed from the list.

The X display drivers also use functions associated with the XDI entry points RemoveCursors() and ReplaceCursors() to help the X server remove and replace the X and Starbase cursors before and after rendering operations. Unlike the routines used by the Starbase drivers, these routines accept flags to perform selective removal of Starbase cursors, the X cursor, or both. This is accomplished without the X server's having to know very much about the cursors' relative stacking order or other details. Once the X cursor is removed, it remains removed until the device is unlocked. The principal reason for not replacing the X cursor until the last minute is to avoid invalidating the current rendering state.

Removing cursors in the X server can be an expensive process, so care is taken to avoid unnecessary calls to the RemoveCursors() routine. The server keeps a flag for each window to indicate whether cursors have been removed. Since the cursor removal code in the display drivers only removes cursors from visible areas of a window, cursors must be removed before changing the clip list in those cases where the window situation is being modified (e.g., a window is being moved or iconified). The cursors are replaced using the new clip list, thereby drawing them into any newly exposed areas.

Cursor removal is further complicated by Starbase cursors in reserved planes. On the SRX and TurboSRX display systems the fourth overlay plane can be used to hold Starbase cursors. The fourth plane is used for cursors by writing the cursor color into the top eight entries of the color map. Whenever the fourth plane has a one in it, the cursor color will be displayed on the screen, allowing the cursor to be drawn in the overlays without destroying the color already there. Clearing the fourth plane restores the old color. Since these cursors need not be removed for normal rendering, the RemoveCursors() routine typically does not remove them. In some situations, such as changing the stacking order of windows, moving windows, and so on, these cursors must be removed because their associated windows may become fully or partially obscured. These situations are handled by catching them and passing the flag ALL_PLANES to the X display driver when calling RemoveCursors(). Of course use of the ALL_PLANES flag must be remembered so it can be passed to ReplaceCursors() when placing the cursors back on the screen.

Sharing Fonts

The fast alpha/font manager (FA/FM) system is a utility package that Starbase applications use to display raster text. This proprietary system was originally developed for the HP Windows/9000 and Starbase graphics environment. Being an early proprietary system, FA/FM could not take advantage of any of the work done by public domain systems such as X. New development for FA/FM, such as the creation of new fonts, had to be done by HP.

During the Starbase/X11 Merge design phase, the design team saw the opportunity to remove FA/FM's reliance on proprietary fonts and share the font files associated with the X Window System. The team set about designing a new font loading system that could be shared by both the X server and the FA/FM libraries. In addition, the FA/FM system was reengineered to render with X fonts. There were good reasons to design the new system. By removing FA/FM's reliance on proprietary fonts and allowing FA/FM to use the same font files as X, we anticipated that FA/FM would have a richer set of fonts to draw from. Whenever a new font was distributed for X, it could be used by FA/FM as well. X fonts are distributed in a format called Binary

Distribution Format, or BDF. This has become a de facto standard in the workstation marketplace. Font vendors typically make their fonts available in BDF format. BDF fonts are usually translated by workstation vendors into Server Natural Format (SNF) for efficiency in storage and loading.

We also saw the opportunity to conserve system resources. While the X server is running, offscreen memory and system RAM are heavily used. Therefore, it was decided that with proper design and engineering, we could create a system that allowed both FA/FM and X not only to share font files, but also to share the actual fonts in virtual memory and offscreen memory.

The core of the font sharing system is the font loader. Early in the design phase it was decided that the easiest way to share fonts was to write a single font loading system that could be used by the X server and the FA/FM library. This shared loader's responsibility is to read the font file from disk into shared memory, making the font available to requesting processes.

At the most basic level, the font loader is quite simple. When a request is made to load a font, the font loader does the following:

```
Locate and verify that the font file is a valid X font
If font is in shared memory
        establish pointers
        return
Else allocate the necessary virtual memory
Create a shared memory object in the GRM's
  shared memory space
Load the font's disk image into the shared
  memory
Establish pointers
Return
```

The GRM shared memory object created by the font loader is the data structure labeled "Font Object" in Fig. 4. As long as a particular font remains loaded, any further requests to load this font will result in the loader's finding the font in shared memory because the same code is used by Starbase applications and the X server to load fonts. This ensures that at no time will there be more than one copy of a font in memory.

There were some additional requirements that had to be met for this new technology to be acceptable.

- Object Code Compatibility. Even though the font files used by FA/FM were changing, we had to ensure that programs that used the old technology would still work.
- Relinked applications had to work. We had to ensure that relinking an application to use the new FA/FM font technology did not cause it to break, even though the application might contain absolute pathnames of fonts.

The first requirement meant that whatever was done for the reengineered FA/FM system, the fonts that were currently used by the old FA/FM system must remain where they were in the file system so that old object code references would still function.

The second requirement could have been met easily if not for the first requirement. For example, if an object module that contained a request to load the font /usr/lib/raster/6×8/lp.8U was relinked, it had to be able to find a font that was in the X font format from this pathname, even though the exact file named contained the original FA/FM font file. To get around this problem and to satisfy the first requirement, the directory structure used for FA/FM fonts was modified. It was decided that any directory that had old-style FA/FM font files in it would have a subdirectory named SNF. This SNF directory would contain analogs to the FA/FM font files, but in the X font format. Fig. 8 shows the old and new directories formats.

With this scheme, all of the old-format FA/FM font files can remain untouched, and the modified directory structure satisfies the first requirement.

To meet the second requirement, the method used by the font loader to find a font had to be expanded to accommodate the new directory structure. Instead of just accepting the pathname given it, it had to be able to search a little further. Thus, the first step of the loader process was expanded to:

```
Look at the name given
If valid font file, load it
else
        insert "/SNF" into path
        look for valid font in this path
        if found, load it
        else error
```

With the new font loader, fonts need to be loaded into memory only once no matter how many applications are using them. Backwards compatibility with the old FA/FM system is preserved. The X server and the FA/FM system now share the source code to accomplish font loading, thus ensuring compatibility and reducing maintenance requirements.

Sharing the Color Map

One of the recurring themes of the Starbase/X11 merge project was how to make X and Starbase share resources

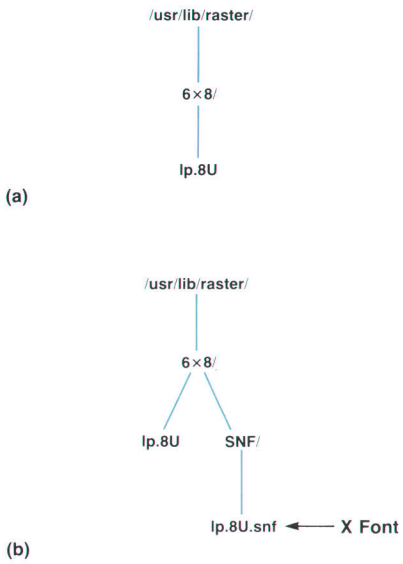


Fig. 8. *The old (a) and new (b) directory structures used to find and load FA/FM fonts.*

that they previously believed they each controlled exclusively. One of these resources that had to be arbitrated was the color map and display controls like display enables and blink control.

Notions of Color Map

The X concept of a color map was modified quite a bit from Version 10 to Version 11 of the X Window System. In Version 10 there was a single color map that every client allocated colors from. When all of the colors were used, the client simply made do with what it had or exited. In Version 11, the concept of a virtual color map was designed. Multiple color maps can be created regardless of how many color maps the hardware can support. In fact, every window can have a different color map. The color maps get installed by a window manager according to some policy usually established by the user. This way, every window can use the entire range of colors that a particular display has to offer. Fig. 9 illustrates the concept of virtual color maps.

Starbase, on the other hand, maintains the single color map notion. Starbase is designed to believe that it is always running to a raw display device and that it has complete control over the device. It also assumes that there is a single hardware color map and that it writes directly to it.

The Needs

The solution for the X server sharing the color map with a Starbase application was simple. Every time a Starbase application opens a window and requests that the color map be initialized, a new X color map is created for that window. In this way, Starbase applications that believe that they have complete control over the color map can run without modification. This solution easily takes care of the problem of how to emulate a single hardware color map with exclusive access for a Starbase application.

This solution does not answer the question of how Starbase applications can read from and write to the color map or how Starbase can share the color map with other X applications. The first option explored was to have Starbase use the standard X color map calls. There were a number of problems with this option. Starbase has a different notion than X does of how some color maps look. For example, in X it is possible to write only to the red bank of a particular color map entry. This is not true of Starbase. For example, for 24-bit displays, Starbase looks at the color map as a single 256-entry array of RGB values that can only be written as tuples. X views this same color map as three separate banks of color maps, representing the red, green, and blue banks of entries.

There was also the question of performance. Some Starbase applications use rapid alterations of the color map to achieve certain visual effects. Using the X color map mechanisms, the overhead of X server communication might prove to be a bottleneck for performance.

Finally, Starbase allows the manipulation of more than just the values of the color map. The shared memory version of the X color map includes additional attributes such as the display enable, color blinking, and color map blending. Also, information about transparent colors is included in overlay plane color maps. None of this information can be manipulated using the standard X color facilities.

The Solution

The design team agreed that Starbase's needs were beyond the capabilities provided by X and a new approach was needed. The approach finally agreed on was for each X color map to have an analog that the X and Starbase display drivers dealt with called a display state (see Fig. 10). These display states are created in shared memory every time a new X color map is created, and they can be manipulated by X or directly by Starbase clients. As information is written to the display state by a display driver, the display state is checked to see if it is installed in the hardware. If it is, then the hardware values and the display state are changed. If not, then only the software values in the display state are changed.

Since the display state is in the shared memory and is managed by the graphics resource manager, Starbase applications can now manipulate it. Now when a Starbase applications opens a window and requests initialization, the driver performs the following operations:

- Create an X color map (this operation creates a display

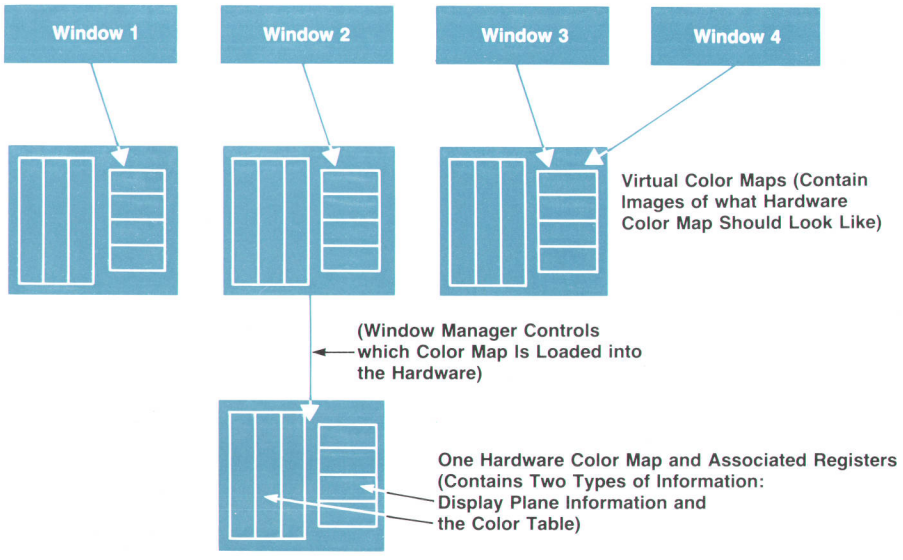


Fig. 9. *Virtual color maps.*

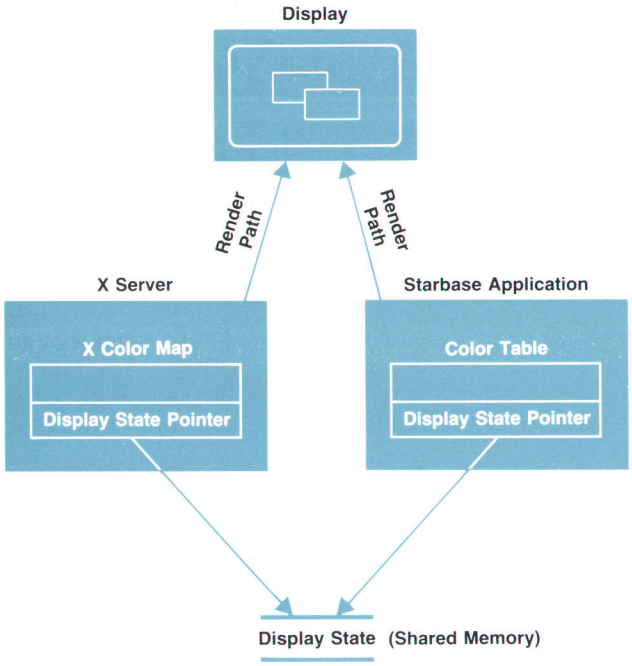


Fig. 10. *The architecture for sharing the color map in Starbase/X11.*

state in shared memory).

- Associate the color map with the X window.
- Establish pointers to the shared memory display state data structure.
- Initialize the display state.

Whenever a Starbase application makes changes to the display state, the Starbase driver does so directly, not using the X color map routines. In this way, it is not slowed by the overhead of the X server communication mechanism. And since the Starbase driver creates its own color map, it assumes that it can do anything with it that could normally be done with the hardware color map.

When a window is opened without the INIT* flag, Starbase asks the X server which color map is associated with the window. It then connects to the display state of that color map in shared memory. In this mode Starbase respects the restrictions placed on the color maps by the X server protocol. For example, Starbase will not change a color map cell if the X server has marked it as read-only. The X server also does not allow a Starbase program to change the display enable register. This allows a Starbase application to continue to use Starbase library calls to modify the color map, but still cooperate fully with other X clients.

Only one problem was left to resolve: how to communicate changes made by Starbase applications to the shared display state data structure to the X server? An X server extension called XHPSynchronizeColorRange was created to solve this problem. When a Starbase display driver alters the values in a display state, it then calls this extension routine. The X server then reads the current values of the display state and updates its notion of the color map's contents.

*INIT is a standard flag used with gopen that implies clearing of the open display planes and the initialization of the color map to Starbase default values.

Backing Store

The backing store is a piece of memory where the contents of a window are backed up in case the window gets destroyed or obscured by some user action, such as iconification or resizing, or by the action of another program. The X server supports backing store on a per-window basis. If an X client requests the server to maintain a backing store for an window, the server will do so, if possible.

Fig. 11 illustrates the use of backing store in the standard X environment. The contents of a window and its backing store are shown in different frames. Assume initially that window A is completely visible and has a picture of an arrow on the screen. At this stage its backing store is empty (frame 1). When window B is placed on top of window A, window A is obscured and the picture in the obscured region is damaged. If window A was created with a backing store, the server will intervene before the damage takes place. When the server realizes that the surface of window A is going to be encroached upon by some other window, the server saves the picture from window A to its backing store (frame 2). When window B is removed, the picture in the unobscured region of the window A has to be updated (frame 3). If window A has a backing store, the server copies the appropriate region from the backing store and recreates the picture in window A (frame 4).

If window A has no backing store, then the only way of updating the picture would be to send an expose event notice to the client owning window A. The expose event tells the client that a region or regions of its window have

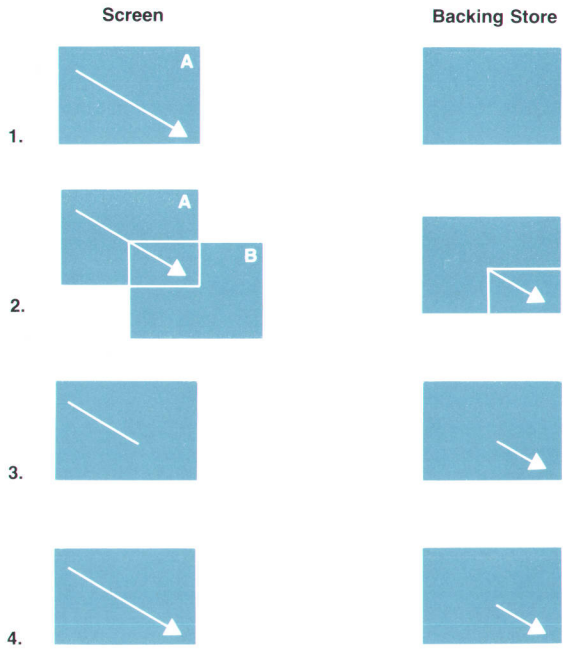


Fig. 11. *Views of a window and its backing store. In frame 1 window A is completely visible and backing store is empty, in frame 2 window A is partially obscured by window B. In frame 3 window A is unobscured and part of the screen picture is missing, and in frame 4 the missing part of the picture is copied from backing store to window A without intervention by the client.*

become exposed, that is, the picture contained in that region may have become inconsistent. If the client chooses to, it can update the picture by sending appropriate rendering instructions to the server. In many graphics applications, it ends up redrawing all objects in the window even though only a small part of the window may have been damaged.

For complex applications, redrawing the entire window is a time-consuming event. The standard X11 server does not guarantee that all implementations will support backing store. The burden of redrawing a window is left to the X clients. All X applications must be knowledgeable about expose events and must be able to deal with them.

The Starbase/X11 server, however, must provide backing store support and cannot depend on the clients' ability to deal with expose events, since the Starbase libraries and most Starbase applications have no notion of expose events and do not know how to handle them. A Starbase application running in an X window would be unable to refresh a window after the window became unobscured, so the X server must update it from the backing store. The X server not only must support backing store, it must also make the backing store a sharable object between its own display drivers and the Starbase display drivers. Therefore, the Starbase/X11 X server employs "smart" rendering functions to share the backing store between the X and Starbase applications.

HP support of the backing store capability in the X server dates back to the days of Version 10 of the X Window System. In the HP implementation of the X10 server this capability was called the retained raster facility.

The Starbase/X11 version of the X server operation of backing store was guided by two considerations:

- Operations involving backing store should be as fast as possible.
- In a window with backing store a pixel must never be rendered twice. If the pixel is in the visible portion of the window, it must be rendered on the screen; otherwise it will be rendered in the backing store.

Allocation Policy

The backing store of a window is always of the same size as the window it is backing up. The X server always tries to accommodate the backing store in the offscreen frame buffer. With the assistance of the display hardware, operations on backing store resident in the offscreen frame buffer are as fast as those on the screen. However, the frame buffer is a limited resource, and there will be occasions when there will not be enough space in the frame buffer for a backing store operation. When this happens, the X server will place the backing store in the virtual memory.

Direct hardware access windows (DHA windows) are shared between the X server and the Starbase application. If at any time in its life a DHA window is declared to be a backing store window, the X server will ask the graphics resource manager for a portion of offscreen memory large enough to fit the window. If none exists, the X server will ask the GRM for a portion of shared memory so that both the X server and the Starbase application can render to the shared backing store. However, like the frame buffer, shared memory is also a limited resource. Thus, there is no guarantee that sufficient space will be available in the shared memory at the time the allocation request is made to the GRM. If the GRM cannot provide the needed amount of shared memory, the server will declare the DHA window to have no backing store.

MOMA windows are never provided with backing store. MOMA windows employ transform engines in the hardware to accelerate their rendering performance. There is no way to take advantage of the hardware transform engines to render to the backing store if the latter is in virtual memory. Since we cannot guarantee that the backing store will be in offscreen memory, the X server does not support backing store for MOMA windows. Therefore, if a window with backing store becomes a MOMA window, the X server will dispose of its backing store.

Smart Driver Functions

The X server employs smart driver functions to render to its drawables. A drawable is a two-dimensional window or a pixmap that X and Starbase can draw on and treat as a single unit. These driver functions are called smart because they can distinguish between different types of drawables, such as windows without backing store, windows with backing store in frame buffer, windows with backing store in virtual memory, and pixmaps in virtual memory.

When a smart driver function is called to render to a window, the function can determine whether the window has a backing store. If the window has a backing store the driver can determine the location of the backing store, which can be in the frame buffer, virtual memory, or GRM shared memory. Further, the driver can figure out which parts of the backing store represent obscured regions of the window. With this knowledge, the smart functions render the necessary pixels either on the screen or in the backing store. It is never necessary to render to a pixel twice.

To make backing store sharable between X and a DHA Starbase client, the X server HP extension XHPRegisterWindow() is used to create the backing store object shown in Fig. 4. The following information is contained in this object:

- Drawable Type (drawable_type). An integer flag representing the backing store attributes of the window. The values indicate whether the window has backing store and whether it is located in the offscreen frame buffer memory, virtual memory, or GRM shared memory.
- Backing Store Stamp (bs_stamp). An integer counter that is incremented whenever the state of the window's backing store changes. This is a trigger to the client that it needs to obtain new backing store information from the shared memory object.
- Shared Memory Offset (sm_offset). A pointer to the start of backing store if it is located in shared memory. The value of this pointer is an offset relative to the beginning of the shared memory segment. The client must add its own shared memory base address to determine the true absolute address.
- Shared Memory Stride (sm_stride). An integer value representing the width of the shared memory backing store pixmap in bytes.
- Backing Store X Offset (bs_offset_x). An integer value representing the frame buffer x offset of backing store if it is in frame buffer offscreen memory.

- Backing Store Y Offset (bs_offset_y). An integer value representing the frame buffer y offset of backing store if it is in frame buffer offscreen memory.
- Backing Store Planes (bs_planes). An integer bit mask representing the display bit planes that are managed by backing store.
- Backing Store Pixel (bs_pixel). An integer representing the value to be placed in the bit planes not managed by backing store.

Deep Backing Store

Starbase supports 24-plane deep windows. Therefore, it was necessary to develop a method for the X server to support a 24-bit-per-pixel backing store. The main problem was determining how deep backing stores can be organized. In 24-plane-deep displays, the frame buffer is organized as three memory banks, each eight planes deep. The three banks are the red, green, and blue banks. As long as the backing store is placed in the frame buffer, there is no problem. The RGB components of each pixel are stored in the corresponding bank. There is a problem, however, when the backing store must be placed in virtual or shared memory.

In the X server, rendering to the virtual memory is done using the memory drivers leveraged from the Starbase library. There are two main components of the memory driver: the bit driver and the byte driver. The bit driver is used to draw on one-bit-per-pixel virtual memory pixmaps, and the byte-driver is used for one byte-per-pixel virtual memory pixmaps. In implementing the deep backing store we could have written a new memory driver for drawing to 24-bit-per-pixel virtual memory pixmaps or organized the deep backing store so that the existing memory drivers could be used without any modification. We chose the latter solution (see Fig. 12).

The organization of the deep virtual memory backing store mirrors that of the deep frame buffer. The deep virtual memory backing store is organized as three software banks, each one byte deep, corresponding to RGB banks in the hardware (see Fig. 12c). With this organization we are able to use the byte drivers without any change. However, for each drawing operation we call the byte driver three times—once for each software bank. This organization also simplifies the process of copying data from the virtual memory backing store to the screen because the data from a software bank is simply moved to the corresponding hardware bank.

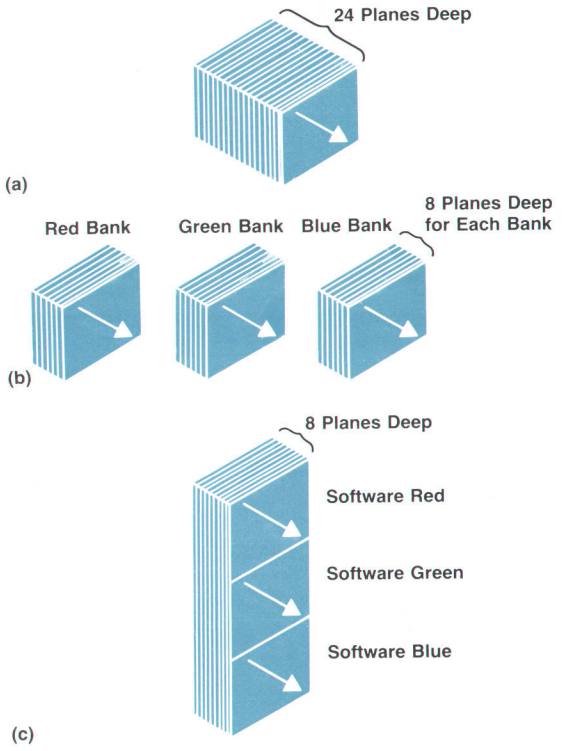


Fig. 12. *(a) A 24-plane-deep window on the screen. Of course the physical depth of display memory is not seen by the user. (b) 24-plane-deep backing store in offscreen frame buffer memory organized in three hardware banks of 8 planes each. The picture on the display is replicated on the three banks. (c) 24-plane-deep backing store in virtual memory. This is a contiguous piece of memory organized in three compartments. Each compartment is a software bank mirroring the hardware banks.*

Sharing Overlay and Image Planes in the Starbase/X11 Merge System

Developing a method to take full advantage of the capabilities of display memory was one of the challenges of the Starbase/X11 Merge project.

by Steven P. Hiebert, John J. Lang, and Keith A. Marchington

DEPENDING ON THE DISPLAY DEVICE, the X server allows users to configure a display in four fundamental display modes: image mode, overlay mode, stacked mode, and combined mode (see Fig. 1). The display mode determines how the hardware display memory is used. This article describes the rationale for the different display modes and how each of them works. The combined mode is discussed in greater detail than the others because it is the most sophisticated mode and it is available on the TurboSRX 3D graphics accelerator display system.

HP offers a wide variety of display hardware for its workstation products. This display hardware ranges from low-resolution monochrome displays to high-resolution displays with 16 million colors and 3D acceleration hardware. Using the full range of display capability in the display hardware was one of the challenges for the Starbase/X11 Merge design team.

One of the underlying philosophies of the X Window System is that it provides the tools to build different user interfaces, but it does not enforce any particular user interface standard. Thus X provides mechanisms, not policy. To maintain this philosophy, it was decided that the X server would provide the different display modes for the X Window System and allow the user to choose the display mode most appropriate for the application.

Overlay and Image Planes

All display systems for HP's workstations have at least one and as many as 24 planes of display memory. In addition, some of the more sophisticated display systems have additional display memory called overlay planes. The overlay planes are so named because they appear on top of, or over, the image planes. For example, if the overlay planes of a display are enabled and each pixel is set to black, then the image planes would not be visible. Areas of the overlay planes must be disabled or made transparent to view the image planes. Overlay planes can be set to a transparent color so that the image planes can be seen. Existing HP displays have from zero to four overlay planes.

The image planes are used primarily for rendering complex images and usually have more capabilities than overlay planes. For example, on the TurboSRX display system, the 3D accelerator can clip to an arbitrary set of rectangles in the image planes, but not in the overlay planes. Overlay planes have a number of uses, but primarily they are used to display information like text and menus. In this way

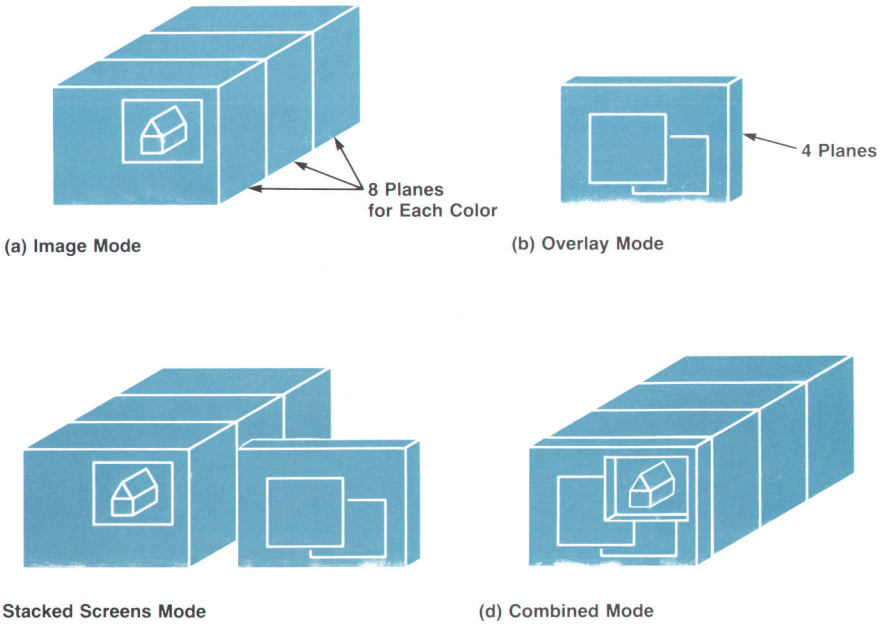


Fig. 1. *An illustration of the different display modes. (a) Image mode. All rendering by X is done only to the image planes of the display. (b) Overlay mode. All X rendering is done only to the overlay planes, and the image planes can be used by other applications. To see what is on the image planes the overlay planes would have to be made transparent. (c) Stacked screens mode. The overlay and image planes are treated as two separate screens. (d) Combined mode. Implemented primarily to support the display capabilities of TurboSRX, the combined mode uses the overlay and image planes as one screen.*

rendering in the image planes is not damaged by menus or text, and costly redraws of pictures in the image planes are prevented. With some 3D graphics or complicated 2D graphics, such redraws can take many minutes.

The overlay and image planes are located in the frame buffer and, as shown in Fig. 2, each plane is organized into on-screen and offscreen memory.

Image Mode

Every HP display system supports the image mode and all but the TurboSRX will default to image mode if the user does not specify a display mode. In the image mode, the X server performs all rendering only on the image planes available on the display device. If the display device has any overlay planes they are set to transparent in this mode. See Fig. 1a.

Overlay Mode

The overlay mode is almost identical to the image mode, except that the overlay planes of the display device are used by X rendering calls, and the image planes are free to be used by other applications such as Starbase graphics applications. A good example of this configuration is the HP 9000 Series 300 and 800 SRX (solids rendering acceleration) display system. The 3D acceleration hardware of the SRX is not capable of clipping to window boundaries, so it is not useful in a window environment. For the 3D acceleration hardware to be useful, it must have unobstructed access to the full, unobscured image planes. To run with this hardware configuration, a window-based application can provide all user interface components (e.g., windows and menus) in the overlay planes using the X display driver, and use the 3D accelerator for more complex rendering in the image planes. By creating a transparent window in the overlay plane, or by setting the window system's root window to transparent, the image planes can be made viewable. On the SRX display, this is the only way to use the 3D graphics accelerator and a window system such as X at the same time.

Stacked Screens Mode

In the stacked screens mode the overlay planes are used as one screen and the image planes as another (see Fig. 1c). In this way, the window system has twice as much screen "real estate." Stacked screens mode is literally the image mode and overlay mode running simultaneously. The screens are stacked one on top of the other with the visible screen being the one where the mouse cursor is located. To get from one screen to the other, the user simply moves the mouse off the edge of the current screen. The other screen is made visible as the mouse enters it. All of the normal capabilities of X are available in both the image and the overlay screens, and all of the restrictions of the image and overlay modes apply.

Stacked screens mode is particularly popular with software developers because it is possible to make twice as much information easily viewable. This means that a developer can have a debugger, terminal emulators, editors, code viewers and other applications all running at the same time and viewable.

Combined Mode

Image, overlay, and stacked screens modes were available in the X Window System before the Starbase/X11 Merge project. However, the Starbase/X11 Merge project's goal was to provide full-performance Starbase graphics in X windows wherever possible, and since the TurboSRX display, which is the successor to the SRX display system, has the hardware necessary to do accelerated graphics in windows, this meant that we needed to provide accelerated graphics in windows as well. This could have been done in image mode on the TurboSRX, but it would not have been as aesthetically pleasing.

The design team decided that a new approach was needed for the TurboSRX. This new approach is called the combined mode. The combined mode uses all of the planes of the display system (both image and overlay) as a single screen, making it look to the application as if there were simply one contiguous set of planes with a variety of different capabilities (see Fig. 1d). Using both the overlay and the image planes as a single screen is basically the opposite of how stacked mode works. In stacked mode the image and overlay planes are treated as two separate screens. With the combined mode the capabilities of the TurboSRX and X can work together.

TurboSRX Capabilities

Many of the capabilities provided by the HP 9000 Series 300 and 800 TurboSRX graphics subsystem are also provided by its predecessor, the SRX. These capabilities include:

- Image Planes. There can be 8 to 24 planes of image memory plugged into the display system. The system can be used as an eight-bit pseudocolor device (CMAP_NORMAL mode) offering 256 colors simultaneously or as a 24-bit color device (CMAP_FULL mode) offering over 16 million colors simultaneously.
- Overlay Planes. Each display system has three or four planes of memory that overlay (or are in front of) any other display memory. The original intention for these planes was to use them for floating text, cursors, or menus.
- Double Buffering. The image planes can be partitioned as pairs of banks in a variety of ways for double buffering. The most common configurations are to divide them into two eight-bit banks in CMAP_NORMAL mode and into two 12-bit banks in CMAP_FULL mode.
- Color Map Mode Hardware. The color map mode hardware enables the display system to run either in the CMAP_NORMAL mode or the CMAP_FULL mode. If 24 planes of image memory are plugged into the display system, in CMAP_NORMAL mode each pixel is interpreted by taking the eight-bit pixel value out of the low bank of display memory and using it as an index into a table of RGB (red, green, blue) values to determine what color a particular pixel on the display should be. In CMAP_FULL mode, each of the three eight-bit banks of display memory is read to determine which red, green, and blue value should be used on the display. By writing to a hardware mode register, these modes can be dynamically switched and different windows on the display screen can be dis-

played in different color map modes.

- 3D Graphics Hardware. Both systems have the ability to render complex 3D graphics, providing realistic images on the display. The front cover of this issue shows an example of the realistic images that can be produced using combined mode on a TurboSRX display system. The 3D images (car, engine, and gears) are located in the image plane, and the other items on the display are located in the overlay plane.

Capabilities available in the TurboSRX but not in the SRX include:

- Hardware Cursor. Two planes of memory (in addition to the overlay and image planes) are available for the display of cursors. This feature allows a hardware cursor to be placed on the display without disturbing the contents of any of the image or overlay planes beneath it. The hardware cursor also offers the advantage of not having to remove the cursor to render, since it resides in its own plane of display memory. Not removing the cursor before rendering provides better performance for rendering routines.
- MOMA Window Support. From the perspective of the Starbase/X11 Merge system, this is probably the most significant feature on TurboSRX. MOMA (multiple, obscurable, movable, accelerated) window support allows the TurboSRX accelerated graphics capabilities to be used in a windowed environment by providing special clipping hardware. This clipping hardware allows the TurboSRX graphics accelerator to render only to the exposed rectangles of a window. The TurboSRX hardware has support for a maximum of 32 clipping rectangles for MOMA windows, which is an adequate number for most window systems, but a small number for the X Window System.

With these TurboSRX features in mind, the design team focused on designing the Starbase/X11 Merge system to take full advantage of the hardware capabilities of the TurboSRX. This resulted in the following design goals for the combined display mode:

- Provide support for MOMA windows that would allow Starbase applications to use the 3D graphics accelerator in X windows.
- Support eight-bit and 24-bit color modes. Make 8-bit pseudocolor and 24-bit color with double buffering available to applications.
- Maintain the visual aesthetics of the system. When possible, minimize the damage that different hardware modes and different color maps cause to the appearance of the display when they are displayed simultaneously.
- Provide a state-of-the-art X server implementation. Reconcile the capabilities of the X Window System, Version 11 with the capabilities of TurboSRX.

The Architecture

With X11, a number of new concepts were introduced to increase the capabilities of X such that it could be run on the entire range of today's display hardware as well as any future display hardware that might be developed. The concept in X11 that is most important to the combined mode is called the "visual." The visual is the mechanism X uses to describe the capabilities of a particular display's hardware. The visual structure includes:

- Class. The class describes how a color is mapped from memory to the display. There are two major classes, static and dynamic, and subclasses of each. The subclasses include gray, mapped color, and decomposed color. Static and dynamic classes are defined at X server start-up time. Static classes cannot be changed by application programs, but dynamic classes are definable and changeable in the application program. The gray subclass means that all the colors in the color map are shades of gray. For the mapped color subclass, one-byte pixel values from the frame buffer are used to index into a color map of RGB tuples which describe the color to be displayed (see Fig. 3a). For the decomposed color subclass, a three-byte pixel value is used to get the color value from the color map. The first byte is used for red, the second byte for green, and the last byte for blue (see Fig. 3b). The mapped color subclass allows up to 256 colors and the decomposed subclass allows up to 16 million colors. Each entry in the color map table represents a color intensity (shade). For instance, the value 10 might represent dim RGB intensities and 220 would represent bright RGB intensities. These red, green, and blue intensities are mixed together to produce the displayed color. Putting these attributes of color maps together (class and subclass) allows the device to support up to six types of color maps. Table I shows the X color map types.

Table I
X Color Map Types

Subclass	Static	Dynamic
Gray	StaticGray	GrayScale
Mapped Color	StaticColor	PseudoColor
Decomposed Color	TrueColor	DirectColor

- Color map entries. The number of different color map entries available for use by client applications.
- Bits of RGB information. How many bits of resolution are available to describe red, green, and blue color values.

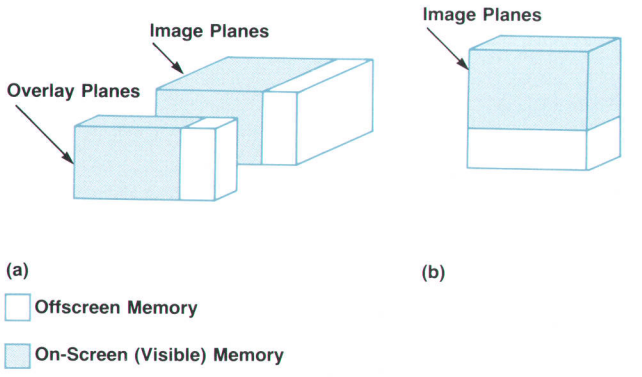


Fig. 2. *The organization of the image planes in the frame buffer. (a) A display system containing both image and overlay planes (e.g., the HP 98550A Color Graphics Board). (b) A display system with only image planes in the frame buffer (e.g., the HP 98547A).*

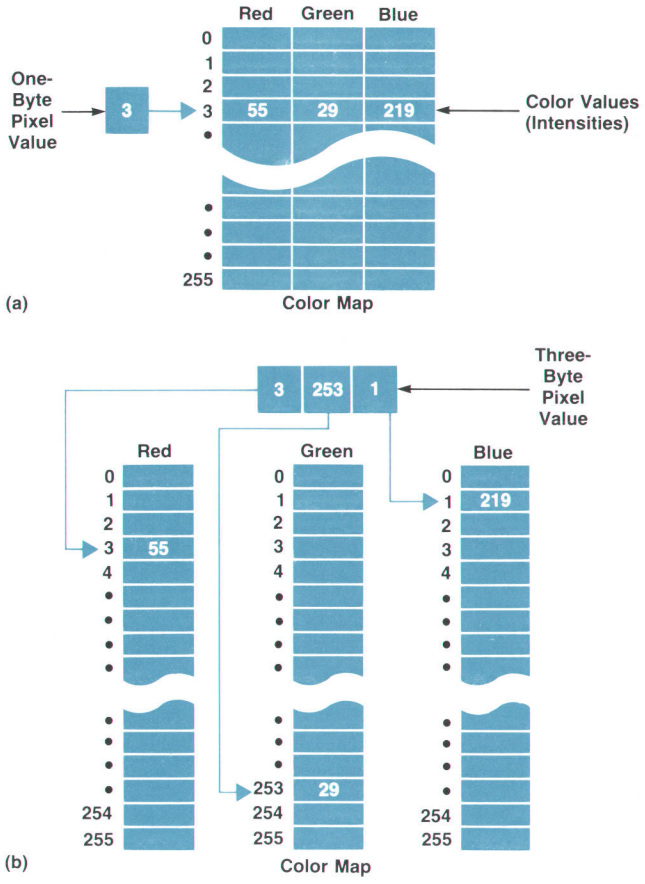


Fig. 3. *(a) Mapped color subclass. A one-byte pixel value is used to index into the color map to obtain the RGB tuple. (b) Decomposed color subclass. A three-byte pixel value contains an index for each primary color list in the color map.*

- Planes. The number of planes of display memory available on the display device.

X11 makes it possible to have more than one of these visuals available on a given screen at the same time. With multiple visuals, it is possible to create a mode that incorporates the capabilities of both the image and the overlay planes of the TurboSRX so that the full range of the display's capabilities are available to applications. We decided to treat the image and overlay planes as a single screen with the overlay planes represented by one three-or-four-plane PseudoColor visual type. The number of planes is dependent on how the user sets up the device file for them. The image planes, with their CMAP_NORMAL and CMAP_FULL modes, are allowed to have either an eight-bit PseudoColor visual type, a 24-bit DirectColor visual type, or both simultaneously. Another option allows an eight-bit double-buffered PseudoColor visual type for image planes and a 12-bit double-buffered DirectColor visual type for image planes.

In combined mode, the root window for the screen always resides in the overlay planes, and the overlay plane visual is the default visual for the screen. Any client that simply asks for a window to be created with the default visual of the screen ends up residing in the overlay planes. For an application to create a window in the image planes, it has to request the visual information from the server and specifically request the desired visual type.

The color map modes CMAP_NORMAL and CMAP_FULL in the image planes are handled through virtual color maps. Virtual color maps are an image of what the window or client thinks the hardware color map looks like. As was described in the article on shared display resources on page 20, each color map in the Starbase/X11 Merge system has an analog called a display state, which is used by the display drivers. Each display state contains the current color values for a device's color map, some device-specific information about which planes of the display are enabled, and in the case of the TurboSRX, the color map mode of the hardware. X provides a way for a program to control which color map is currently loaded into the hardware (this is called validating the color map). Usually a special X client, such as a window manager, is the only program that changes which color map is loaded (validated). The window manager may have several methods for the user to specify which color map is loaded. Therefore, when the color map for an eight-bit PseudoCcolor window is installed in the image planes, the hardware will be switched to CMAP_NORMAL mode, and when the color map for a 24-bit DirectColor window is installed, the hardware will be switched to CMAP_FULL mode. Fig. 9 on page 29 illustrates the virtual color map concept.

The result of this approach is that most windows are created in the overlay planes. Most X server clients such as window managers and terminal emulators use the default visual. Applications that request visual types that are in the image planes can change the color map in the image planes and use one of the color map modes without affecting the visual appearance of the windows in the overlay planes. Most of this color map control was provided for Starbase applications because they usually assume that they can change the color map at will. As a result a Starbase application creates its own virtual memory color map for a window that it opens.

This design allows the TurboSRX to be used in windows and satisfies all of the design goals for the TurboSRX display driver in the X server. With most of the windows in the overlay planes, their clipping regions do not have to be included in the hardware clip list for the accelerator. This helps us live with the 32-clip-rectangle restriction of the TurboSRX and achieve the full performance of a Starbase application running in X.

Having most of the commonly used windows in the overlay planes allows combined mode to maintain visual aesthetics at the highest possible level, while still allowing both eight-bit and 24-bit windows in the image planes. As a counterexample, take the case of image mode. If image mode were to allow both eight-bit and 24-bit windows simultaneously, one of those two visual types would have to be the default. If an application created a window of a visual type other than the default and its display state were installed, it would change the hardware color map mode and all of the windows of the default visual type would become incorrect in appearance. In fact, the windows, including the root window, would become completely indecipherable. However, with combined mode, when the hardware color map changes, the windows in the overlay plane (where most applications reside) remain visually

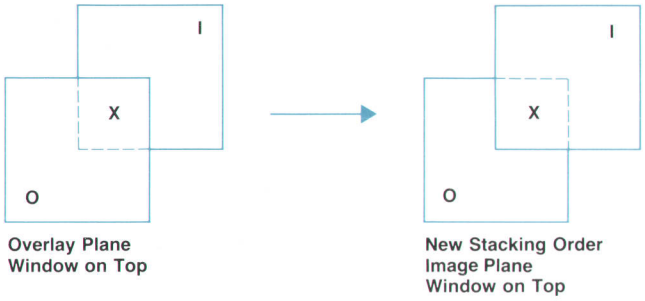


Fig. 4. *When the stacking order is changed, area X becomes the newly exposed area in the image plane. In the original clipping algorithm, besides causing area X to be painted transparent in the overlay plane, the image plane is considered to be damaged, causing the image plane to be cleared to the background color and an exposure event sent to the client owning window I. The client would then rerender area X in the image plane.*

correct and only image plane windows become visually incorrect.

This design also provides a very straightforward view for an X application. A client application can simply connect to the server and request windows of the default type and get windows in the overlay planes. Or, using the XGetVisualInfo routine, the client application can interrogate the server for all of its visuals or a particular visual it is interested in. The application never worries whether it is in overlay planes or image planes. The server automatically places the window in the appropriate planes without intervention by the application.

Implementation

The architecture described above fits very neatly into the the X model, and for the most part, the implementation of combined mode was straightforward. But there were some challenges in the implementation that resulted in some interesting solutions. The two most challenging areas were how to allow the user to see through the overlay plane to the image plane windows and how to clip windows and generate exposures for only those areas of windows that were actually damaged by other windows. A window that needs exposure is one that is covered up and needs to be seen. To see a window that resides in the image planes, the overlay planes must be made transparent. At first, creating this transparent hole seemed like a difficult task, but as it turned out, the X server architecture allowed this to be handled quite easily. Whenever an area of a window is exposed, the server is required to paint the window's background. At this point, the X server determines if the window being painted is in the image planes, and if it is, simply makes the same area of the overlay planes transparent. In this way all visible regions of the image plane window have a corresponding area in the overlay planes painted a transparent color.

Combined Mode Clipping

To solve the problem of clipping windows and generating exposures for damaged windows, and to make full use of the capabilities of the TurboSRX hardware, the clipping algorithm used in the X server had to be modified. In the original X server, the clipping algorithm made no distinction between overlay and image planes when computing clip lists for windows. Lacking this distinction, creating a window in the overlay planes would cause the server to conclude that any windows in the image planes obscured by the overlay plane window were damaged. When the overlay plane window was moved or destroyed, newly exposed areas of the image plane window would be cleared to the window's background color and an exposure event would be sent to the client owning the image plane window. The exposure event tells the client that it must rerender to the image plane (see Fig. 4). The modification of the clipping algorithm allows windows in the overlay planes to be created and destroyed without affecting windows in the image planes.

For both clipping algorithms, new clip lists are computed whenever an action is taken that could change the clip list (e.g., changing the stacking order of the windows on the screen). The function xosValidateTree() is used to compute the new clip lists. xosValidateTree() adds the visible portions of any children of the parent window to be reclipped back into the parent window's clip list and then, passing the parent's clip list as the working universe, calls the routine xosComputeClips() to let each of the parent window's children, and the children's children, and so on recompute their clip lists. The working universe includes the visible areas of the parent window. Upon return from xosComputeClips(), the working universe is the parent's new clip list. By subtracting the old clip list from the new clip list the parent can compute which areas have been newly exposed. That is, any area in the new clip list that is not in the old clip list must be newly exposed.

The modification of the clipping algorithm to support combined mode consists mainly of computing two clip lists for all the windows on the screen. One set of clip lists, which we can call the old-style clip lists, is generated based on the unmodified clipping algorithm described above (i.e., these clip lists contain windows from both the image and the overlay planes). The second set of clip lists is computed taking only the image plane windows into account (image-only clip list). Within the X server, image plane windows use the image only clip list as the default clip list, and the overlay planes use the old-style clip list as the default. Both image and overlay plane windows use the old-style clip list for cursor removal. Since either type of window can have children or subwindows of the other type, windows on both planes must keep the image-only and old-style clip lists available.

In the new combined mode algorithm, rendering to the image plane is done only when there are changes to the windows in that plane and not because of changes to windows in the overlay plane. The image-only clip list is used to handle rendering to image plane windows. The old-style clip list is used to determine which areas of the overlay plane windows must be painted transparent to expose windows in the image plane.

Combined mode clipping allows rendering to an image plane window while it is obscured by an overlay plane window. Since the root window is always in the overlay planes, rendering can even take place to an image plane window that is iconified. The server must take care, however, to avoid rendering to areas of iconified image plane windows used by other image plane windows. Image windows that are not iconified are automatically removed from the allowable rendering area by the old clipping method. Extra programming was required to remove iconified image plane windows from the allowable rendering areas of other iconified image plane windows. That is, if two iconified image plane windows overlap, neither may render to the overlapping area. When one or the other of the iconified windows is mapped, it will get an exposure event for that overlapping area.

Conclusion

Combined mode is a solution to the complex problem of how to support a high-end display system in the best possible way. Combined mode offers some capabilities that allow the TurboSRX display system to work at its full potential in an X environment. With the addition of combined mode, the X server now offers four different display modes of operation to take full advantage of the broad range of display hardware for HP workstations.

Sharing Input Devices in the Starbase/X11 Merge System

To provide support for the full set of HP input devices and to provide access to these devices for Starbase applications running in the X environment, extensions were added to the X core input devices: the keyboard and the pointer.

by Ian A. Elliott and George M. Sachs

STANDARD X SERVERS SUPPORT two input devices: the pointer (mouse, tablet, light pen, etc.) and the keyboard. These devices are known as the core input devices. The X server sends information from the input devices to client programs in packets called "events." The keyboard generates key events, while the pointer generates button or motion events. These events contain information that includes the absolute location in two dimensions where the event occurred, the location relative to the X window in which the event occurred, and a timestamp. For key and button events, there is also a field that tells which key or button was pressed.

In a typical X environment, multiple application programs called clients run simultaneously. Each has its own window or set of windows and all share the core input devices. The X server arbitrates which client gets a particular input event by determining which window has the "input focus." The focus window, which is the window that is allowed to receive input from input devices, is normally either the smallest window that contains the pointer, or is an arbitrary window explicitly established as the focus window by a protocol request made by a client program.

We faced two major problems in the area of input device support for Starbase/X11 Merge: how to provide the ability to use the full set of Hewlett-Packard input devices in an X environment, and how to access those devices through Starbase in that environment. The first problem arose because there is currently no X standard for using other input devices in addition to the core devices. If additional devices were supported, there is no provision within the defined core events for determining which device generated the event. There is also no provision in the existing events for reporting data of more than two dimensions, or motion data whose resolution is different from that of the screen. The problem with Starbase was that prior to this project, Starbase did not provide a way for multiple programs to share input devices. The only input devices that could be shared were those for which a window system arbitrated the sharing and allowed Starbase input. These devices included the HP Windows/9000 locator and the X Version 10 pointer and keyboard.

To overcome these problems the goals established to provide sharing of input devices in the Starbase/X11 Merge system included:

- Support a wider range of input devices including the core devices, and ensure that all the devices supported have the same functionality as that provided by the core devices.
- Support all input devices that follow the HP-HIL (Hewlett-Packard Human Interface Link) specification[1] and

X Input Protocol and X Input Extensions

The core protocol of the X Window System provides a standard syntax for making requests to the X servers. The syntax describes the sequence of bytes that make up each of the protocol requests. For example, the XSetInputFocus request, which allows a client to choose which window should receive input from the keyboard, has the following format:

Length (bytes)	Value	Meaning
1	42	XSetInputFocus Request ID
1	0, 1, or 2	Revert-to-Window Parameter
2	3	Request Length (in four-byte words)
4	0, 1, or a Window ID	Focus-Window Parameter
4	Timestamp Information	Focus-Time Parameter

The information in a protocol request like the one above tells what request is being made (XSetInputFocus), the length of the request (three four-byte words, or 12 bytes), and the values of any parameters the request has. The parameters in the request specify which window should receive input from the keyboard (the Focus-Window parameter), which window should receive input if the focus window disappears (Revert-to-Window parameter), and when the XSetInputFocus request should take effect (Focus-Time parameter). The 0, 1 and 2 values in the parameters are special constants that indicate no window, whichever window contains the X pointer, and whichever window was named as the parent of the focus window, respectively.

X was designed to allow individual vendors such as Hewlett-Packard to extend the protocol by defining new requests that can be interpreted by X servers in the same way as standard X requests. For example, the HP input extension provides a protocol request named XHPSetDeviceFocus. This request allows a client program to choose which window should receive input from some input device other than the keyboard or mouse. The request has the following format:

Length (bytes)	Value	Meaning
1	128 ≤ Number ≤ 255	ID of HP Input Extension
1	8	XHPSetDeviceFocus Request ID
2	5	Request Length (in four-byte words)
4	0, 1, or a Window ID	Focus-Window Parameter
4	Device Identifier	Focus-Device Parameter
4	Timestamp Information	Focus-Time Parameter
1	0, 1, or 2	Revert-to-Window Parameter
3	Unused Bytes	

The request begins with a number that identifies the extension that implements the request and distinguishes the request from core protocol requests. The next byte identifies the request within the extension. The length, Focus-Window, Focus-Time, and Revert-to-Window parameters serve the same purpose as they do for the XSetInputFocus request described above. The Focus-Device parameter identifies the input device for which the client program making the request wishes to control the destination of the input.

are supported by the HP-UX operating system.

- Allow the choice of the core devices to be easily configured and provide reasonable defaults if no choice is made.

For Starbase applications the following additional goals were established:

- Provide full functionality for Starbase applications using input devices in an X window.
- Ensure that the design does not require source code changes in the Starbase application, except for the possible exception of the call to the gopen function which is used to open an input device.
- Allow multiple programs to access and share the same input devices simultaneously.

HP-HIL Input Devices

HP-HIL input devices are grouped into three general categories by the Starbase/X11 server. First, there are keyboards and keyboard-like devices such as all of the different HP language keyboards, the HP 92916A Bar Code Reader, and the HP 46086A 32-Button Box programmable function keys. These devices either generate keycode data, or as in the case of the barcode reader, generate USASCII data which can be translated to keycodes. The second group of input devices are those that generate absolute positional data as well as button information. These include graphics tablets and touchscreens. The existing devices of this type report absolute positions for two axes, and may report zero, one, three, or four buttons. The third group of input devices are those that generate relative motion data. These include two-button and three-button HP-HIL mice such as the HP 46095A 3-Button (quadrature) Mouse, the M1309A Trackball, the HP 46085A Control Dial Module (nine-knob box), and the HP 46083A Knob (one-knob box). The existing devices of this type may report two or three axes of motion and report zero, two, or three buttons.

There are a few HP-HIL devices that are not easily categorized. For example, the HP 46084A ID Module, which is used to prevent unauthorized software duplication, does not generate any input, but occupies a position on the HP-HIL. It currently cannot be accessed through the X server. A client program can access it directly, but not across a network. Audio extension modules, such as the HP 46082A, do not occupy a position on the HP-HIL, but X functions exist to access the beeper contained in the module.

Core Input Devices

Up to seven input devices can be attached to one HP-HIL. There is no standard definition in X for determining which of those devices should be used as the pointer or the keyboard. In Starbase/X11 Merge, explicit specification of the core devices is done through a configuration file. The name of the configuration file is constructed using the display number specified by the user when X is invoked. Because that number is under the control of the user, mul-

tiple configuration files with different names can exist and can be used to specify different input devices as the core devices. When a device is chosen, it can be specified either by giving the name of its device file and its intended use, or by giving an ordinal position (first, second, etc.) and the type of device, along with its intended use. The position of the device is relative to other input devices of the same type on the HP-HIL, with the first device being the one closest to the computer. For example, a graphics tablet can be specified as the pointer device with a line in the configuration file of the form /dev/hil2 pointer, or with a line of the form first tablet pointer.

It is possible to specify explicitly that the server operate with no pointer device or no keyboard device, or both. In addition, the keyboard can be specified as the keyboard device and the pointer device. This feature is provided for working environments where it is not desirable to have a separate pointer device. If a keyboard is used as the pointer device, the user can specify in the X server configuration file which keys cause the pointer to move and the magnitude of movement. These keys are taken over by X and are not available for use by client programs. To prevent conflicts in the use of these keys between X and client programs, it is possible to specify that the keys should be used for pointer movement only if a specified set of the modifier keys (e.g., left **Shift**, right **Shift**, **CTRL**, left **Extend char** and right **Extend char**) are pressed at the same time. The user can also specify which keys should be interpreted as buttons for the pointer device.

Default choices for the core devices reflect the devices most commonly used as the default keyboard or pointer device. For example, if a keyboard is attached to the HP-HIL and can be opened by the X server, it is used as the keyboard device. If more than one keyboard is attached, the last one, that is, the one most distant from the computer on the HP-HIL, is used. If no keyboard can be opened by the server, the last key device, such as a barcode reader or 32-button module, is used. For the default core pointer device, if an HP-HIL mouse is attached to the HP-HIL, it is used as the pointer device. If no mouse can be opened by the server, the last device on the HP-HIL that can generate motion data is used. If no such device can be found, the keyboard is used as the pointer device. If the motion device chosen is one that can report more than two axes of motion, axes beyond the first two are ignored.

Some additional functionality was provided for HP 9000 Series 800 Computers. These machines are capable of supporting up to four HP-HIL loops, each of which can be associated with a set of input devices. Our goal for these machines was to provide maximum flexibility in specifying input devices while still providing reasonable defaults if no specification is made. The method chosen provides a default based on the display number specified when X is invoked. This display number is used to determine which configuration files are used in initializing the server.

The user can specify an HP-UX path to be searched for all input devices or the path to be used for an individual input device. This functionality was implemented to allow the HP-HIL path to be explicitly chosen on HP 9000 Series 800 computers. However, it also proved useful during project testing. A test tool that was written to simulate HP-HIL driver input used this feature to simulate input from various input devices. The result was greater flexibility in testing various combinations of hardware. See the article on page 42 for more information about project testing.

HP Input Extensions

Although the core protocol of the X Window System is standard across all vendors, X was also designed to allow individual vendors to implement extensions to that protocol. This allows vendors to add functions that are specific to their hardware or software requirements, or that are not included in the core protocol. If these extensions are found to be useful for the general X community, a procedure exists to propose them as standards to be included in future releases of X.

This was the method chosen to add support for HP-HIL devices within the X server. It provided a solution that met the needs of X clients, while also providing Starbase drivers with information from input devices that could not be reported through the core X protocol. See the box on page 39 for an example of X protocol and X extension format.

There are two parts to most X extensions: library functions to invoke the protocol requests it defines, and a server portion to process the requests and implement the functions. The X protocol defines the format of requests in the X library. An input X extension is more complicated than other X extensions because it also involves the creation of new input events, code to generate the events within the server, a means to allow clients to ask to receive those events, and code to route the events to the appropriate clients. Unlike many extensions, input X extensions require additions to both the device independent and device dependent portions of the server.

To provide functionality equivalent to that provided for the core devices, it was necessary to implement protocol requests that are analogous to core protocol requests and also allow the user to specify which device should be manipulated. These functions include the ability to select input events from a device, control the focus of that device, and "grab" (temporarily take exclusive control of) a device.

Other necessary functions include those that allow a client to list all the input devices available to the X server, and functions to enable and disable those devices. Also, input events for this extension were defined so that more than two dimensions of motion data could be reported.

Technical Issues and Trade-offs

The major input extension implementation issue we encountered was how to treat input devices other than the pointer that report motion data. The position of a typical pointer, such as a mouse, is tracked by the server and a cursor is echoed at that position on the display by the server. A keyboard takes its position from the pointer, and its focus is either explicitly set or is determined by the position of the pointer.* It was obvious that additional key devices should be treated like the keyboard, but it was not obvious how additional motion devices should be treated.

The alternatives were either to treat all devices supported through the extension like the keyboard or to treat addi-

*When a keyboard key is pressed, one of the parameters returned to the application is a pointer (cursor) position.

tional motion devices like the pointer. If they were all treated like the pointer, the server would have to track their position and echo a cursor for them, and not allow their focus to be explicitly set by the client. If they were absolute devices, their input would have to be scaled to the screen. If instead they were treated like the keyboard, the server would not have to track their position individually but would take it from the position of the pointer. The server would not echo a cursor for them, but would leave that up to clients and allow their focus to be explicitly set. To give clients maximum flexibility, it was decided to treat all devices supported through the extension like the keyboard.

Input Devices and Starbase

The Starbase library provides functions to open input devices and to receive two- or three-dimensional world-coordinate input. Several device drivers exist that allow Starbase to receive input from different devices or from the same device in different environments. In some of these environments, access to input devices has been exclusive, allowing only one program at a time to open and access a device. Shared devices for Starbase applications have been supported under previous HP window systems, but only for a pointer and a keyboard. Therefore, the major Starbase contribution to this project has been providing the ability for multiple programs to share all input devices.

At first it was not known how to achieve the desired device sharing functionality. However, once it was determined that an input extension would be provided, the basic approach was to provide device driver code that uses either core or X extension Xlib calls to obtain input from the requested devices. In this manner, the X server provides shared access to all devices for both Starbase and X clients (see Fig. 1). The X server arbitrates the sharing of input devices between programs, and applies normal focus rules to Starbase and X programs. The new device driver code is similar to the existing Starbase HP-HIL driver code, differing only in how it obtains input from a device.

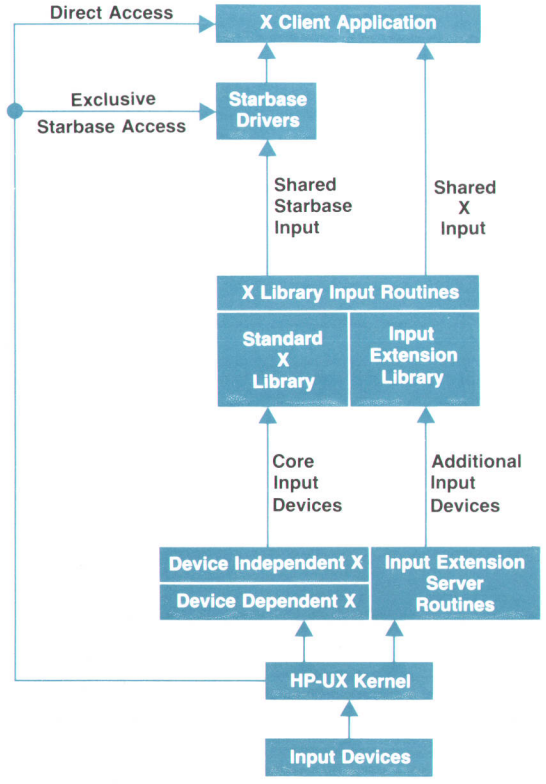


Fig. 1. *Input data flow in the Starbase/X11 Merge X server.*

The syntax of the gopen request, which describes the input device to be opened, was enhanced to allow the specification of an input device and window combination. This allows the driver to make a request in the form expected by the X server to open that device and request input from it. Since many Starbase programs specify this information through HP-UX environment variables or program parameters, they can take advantage of the enhanced syntax without changing the source code of the program.

It was possible to access the core input devices through Starbase input requests in previous releases of X, and compatibility has been maintained so that client programs can continue to access these devices as before. However, in previous releases of X, except for the keyboard and pointer, it was not possible to access input devices in a manner that would allow them to be shared among programs. Also, it was not possible to access them across a network. As a result of this project, programs can take full advantage of the window system and network, while continuing to use additional devices and access them for Starbase input.

Direct Access to Input Devices

Client programs can open and access input devices directly that are not in use by the X server. This allows a program that was not written for a windowed environment to continue to work. However, only one instance of that program can be run at a time, thus preventing other X clients from using that device. Although a good feature for existing programs that do not require a windowed environment, direct accessing of input devices is not a recommended practice for any newly written or ported programs.

The core pointer and keyboard devices cannot be directly accessed by client programs, since the X server opens those devices.

Conclusion

The result of this project is that existing applications are supported, and an easy transition to a windowed environment is provided for them. As shown by Fig. 1, programs have a number of optional ways to access the input devices. Exclusive access to input devices other than the core devices is supported, although not recommended for new clients. Shared access through X libraries is supported for both core and extension input devices. Shared access through Starbase input routines is supported for both core and extension input devices, and is provided in a way that minimizes changes to existing Starbase programs.

Acknowledgments

Mike Stroyan created the original Starbase input tracking mechanism and helped in developing the Merge input model.

Reference

1. R.S. Starr, "The Hewlett-Packard Human Interface Link," *Hewlett-Packard Journal*, Vol. 38, no. 6, June 1987, pp. 8-12.

Sharing Testing Responsibilities in the Starbase/X11 Merge System

The testing process for the Starbase/X11 Merge software involved setting realizable quality goals, and using extensive test suites and test tools to measure and automate the process.

by John M. Brown and Thomas J. Gilg

WITH THE DEVELOPMENT OF the Starbase/X11 Merge environment, new forms of testing had to be considered. Before the Starbase/X11 Merge project, the X test suites consisted of nearly 450 tests, and the Starbase test suite contained nearly 400 tests run across an average of 40 hardware configurations. The challenge was to make the appropriate modifications to this extensive set of tests to make them useful in the Starbase/X11 Merge environment. In areas where the existing test suites were inadequate, new test tools and tests had to be developed.

Test and Quality Goals

The combination of existing and new test suites needed to ensure adequate code coverage. Adequate code coverage in this context means exercising all procedural interfaces (i.e., X and Starbase library calls), and the in-depth testing of each procedure. An HP software tool known as the branch flow analyzer (BFA) was used to measure code coverage. Code quality was measured in terms of defect densities and defect arrival rates. The project quality goals were stated in terms of acceptable defect densities (defects per 10 KNCSS*) for each class of defect severity. Furthermore, defect arrival rates (defects per 1,000 test hours) were closely monitored throughout the project, and objectives were set to achieve specific diminishing arrival rates at project checkpoints.

Strategy

Existing test technologies for X and Starbase were reviewed for their suitability in testing the Starbase/X11 Merge system. In several cases, the existing technologies and their related test suites required no modifications. In other cases, weaknesses were identified and an effort was undertaken to enhance the remaining test tools and test suites. With nearly 850 pre-Starbase/X11 Merge tests and several hundred megabytes worth of time-proven archives, the value of such an undertaking was obvious. Two test strategies were undertaken. First, new tests were developed that could be directly incorporated into the existing test suites. Second, for all the test scenarios not covered, new test tools and tests were developed.

In all cases, a high priority was placed on the automation of tests. A best-case scenario was envisioned in which all the code changes, deletions, and additions developed in one day would be tested overnight on all available resources, and a summary of the test results would be generated automatically for inspection by the engineers the following day. In addition to the testing effort, code reviews helped round out the quality assurance effort. A code review or code walkthrough was conducted for each new code module. Attendance included the code author, a moderator or code reader, and several reviewing engineers.

*Thousands of noncomment source statements.

Testing Measures

To help guide the testing effort, several test and quality metrics were identified and used. These metrics include:

- Branch Flow Analyzer (BFA) Coverage. The branch flow analyzer provides a measure of how well all the code in the software under test is exercised (covered) during the testing effort. To use the BFA, the source file to be tested is run through a BFA preprocessor which places counters at all conditional statements and at the beginning of all procedures (see Fig 1a). The source file produced by the preprocessor is then compiled in a standard manner. When the program is run, the counters embedded in the code update an external disk-based data base, which can later be analyzed. Analysis of the BFA data base provides a summary of which procedures are called and a breakdown for each procedure is given showing which conditional paths were executed, or more important, missed (see Fig. 1b). The BFA tool identified unexercised sections of code to be targeted when writing new tests.
- Defect Density. To measure the current product quality, the defect density described the expected number of severity weighted defects (critical, serious, low) per 10 KNCSS.
- Defect Arrival Rate. As a way to sense trends in quality, the defect arrival rate described the number of defects found per 1000 hours of testing.
- Continuous Hours of Operation. A continuous hours of operation test was frequently executed to give an indication of X server robustness, and to reveal any long-term execution side effects (e.g., memory utilization growth).

Engineer Test Suites

The end users for the Starbase/X11 Merge product are software engineers who develop high-performance graphics applications running in windowed environments. With

```
extern bfarecord ();
extern bfareport ();

main (argc, argv)
int argc;
char *argv[];

        bfarecord ("main',1);

                char line [1000], *s;
                long lineno = 0;
                int except = 0, number = 0;

                while (--argc > 0 && (*++argv) [0] == '-') {
                        bfarecord ("main",2);
                        for (s = argv [0]+1; *s != '\0'; s++) {
                                bfarecord ("main",3);
                                switch (*s)
                                {
                                case 'x' :
                                        bfarecord ("main",4);
                                        except = 1;
                                        break;
                                case 'n' :
                                        bfarecord ("main",5);
                                        number = 1;
                                        break;
                                default:
                                        bfarecord ("main", 6);
                                        printf("find: illegal option %c\n",*s);
                                        argc = 0;
                                        break;
                                }
                        }
                {
                if (argc != 1)  {
                        bfarecord ("main",7);
                        printf ("Usage: find -x -n pattern\n");
                else{
                        bfarecord ("main",8);
                        while getline (line, 1000) > 0   {
                                bfarecord ("main", 9;
                                {
                                        lineno++;
                                        if ((index line, *argv) >= 0) != except){
                                          bfarecord ("main",10);
                                          {
                                                if (number) {
                                                  bfarecord ("main",11);
                                                  printf ("%1d:",1);
                                                }
                                                else bfarecord("main",12);
                                                printf ("%s",line);
                                          }
                                        }
                                        else bfarecord ("main",13);
                                }
                        }
                }
        }
        bfareport ("-rreport");
}
```

(a)

```
function  # times  existing  # branches  % of
name      invoked  branches  hit         branches hit
=============================================================

main          1        13          9           69

index        14         5          5          100

getline      15         4          4          100
-------------------------------------------------------------
Totals                 22         18           82

A * preceding the function name indicates the function
was not hit

      3 functions in the program: 0 not hit
      100% of the functions were entered
```

(b)

Fig. 1. *(a) A BFA instrumented source file. The names of the instrumented functions are highlighted. The underlined lines of code were inserted by the BFA preprocessor. They are calls to the routine* bfarecord *which handles the accounting on the software being tested. (b) The summary test report provided by BFA after the instrumented program is run.*

this information we figured that some of the best test cases could be leveraged from the engineers developing the Starbase/X11 Merge code. Therefore, an effort was made to formalize the process that engineers naturally go through when trying a new version of the X server for the first time. All engineers were required to develop a short list describing the types of tests they normally tried. When an integration cycle approached, all engineers ran through their minisuites and provided feedback. With little additional effort, such testing proved valuable.

Starbase Test Suite

The Starbase test suite has traditionally been used to perform testing of the Starbase graphics library on all of HP's supported graphics display devices and workstation configurations. The test suite consists of nearly 400 test programs, archive files of expected results, and various shell scripts and C programs that control test suite automation.

When a test program is run as part of an automated session, the resulting standard output and errors are compared against the expected result archives. In addition, representations of the various graphics images that may have been generated by the test program are compared with the archives. Specific differences between actual and expected results are noted in a test suite log file, and simple pass/fail information is placed in a summary file.

Before the Starbase/X11 Merge system, the test suite was used to test Starbase running only on a raw display device rather than in a windowed environment. With the advent of the Starbase/X11 Merge system, there was a need to enhance our Starbase testing approach to include not only raw device testing, but also testing of Starbase in the X Window System environment.

Starbase test programs in the Starbase/X11 Merge environment take two basic forms:

- Window Naive. A window naive test can run either in raw mode or in X. The test itself has no knowledge of X, and does not create X windows itself, but instead relies on an outside mechanism to create the windows and direct the test to those windows.
- Window Smart. A window smart test can only run in X. By definition, a window smart program makes X calls, and usually creates its own output windows.

The enhancements made to the Starbase test suite had to be able to support both varieties of test programs. An additional goal of the changes was to leverage as much of the existing test suite as possible. To test window naive Starbase programs, the test suite was modified so that it could recreate various selected X window scenarios and then run test programs in each scenario. Since window naive programs can be run on a raw display or in an X environment, we were able to use a set of the existing test programs in these scenarios. Of course, new archives of expected results had to be created for each scenario.

To cover window smart testing, an additional X window scenario was used in the test suite. Also, since none of the existing test suite programs contained both Starbase and X library calls, a set of new test programs had to be written to test this new functionality adequately. Areas of particular testing attention included text fonts, cursors and echoes,

color map manipulation, backing store, double buffering, and z-buffering.

Once the changes to the test suite were in place, the suite was run nightly in a test center stocked with a complete set of graphics display devices and workstation configurations. An additional set of tools was developed to gather and report test results automatically from each configuration on a daily basis. This was done even during the latter part of the Starbase/X11 Merge project implementation phase and it enabled developers to track the quality of their code as it was being completed. During the testing and release phases, the nightly test suite results helped ensure continuing improvement in code quality and stability.

X Test Consortium Test Suites

Through HP's affiliation with the X Test Consortium, several X test suites were acquired. The Digital Equipment Corporation's X test suite (nearly 350 tests) tests each call available in the Xlib library. The tests themselves come in two categories: good-only tests or centerline tests which just test for expected functionality. Validate and error tests expand on the centerline tests by checking for robustness using invalid parameters and erroneous functionality.

The Sequent Computer Corporation, which is a member of the X Test Consortium, provided an X test suite that consists of nearly 125 tests that exercise the server at the X protocol level. The tests themselves do not use Xlib, but instead contain custom buffering routines to send X protocol requests to and receive replies from the server. The object of these tests is to see how well the X server handles malformed protocol packets not normally generated through the X library calls.

Early in the testing effort, the decision was made to make the X Test Consortium suites more manageable by controlling them with HP's scaffold automation tool.[1] The scaffold provided the framework to manage the large body of tests, and also provided some input and output archiving. With the scaffold in place, the test suites were run nightly by an HP-UX cron script on all unoccupied workstations used by the Starbase/X11 Merge development team.

HP-HIL and Input Extension Test Suite

With the addition of several input extensions to the X server, a new input extension test suite had to be developed. Previous input testing tools proved to be inadequate for three reasons:

- HP-HIL (HP Human Interface Loop) activity was usually captured after some processing of the HP-HIL activity had already occurred.
- Previous test tools required that the code under test be modified to accommodate the test mechanism.
- Previous test tools could only handle keyboard and mouse activity, thereby excluding the new HP input extensions to the server.

The HP-HIL simulator, which was leveraged from an existing HP Windows/9000 test tool, allows multiple HP-HIL devices to be simulated and tested at once, including the new input extensions. The HP-HIL simulator operates in record/playback modes. The record mode requires the HP-HIL devices, the simulator, and a tester to run the test and use the HP-HIL devices. When it is recording, the simulator captures all HP-HIL activity and puts it into a file. In playback mode, the simulator uses the file captured during the record mode in place of the real HP-HIL devices. The tester only needs to start the test program in playback mode. All of the HP-HIL data, regardless of its source, is sent to the server.

The HP-HIL simulator is installed by creating a pty (pseudo tty) in the /tmp directory for each input device on the HP-HIL loop. This sets up a communication path between the ptys and the real HP-HIL devices. To ensure that the X server will use the ptys in /tmp, an appropriate entry is made in the server's Xndevices file to change each device's path from /dev to /tmp. The Xndevices file is used by the server to determine its input device locations.

When a recording test session is started and the server tries to open what it thinks is an HP-HIL device, it is connected to a pty and the HP-HIL simulator is triggered to open the real HP-HIL device. Once this is done, the HP-HIL simulator, transparent to the test program, passes all HP-HIL device activity back and forth while saving all HP-HIL activity along with timing data into a file. The timing data ensures that realistic playback is provided. Fig. 2a shows the setup for test recording.

For HP-HIL playback, the file that was saved during recording is simply read by the simulator, and the appropriate HP-HIL activity is generated in the same time sequence it was recorded and fed into the pty. During the playback sessions the real HP-HIL devices do not have to be present on the HP-HIL loop. This facility allows suites recorded using the HP-HIL simulator to run on any machine without concern for the presence of HP-HIL devices—which are sometimes hard to find. The setup for playback is shown in Fig. 2b.

The HP-HIL simulator was used to test the server input extensions, and was then incorporated into the test scaf-

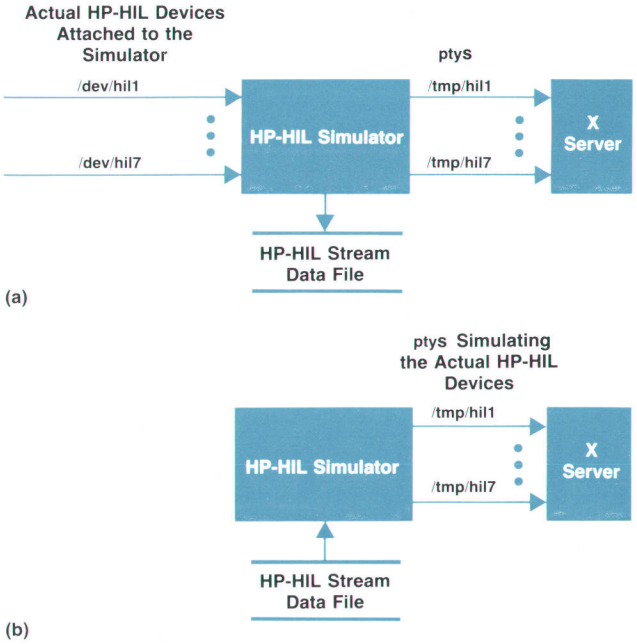


Fig. 2. *(a) Initial setup for recording data from HP-HIL devices. (b) Setup for playback.*

fold. The simulator was also used by another group to simulate foreign versions of the HP-HIL keyboard to test native language support (NLS) functionality.

GRM Test Suite

The graphics resource manager (GRM) is composed of a daemon process and a client interface library. The suite of tests that was developed for the GRM system is partitioned according to the various functional components of the system. A test module was developed for each of the following functional categories:

- Client/Server Protocol. The serial data stream between the GRM client and the GRM daemon.
- Object Allocation (including semaphores). The maintenance of all display hardware resource allocations.
- Offscreen Memory Management. The allocation and deallocation of three-dimensional blocks of offscreen memory.
- Shared Memory Management. The creation, allocation, and deallocation of chunks of shared memory.
- Sequence Control. The maintenance of request sequences for multiple processes.
- Listing of Objects. The wild-card matching and listing of all GRM objects.

With the exception of the protocol test module, all of these test modules tested the operation of the GRM daemon through the standard GRM interface library. For the protocol test module, some library routines were replaced with altered copies of the original library routines to achieve the desired test procedure.

Although the GRM daemon is designed to operate with multiple clients, the tests were designed to have exclusive use of the GRM. If another GRM client process was detected by the test process, the test would identify the error and exit. Since only one GRM daemon will run on a single host at any particular time, the test environment had to be free of any graphics applications that used Starbase or the X server.

XDI Test Harness

The X driver interface, or XDI, has about four dozen entry points in the device dependent portion of the X server. The X driver interface provides an interface between a translation module and the low-level X display drivers that perform the actual display control and rendering operations on the display hardware. The translation module is responsible for translating requests from the device independent portion of the X server into a form suitable for the X display drivers. This architecture allowed independent development by HP engineers in two different organizations and locations, and provided a platform for code sharing. The X server code was done at HP's Corvallis Information System Organization, and the display drivers (for X and Starbase) were done at HP's Graphic Technology Division. The article on page 6 describes the Starbase/X11 Merge X server and the XDI, and Fig. 2 on page 9 shows the X server architecture.

With the significant advantages of this newly defined interface, there came corresponding new testing demands, because high-quality, well-tested X display drivers had to be delivered at regular intervals, and these drivers had to be developed whether or not any server code was available.

While much of the underlying driver code was shared by the Starbase driver code, the X driver interface was tailored to the needs of the X server. The differences between the Starbase driver interface and XDI were sufficient to prohibit direct use of the Starbase test suite. Since the test suite could not be directly used, other approaches were explored that would meet our testing needs and leverage as much of the existing test suite technology as possible.

To provide a tool for debugging and automated testing, the XDI test harness was developed. The harness provides:

- A user interface for each XDI entry point
- A means for importing and manipulating the associated data structures
- Support for a subset of C programming language commands.

What makes the harness an unusual testing tool is the way in which it acts as an interpreter that receives input commands either interactively or from text script files.

The XDI test harness offers several advantages over more traditional testing approaches that involve compiling various test programs and then linking each of them with the code under test. The harness needs to be linked only once with the code under test, and since the harness is interpreter-based, any number of test programs can be run without the need to link each one. The harness also makes test programs easier to write and modify because it provides a convenient interface to the XDI entry points and the ability to manipulate data structures. Finally, disk space is conserved because only the harness and not the numerous test programs need to be linked with the large driver libraries.

As a result of these advantages, the XDI test harness proved to be a useful tool for XDI code development and debugging. In addition, with relatively minor changes to the Starbase test suite tools, the XDI test harness was integrated directly into the test suite. An extensive set of new harness test programs was developed to test all the types of graphics display devices supported by the Starbase/X11 Merge system. Once the tools and test programs were in place, this new XDI test suite was run nightly in the test center.

Interactive Testing

While it was desired to automate as many tests as possible, not all server activities could be automated. Furthermore, a measure of randomness not provided by the automated tests needed to be added. Areas especially suited for this type of testing included object manipulations with the X cursor (e.g., moving a window), screen changes when running in a stacked screens mode, multiserver environment, and Starbase echoes (cursors operated by Starbase) in X. Usually, interactive testing allowed a wider range of scenarios to be tried. When certain scenarios were identified as productive, an attempt was made to automate them.

Conclusion

With approximately 500 KNCSS between X and Starbase, and over 80 different hardware configurations, testing the Starbase/X11 Merge system proved to be very challenging. Available test tools and test suites provided the bulk of

our automated tests, while the branch flow analyzer coverage led to the development of new test tools and many new tests. During the latter half of the Starbase/X11 Merge project, we realized there was a need for more user-interactive tests. While automated tests are indispensable, we found that a great many interesting and important defects can be uncovered with the randomness provided by user-interactive testing.

Acknowledgments

We would like to thank Jim Andreas and Courtney Loomis for their efforts on the X Text Consortium and graphics resource manager test suites. We would also like to thank Mike Mayeda, Jan Kok, and John Waitz for playing key roles in redesigning and executing the Starbase test suite technology in the X Window System, and Jennefer Wood, who designed and built the HP-HIL simulator.

References

1. C.D. Fuget and B.J. Scott, "Tools for Automating Software Test Package Execution," *Hewlett-Packard Journal*, Vol. 37, no. 3, March 1986. pp. 24-28.

Authors

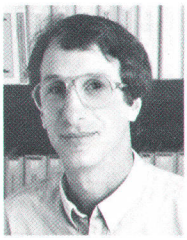
December 1989

6 Starbase/X11 Merge System

David J. Sweetser

As project manager, David Sweetser brought several years of experience in Starbase driver development and high-performance 2D and 3D graphics to the Starbase/X11 Merge project. Born in Woodland, California, he received a BSEE degree (1971) and an MSEE degree (1972) from Harvey Mudd College. In 1977, he joined HP's Corvallis Division, working on I/O hardware and software for the HP-85 Computer. He later transferred to the Graphics Technology Division, where his responsibilities have included the design of graphics accelerators and the development of graphics drivers, most recently for the Starbase/X11 Merge. Dave is married, has a son and a daughter, and lives in Fort Collins, Colorado. He spends his spare time white-water rafting, mountain biking, cross-country skiing, hiking, and playing tennis and volleyball. He has contributed two previous articles to the HP Journal.

Kenneth H. Bronstein



School teaching and textile production management are two of several vocations Ken Bronstein pursued prior to joining HP's Corvallis Division in 1981. Since then, he has worked on HP Integral PC applications and OS support, custom ROM development for the Integral PC, and the Starbase driver for the X10 system. He was a project manager on the Starbase/X11 Merge project. Born in Chicago, Illinois, Ken received a BA degree from Washington University in St. Louis in 1973 and an MS degree in computer science from the University of Oregon at Eugene in 1981. He is married, has a son and a daughter, and lives in Corvallis, Oregon. An amateur astronomer, he also finds time for hiking, organic gardening, and backgammon.

William R. Yoder

Bill Yoder has served as a senior technical writer, as a software engineer on a DOS/Pascal workstation and X10, and, more recently, as project leader of the Starbase/X11 Merge server team. He is now release manager of the HP-UX 8.0 X Window System. He received a BA degree in history and literature from Harvard University (1972) and an MA degree in English from the University of California at Santa Barbara (1976) and taught high school and college English for six years before joining HP in 1980. He went on to earn a BS degree (1985) and an MS degree (1988) in computer science from Oregon State University and Stanford University, respectively. He has written a dozen user and programmer manuals for the HP-75 and HP-85/86/87 Personal Computers, UCSD Pascal, and the HP Integral PC, and he programmed and authored the tutorial disc for the Integral PC. Born in Bloomington, Illinois, Bill is married, has a son, and lives in Corvallis, Oregon. He is a member of the ACM SIGGRAPH and SIGCOM, the IEEE Computer Society, and Computer Professionals for Social Responsibility. He coaches in the American Youth Soccer Organization and enjoys aerobie, jogging, playing guitar, reading fiction, and participating in local politics.

12 Starbase/X11 Display Objects

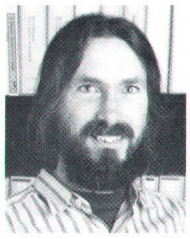
Robert C. Cline

Born in Oceanside, California, Bob Cline received his BS degree from the University of Massachusetts in 1976 and his MS degree from Indiana University in 1978. He came to HP that same year and worked on the HP Integral PC, the X10 clients, and the X11 server before joining the Starbase/X11 Merge project team. He is currently working on the HP NewWave Environment. Bob's specialties are operating systems and data communications systems. He has coauthored articles on the UNIX operating system for the HP Journal and for Byte Magazine. He is single, lives in Corvallis, Oregon, and devotes his spare time to camping, hiking, swimming, and golf.

James R. Andreas

Before joining HP, Jim Andreas designed hardware and software for digital plotters. In 1980, he joined HP's Corvallis Division, where he was involved in research and development of the HP-85 Personal Computer, the HP 9807A Integral Computer, and the HP Vectra CS. He has spent the past two years working on the X Window System and was a member of the R&D group that designed the graphics resource manager of the Starbase/X11 Merge system. He coauthored a previous article in the HP Journal on a UNIX operating system. A 1977 graduate of the Massachusetts Institute of Technology with a BS degree in electrical engineering and computer science, he also holds an MS degree in computer science from Oregon State University (1988). Jim was born in Silverton, Oregon and lives in Corvallis, Oregon with his wife and two daughters. He and his wife are active in ballroom dancing. He also collects and raises Japanese maples.

Courtney Loomis



Born in Honolulu, Hawaii, Courtney Loomis came to HP by way of Oregon State University, where he earned his BS degree (1979) and MS degree (1982) in electrical and computer engineering. He designed the keyboard controller and memory modules for the HP Portable Plus and developed the keyboard controller firmware for the HP Vectra CS portable computer. He is currently applying his expertise to user interface management systems. He was part of the Starbase/X11 Merge team who designed and tested the Graphics Resource Manager. Courtney is single and lives in Corvallis, Oregon. His outside interests are birdwatching, botany, cross-country skiing, mountaineering, canoeing, and visiting Western deserts.

20 Starbase/X11 Display Resources

Steven P. Hiebert

Author's biography appears elsewhere in this section.

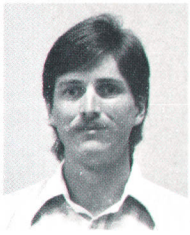
Keith A. Marchington

Author's biography appears elsewhere in this section.

Michael H. Stroyan

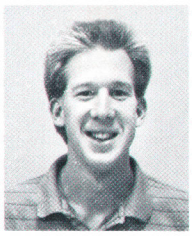
Mike Stroyan's special interests are in computer graphics. After receiving his BS degree in computer science from Colorado State University in 1982, he joined HP and worked on graphics for the HP 9000 Series 200 HP-UX operating system and Starbase. After joining the Starbase/X11 Merge project team, he concentrated primarily on the input and display drivers. Mike is a member of the ACM SIGGRAPH. He is single, lives in Fort Collins, Colorado, and enjoys skiing, volleyball, and swimming. He was born in Lockport, New York.

John J. Lang



John Lang has worked on Starbase projects since he joined HP in 1985. He has contributed primarily to the TurboSRX drivers and the SRX driver of the Starbase/X11 Merge system. Born in Lompoc, California, he received his BS degree in wildlife biology in 1982 and his MS in computer science in 1985, both from Colorado State University. John is single, enjoys skiing, and coaches a coed HP soccer team in a citywide league in Fort Collins, Colorado, where he lives. He is a member of the ACM.

Jeff R. Boyton



Jeff Boyton is a graphics engineer responsible for the X Window backing store and for the pixmap (memory) portion of the X Window System. A 1986 graduate of the Michigan Technological University with a BS degree in computer science, he joined HP the same year. He now has engineering responsibility for the TurboSRX driver and administrates defect tracking for Starbase. Away from work, he is a frequent participant in volleyball tournaments. Jeff is married and lives in Fort Collins, Colorado. He is a member of the ACM.

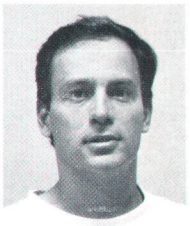
Sankar L. Chakrabarti

Sankar Chakrabarti received degrees in chemistry at Calcutta University and Kalyani Universities, and a PhD degree in chemistry and molecular biology from the Tata Institute of Fundamental Research in Bombay. He also pursued basic research in molecular biology at Harvard Medical School and the University of Oregon. A major career change brought him to Intel Corporation and later to HP's Corvallis Division in 1981. Since then, he has worked on the X10 Window System and the Integral PC before joining the team that developed the backing store portion of the Starbase/X11 Merge system. Born in West Bengal, India, Sankar lives with his wife and two children in Corvallis, Oregon, where he helps coach his son's soccer team. He has an MS degree in computer science from Oregon State University (1985).

Jens R. Owen

As a member of the technical staff of HP's Graphics Technology Division, Jens Owen worked on the design of raster fonts for the Starbase/X11 Merge project. Born in Gaester, Denmark, he grew up in Denver and received his BS degree from Colorado State University in 1986. He joined HP after graduation. Jens is newly married and lives in Fort Collins, Colorado. His hobbies include snowboarding, volleyball, and mountain biking.

John A. Waitz



A graduate of Colorado State University with a BA degree (1980) and MSCS degree (1982), John Waitz has worked on various Starbase assignments since joining HP in 1983. His contribution to the Starbase/X11 Merge project was in the development of Starbase and X Window display drivers and driver utilities. John was born in Philadelphia, Pennsylvania, but grew up in Boulder, Colorado. He is single and lives in Fort Collins, Colorado. He plays the piano, sings in a community jazz choir, and enjoys tennis, running, and bicycling. He is a member of the ACM and the IEEE.

Peter R. Robinson

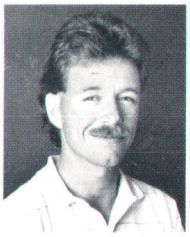
Peter Robinson was a design engineer for the XDI interface and direct hardware access (DHA) extensions to the Starbase/X11 Merge server. His earlier assignments include work on the HP Integral PC and HP Portable Plus projects. Born in Lancaster, California, he received his BS degree in computer science from the University of California at Los Angeles in 1975 and his MS degree in electrical engineering from Stanford University in 1978. He joined HP in 1980 at Corvallis, Oregon, where he now resides. Peter and his wife provide home schooling to their three daughters, for which they design the curriculum. He also plays the piano and accompanies his family on bicycle tours.

33 — Starbase/X11 Memory Planes

John J. Lang

Author's biography appears elsewhere in this section.

Keith A. Marchington



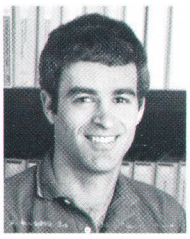
Development of global display controls and server operational modes for the X Window System server are Keith Marchington's primary contributions to the Starbase/X11 Merge project. He has also been a product marketing engineer on the X10 Window system and a product support engineer for various Corvallis products. Born in Portland, Oregon, he lives in Corvallis, Oregon, where he joined HP in 1979. He has BS degrees in computer science and mathematical sciences from Oregon State University (1981). Keith is a single parent who enjoys sports, reading, and a variety of activities with his five-year-old son.

Steven P. Hiebert

A former U.S. Air Force staff sergeant, Steve Hiebert was born in Portland, Oregon. He received his BS degree in mathematics from Portland State University in 1976. He joined HP in 1981 and worked on the compilers and compiler utilities for the HP Integral PC. Before that, he worked on Pascal compiler implementation and support at Electro Scientific Industries and was the department manager for systems programming at TimeShare Corporation. On the Starbase/X11 Merge project, Steve's responsibilities included the cursors, locks, rendering state, combined mode, and stacked-screens mode of the X Window System. He is single and lives in Corvallis, Oregon. He is a member of the IEEE and the ACM.

38 — Starbase/X11 Input Devices

George M. Sachs



As an R&D engineer on the Starbase/X11 Merge project, George Sachs was responsible for the support of input devices by the X Window server and for proposing and implementing standardized support for additional input devices through the X Consortium. He also has been a software quality engineer on various other projects. Before coming to HP in

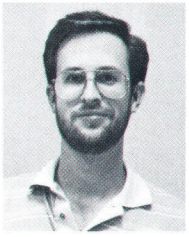
1981, he was a software development engineer developing a relational data base management system at IBM. He earned his BS degrees in computer science and zoology in 1978 at Oregon State University. Born in Palo Alto, California, George is married, has two children, and lives in Corvallis, Oregon. He is active in his church and lists as his hobbies city-league softball, water and snow skiing, fishing, camping, and woodworking.

Ian A. Elliott

After receiving his BS (1984) and MS (1987) degrees from the University of Utah and serving as assistant director of the University's Department of Bioengineering Surface Analysis Laboratory, Ian Elliott joined HP and began working on the Starbase/X11 Merge project. His most recent contributions have been to the X Window System, Starbase device driver, and general architecture. He authored a 1983 article about surface analysis for the Journal of Electron Spectroscopy and Related Phenomena. Ian, who was born in Detroit, Michigan, lives in Fort Collins, Colorado with his wife and new daughter. He spends most of his free time fixing up their recently purchased home.

42 Starbase/X11 Testing

John M. Brown



London, Kentucky was home to John Brown until he completed his BSEE degree in 1980 at the University of Kentucky. His specialty of high-performance 3D graphics rendering is the result of several years spent developing software for sonar systems at IBM, and hardware and software graphics architectures at General Electric. He joined HP in 1988 and assumed responsibility for quality assurance and testing for the Starbase/X11 Merge project. John is married, has two children, and lives in Ft. Collins, Colorado. He enjoys running, snow-skiing, and bicycling.

Thomas J. Gilg

Thomas Gilg earned his BS (1986) and MS (1988) degrees in computer science from Montana State University, where he was a graduate research assistant working on the remote electronic animal data system (READS) project at GeoResearch, Inc. He has worked on the Starbase/X11 Merge project since joining HP in 1988, particularly in testing for the merge server, concentrating on mixed-mode tests. Thomas is an avid fisherman; through his membership in the Northwest Association of Steelheaders he is active in the preservation of fishing habitats. He also enjoys back-country traveling, hiking, and cross-country skiing. He was born in Casper, Wyoming, is single, and lives in Corvallis, Oregon.

50 CD-ROM Source Access System

B. David Cathell

As a project manager on the HP Source Reader development, David Cathell's contributions focused on the design and, eventually, the implementation of the system. When he joined HP in 1978, he had completed some ten years of system programming experience at Control Data Corporation. Since then, he has held positions as a senior programming analyst for manufacturing applications and as a field systems engineer for commercial installations. He wrote and presented a paper at the 1984 HP 3000 Users Group (now Interex) semiannual conference describing his research into the rules of Image. David was born in Showell, Maryland, and lives in San Jose, California. His hobbies are woodworking and reading. He and his wife enjoy traveling to Hawaii every year.

Michael B. Kalstein

Born in Elkins Park, Pennsylvania, Mike Kalstein graduated in 1976 with a BA degree from Temple University. He came to HP after receiving his MS degree in computer science from the University of Illinois in 1979. He soon joined the Santa Clara, California, sales office to become a systems engineer responsible for presale support, training, and field software coordination. In later assignments as on-line support manager and systems escalation engineer at HP's Commercial Systems Division, he directed technical support for MPE systems and administered the engineering staff. His contributions to the HP Source Reader project included developing part of the access and filtering programs and coordinating the production of some of the CDs. Mike and his wife are expecting their second child in December. His leisure interests include playing piano and guitar, song-writing, tennis, bicycling, walking, and home improvements. He lives in Campbell, California.

Stephen J. Pearce

When he joined HP in 1973, Steve Pearce had completed a four-year internship in electronic R&D at the British Ministry of Defense. Since then, his assignments have included positions as a bench engineer on microwave equipment, as a customer engineer on computer systems, and as both a technical support engineer and a response center engineer on the HP 3000 Computer. On the Source Reader project, he was the programmer for the access program. Steve holds the equivalent of a BSc degree in electronic engineering from the Worcester Technical College. He was born in Bromsgrove, Worcestershire, has two children, and lives in San Jose, California. His favorite leisure activities are astronomy and listening to music.

58 Transmission Line Effects

Rainer Plitschka

As project manager for the HP 82000 IC Evaluation System, Rainer Plitschka's responsibilities included the pin electronics, device-under-test interfacing, dc parametric measurement unit, mechanics, and power supply. In earlier projects, he contributed to the design of the HP 8112A Pulse Generator and the HP 8116A Pulse/Function Generator. He joined HP in 1979, after completing his studies at the University of Karlsruhe, where he earned his engineering diploma. Rainer presented a paper on the subject of transmission line effects in the test of high-speed devices at the European Test Conference 1989 in Paris. Born in Stuttgart, he is married, has three boys, and lives in Herrenberg, Baden-Württemberg. In his off-hours, he enjoys rebuilding an old home, woodworking, and photography.

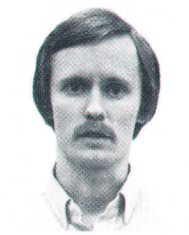
74 Custom VLSI

Larry J. Thayer

Larry Thayer worked on the HP 9000 Series 500 Computer and the SRX graphics system scan-conversion chip before assuming responsibility for the HP TurboSRX pixel processor. Presently, he is designing VLSI for a future graphics hardware product. He graduated from Ohio State University with a BSEE degree in 1978 and an MS degree in 1979. He is coauthor of a 1984 HP Journal article about the HP 9000 Series 500 Computer and a 1986 SIGGRAPH paper describing the SRX scan-conversion chip. A native of Lancaster, Ohio, Larry lives in Fort Collins, Colorado with his wife and two children. His hobbies include basketball, softball, and church and family activities.

78 Radiosity

David A. Burgoon



Dave Burgoon wrote microcode for the TurboSRX transform engine and is now working on next-generation transform engine hardware design. He joined HP in 1981, after he received his BS degree in electrical engineering from the University of Toledo. He earned his MS degree in computer science in 1989 from Colorado State University. His professional interests include fast rendering methods and hardware architectures for computer graphics. He authored an article describing the SRX accelerator hardware architecture that appeared in the July 1987 issue of Electronic Design. Dave enjoys repairing automobiles and televisions and reading science fiction. He is married, has a daughter, and makes his home in Fort Collins, Colorado.

A Compiled Source Access System Using CD-ROM and Personal Computers

HP Source Reader is in use in virtually every HP support facility around the world, giving local support engineers fast access to complete source code listings for MPE, the HP 3000 Computer operating system.

by B. David Cathell, Michael B. Kalstein, and Stephen J. Pearce

HP SOURCE READER IS A SYSTEM for accessing compiled source code stored on compact disk read-only memory (CD-ROM) for purposes of system debugging. The source code is stored in a proprietary format that optimizes retrieval by the access program running on an HP Vectra Computer.

HP Source Reader facilitates quick and efficient debugging of HP 3000 Computer systems by allowing the user to display source code at any point within a specified procedure or segment. The user can then quickly scroll the display or jump to any other location with precise control. Relevant information can be "popped" onto the screen in seconds. This includes identifier definitions, reference materials, and the assembly code corresponding to each source line. The program also provides many useful auxiliary functions including searching, printing, logging, and a comprehensive set of customization options. A context sensitive help facility eliminates the need to consult written documentation.

Unlike other source browsing systems, HP Source Reader was written by and for engineers who debug HP 3000 Computers. The user interface is designed to be familiar to support engineers who may not be knowledgeable about personal computers. The program prompts users for information in the same format as other tools they use. In addition, to make the program easy to use, HP Source Reader takes full advantage of the personal computer user interface, including keyboard, mouse, pop-up windows, and menus.

To our knowledge, HP Source Reader is the first system in the industry that combines the convenience of one-step source retrieval with the power of the CD-ROM and personal computer (PC) technologies.

HP 3000 Debugging—Before

HP 3000 Computers are debugged, for the most part, by analyzing dumps of the computer's memory. When a system fails, the operator dumps the memory to magnetic tape and then restarts the computer. The tape is forwarded to HP where it is formatted and analyzed by an engineer. The engineer must examine source code while reading the dump, comparing the failed system to what the source code indicates should happen when the system is running normally. The engineer is constantly alternating between the source code and the dump throughout the analysis of the problem.

Historically, memory dumps have been printed on paper. This worked fine when the HP 3000 contained less main memory, but this practice has gradually become untenable with the advent of larger and larger systems. Therefore, interactive tools have been developed that allow a dump to be analyzed in an on-line mode. Over time, these interactive tools have been enhanced to the point where they are now powerful on-line tools that allow engineers to locate and format specific information in a memory dump easily. However, as these tools have matured, no parallel progress has occurred allowing efficient on-line examination of source code. Engineers have continued to depend on printed listings stored in a shared library area.

Fig. 1 shows the complex manual process that must be followed to locate specific source code in a listing from information presented in the memory dump. It should be apparent that this is exceptionally tedious.

Project History

In 1986, we began to rethink the strategy for the use of workstations within our organization (HP Commercial Systems Support). It seemed apparent that real productivity gains could be made by engineers, managers, and support personnel through the use of readily available PCs and software.

At the same time, we recognized that it was becoming feasible to marry the PC to the emerging technology of optical media. This marriage could provide a platform for an engineer to access the massive amount of information required for system level support of MPE, the HP 3000 operating system.

It became apparent that much of the time spent in analyzing system failures was not bringing expertise to bear on the problem. Engineers were spending too much time in overhead activities—walking to a library, finding listings, and engaging in the long tedious process of source location. We felt this strictly mechanical work could and should be automated.

We refined our ideas sufficiently to produce a PC-based demo version of a program. This gave us the opportunity to evaluate the user interface with feedback from engineers who would be users of the actual program, if and when it was produced. It also gave us a method to communicate our vision to software developers who had experience in

the CD-ROM industry but who did not necessarily have knowledge of our particular activities in supporting the HP 3000.

Unfortunately, when we surveyed what was available in an attempt to save development effort, all we found were natural-language-based keyword indexing data base engines. These are not a viable solution because there is a substantial difference between natural language and computer language. For instance, the word "ball" has a small number of meanings which are reasonably consistent from document to document. However, the variable "x" may have many different meanings depending on where it appears in the source code.

Eventually, we concluded that there were no existing solutions that we could leverage to meet our needs—we would have to develop a prototype. This first prototype was the proof of the validity of the concept. It had enough positive aspects to justify the resources to rewrite and then extend the programs.

The main body of effort is now complete and the generation of HP Source Reader CD-ROMs is becoming a routine manufacturing effort. The only remaining tasks involve small utility programs to automate some partially manual processes. In addition, we plan to continue to expand the functionality of the access program as good ideas are suggested and as time permits their implementation.

Project Goals

At the beginning of the project our overriding objective was to improve the efficiency of HP 3000 system debugging. To achieve this, we established the following goals:

- Elimination of paper listings to save time, space, and mundane labor.
- Full use of emerging technology to make engineers' time as productive as possible.
- Ease of use to minimize learning time and errors.
- Minimal impact on organizations that supply source code to avoid the need to reformat source code or modify procedures.
- Cost-effectiveness to make it easy for support organizations to justify the expense required.

CD-ROMs and PCs

CD-ROM is a logical choice for the paperless environ-

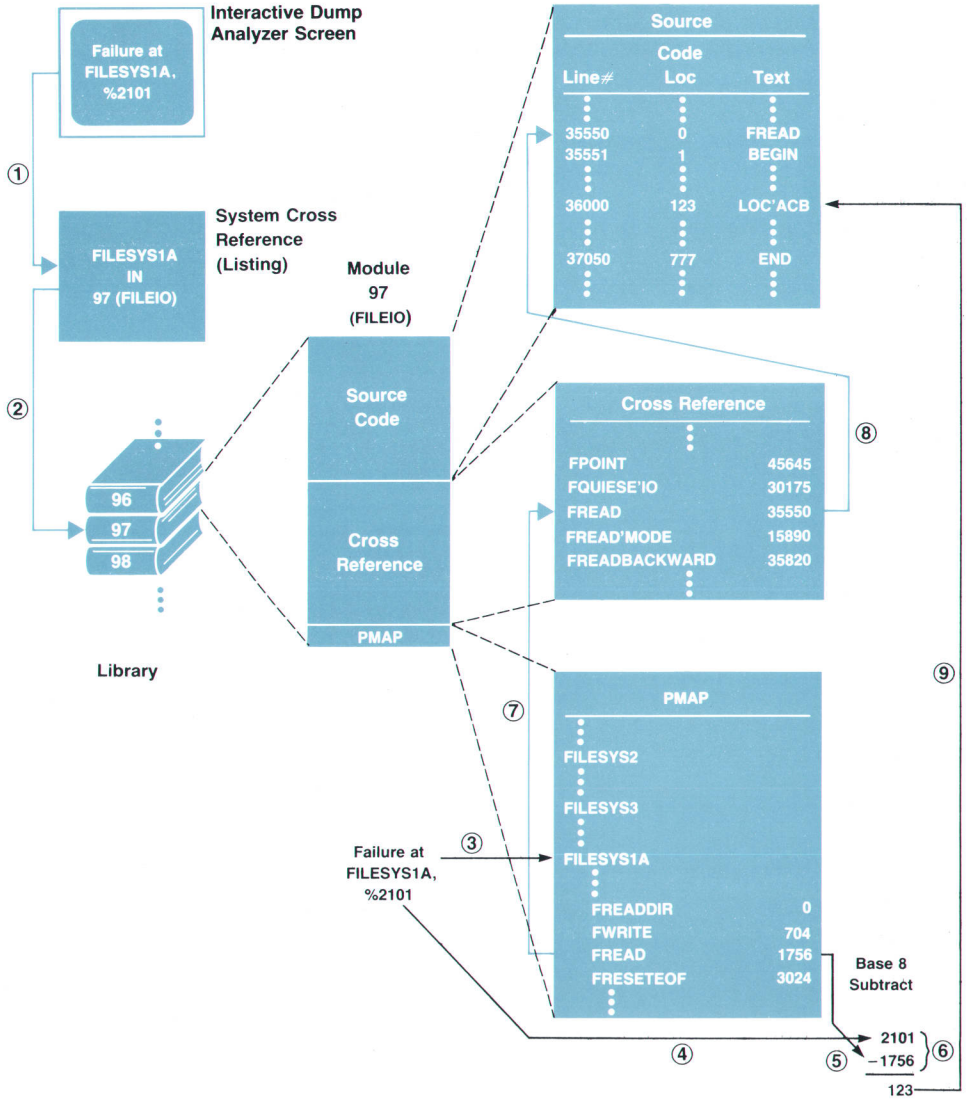


Fig. 1. *The traditional method of source code location.*

ment. A CD-ROM, a 4½ inch plastic disk, can hold the equivalent of a 35-foot stack of paper listings. This is sufficient capacity to contain an entire release of the HP 3000 operating system and its supporting software. The disks are inexpensive enough that each engineer can have a set. Unlike paper, optical media are machine readable, allowing for a wide variety of automated access techniques. Because the CD-ROM is read-only, it cannot be overwritten; it always has the integrity it had when it was manufactured.

The HP Vectra PC is an excellent system for implementing a high-technology, ergonomic access program. It has many features that make it comfortable for users—a mouse, a full-color display, and the ability to pop up and remove windows and menus as necessary. Since the Vectra is a personal computer, each engineer has private use of the system and its performance is not impacted by other users competing for resources. In addition, the Vectra supports a vast array of commercially available software and hardware products. Some of these products provide mechanisms for switching quickly between the source code and the dump, capturing parts of both in an integrated document. The Vectra is widely available within HP and is already in use in many offices that would need to use HP Source Reader.

Fig. 2 shows how HP Source Reader is used to accomplish the task shown in Fig. 1. These two diagrams clearly show the reduction in manual effort brought about by the access program.

HP Source Reader

HP Source Reader consists of two main parts. The first is the data preparation system, which is used to generate the CD-ROMs from the compiled source code as it is produced by the lab. The second is the access program that runs on the Vectra, which is used to locate and display the source code stored on the CD-ROM.

CD-ROMs are generated whenever a new version of the MPE V or MPE XL operating system is about to be released. Each disk contains all the modules associated with a given version. Fig. 3 shows the process flow used to convert the data from its original form (in the lab) to its final form (on the CD-ROM). Raw source code is maintained in the lab, then compiled with the output listing files submitted for inclusion on the CD-ROM. The compiler listings are processed in a series of steps to produce a magnetic tape set. The tapes are sent to a mastering facility, which manufactures the disks.

Structure of the CD-ROM

The CD-ROM has exactly the same physical structure as the now familiar audio CD. The only real difference between the two is the meaning of the information recorded on the optical media, which represents computer data in the case of the CD-ROM and digitized music on the audio CD. Data is recorded as a series of pits positioned in a continuous spiral (beginning at the center of the disk). The pits are read as ones and zeros when illuminated by a laser source. The bits are evenly spaced, requiring the drive to vary the rate of rotation to maintain a constant linear velocity. Additional bits are used to provide a high level of error correction.

Additional structure is imposed to make it possible to use the CD as a random-access device. A standard layout of the disk directories and files known as the High Sierra standard was proposed and widely accepted within the industry. Microsoft Corporation was active in the definition of the standard and quickly produced an intermediate level driver that makes all High Sierra CD-ROMs look like very large standard DOS discs (albeit read-only). CD-ROMs recorded using this standard have approximately 550,000,000 bytes of available disk space for data and directories. The wide acceptance of this standard and the availability of the Microsoft CD-ROM extensions made it possible for our project to develop our access program using the normal DOS file functions.

From the very beginning of the project, it was evident to us that the organization of the many files that would be on the CD-ROM was of paramount importance. A poor choice would have resulted in terrible performance. The resulting design makes extensive use of DOS subdirectories to group modules in a pattern logically similar to that of MPE. Fig. 4 shows the directory structure of the CD-ROM. The root directory contains only a file describing the contents of the CD-ROM. The second level subdirectories are of three types—one for system libraries, one for programs, and one for the reference documents.

The system library subdirectory contains only a file listing all the entry points and segments for that library. The modules themselves are located in subdirectories below the system library directory. Each module subdirectory contains a set of files containing the compressed source code, identifiers, cross reference, procedure map, and optionally, the object code for that module.

The program subdirectories contain a set of files containing the compressed source code, identifiers, cross reference, procedure map, and optionally, the object code for that program.

The document subdirectory contains a set of files containing the compressed text, page list, table of contents,

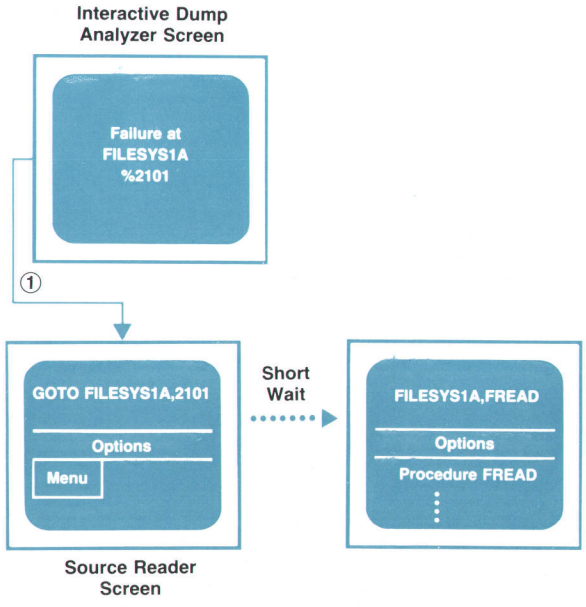


Fig. 2. *HP Source Reader method of source code location.*

and index for each document.

In addition to providing good performance, this structure has proved to be quite robust—only small extensions were required to include the changes brought about by MPE XL. Originally, we had only one system library subdirectory, and now we have three. In addition, a new directory type was defined for include files (these are files that are incorporated into the source code of multiple modules to provide common definitions, etc.). In the case of MPE XL, a set of files containing the compressed source code, identifiers, and cross reference for the large include file DWORLD resides in that directory. Thus this shared information is recorded only once, greatly reducing the amount of disk space required.

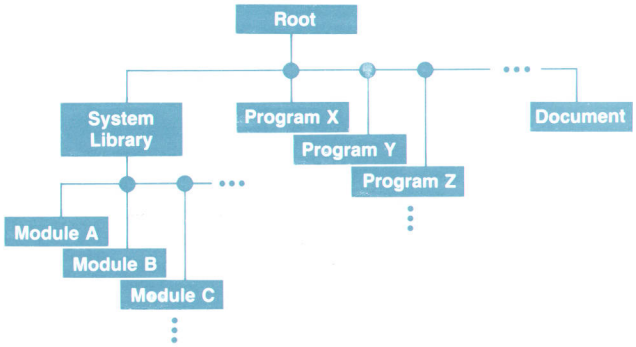


Fig. 4. *CD-ROM directory structure.*

Filters

In the compact disc industry, a filter is a program that reads some form of data and reformats it for use on a CD-ROM. The files on the CD-ROM are designed and organized to facilitate rapid retrieval of the desired information. The application designer can take advantage of the fact that optical media can be read but not written. Thus it is desirable to do as much processing as possible during the data preparation phase. This should result in less processing and, presumably, faster data retrieval by the access and display programs.

For the HP Source Reader project to succeed, we had to minimize any additional effort that might be required of other organizations. In our case, that meant that the input data for the filter program would have to be the same compiler-generated listing files that were already supplied for each release of MPE. These are exactly the same files that we previously printed and archived in our library.

The initial prototype filter was for SPL, the primary language used in MPE V. The result was tantalizing in that it gave us a glimpse of the tool that we had envisioned.

We learned from this prototype when we began the design of the filter for Pascal/XL (the primary language used in MPE XL). The major goal was to automate the processing of the huge number of listing files. The logical solution was a data base that would contain enough information about each operating system module to make the need for human intervention minimal. Thus the filter could locate files on the system to be filtered, determine which filter was to be used, and record the results of the filtering in the data base. This goal also dictated that filtering be done on a more powerful computer system than a PC, and an HP 3000 Series 70 was chosen.

A secondary goal was that the overall environment and the program structure be suitable for extending the filter for other programming languages. Proper design of the data

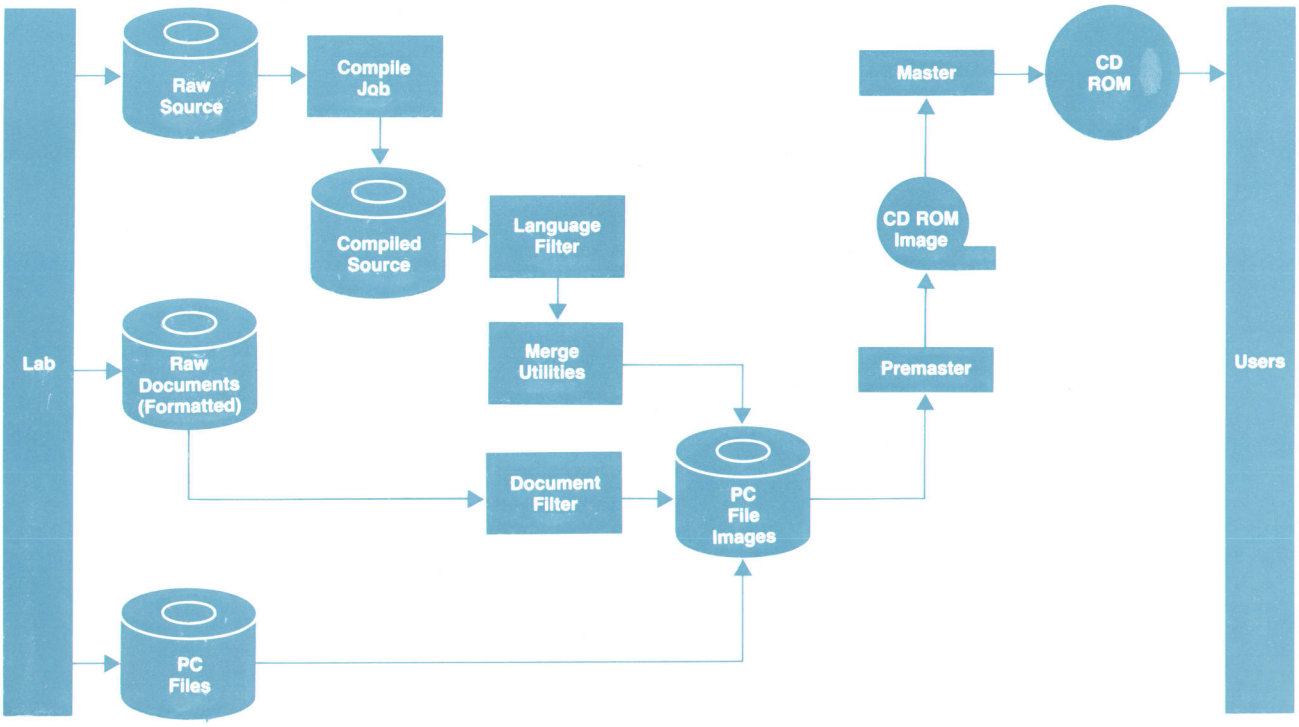


Fig. 3. *CD-ROM production process.*

base would easily allow extending the environment. To facilitate extending the filter program itself, we chose a three-pass philosophy.

The first pass parses each input record and determines what part of the listing it represents. It then reformats information to be retained and writes it to the appropriate intermediate file. The second pass performs certain cleanup tasks such as removing duplicate information regarding identifiers. The final pass generates the target files for the CD-ROM.

Although the first implementation using this three-pass philosophy was for Pascal/XL, we found that more than 95% of the code was retained when we extended the program to handle Pascal/3000 (the MPE V version of Pascal). The second and third passes were only minimally changed. Perhaps this result is not very surprising in the case of such closely related Pascal compilers. However, we found that more than 90% was retained when we implemented the SPL version of the filter. With these three filters, we can now process 99% of the modules for both MPE V and MPE XL.

Most of the processing is done by the filters. However, there is a need to accommodate certain complex modules that consist of multiple compilation units that may even be written in different languages. To keep the process as simple as possible, we filter each submodule and later employ a merge utility, which we also developed. This program uses the data base to determine which submodules need to be merged. The source, identifier, cross reference, and optional object files are retained but the procedure map files are combined. Each procedure entry in the merged map file indicates which submodule contains it.

Writing the filters was not a trivial task. We encountered numerous difficulties. The biggest challenge was posed by inaccuracies in the compiled output. The filters detected numerous cases of cross references that didn't exist or were on pages other than what the compiler reported. The Pascal compilers support long identifier names but truncate them in many places.

Additional challenges were provided by programmers. Some use NOLIST compiler directives to turn off listing output. Others use the DEFINE construct in SPL to improve readability and shorten the code. Still others use different cross-reference programs whose formats are different from the ones for which the filters were written.

Premastering and Mastering

Premastering is the process of converting files from standard DOS format to High Sierra format. The files output by the filters are standard DOS file images, while compact discs are recorded according to the High Sierra standard. Premastering changes the structure, not the content, of the files. The conversion is done on a CD Publisher system manufactured by Meridian Data Systems. The output of the CD Publisher is a set of master tapes, which are then sent to a compact disk mastering facility.

The mastering vendor takes the tapes and creates a CD-ROM master with the same data structure. This will be used to press CD-ROMs by a process identical to that used for audio CDs. The finished CDs are sent back to HP for packaging and distribution.

Access Program Design Philosophy

As mentioned above, a major goal for this project was to make the access program easy to use. This was especially important because most of the engineers who use it are not knowledgeable about personal computers. Therefore, we designed the screen layout with the major commands permanently displayed on the second line. Above that line is an area that identifies the current procedure. It is also used for dialog for commands that require it. The remainder of the screen is used for displaying source code.

Commands are invoked by pointing at them with the mouse. For systems without a mouse, the command can be selected by pressing the slash key (/) followed by the first letter of the command. When a command is selected, a menu drops down from the command line listing the subcommands. The user can point to the desired subcommand with the mouse or type the first letter of the subcommand. Prompts for additional information can be displayed on the top line or in dialog boxes if more room is needed.

Many of the commands require information such as the name of a procedure or a variable. We recognized that, while the program is in use, this information is probably already displayed on the screen. Therefore, we permit the user to move the alpha cursor by pointing at a screen position with the mouse, then selecting the command. When the user is prompted for the name of a procedure or variable, the access program automatically displays the identifier above the cursor as the default value.

Another design decision was the extensive use of windows—temporary boxes that overlay the main screen and contain information gathered from some other place in the listing. For example, if the user wants to know more about a variable used in the currently displayed code, the information is displayed in a window overlaying the top of the code area. Once the user has finished with the window it is removed and the code area is restored to its previous condition.

Although HP Source Reader uses many windows, it is not a Microsoft® Windows application. At the time the project began, MS Windows was not an established product. There was little known about OS/2 and Presentation Manager. Therefore, we decided to implement the access program as a character-based DOS application capable of running in various environments including DOS, MS Windows, and Quarterdeck DesqView.™

However, we also decided to structure the program in such a way that converting to MS Windows or Presentation Manager would be feasible without a complete rewrite. Thus, the program has a main loop, which checks for a user action (keystroke, mouse movement, or mouse button press). Control then passes to a routine based on the current internal state. That routine performs some action, possibly changes the internal state, and returns to the main loop.

Another important attribute of the access program is the speed of scrolling—we wanted it to be as fast as possible. Unfortunately, the access speed of the CD-ROM is only a bit faster than that of a flexible disk drive. Since most of our disk access is sequential, we implemented a buffering algorithm using buffers that are one sector long (2048 bytes). A pool of buffers is allocated when the program initiates. The exact number depends on the amount of

memory available on the system. Buffers are linked in order from most recently used to least recently used. When a new one is needed, the least recently used buffer is cleared and reused. This results in faster access than simply reading the individual records one at a time. Furthermore, the display of information already in the buffers is very rapid, since no I/O is required.

HP Source Reader is written in Turbo Pascal from Borland International with extensive use of routines in Turbo Power Tools Plus from Blaise Computing. The program employs numerous overlays which are carefully organized to preclude the possibility of thrashing.

Access Program Command Overview

The HP Source Reader access program is designed to provide engineers with the most flexible interface possible—one that provides commands that allow the required code to be located with minimum delay. The program was developed by engineers who would use it in day-to-day work, so the command structure chosen complements the data provided by current tools.

The main commands and subcommands of HP Source Reader are as follows:

GOTO

This is probably the most important command in the access program. It allows the user to select the exact code to display. Subcommands allow different types of access to the source. In MPE, each module is located either in a library or an application program. In MPE V/E and MPE XL compatibility mode, procedures are grouped into segments. In MPE XL native mode, segmentation is not used. To provide a consistent user interface, HP Source Reader defines "native mode segment" to be interchangeable with "module." GOTO has six subcommands.

SEGMENT. Allows the user to select a segment/module name to be used for the starting point for displaying source code. Optionally, the user can also provide an offset from that starting point. The user can limit the search domain to specific libraries to reduce search time.

PROCEDURE. Identical to GOTO SEGMENT except that the user provides a procedure name as the starting point.

ENTRY. Equivalent to GOTO PROCEDURE with an implicit offset to the main entry point of the procedure. This bypasses declarations and nested subroutines, procedures, and functions.

CALL. Equivalent to GOTO ENTRY, plus the current module and location are saved in a logfile. allowing the user to return to this point at a later time. This mimics the call and return mechanism used by a computer.

RETURN. Allows the user to return to a place in the source code that was saved in the logfile as a result of an earlier GOTO CALL.

APPLICATION. Allows the user to select an application program to be displayed instead of a library module.

IDENTIFY

This command displays information regarding identifiers defined in the source code. Three subcommands select different information to display.

VALUE. The identifier map information supplied by the compiler for the selected identifier is displayed in a window. This includes type, class, and location or value.

DEFINITION. The source code containing the definition of an identifier is displayed in a scrollable window.

LOCAL VARS. The identifier map information supplied by the compiler for all the local identifiers in the current procedure is displayed in a scrollable window.

SEARCH

This command finds a specific item or pattern in the current module. Three subcommands determine the search method. Each can be done in a forward or backward direction.

IDENTIFIERS. Finds the next or previous occurrence of an identifier as supplied by the compiler cross-reference table.

TEXT. Searches forward or backward for text matching a pattern, which can include wildcard characters for increased flexibility.

LEVEL. Searches in the required direction for a specific block level. The block level is a function of the BEGIN-END statements in Pascal and SPL. Each BEGIN increments the level number, and each END decrements it.

DISPLAY

This command switches the display between code and supplementary information while retaining the previously displayed information. Seven subcommands select what information to display.

CODE. Returns to the source code display.

PMAP. Displays the procedure map for the current module. This lists procedures with segment offsets, if applicable.

REFERENCE. Displays the current page of the current reference document. Useful documents such as internal specifications are included on the CD-ROM.

LIBRARY/MODULE/APPLICATION. Displays a list of procedures, module/segments, or applications whose names match a pattern.

STACK. Displays the current logfile CALL history.

TOGGLE

This command controls the state of three binary switches.

ABSOLUTE/RELATIVE. Alters the way that code offsets are displayed. They can be ABSOLUTE (using the segment as a base) or RELATIVE (using the procedure as a base).

HEX/OCTAL. Alters the radix of code offsets.

SOURCE ONLY/INNERLIST. Displays source code only or source code interspersed with the corresponding assembly instructions generated for each source line.

PRINT

This command prints information to a printer or file. There are subcommands to control what is printed.

REFERENCE

This command selects a specific document or a location in that document using the table of contents or index.

CONFIGURE

This command is used to customize the program by selecting miscellaneous options for the access program to use. These include display colors, screen size, printer, function keys, and CD-ROM drive location.

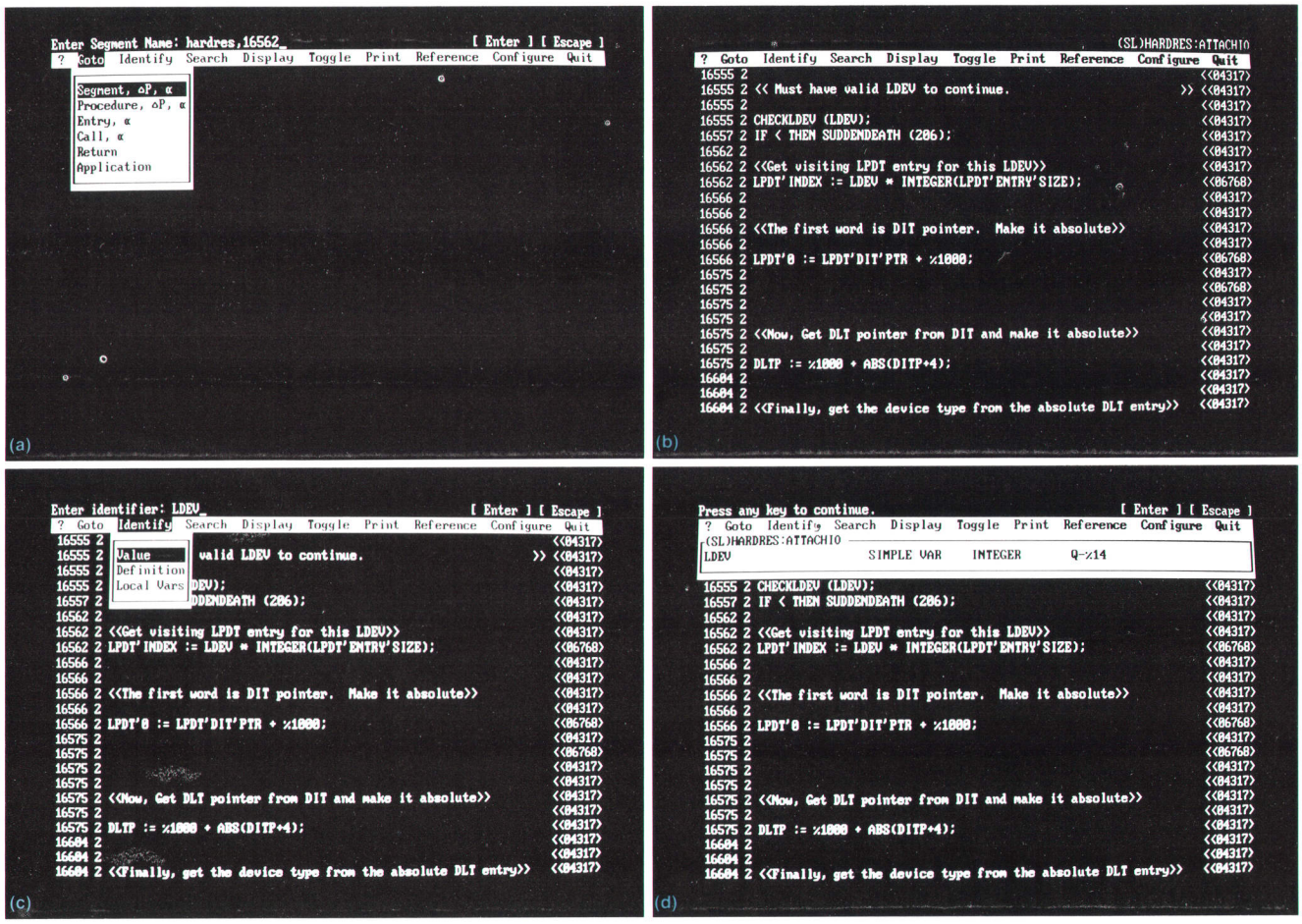


Fig. 5. *Screens from a typical HP Source Reader session. (a) The* GOTO SEGMENT HARDRES *command is entered. (b) Resulting screen. (c) Invoking the* IDENTIFY VALUE *command. (d) Resulting screen.*

HELP

Context sensitive help text is provided to assist with any difficulty using the program. For example, if the user is being prompted for some input, the HELP command displays text that explains the exact nature of the input required. This is most useful for a novice user, but even experienced users may need assistance from time to time with infrequently used features.

QUIT

This command gracefully exits from the program. A special logfile entry is made, saving the current location. This allows the user to issue a GOTO RETURN command the next time the program is run to resume displaying the source code that was being displayed when HP Source Reader was last terminated.

An Example

Fig. 5 shows part of a typical HP Source Reader session. An engineer is trying to locate the source line that aborted the system. From the memory dump the engineer has determined that the code aborted in segment HARDRES at octal offset 16562. The engineer switches from the dump analysis tool to HP Source Reader. Fig. 5a shows the screen after the engineer selects the GOTO SEGMENT command and types the segment name and offset. HP Source Reader locates the source code at that location, resulting in the screen shown in Fig. 5b. The cursor is positioned on the source line corresponding to the return point from the call to SUDDENDEATH—the engineer has found the call that aborted the system.

From the code, it is apparent to the engineer that SUDDENDEATH is called if CHECKLDEV determines that the value of the variable LDEV is invalid. The engineer then needs to examine LDEV in the dump to determine what value it contained when the check failed. The engineer uses the mouse to point to LDEV on the screen, then invokes the IDENTIFY VALUE command. Fig. 5c shows the screen for doing this. HP Source Reader locates the identifier map information for LDEV and displays it in a window as shown in Fig. 5d. The engineer now knows that LDEV is found at location Q-%14, and therefore switches back to the dump analysis tool and examines the value of LDEV found at that location in the memory dump.

Conclusions

HP Source Reader provides substantial increases in productivity based on our personal experience, feedback from

support engineers, and management analysis. The time that it takes an engineer to locate a specific source location has been reduced from several minutes to a few seconds. Further savings are achieved by direct access to supporting information such as identifier maps, assembly code, reference materials, and other sources. Significant cost savings are achieved by the elimination of paper listings. These savings include computer time, consumable items, labor for printing and binding, and storage costs.

HP Source Reader represents an important contribution to HP's commitment to customer satisfaction in support. Local support engineers now have fast access to complete source listings. Previously, maintaining such listings in every HP support office was not cost-effective. Today, more problems are resolved by field support personnel. Customers realize this as improved system availability.

HP Source Reader is now in use in virtually every HP support office around the world. Engineers tell us it is indispensable, and managers at all levels have gone out of their way to report that HP Source Reader has dramatically improved problem resolution time.

HP Source Reader successfully combines new optical media technology with the ease of use and power of the PC. Designed with HP's traditional "next bench" development philosophy, it seems to be developing into the method of choice for MPE system support engineers who analyze memory dumps.

Acknowledgments

We would like to acknowledge Rob Williams, Mark Muntean, Jim Schrempp, and Danny Wong for having the vision to provide the commitment to the project. Gary Robillard deserves the credit for his efforts in coordinating and filtering several versions of MPE, without which several CD-ROMs would not yet have been released.

Correction

In the left column on page 99 of the October 1989 issue, the words "parallel" and "perpendicular" are transposed in equations 3 and 4, Fig. 2, and the associated text. Fig. 2a on page 99 shows reflectivity $R(\theta)$, not reflection coefficient $r(\theta)$ as stated. ($R(\theta) = r^2(\theta)$.) Fig. 2b shows $R^2(\theta)$, which is the fraction of light reflected after two reflections. Also, Brewster's angle θ_B is approximately 61° instead of 59° as shown.

Transmission Line Effects in Testing High-Speed Devices with a High-Performance Test System

The testing of high-speed, high-pin-count ICs that are not designed to drive transmission lines can be a problem, since the tester-to-device interconnection almost always acts like a transmission line. The HP 82000 IC Evaluation System uses a resistive divider technique to test CMOS and other high-speed devices accurately.

by Rainer Plitschka

TODAY'S STATE-OF-THE-ART DIGITAL ASICs (application-specific integrated circuits) are characterized by faster and faster clock rates and signal transition times. In testing these devices, delivering the test signals to the device under test (DUT) and precisely measuring the response of the DUT can be a problem. To maintain signal fidelity, transmission line techniques have to be applied to the tester-to-DUT interconnection.

This paper illustrates how this critical signal path is implemented in the HP 82000 IC Evaluation System to obtain high-precision timing and level measurements even for difficult-to-test CMOS devices. The HP 82000 offers a resistive divider arrangement that provides terminated transmission lines to the inputs and outputs of the DUT. This makes it possible to test low-output-current devices up to their maximum operating frequencies. The HP 82000 tester also offers good threshold accuracy, low minimum detectable signal amplitude, and system software that supports adjustment of the compare thresholds according to the selected divide ratio.

Whether an interconnection between the tester pin electronics and the DUT should be considered a transmission line depends on the interconnection length and the transition time of the driving circuitry. If

$$t_{pd} > t_r/8 \qquad (1)$$

where t_r is the equivalent transition time (0 to 100%) and t_{pd} is the propagation delay (electrical length) of the interconnection, then the interconnection has to be treated as a transmission line.[1] For delays less than 1/8 of the transition time, the interconnection can be considered a lumped element.

Table I shows propagation velocities of signals in different types of transmission lines. Using equation 1 for a typical ECL output or a modern CMOS output with a 20-to-80% transition time of 1 ns, or 1.67 ns for 0 to 100%, and using Table I for signal velocities, we can compute a maximum interconnection length of 1.25 inch (3.1 cm) for a microstrip line. There are no high-pin-count testers that even come close to such a short interconnection length between the pin electronics and the DUT. Therefore, a transmission line model must be used.

Table I
Signal Velocity in Different Transmission Line Media

Type	Velocity
Coax, air	1 ft (30 cm) per ns
Coax, foam-filled	8 in (20 cm) per ns
Microstrip, FR4	6 in (15 cm) per ns

Transmission Line Impedance

Besides signal velocity, the line impedance Z_l is a characteristic parameter of a transmission line. The value of Z_l depends on the line type, geometric factors, and the electrical parameters of the materials used. Table II shows typical values and tolerances. Note that Z_l typically lies within a small range of values, and that quite high tolerances are usual.

Table II
Transmission Line Impedance Characteristics

Line Type	Range of Z_l	Tolerance
Coax, foam-filled	50 to 100 Ω	2 to 10%
Microstrip, FR4	30 to 120 Ω	5 to 20%

The choice of a value for Z_l in a high-speed tester environment is influenced by three major factors. First, the outputs of ECL devices normally are designed to operate at $Z_l = 50\Omega$. However, 25Ω and 100Ω outputs exist.

Second, connecting a capacitance C to the end of a transmission line forms a low-pass filter. This occurs in a tester when a DUT with input capacitance C_{in} is connected to a driver. It also occurs at a comparator input, which has a lumped capacitance C_{lumped} (see Fig. 1). The low-pass filter's step response transition time t_s (10% to 90%) is:

$$t_s = 2.2\tau = 2.2(Z_l C). \quad (2)$$

A signal with transition time t_t at the input to the filter will be slowed down to a transition time of t_{res} at the output:

$$t_{res} = \sqrt{t_s^2 + t_t^2} \quad (3)$$

which adds additional delay at every point of the original transition. For the 50% point this delay is approximated by the factors shown in Table III.

Table III
Delay for the 50% Point of a Transition Caused by Low-Pass Filtering

	$t_s << t_t$	$t_s = t_t$	$t_s >> t_t$
Delay at 50%	$0.7Z_lC$	$0.9Z_lC$	$1.0Z_lC$

As a consequence, the impedance Z_l should be as low as possible, because C (that is, C_{in} or C_{lumped}) is always nonzero.

The third factor influencing the value of Z_l is the required source current capability. To minimize it, Z_l should be as low as possible. To generate a voltage step V_s to propagate along the transmission line, the source has to provide current I_s according to Ohm's law:

$$I_s = V_s/Z_l. \quad (4)$$

This is true for both the tester's driver circuit and the DUT. Proper design of the driver circuit will ensure sufficient drive current. However, some DUT outputs, especially CMOS, cannot provide the current required over the entire range of Z_l values shown in Table II.

As a result of these considerations, a tester in which both accuracy and speed are important will use an impedance Z_l of 50Ω.

Termination Models

To maintain pulse performance, a terminated signal distribution system has to be used. Two methods of performing the termination are possible: parallel and series.

Parallel termination uses a resistor $R_t = Z_l$ at the end of the transmission line, as shown in Fig. 2. At time t = 0, a voltage step V_0 is generated by the the source. The forward wave will see the line as a resistor $R = Z_l$, and therefore $V_1(t = 0) = V_0$. At $t = t_{pd}$ the wave has reached the end of the transmission line, and because $R_t = Z_l$, $V_2(t = t_{pd}) = V_0$. No further reflections occur. The current that must be provided by the source is $I_0 = V_0/Z_l$, and it flows as long as $V_1(t) = V_0$. This model is applicable for ECL outputs. The resistor R_t is connected to −2V.

A version of this paper was originally presented at the IEEE European Test Conference, Paris, 1989.

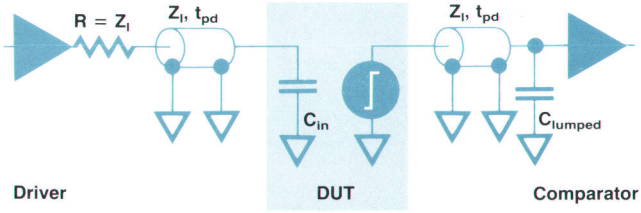


Fig. 1. *DUT interconnection model showing low-pass filters caused by capacitive loadings.*

The series termination method uses a resistor $R_s = Z_l$ in series between the source and the transmission line, as shown in Fig. 3. At time t = 0, a voltage step V_0 is generated by the source. The forward wave will see the line as a resistor $R = Z_l$. Because of voltage splitting between R_s and Z_l, $V_2(t = 0) = V_0/2$. At $t = t_{pd}$ the wave has reached the end of the transmission line, and because of reflection at the open end, $V_3(t = t_{pd}) = 2V_2(t = 0) = V_0$. After $t = 2t_{pd}$, the reflected wave will reach the source side, giving $V_2(t = 2t_{pd}) = V_0$. No further reflections occur, since the source side is terminated. The current I_0 to be provided by the source is $I_0 = V_0/2Z_l$ for the time $2t_{pd}$. This termination model is appropriate for a driver circuit in the tester.

Unterminated Environment

When connecting a source with $R_{out} \neq Z_l$ to a transmission line there is no matching element in the circuitry. This situation arises when a DUT output, such as CMOS or TTL, is connected to a tester channel in which the driver has been set to high impedance and a high-impedance comparator is used. Fig. 4 shows the resulting waveforms for $R_{out} > Z_l$ and $R_{out} < Z_l$. As can be seen, the transmission line mismatch creates a series of pulses that reflect back and forth (ringing). The amplitude and number of steps depend on the magnitude of the mismatch, and the duration

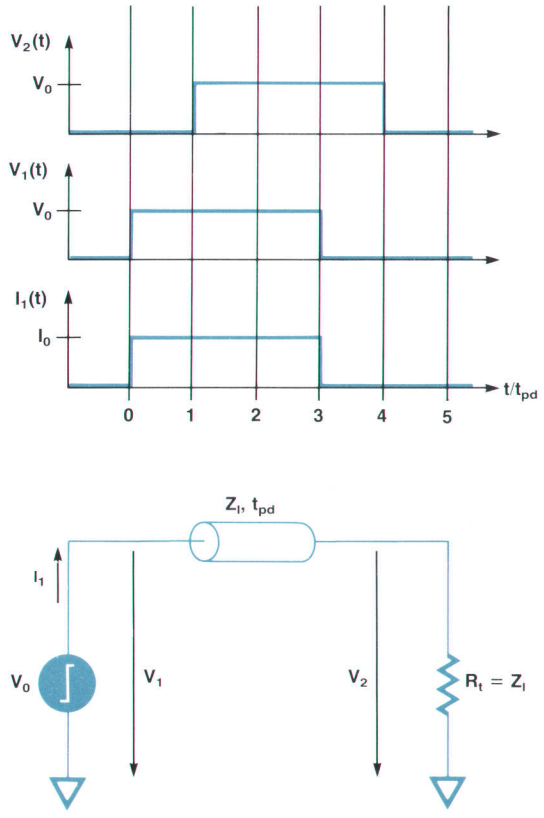


Fig. 2. *Parallel termination model.*

depends on the propagation delay of the line and the number of steps.

Under these conditions, accurate timing and level measurements are not easy.[2] For repeatability of measurement results, the ringing should be completely settled before a measurement is made. Therefore, the device has to be tested at data rates far lower than maximum. Fig. 5 shows the relationship between the maximum possible test frequency and the electrical length of the interconnection for various degrees of mismatch (i.e., different device impedances), assuming two different settling criteria. One of the two curves assumes that the waveform is allowed to settle within 10% of its final value before an opposite transition can be started. The other assumes 1%.

Tester Parasitics

The basic elements of a tester's pin electronics are a transmission line, a driver, and a comparator. There is normally also an ac/dc switch for performing dc measurements. This switch, implemented using a relay, can cause problems. However, by proper selection of the relay type and careful design, the transmission line impedance can be maintained without significant parasitics.

For stimulating the DUT, the driver output signal is fed to the pin. Because of the input capacitance of the fixturing and the pin capacitance (C_{in}), the driver transitions will be slowed. This causes a delay as discussed above. Equations 2 and 3 and Table III can be used to calculate the delay. Also, because of input leakage currents flowing through the driver's source impedance ($R = Z_l = 50\Omega$), the driver levels will change. For example, for an ECL device with $I_{ih} = 500\ \mu A$ typically, there will be a voltage drop of $I_{ih}Z_l = 25$ mV.* Further problems will not occur.

For receiving DUT data, the comparator can be used in two different modes (Fig. 6): high-impedance (high-Z) and terminated (parallel).

In the high-Z mode, the driver is switched to high impedance, resulting in a capacitance C_{lumped} formed by the parasitics of the amplifier's switched-off transistors. Assuming a value of 3 pF for the compare chip (C_c) and 20 pF for the driver (C_d), $C_{lumped} = C_c + C_d = 23$ pF and the resulting step response time for the comparator input voltage is 2 ns.

In the terminated mode, the tester's driver is used for termination, eliminating the capacitance C_d. Only the comparator's input capacitance will limit the bandwidth, giving a step response time of $2.2(C_cZ_l)/2 = 165$ ps. This value is equivalent to an analog input bandwidth of 2 GHz.

Fig. 7 shows the step response as a shmoo plot. The stimulus was a pulse with a transition time of $t_r = 200$ ps from an HP 8131A Pulse Generator. The measured value of the 10-to-90% transition is $t_m = 275$ ps. The resulting intrinsic transition time t_i of the comparator is therefore:

$$t_i = \sqrt{t_m^2 - t_r^2} = 165 \text{ ps.}$$

*In this paper, the subscripts o and i indicate output and input parameters, respectively, and the subscripts h and l indicate high and low logic levels, respectively. Subscripts s, d, and g indicate the source, drain, and gate, respectively, of a field-effect transistor.

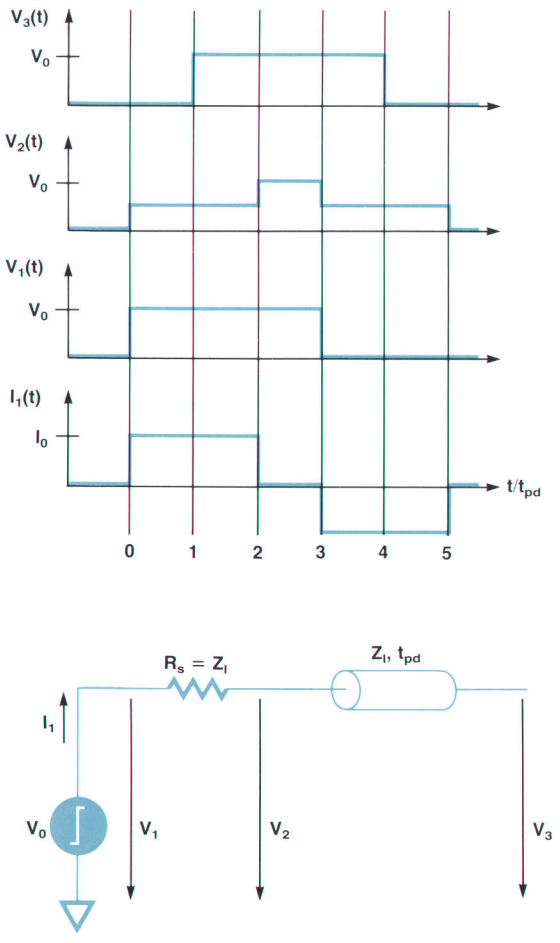


Fig. 3. *Series termination model.*

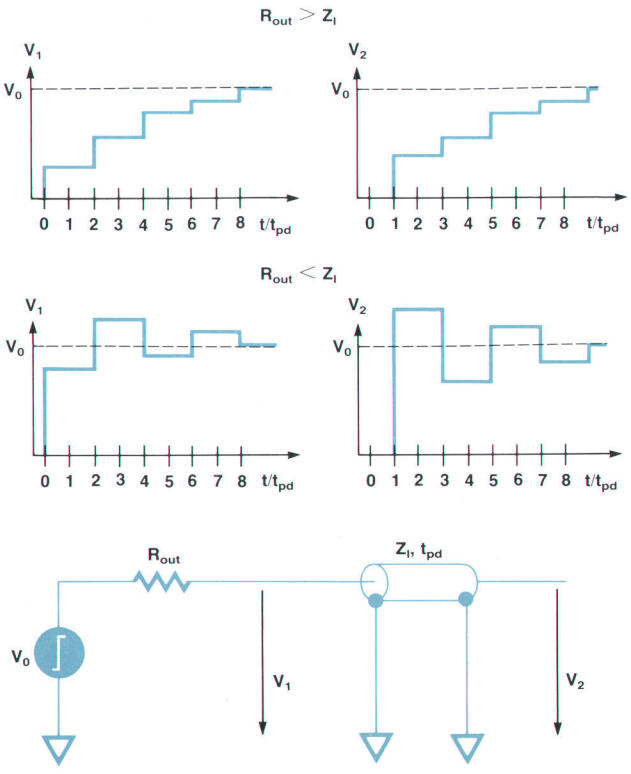


Fig. 4. *Unterminated model.*

Interfacing CMOS Devices

CMOS devices are usually unable to drive transmission lines. The output impedance of CMOS devices does not match typical transmission line impedances, and static power dissipation, which occurs when driving a terminated transmission line, may damage a CMOS device.

Fig. 8 shows the operating characteristics of a CMOS output buffer cell.[3] The specified dc parameters V_{ohmin} at I_{oh} and V_{olmax} at I_{ol} are marked.

The output resistance is not linear. For high V_{ds}, the cell acts as a current source. For low V_{ds}, it is a voltage source with a low resistance. The large-signal output resistance R_{out} can be defined for either high or low output by:

$$R_{outl} = V_{ds}/I_d$$
$$R_{outh} = (V_{dd} - V_{ds})/I_d \quad (5)$$

where V_{ds} and I_d are corresponding values on the curves.

The worst-case output resistance, R_{maxl} or R_{maxh}, is defined when $V_{ds} = V_{ohmin}$ or V_{olmax} and $I_d = I_{oh}$ or I_{ol}. Note that there are major differences between typical and worst-case resistances. The resistance also varies with the operating temperature.

A CMOS output connected to a capacitance C will perform as shown in Fig. 9. Assume that the source FET is turned on at $t = 0$ with $V_{ds} = V_{dd}$. The capacitor is charged with constant current, resulting in a linear ramping voltage. As the capacitor voltage increases, V_{ds} and the output resistance decrease. This decreases current flow into the capacitor, which slows the voltage ramp. The resulting capacitor voltage waveform resembles an exponential curve.

The performance of a CMOS output driving a resistive load R_{load} connected to a voltage source V_{load} is shown in Fig. 10. The output voltage can be obtained by drawing a line defined by $V_{ds} = V_{load}$ and $I_d = V_{load}/R_{load}$. The intersections with the FET characteristics define the output voltage and current for source and sink operation. The transition times depend only on the internal switching. Loading to the dc specifications can be obtained by using values for R_{load} and V_{load} calculated as follows:

$$R_{load} = (V_{ohmin} - V_{olmax})/(|I_{oh}| + |I_{ol}|)$$
$$V_{load} = \frac{V_{ohmin}|I_{ol}| + V_{olmax}|I_{oh}|}{|I_{oh}| + |I_{ol}|}. \quad (6)$$

A worst-case device loaded to I_{oh} or I_{ol} will have an output voltage of V_{ohmin} or V_{olmax}, respectively. A typical device will have an output voltage greater than V_{ohmin} or less than V_{olmax}, respectively.

CMOS Driving a Transmission Line

Connection of a CMOS output directly to an open-ended transmission line is shown in Fig. 11. Assume that $R_{out} > Z_l$ and a positive transition occurs at $t = 0$. At $t = t_{pd}$ the device output will correspond to the intersection of the FET's characteristic and the load line defined by $V_{ds} = V_{dd}$ and $I_d = V_{dd}/Z_l$. Calculating the device's output resistance using equation 5, the output waveform behavior can be predicted as discussed above for the unterminated environment. Because of the nonlinear output resistance, slightly different waveforms may occur depending on the actual V_{ds} and I_d. When R_{out} is less than Z_l, the second step on the source side may be higher than V_{dd}. If clamping diodes are included between the output and V_{dd}, this reflection can be reduced and further reflections will be inverted.

For CMOS outputs with $R_{out} < Z_l$, termination can be achieved by adding a resistor R_s between the output and the transmission line. The value should be:

$$R_s = Z_l - R_{out}.$$

This is the series termination model, which gives correct pulse performance. This method has been suggested in the

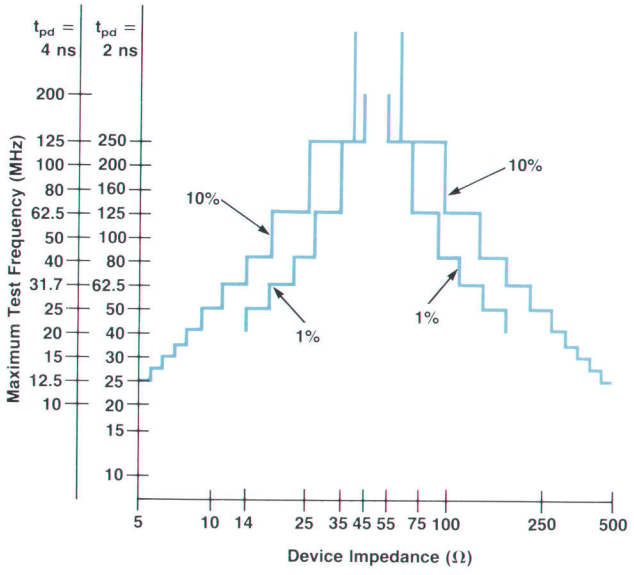


Fig. 5. *Maximum test frequency in an unterminated environment. Any DUT switching time is assumed to be zero and any transition time is assumed to be zero.*

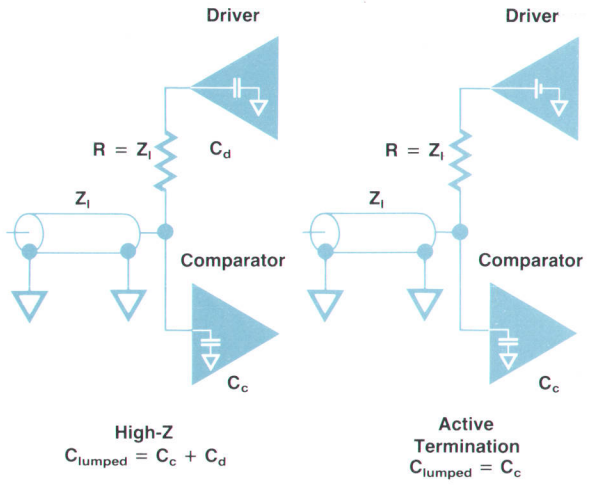


Fig. C. *Operating modes of the receiver path.*

past.[4] However, its practical applicability is limited, because the output resistance for positive and negative transitions is generally not equal, and the output resistance changes from sample to sample and is not stable with temperature.

The Resistive Divider Solution

The resistive divider provides a solution to the problem of embedding a CMOS device in a transmission line environment. This technique is implemented in the HP 82000 IC Evaluation System.

The operating principle of the resistive divider is to apply a definable dc load to the DUT. Signal fidelity is maintained because the signal is fed into a parallel-terminated system; therefore, no reflections occur.

Fig. 12 shows a schematic diagram of the resistive divider. The resistor R_t is built into the tester. The resistor R_s is selected by the user to give an appropriate divide ratio for the particular DUT. R_s is installed on the DUT board, which interfaces the DUT to the tester and is different for each DUT. The user then tells the HP 82000 software what the divide ratio is. The termination voltage V_t in Fig. 12 is also selected by the user.

Besides providing a terminated transmission line environment, the resistive divider puts only a very small capacitive load on the DUT (shown as C_{par} in Fig. 12). A value as low as 2 pF can be obtained if R_s is close to the DUT pin. This is possible using ceramic blade probes with printed resistors. For high-pin-count devices (up to 512 pins), the tester's DUT board can be laid out with easily installable resistors, keeping parasitics below 10 pF.

The length of transmission line between the DUT and the comparator does not affect the capacitive and resistive loading on the DUT. The termination is done by the tester's driver, which is part of the I/O channel. Therefore, the lumped capacitance that occurs if the driver is switched to high impedance is eliminated. This ensures a wide bandwidth for the compare path as discussed earlier.

The DUT output levels detected will be reduced by the divide ratio. The resulting comparator input voltages can be calculated by:

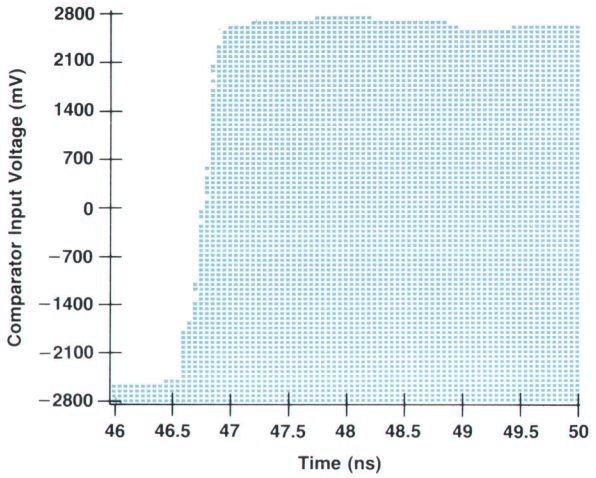


Fig. 7. *Shmoo plot of the step response at the tester input for an input signal with t_r = 200 ps.*

$$V_{cmp} = \frac{V_o R_t + V_t R_s}{R_s + R_t} \qquad (7)$$

where V_o is the actual high or low output voltage under the defined load. This equation can also be used for calculating the appropriate threshold setting. For ease of use, this calculation is embedded in the HP 82000 tester software, so that a user always thinks in terms of noncompressed signals.

Resistive Divider Parameters

The selectable parameters of the divider are R_s and V_t. There are several choices for defining the DUT load. Device loading according to dc specifications is normally the best choice. The DUT's maximum power consumption will never be exceeded and throughput is improved. If the ac test is performed with the specified dc loading, the need for further dc fanout measurements is eliminated.

Dc loading specifications can be converted to resistive divider parameters using:

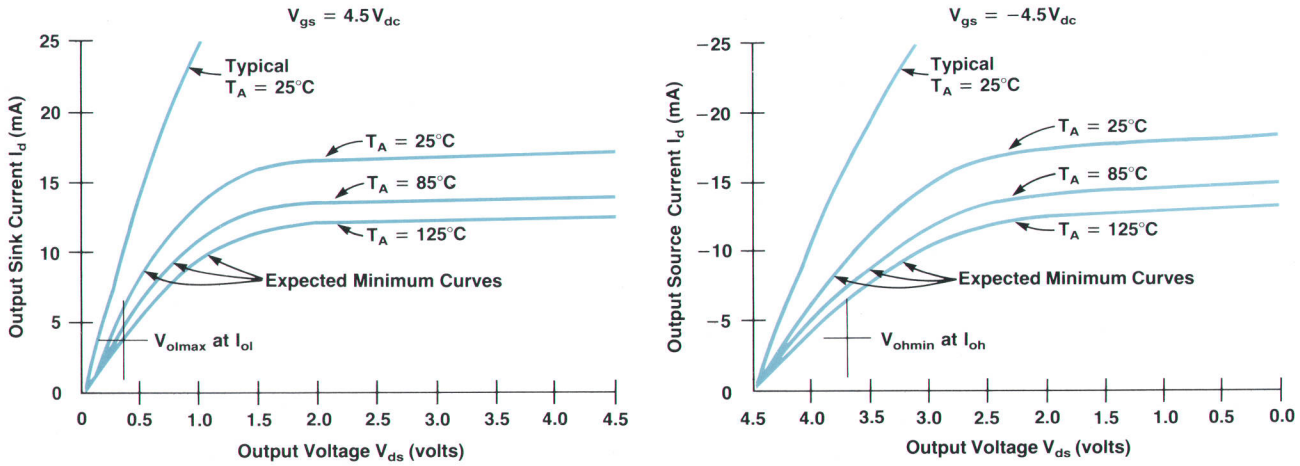


Fig. 8. *Output characteristics (I_d vs V_{ds}) of a CMOS output buffer, where I_d is the drain current and V_{ds} is the drain-to-source voltage across the FET.*

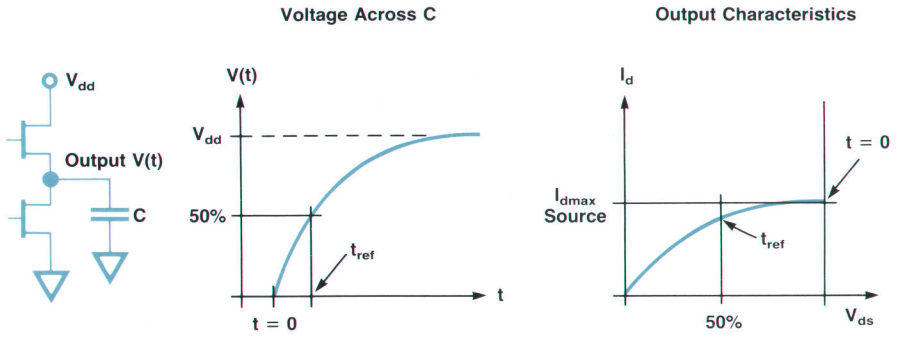


Fig. 9. *CMOS output driving a capacitive load.*

$$R_s = R_{load} - R_t = (V_{oh} - V_{ol})/(|I_{oh}| + |I_{ol}|) - R_t \quad (8)$$

$$V_t = V_{load} = \frac{V_{oh}|I_{ol}| + V_{ol}|I_{oh}|}{|I_{oh}| + |I_{ol}|}$$

Special ac loads are defined for timing measurements as shown in Fig. 13. These ac loads can be converted to resistive divider parameters by the Thevenin equations:

$$R_s = 1/(1/R_1 + 1/R_2) - R_t$$
$$V_t = V_{dd}R_2/(R_1 + R_2). \quad (9)$$

There are situations where the values of R_s and V_t calculated using equation 8 cannot be used. Changing the loading will change the output levels. The changed values can be obtained from the output characteristic curves. The actual values are defined by the intersection of the load line with the FET curve. The worst-case values can be obtained from the intersection of the load line and the worst-case output resistance line, as shown in Fig. 14. These modified levels can be calculated using:

$$V_{ol}' = (V_tR_{maxl} + V_{opl}R_{load})/(R_{maxl} + R_{load})$$
$$I_{ol}' = (V_t - V_{opl})/(R_{maxl} + R_{load}) \quad (10)$$
$$V_{oh}' = (V_tR_{maxh} + V_{oph}R_{load})/(R_{maxh} + R_{load})$$
$$I_{oh}' = (V_t - V_{oph})/(R_{maxh} + R_{load})$$

where V_{opl} and V_{oph} are the low and high level open-circuit output voltages, and the values for worst-case output resistance are given by equation 5.

Modified loading may result in higher power dissipation for one of the output levels. It is recommended that power consumption be checked using:

$$P_{dl} = V_{ol}'I_{ol}'$$
$$P_{dh} = (V_{dd} - V_{oh}')I_{oh}'. \quad (11)$$

Practical tests have shown no problems as long as the level change caused is less than 500 mV.

To measure the reduced output signals resulting from the resistive divider, the comparator must be designed to detect small amplitudes. Two parameters affect this ability: comparator hysteresis and open-loop gain. The hysteresis is a positive feedback effect to ensure the comparator's stability. The open-loop gain is the comparator's amplifying factor for small signals, and is frequency dependent. Both parameters affect the finite voltage swing (overdrive) around the threshold that has to be applied to the comparator input to obtain output switching (see Fig. 15).

A high-performance comparator design will ensure that the necessary overdrive will be constant up to the maximum data rate. Smaller pulses can be detected as long as sufficient overdrive is applied. For detection of a single transition, the value for the flat section of Fig. 15 applies, and the limiting element is the input signal bandwidth.

With a value of ±20 mV for the overdrive, and assuming a dc accuracy of ±10 mV, the signal's amplitude at the comparator input should be greater than 60 mV. A TTL-compatible output will generate a swing of at least 2V. Within a 50Ω environment, this allows a maximum divide ratio r_{max} and a maximum value for R_s of:

$$r_{max} = 33$$
$$R_{smax} = 1600\Omega. \quad (12)$$

This means that devices having output currents

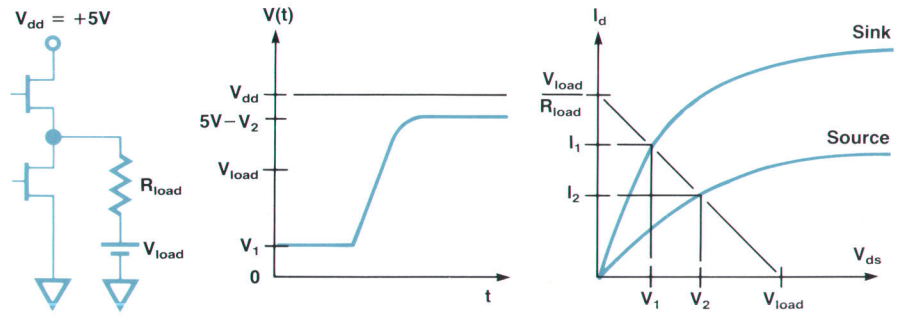


Fig. 10. *CMOS output driving a resistive load.*

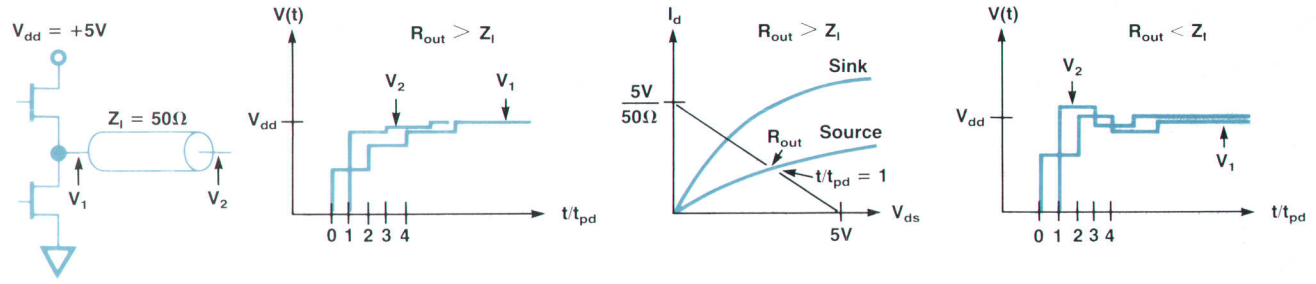


Fig. 11. *CMOS output driving an open-ended transmission line.*

$|I_{oh}| + |I_{ol}| \geq 2V/1600\Omega \approx 1.2$ mA at TTL output levels can be tested.

DUT Power Dissipation

At higher test frequencies, the resistive divider results in less DUT power consumption than a capacitive load, as shown in Fig. 16.

I/O Pin Considerations

The resistive divider is applicable to I/O pins, with some additional considerations.

The tester's driver can generate a two-level signal. These levels should be set according to the DUT's input requirements, that is, $V_l \leq V_{ilmax}$, $V_h \geq V_{ihmin}$. When receiving signals from the DUT, one of these levels has to be used for termination. This may mean that the calculated V_t does not match the driving requirements. V_t and R_s should be set to:

$$V_t \geq V_{ihmin} \text{ if } I_{ol} \geq I_{oh}$$

$$R_s = (V_{ohmin} - V_t)/I_{oh} \qquad (13)$$

or $$V_t \leq V_{ilmax} \text{ if } I_{ol} \leq I_{oh}$$

$$R_s = (V_t - V_{olmax})/I_{ol}.$$

The value of R_s is modified to ensure that none of the output states will be loaded more than specified. This will occur if only V_t is modified. Note that one level remains less loaded. If DUT power dissipation is not critical, the value of R_s need not be modified.

The device can be stimulated via the series resistor. CMOS normally has negligible input current, so no level errors occur. The input capacitance and R_s form a low-pass filter, which limits the data rate and causes a delay at the 50% point on the transition:

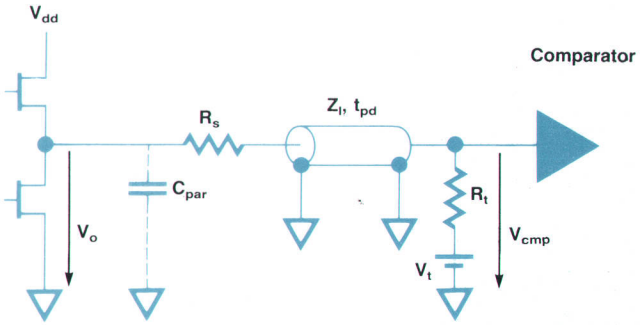


Fig. 12. *Resistive divider model.*

$$\text{Data rate} = 2.3(Z_l + R_s)(C_{in} + C_{par})$$

$$\text{Delay at 50\%} = 1.0(Z_l + R_s)(C_{in} + C_{par}). \qquad (14)$$

Table IV shows values for the maximum data rate obtainable and the corresponding delay for the 50% point. Also shown is the obtainable accuracy assuming a variation of 1 pF for the capacitance.

Table IV
Low-Pass Filter Effects on Drive Signal
(Driver transition time = 2 ns. $C_{in} + C_{par}$ = 10 pF.)

$R_s(\Omega)$	τ	Delay at 50%	Delta Delay at 1 pF
100	1.5 ns	1.3 ns	135 ps
200	2.5 ns	2.2 ns	225 ps
500	5.5 ns	5.5 ns	550 ps
1000	10.5 ns	11.5 ns	1150 ps

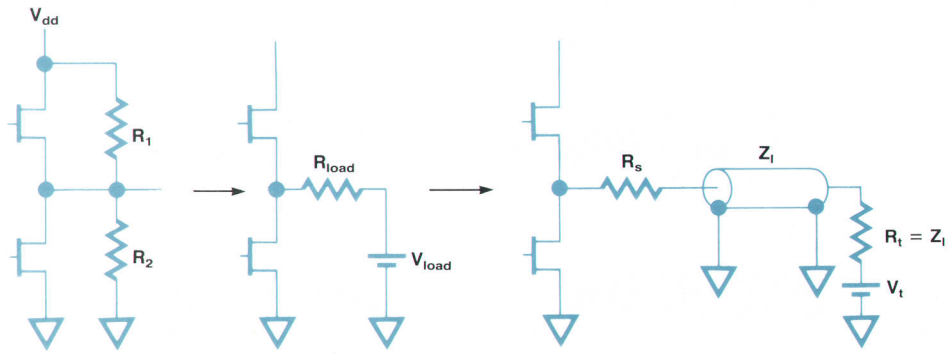


Fig. 13. *Transformation of a desired load to resistive divider parameters.*

CMOS Device Measurement Results

HCMOS Example

Fig. 1 shows the signal obtained at the HP 82000 tester comparator input from an HCMOS output. Switching characteristics (at V_{dd} = 4.5V, T_a = 25°C, load capacitance C_l = 50 pF) are: transition time ≤ 8 ns, propagation delay ≤ 38 ns.

For comparison, Fig. 2 shows the signal obtained with the same output connected to an open-ended transmission line. Significant influences are introduced by the transmission line environment.

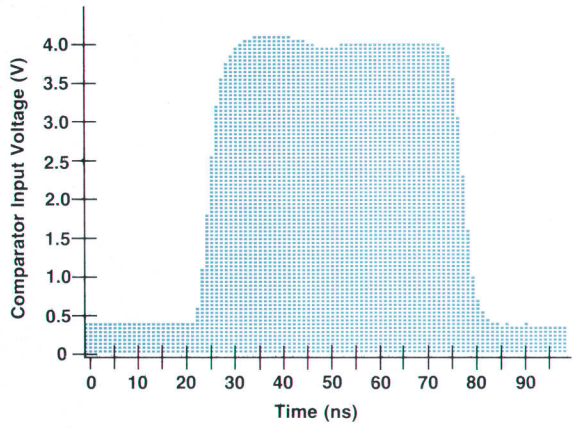


Fig. 1. *Shmoo plot of comparator input signal from an HCMOS output buffer with 4-mA source/sink capability, loaded by a resistive divider with parameters R_s = 200 ohms, V_t = 2.2V.*

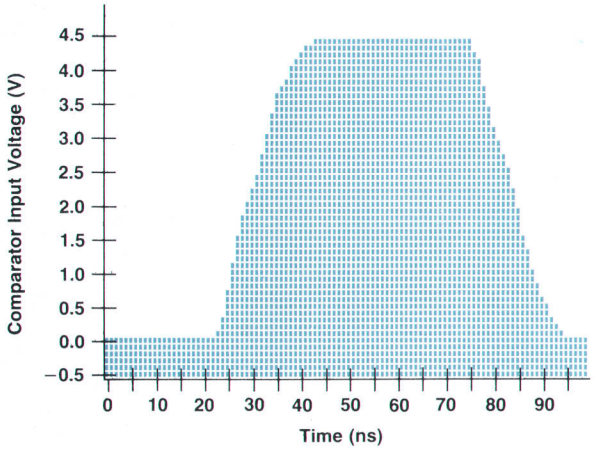


Fig. 2. *Shmoo plot of comparator input signal from an HCMOS output buffer with 4-mA source/sink capability, loaded by an open-ended transmission line (t_{pd} = 3 ns) with the comparator in high-Z mode (lumped capacitance = 23 pF).*

CMOS 14000 Family Example

Fig. 3 shows the signal obtained at the comparator input from a CMOS 14000 family output. Switching characteristics (at V_{dd} = 5V, T_a = 25°C, C_l = 50 pF) are: transition time ≤ 33 ns + 1.35 ns/pF, propagation delay ≤80 ns + 0.9 ns/pF. To show the comparator's sensitivity, the waveform is not back-calculated according to equation 7 of the accompanying article (that is, V_{cmp} is shown, not V_o). Such a calculation would result in values of 3.97V for the high level and 1.14V for the low level.

For comparison, Fig. 4 shows the signal obtained with the same output connected to an open-ended transmission line. Because of the slow transitions, the transmission line acts as capacitive loading.

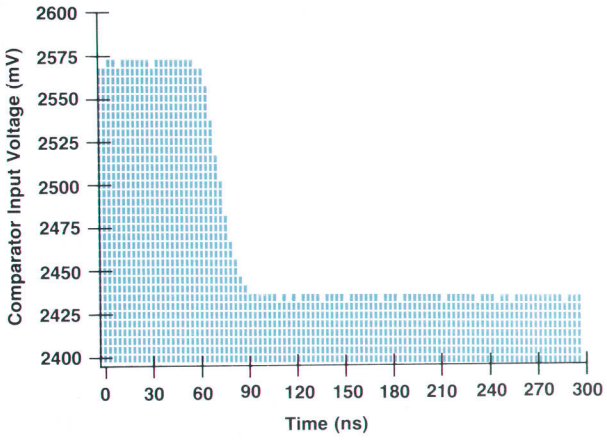


Fig. 3. *CMOS MC14000 family output signal (V_{oh} = 2.5V at I_{oh} = 2.1 mA, V_{ol} = 0.4V at I_{ol} = 0.44 mA) with resistive divider parameters R_s = 1000Ω, V_t = 2.5V.*

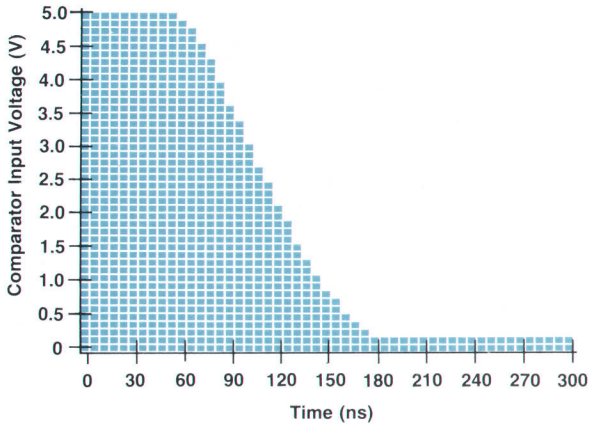


Fig. 4. *CMOS MC14000 family output signal (V_{oh} = 2.5V at I_{oh} = 2.1 mA, V_{ol} = 0.4V at I_{ol} = 0.44mA) with open-ended transmission line (t_{pd} = 3 ns), comparator in high-Z mode (lumped capacitance = 23 pF, resulting in a load of 50 pF total).*

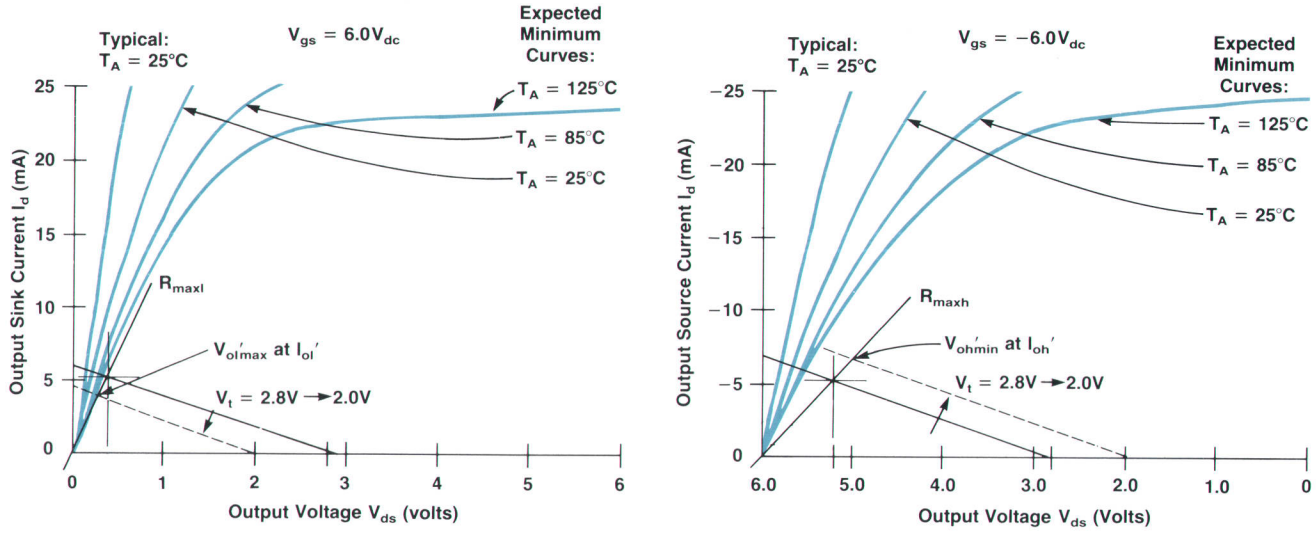


Fig. 14. *Loading resulting from modifying V_t.*

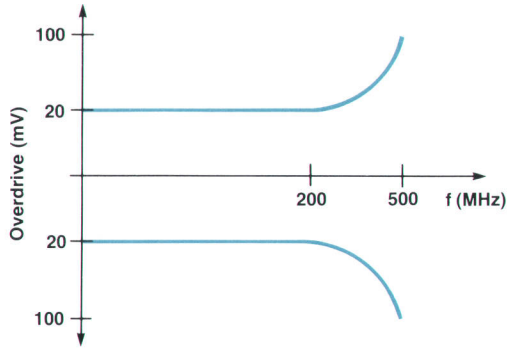


Fig. 15. *Comparator overdrive as a function of data rate.*

DC Accuracy with Resistive Divider

For ease of use, the tester's software takes care of the appropriate calculations of the user's comparator thresholds. This is done using equation 7.

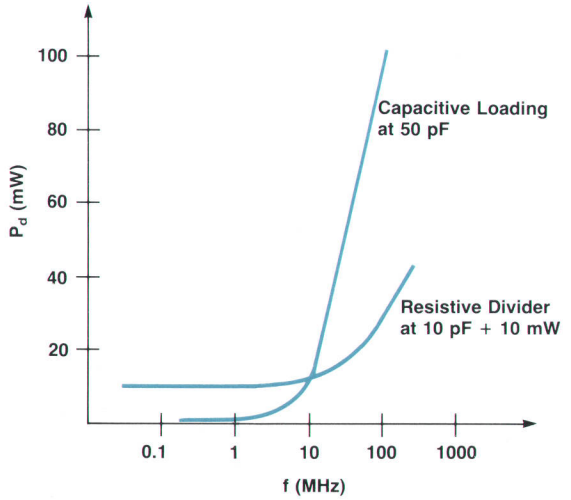


Fig. 16. *Device under test power dissipation as a function of frequency for capacitive loading at 50 pF and for a resistive divider with 10-mW dc loading + 10-pF capacitance.*

Thinking in terms of the noncompressed thresholds will affect the dc accuracy. There are four sources of error in setting the desired comparison threshold:

- Termination source error: dV_t (mV)
- Comparator threshold error: dV_{th} (mV)
- Tolerance on R_s: dR_s (%)
- Tolerance on R_t: dR_t (%).

The total accuracy for the desired threshold (dV_p) can be calculated as:

$$dV_p = rdV_{th} + (r-1)dV_t + (1 - 1/r)(V_{th} - V_t)(dR_t - dR_s) \quad (15)$$

where r is the divide factor: $r = (R_s + R_t)/R_t$.

Using 1% resistors and assuming 10-mV basic accuracy for the threshold and termination voltages, an accuracy $dV_p \leq 2r(10\ mV)$ can be obtained.

DC Measurement Capability

When the loading on the DUT pin matches the dc specifications, further fanout measurements are not necessary, but can be made anyway. The presence of R_s will cause a voltage drop when a load current I_f is forced at the output (Fig. 17). It is necessary to consider the drop when program-

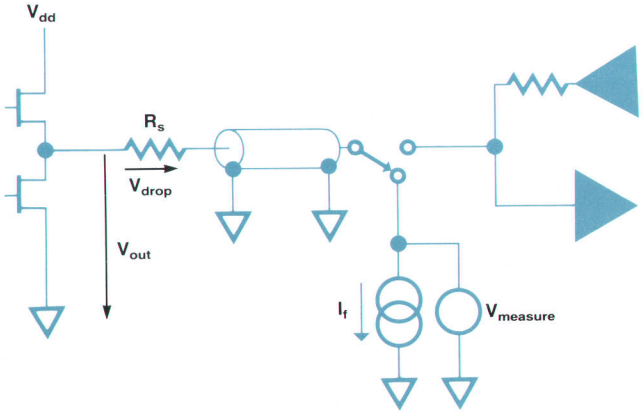


Fig. 17. *Dc measurement path using the resistive divider.*

ming the compliance voltage of the tester's parametric measurement units (PMU). Since R_s and the forced current are known, the actual output level can easily be calculated with sufficient accuracy. It is:

$$V_{out} = V_{measure} - V_{drop} = V_{measure} - R_s I_f. \quad (16)$$

For best results, 0.1% resistors are recommended for R_s.

Summary

In the HP 82000 IC Evaluation System, the resistive divider method offers advantages in operating speed and measurement accuracy. The method has its restrictions and does not ensure testability of every DUT.

References

1. D. Royle, "Rules tell whether interconnections act like transmission lines," *EDN*, June 23, 1988, pp. 131-136.
2. M.R. Barber, "Timing measurements on CMOS VLSI devices designed to drive TTL loads," *International Test Conference 1986*, Paper 4.4.
3. *High-Speed CMOS*, Volume 3, Motorola Semiconductor Corporation, 1984.
4. G.C. Cox, "Transmission line testing of CMOS," *International Test Conference 1987*, Paper 20.1.

HEWLETT-PACKARD JOURNAL

Volume 40 January 1989 through December 1989

Hewlett-Packard Company, 3200 Hillview Avenue, Palo Alto, California 94304 U.S.A.
Hewlett-Packard Marcom Operations Europe, P.O. Box 529, 1180 AM Amstelveen, The Netherlands
Hewlett-Packard (Canada) Ltd., 6877 Goreway Drive, Mississauga, Ontario L4V 1M8 Canada
Yokogawa-Hewlett-Packard Ltd., Suginami-ku, Tokyo 168 Japan

PART 1: Chronological Index

June 1989

A Data Base for Real-Time Applications and Environments, *Feyzi Fatehi, Cynthia Givens, Le T. Hong, Michael R. Light, Ching-Chao Liu, and Michael J. Wright*

New Midrange Members of the Hewlett-Packard Precision Architecture Computer Family, *Thomas O. Meyer, Russell C. Brockmann, Jeffrey G. Hargis, John Keller, and Floyd E. Moore*

Double-Sided Surface Mount Process

Data Compression in a Half-Inch Reel-to-Reel Tape Drive, *Mark J. Bianchi, Jeffery J. Kato, and David J. Van Maren*

Maximizing Tape Capacity by Super-Blocking, *David J. Van Maren, Mark J. Bianchi, and Jeffery J. Kato*

High-Speed Lightwave Component Analysis, *Roger W. Wong, Paul Hernday, Michael G. Hart, and Geraldine A. Conrad*

OTDR versus OFDR

Design and Operation of High-Frequency Lightwave Sources and Receivers, *Robert D. Albin, Kent W. Leyde, Rollin F. Rawson, and Kenneth W. Shaughnessy*

High-Speed PIN Infrared Photodetectors for HP Lightwave Receivers

Videoscope: A Nonintrusive Test Tool for Personal Computers, *Myron R. Tuttle and Danny Low*

Videoscope Signature Analyzer Operation

Neural Data Structures: Programming with Neurons, *J. Barry Shackleford*

A New 2D Simulation Model of Electromigration, *Paul J. Marcoux, Paul P. Merchant, Vladimir Naroditsky, and Wulf D. Rehder*

August 1989

An Overview of the HP NewWave Environment, *Ian J. Fuller*

An Object-Based User Interface for the HP NewWave Environment, *Peter S. Showman*

The NewWave Object Management Facility, *John A. Dysart*

The NewWave Office, *Beatrice Lam, Scott A. Hanson, and Anthony J. Day*

Agents and the HP NewWave Application Program Interface, *Glenn R. Stearns*

AI Principles in the Design of the NewWave Agent and API

An Extensible Agent Task Language, *Barbara B. Packard and Charles H. Whelan*

A NewWave Task Language Example

The HP NewWave Environment Help Facility, *Vicky Spilman and Eugene J. Wong*

NewWave Computer-Based Training Development Facility, *Lawrence A. Lynch-Freshner, R. Thomas Watson, Brian B. Egan, and John J. Jencek*

Encapsulation of Applications in the NewWave Environment, *William M. Crow*

Mechanical Design of a New Quarter-Inch Cartridge Tape Drive, *Andrew D. Topham*

Reliability Assessment of a Quarter-Inch Cartridge Tape Drive, *David Gills*

Use of Structured Methods for Real-Time Peripheral Firmware, *Paul F. Bartlett, Paul F. Robinson, Tracey A. Hains, and Mark J. Simms*

Product Development Using Object-Oriented Software Technology, *Thomas F. Kraemer*

Objective-C Coding Example

Object-Oriented Life Cycles

October 1989

40 Years of Chronicling Technical Achievement, *Charles L. Leath*

A Modular Family of High-Performance Signal Generators, *Michael D. McNamee and David L. Platt*

Firmware Development for Modular Instrumentation, *Kerwin D. Kanago, Mark A. Stambaugh, and Brian D. Watkins*

RF Signal Generator Single-Loop Frequency Synthesis, Phase Noise Reduction, and Frequency Modulation, *Brad E. Andersen and Earl C. Herleikson*

Fractional-N Synthesis Module

Delay Line Discriminators and Frequency-Locked Loops

Design Considerations in a Fast Hopping Voltage-Controlled Oscillator, *Barton L. McJunkin and David M. Hoover*

High-Spectral-Purity Frequency Synthesis in a Microwave Signal Generator, *James B. Summers and Douglas R. Snook*

Microwave Signal Generator Output System Design, *Steve R. Fried, Keith L. Fries, and John M. Sims*

"Packageless" Microcircuits

Design of a High-Performance Pulse Modulation System, *Douglas R. Snook and G. Stephen Curtis*

Reducing Radiated Emissions in the Performance Signal Generator Family, *Larry R. Wright and Donald T. Borowski*

Processing and Passivation Techniques for Fabrication of High-Speed InP/InGaAs/InP Mesa Photodetectors, *Susan R. Sloan*

Providing Programmers with a Driver Debug Technique, *Eve M. Tanner*

HP-UX Object Module Structure

Identifying Useful HP-UX Debug Records

Solder Joint Inspection Using Laser Doppler Vibrometry, *Catherine A. Keely*

Laser Doppler Vibrometry

A Model for HP-UX Shared Libraries Using Shared Memory on HP Precision Architecture Computers, *Anastasia M. Martelli*

User-Centered Application Definition: A Methodology and Case Study, *Lucy M. Berlin*

Interviewing Techniques

Storyboarding Techniques

Partially Reflective Light Guides for Optoelectronics Applications, *Carolyn F. Jones*

December 1989

System Design for Compatibility of a High-Performance Graphics Library and the X Window System, *Kenneth H. Bronstein, David J. Sweetser, and William R. Yoder*

The Starbase Graphics Package

The X Window System

Managing and Sharing Display Objects in the Starbase/X11 Merge System, *James R. Andreas, Robert C. Cline, and Courtney Loomis*

Sharing Access to Display Resources in the Starbase/X11 Merge System, *Jeff R. Boyton, Sankar L. Chakrabarti, Steven P. Hiebert, John J. Lang, Jens R. Owen, Keith A. Marchington, Peter R. Robinson, Michael H. Stroyan, and John A. Waitz*

Sharing Overlay and Image Planes in the Starbase/X11 Merge System, *Steven P. Hiebert, John J. Lang, and Keith A. Marchington*

Sharing Input Devices in the Starbase/X11 Merge System, *Ian A. Elliot and George M. Sachs*

X Input Protocol and X Input Extensions

Sharing Testing Responsibilities in the Starbase/X11 Merge System, *John M. Brown and Thomas J. Gilg*

A Compiled Source Access System Using CD-ROM and Personal Computers, *B. David Cathell, Michael B. Kalstein, and Stephen J. Pearce*

Transmission Line Effects in Testing High-Speed Devices with a High-Performance Test System, *Rainer Plitschka*

CMOS Device Measurement Results

Custom VLSI in the 3D Graphics Pipeline, *Larry J. Thayer*

Global Illumination Modeling Using Radiosity, *David A. Burgoon*

PART 2: Subject Index

P

Q

R

S

T

PART 3: Product Index

PART 4: Author Index

Custom VLSI in the 3D Graphics Pipeline

VLSI transform engine, z-cache, and pixel processor chips widen bottlenecks in the pipeline to allow the HP 9000 Series 300 and 800 TurboSRX graphics subsystem to deliver enhanced performance compared to the earlier SRX design.

by Larry J. Thayer

PRODUCTS FOR DISPLAYING 3D GRAPHICS on engineering workstations have been appearing at an ever-increasing rate over the last few years. Products of each succeeding generation are much more interactive and have significantly more capabilities than earlier ones. Fueling the fast-paced change are new algorithms, better architectures, and most important, advances in VLSI (very large-scale integration) processing and design.

Within HP, perhaps the first use of a custom VLSI chip for computer graphics applications was in a graphics display for a desktop computer introduced in 1981. The chip accelerated vector drawing on HP 9845B Computer displays. Our first 3D product, the HP 98700A, introduced in 1985, drew fast wireframe images with the aid of special commercially available video RAM chips. These chips allowed the raster display to be refreshed at the same time the image was changing.

HP's first solids modeling graphics subsystem, the HP 9000 Series 300 and 800 SRX, was introduced in 1986. It uses a proprietary HP process (NMOS-III) to build chips for floating-point operations (essential for fast 3D graphics) and for the scan conversion process (polygon and vector drawing).[1] Another proprietary process (LTCMOS) is used for a chip that caches pixels, thus allowing multiple pixels to be changed per RAM cycle.[2] The upgrade system for the SRX, the TurboSRX, introduced in 1988, uses even more VLSI for increased performance and functionality.

Custom VLSI is the technology of choice for producing interactive 3D graphics for several reasons:

- VLSI devices are a capable source of the very high computation rates needed for fast, interactive graphics. (The scan converter chip used in both the SRX and the TurboSRX is capable of performing over 300 million additions per second.)
- Data flow is pipelined, with each point in the pipeline having a particular function. VLSI chips can be tailored to each function.
- The low-cost potential provided by large-scale integration makes interactive 3D graphics capability available in a workstation that an engineer can afford.

This article describes how the 3D graphics pipeline of the SRX was analyzed, and how custom VLSI was used in the next-generation product, the TurboSRX, to improve the overall graphics performance.

Pipeline Stages

Graphics workstations contain a data pipeline for displaying user graphics data bases (see Fig. 1). The source data is stored in the host system memory, typically in a display list format. This list is simply a file containing a hierarchical list of the graphics primitives needed to draw the image. First in the pipeline is the system CPU, which reads the display list and sends commands to the graphics subsystem. Using the main system CPU for display list processing minimizes system cost and allows the size of the display list to be limited only by the virtual memory space of the processor.

Next in the graphics pipeline is the transform engine block, which resides in the graphics subsystem and consists of one or more microcodable processors (called transform

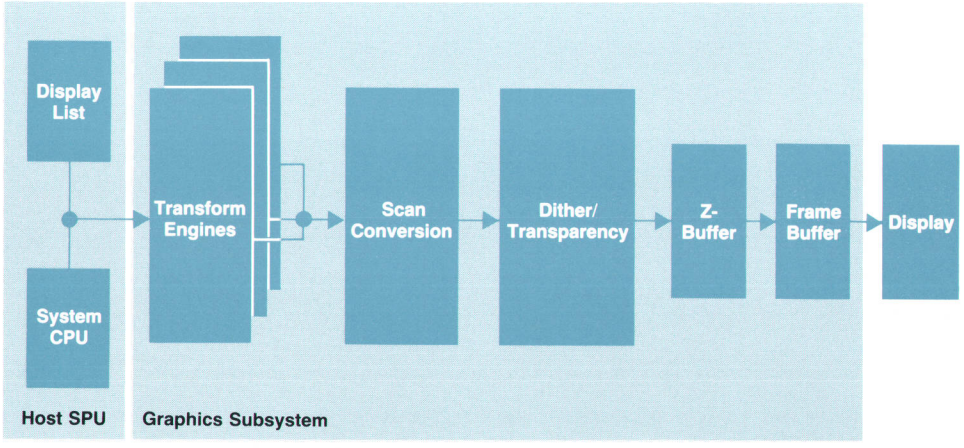


Fig. 1. *The 3D graphics pipeline in the HP 9000 Series 300 and 800 TurboSRX graphics subsystem.*

engines). The transform engine block performs matrix multiply calculations for positioning the image in three-dimensional space, clips the image to the viewing window, calculates polygon vertices for parametric surface commands, and applies lighting calculations for realism.

When the transform engines have finished all necessary calculations, they send the polygon and vector endpoints (in integer device coordinates) to the scan converter. The function of the scan converter is to draw the individual polygons and vectors into the frame buffer where they can be viewed. In the scan conversion process, each pixel in the polygon is calculated individually to determine its x, y, z, red, green, and blue values. The x and y values determine the pixel's location on the screen, the color values allow smooth shading of colors, and the z values are sent to the z-buffer for hidden-surface removal.

After the pixels have been calculated, a dither circuit operates on the color values to provide a greater number of apparent colors, thus allowing true-color images with as few as eight graphics planes. (When 24 planes of frame buffer memory are available, dithering is not used.) Transparency is implemented by drawing alternate pixels of the transparent surface, a technique known as "screen door transparency." The technique gets its name from the screen-door-like pattern used to determine which pixels to draw.

Z-buffering is a general-purpose approach to hidden-surface removal. The z-buffer is simply RAM in which 16 bits are allocated for each pixel on the screen. It works by comparing the z value (depth) of the pixel being drawn to that of the pixel already present at that location, if any. If the new pixel is closer, it is drawn to the frame buffer and the z value is updated to that of the pixel being drawn. If it is farther away, the pixel is not drawn and the z-buffer is not updated.

Comparative Performance

Because the SRX was the first product of its caliber, there were many unknowns about how the product would be used and how it would perform. Since then, much has been learned from our customers and from our own analyses about what features are commonly used and what sizes of polygons are typically drawn. For the purpose of illustration, we will examine two kinds of polygons: small polygons (defined as being 20×20-pixel unconnected quadrilaterals) and large polygons (defined as being larger than 200×200 pixels). The performance metric for small polygons is polygons per second, and large polygons are measured in pixels drawn per second.

Figs. 2 and 3 show the relative performance of different stages in the pipeline. It is important to keep in mind that since the graphics architecture is organized as a pipeline, the performance of the system is determined by the slowest block in the sequence. Note that for small polygons the transform engine block limits the performance on the SRX, with the z-buffer being the next limiter. For large polygons, the z-buffer is the primary culprit, but the dither transparency circuit is right behind.

It was clear from examining the data that to improve performance significantly for both cases, it would be necessary to change more than one functional block.

Transform Engine

Each transform engine consists of a microcodable processor and floating-point chips. (In both the SRX and the TurboSRX, NMOS-III floating-point chips are used.) Because of the many intricate, sophisticated algorithms necessary, it was decided that for the TurboSRX this function should be implemented in the same general-purpose fashion as in the SRX. The approach taken was to use multiple higher-speed transform engines to gain performance. Product packaging limitations prevented a faster discrete imple-

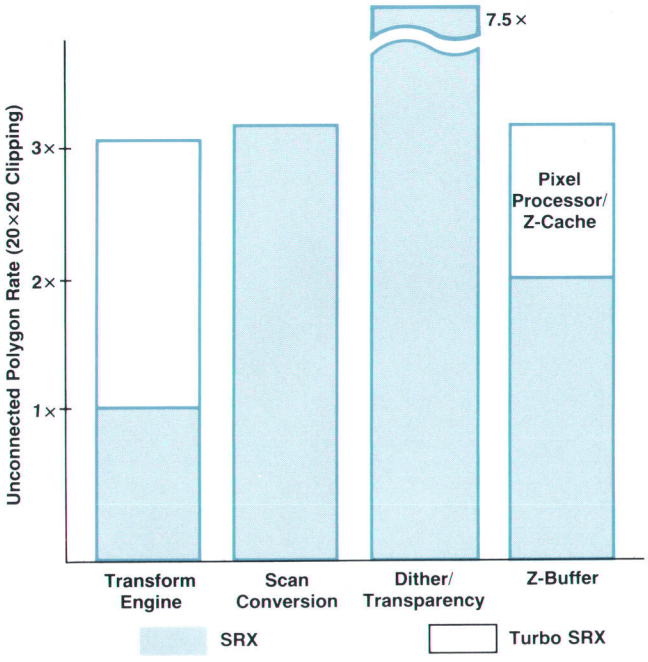


Fig. 2. *Relative performance of 3D graphics pipeline stages for small polygons.*

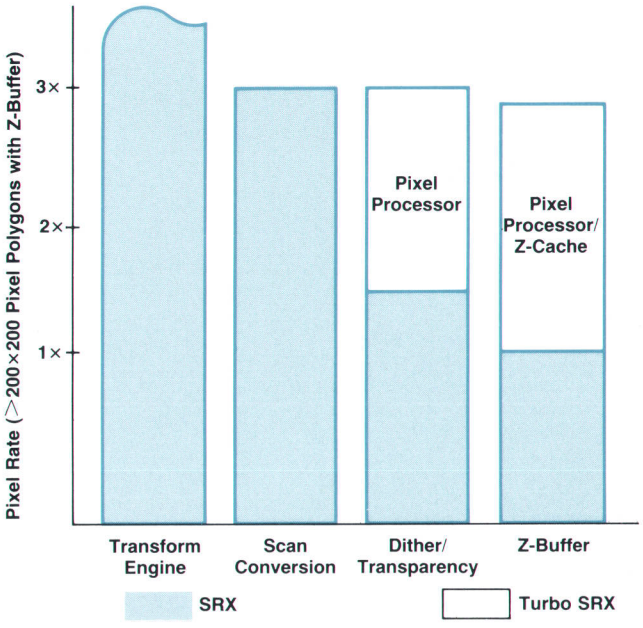


Fig. 3. *Relative performance of 3D graphics pipeline stages for large polygons.*

mentation using bit-slice hardware, so an NMOS-III VLSI chip was designed to enable three improved transform engines to fit into the product. It was dubbed TREIS, which stands for TRansform Engine In Silicon. Integration provides both reduced size and increased performance.

Each transform engine contains the full set of microcode, so any transform engine can execute any graphics operation. One transform engine acts as the master, distributing graphics commands among the three transform engines. Any command can therefore be distributed to the next free transform engine, including the master.

The result is more than a threefold gain in the raw hardware performance in the transform engine stage of the pipeline for small polygons (see Fig. 2). By adding improved microcode and software and some higher-level functions, performance levels up to ten times that of the SRX can be achieved. One higher-level function, quadrilateral mesh, allows the vertices of adjacent quadrilaterals to be transformed, clipped, and lighted a single time, resulting in a net reduction of processing by almost a factor of four.

TREIS (see Fig. 4) is a custom NMOS-III chip containing about 170,000 transistors, including 1536 bytes of pointer RAM and an ALU, in a 272-pin pin-grid array (PGA) package. It outputs a 16-bit microcode address and reads a 68-bit wide microcode word with highly pipelined architecture. It improves performance over the SRX transform engine by combining some two-state activities into one state. Like the SRX transform engine, it connects to HP-proprietary floating-point math chips through a 32-bit floating-point bus for accelerated transformation, clipping, lighting, and parametric surface calculations. The connection to the polygon-rendering chip is through a double-buffered RAM containing polygon and vector vertex addresses, z values, and color data.

Z-Buffer

Once the transform-engine bottleneck was improved, the next performance limitation for small polygons was the speed of the z-buffer. The SRX's z-buffer is in the non-displayed part of the frame buffer. (The frame buffer holds 2048×1024 pixels, but only 1280×1024 can be displayed at a time. Most of what is not displayed can be used as a z-buffer.) While this approach minimizes the cost of low-end systems, maximum performance cannot be obtained when frame buffer and z-buffer accesses cannot be done at the same time.

When drawing with the z-buffer enabled, the SRX must read the z value from the frame buffer, compare the z value of each pixel with the z value present at that location, write the new z value back into the frame buffer if necessary, and write the pixels into the frame buffer if necessary. Using pixel caching allows each access to handle up to eight pixels (the size of a frame buffer "tile") simultaneously.

Most of the z-buffer overhead was eliminated by providing an optional dedicated z-buffer, which allows z-buffer RAM cycles and frame buffer RAM cycles to occur in parallel. In this dedicated z-buffer is another custom chip, the z-cache, which allows multiple z values to be fetched and stored in a single RAM cycle, increases the tile size, and performs comparisons of z values at a rate twice as fast as the SRX.

The z-cache is an LTCMOS standard-cell design containing about 3700 gates, packaged in a 68-pin plastic leaded chip carrier. It is similar in design and size to the pixel cache.[2] It performs fast z comparisons and allows multiple z-buffer operations to take place in a single RAM cycle. One chip per plane is used in the z-buffer.

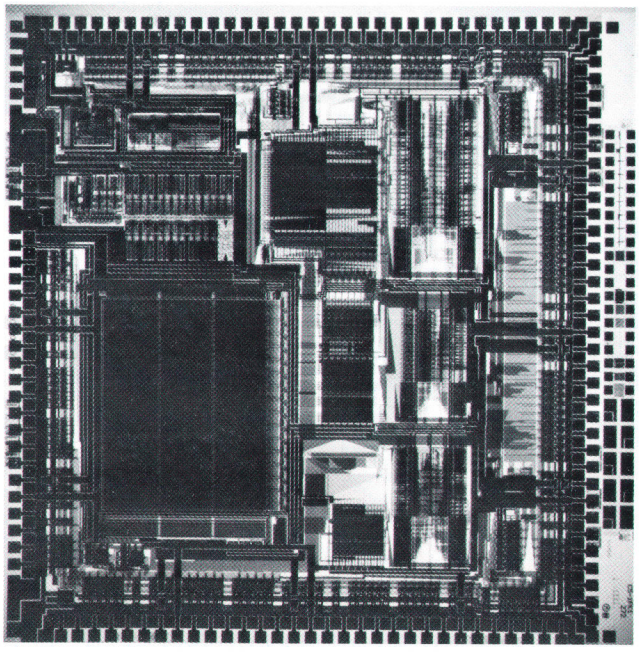


Fig. 4. *TREIS (TRansform Engine In Silicon) chip.*

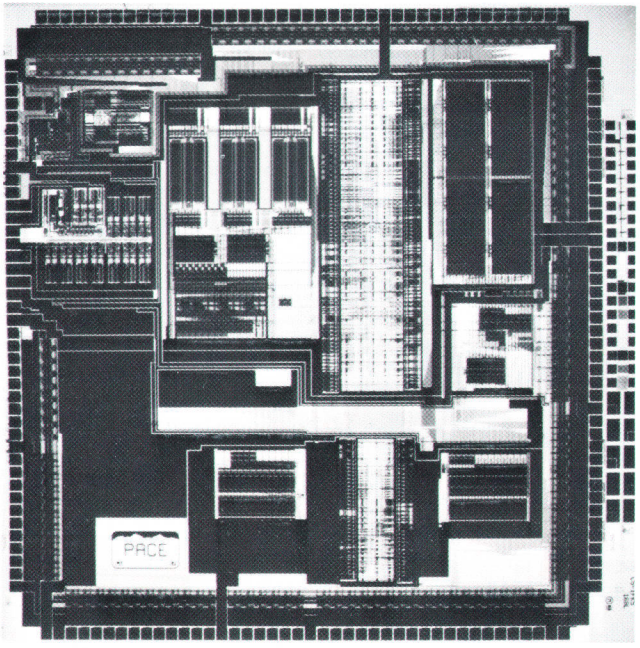


Fig. 5. *Pixel processor chip.*

Pixel Processor

The z-cache chip is still not enough to prevent the z-buffer from limiting overall performance, particularly for large polygons.

In the SRX, whenever a new pixel needs to be written into a tile other than the one accessed by the previous tile, the polygon-rendering chip is held from drawing any more pixels until the new z tile is read. A third custom chip, the pixel processor, was added between the polygon-rendering chip and the z-buffer. It removes that latency by issuing an early warning when a new tile will be needed. This signal is provided far enough in advance of the pixel that the z values can be fetched from the z-buffer before the pixels are drawn. To match the output of the polygon-rendering chip with the z-buffer better, a FIFO buffer was added at the output of the pixel processor. This way, both the polygon-rendering chip and the z-buffer can operate more efficiently.

The pixel processor (see Fig. 5) is a custom NMOS-III chip containing about 110,000 transistors in a 168-pin PGA package. As mentioned earlier, it contains performance improvement features such as the fast dither and transparency operations, the FIFO control, and the early z read signal to prevent slowing down the polygon-rendering chip. In addition, it contains three 1024-byte gamma-correction ROM tables for more accurate color representation, and window clipping operations for up to 32 movable, obscurable, overlapping, accelerated graphics windows. A pipeline valve inside the chip allows fast window operations without emptying the graphics pipeline. All pixel operations inside the pixel processor are performed at the polygon-rendering chip's pixel output speed, so the graphics throughput does not slow down when using any of its features.

Notice in Figs. 2 and 3 that these z-buffer enhancements improve that portion of the pipeline for small polygon performance by about 50% and for large polygons by a factor of three.

Dithering and Transparency

With z-buffer operations streamlined, there was one more stage in the pipeline left to be improved. Dithering and transparency in the SRX are performed with discrete TTL logic. While this does not show up as a performance limiter in the SRX because it is faster than the z-buffer (see Figs. 2 and 3), it would have become the limiting factor in the TurboSRX with the fast z-buffer. Instead of leaving the dither and transparency circuits in TTL, it was decided to include those functions in the pixel processor. This both improves the dither/transparency performance by a factor of two for large polygons (Fig. 3), and improves the reliability and cost of the overall system.

Conclusions

Figs. 2 and 3 reveal that no stage of the TurboSRX pipeline is significantly slower than the others for either small or large polygons. Since the pipeline is fairly well balanced, it might appear that higher performance would require that all parts of the pipeline be replaced, requiring a large amount of product development time and cost. However, as VLSI technology improves, so does the potential improvement of 3D graphics subsystems. Several areas of VLSI technology have been improving lately, including speed, density, packaging, and design productivity. Furthermore, the experience gained on earlier products has pointed the way toward new and better algorithms and architectures. Future graphics products will clearly have to take advantage of these latest advances to meet growing customer expectations.

Acknowledgments

Many individuals contributed to the TurboSRX product. The design teams of the individual chips were as follows. TREIS: Dave Bremner, Bill Cherry, Dan Griffin, Jim Jackson, Gerry Reynolds, and John Young. Z-cache: Andy Goris. Pixel processor: Dale Beucler, Bill Freund, Monish Shah, the author, and James Stewart.

References

1. R.W. Swanson and L.J. Thayer, "A Fast Shaded-Polygon Renderer," Proceedings of SIGGRAPH '86, Dallas, Texas, August 1986, in *Computer Graphics*, Vol. 21, no. 4, August 1986, pp. 95-101.
2. A. Goris, B. Fredrickson, and H. Baeverstad, "A Configurable Pixel Cache for Fast Image Generation," *Computer Graphics and Applications*, Vol. 7, no. 3, March 1987, pp. 24-32.

Global Illumination Modeling Using Radiosity

Radiosity is a complementary method to ray tracing for global illumination modeling. HP 9000 TurboSRX graphics workstations now offer three illumination models: radiosity, ray tracing, and a local illumination model.

by David A. Burgoon

IN COMPUTER GRAPHICS image generation systems, an illumination model can be invoked locally or globally. When invoked locally, only incident light from light sources and object orientation are considered in determining the intensity of light reflected to the observer's eye. Invoked globally, the light that reaches an object by reflection from or transmission through other objects in the scene environment is also considered.

Local illumination models are popular because they produce reasonably realistic rendering and can be computed at interactive rates using hardware acceleration techniques. Global models are usually used when rendering realism is of primary importance. Traditional global illumination modeling methods are extremely computationally intensive. As a result, interactivity is usually sacrificed for the sake of realism.

One of the most familiar local illumination models is that of Phong.[1] Turner Whitted[2] enhanced the Phong model for global use in ray tracing by accounting for the light reflected or transmitted from other objects in the environment.

In the ray tracing procedure, an intersection tree is constructed by tracing a ray from the observer's eye through each pixel into the environment. At each surface intersected by the ray, two branches are added to the tree, representing the spawned reflected and transmitted rays. Each surface intersection is represented by a node in the tree. This process is repeated recursively. The final pixel intensity is determined by traversing the tree starting with the leaves and working toward the root, computing the intensity contribution of each node using the illumination model. The final pixel intensity is the sum of all of these contributions.

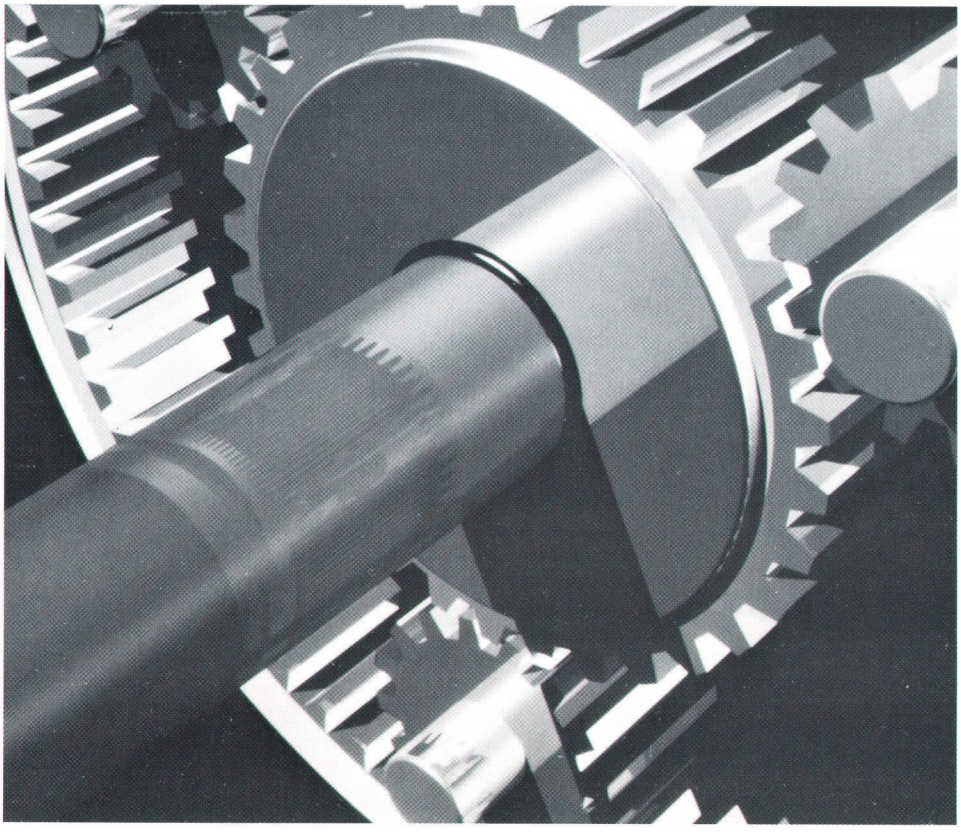


Fig. 1. *These gears were generated on the HP ME Series 30 modeling, design, and drafting system. The ray traced image was rendered using nonuniform rational B-splines. It is a polygonal representation with 3084 polygons and 12 partial polygons.*

Fig. 1 shows an example of a 3D image generated using ray tracing.

Ray tracing is an important rendering method. It has produced some of the most realistic images ever seen to date. However, it is not without its deficiencies.[3,4] In the ray tracing method, realistic shadows are difficult to produce. In particular, penumbras and shadow envelopes are seldom seen in ray traced images. Most ray tracing renderers produce sharp shadow boundaries only.

Most ray tracing systems limit themselves to modeling only point light sources, that is, light sources assumed to emit light that originates from a single point in space. Light sources whose emission comes from a finite area are not readily treated by the method. Only some of the more recent and exotic methods, such as distributed ray tracing and ray tracing with cones, attempt to deal with this limitation.

The reflection models used in ray tracing are usually empirical and approximate. They are often chosen based on subjective results rather than physical laws of energy equilibrium. This disallows the modeling of effects such as color bleeding, where diffuse reflection from one surface causes a soft colored shadow to be seen on another.

Another problem with ray tracing is that it is inherently slow. The computational expense of recursively tracing rays for each pixel on a screen with reasonable resolution (e.g., 1280 by 1024 pixels) can be prohibitive. Furthermore, since the scan conversion and global illumination modeling functions are very tightly coupled, any hardware optimized for scan conversion that may be available is not used. The view dependent nature of the ray tracing algorithm also detracts from the interactivity of the system employing it. Each change in the viewing transformation requires that the entire ray tracing process be repeated to render the new view.

Perhaps the most fundamental flaw of the ray tracing method is that it limits itself to modeling intraenvironment reflections in the specular direction only. Global modeling of diffuse effects is ignored.

Radiosity

The radiosity method, introduced by Goral and others,[5] corrects most of the above deficiencies, but at the expense of introducing some restrictions of its own. The method correctly models the interaction of light between reflecting surfaces if the surfaces are restricted to be perfectly diffuse. It replaces the constant ambient term in Phong's model with an accurate global model. Radiosity has a fundamental energy equilibrium basis, and is derived from methods used in thermal engineering. Fig. 2 shows a 3D image generated using the radiosity method.

In the radiosity method, a (possibly hypothetical) enclosure is constructed around the environment to be rendered. The surfaces or walls of the enclosure completely define the illuminating environment. They consist of light sources and reflecting walls. One or more of the surfaces of the enclosure may be fictitious (e.g., an open window). Each of the surfaces is assumed to be an ideal diffuse reflector, an ideal diffuse emitter, or a combination of the two (Fig. 3).

The radiosity method deals with the equilibrium of radiant energy within the enclosure. The light (or radiosity, which is measured as energy/time/area) leaving a surface i is B_i. It consists of direct emission E_i from the surface plus the reflected portion of light arriving at the surface. The light arriving at i, H_i, is found by summing the contributions from the other $N-1$ surfaces, and from surface i if it is concave. Note that there is no need to treat the emitted

Fig. 2. *This radiosity image of a cathedral with eight bays of windows and columns was done for two bays, and the remaining bays were generated by a step-and-repeat process. It took 7 minutes of preprocessing on an HP 9000 Model 350 to build the data base and subdivide the polygons (meshing), and 12 minutes per step for 40 steps (using progressive refinement) to generate the image (5 minutes and 8 minutes, respectively, on an HP 9000 Model 370). There are 9916 polygons (14,316 after meshing), 26 area lights, and four point lights.*

and reflected energy separately because they are both perfectly diffuse and therefore indistinguishable to the observer. Unlike ray tracing, the history or direction of a ray is lost after reflection from a surface.

The total radiosity leaving a given surface i is therefore

$$B_i = E_i + \rho_i H_i, \tag{1}$$

where B_i = radiosity of surface i. This is the total rate at which radiant energy leaves the surface in terms of energy per unit time per unit area (watts per square meter).

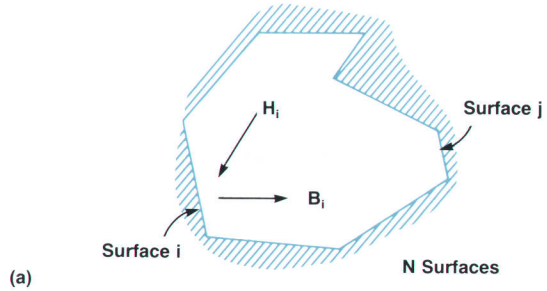
E_i = rate of direct energy emission from surface i per unit time per unit area.

ρ_i = reflectivity of surface i. This represents the fraction of incident light that is reflected back into the hemispherical space surrounding surface i.

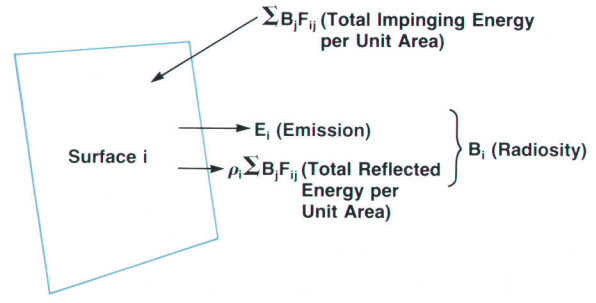
H_i = incident radiant energy arriving at surface i per unit time per unit area (watts per square meter).

H_i is the sum of all the light leaving the N surfaces of the enclosure that "see" surface i. The fraction of the radiant energy leaving a surface j that impinges on surface i is specified by the form factor or configuration factor F_{ji}. The energy per unit time arriving at surface i is therefore

$$H_i A_i = \sum_{j=1}^{N} B_j A_j F_{ji}, \tag{2}$$



(a)



(b)

Fig. 3. *Radiosity relationships. (a) Copyright © 1984 by Goral, Torrance, Greenberg, and Battaile. Used with permission. (b) Copyright © 1986 by Greenberg. Used with permission.*

where A_i is the area of surface i. Dividing through by A_i we have

$$H_i = \sum_{j=1}^{N} B_j \frac{A_j F_{ji}}{A_i}. \tag{3}$$

According to the reciprocal nature of form factors,[5]

$$A_i F_{ij} = A_j F_{ji}. \tag{4}$$

Therefore, H_i is

$$H_i = \sum_{j=1}^{N} B_j F_{ij}. \tag{5}$$

Thus the radiosity at a surface i is

$$B_i = E_i + \rho_i \sum_{j=1}^{N} B_j F_{ij}. \tag{6}$$

This may be rewritten as

$$B_i - \rho_i \sum_{j=1}^{N} B_j F_{ij} = E_i, \tag{7a}$$

or, for i = 1,

$$[1-\rho_1 F_{11} \quad -\rho_1 F_{12} \quad \cdots \quad -\rho_1 F_{1N}] \begin{bmatrix} B_1 \\ B_2 \\ \cdot \\ \cdot \\ B_N \end{bmatrix} = E_1. \tag{7b}$$

Considering all N surfaces i we have

$$\begin{bmatrix} 1-\rho_1 F_{11} & -\rho_1 F_{12} & \cdots & -\rho_1 F_{1N} \\ -\rho_2 F_{21} & 1-\rho_2 F_{22} & \cdots & -\rho_2 F_{2N} \\ \cdot & \cdot & & \cdot \\ \cdot & \cdot & & \cdot \\ -\rho_N F_{N1} & -\rho_N F_{N2} & \cdots & 1-\rho_N F_{NN} \end{bmatrix} \begin{bmatrix} B_1 \\ B_2 \\ \cdot \\ \cdot \\ B_N \end{bmatrix} = \begin{bmatrix} E_1 \\ E_2 \\ \cdot \\ \cdot \\ E_N \end{bmatrix} \tag{7c}$$

This system of N linear equations with N unknown values B_i has parameters E_i, ρ_i, and F_{ij}, which must be known or calculated for each surface. The E_i are nonzero for surfaces that provide illumination to the enclosure. Such surfaces could represent a diffuse area light source or panel, or the first reflection of a directional light source from a diffuse surface. If all of the E_i are zero, then there is no illumination and all of the B_i are zero.

In general, the E_i and ρ_i are functions of the wavelength of the light. They are usually chosen to represent an average value over a bandwidth of radiation, typically red, green, and blue. Once the form factors are calculated, the above matrix equation is solved numerically for the B values for each of three sets of E_i and ρ_i parameters.

The above equation is well-suited to solution using an iterative Gauss-Siedel technique[6] because it is diagonally dominant, that is, the sum of the absolute values of the nondiagonal coefficients in each row is less than the absolute value of the main diagonal term. The solution usually

converges in six to eight iterations.

The aforementioned surfaces generally are not the same as the surfaces of the representation chosen for the geometric model of the scene. For example, if objects are described using polygons, the polygons are usually subdivided into patches or elements (i.e., smaller polygons). These patches become the surfaces of the enclosure.

Once the radiosity for each primary color for each patch has been found, it is mapped onto the vertices of its associated polygon so that the vertex radiosity (color) values can be bilinearly interpolated across the polygon using either Gouraud shading[7] or object-space interpolation. A good way to do this is to set the radiosities at the vertices of patches that are interior to a given polygon to the average of the adjacent patch radiosities and then extrapolate outward to the polygon vertices.

The process of image generation using the radiosity method can be summarized as follows:

- Take the input geometry and subdivide it into patches.
- Calculate form factors.
- Solve for the B_i for each primary color.
- Extrapolate the B_i to polygon vertices and render.

Once the form factors are calculated, they need not be recalculated if colors (ρ) or light sources (E) change. Also, as long as the geometry of the objects remains static, dynamic views of the scene can be generated by merely rerendering. This can be highly interactive on a workstation such as the HP 9000 Model 835 TurboSRX, which has dedicated hardware optimized for polygon rendering.

Form Factor Calculation

We now consider the calculation of F_{ij}, the fraction of the energy leaving surface i impinging on surface j (Fig. 4). Because our surfaces are assumed to be perfectly diffuse, the form factor is purely geometrical in nature. It depends only on the shape, size, position, and orientation of the participating surfaces.

For nonoccluded environments, the form factor from one differential area (i) to another (j) is given by

$$F_{dA_i dA_j} = \frac{\cos\phi_i \cos\phi_j}{\pi r^2} dA_j. \quad (8)$$

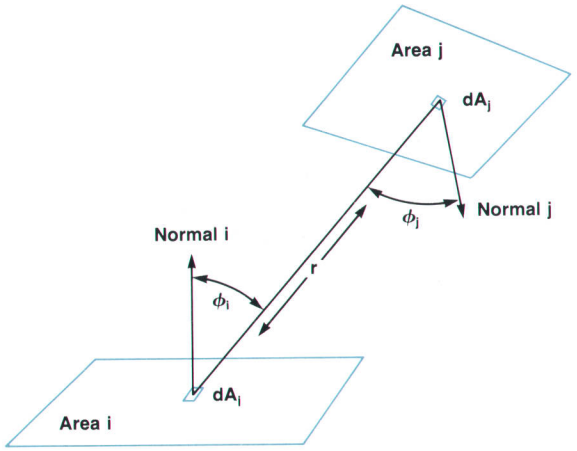


Fig. 4. *Form factor geometry. Copyright © 1986 by Greenberg. Used with permission.*

Integrating over area A_j, the form factor to a finite area or patch is

$$F_{dA_i A_j} = \int_{A_j} \frac{\cos\phi_i \cos\phi_j}{\pi r^2} dA_j. \quad (9)$$

The form factor between finite surfaces (patches) is defined as the area average and is thus

$$F_{ij} = F_{A_i A_j} = \frac{1}{A_i} \int_{A_i} \int_{A_j} \frac{\cos\phi_i \cos\phi_j}{\pi r^2} dA_j\, dA_i. \quad (10)$$

From the symmetry of this equation we can derive the reciprocal relationship given in equation 4. Some other important properties of form factors are:

- From the law of conservation of energy:

$$\sum_{j=1}^{N} F_{ij} = 1. \quad (11)$$

- For any surface that does not see itself (planar or convex):

$$F_{ii} = 0. \quad (12)$$

The Hemicube Algorithm

For occluded environments, equation 10 becomes

$$F_{ij} = F_{A_i A_j} = \frac{1}{A_i} \int_{A_i} \int_{A_j} \frac{\cos\phi_i \cos\phi_j}{\pi r^2} (\text{HID})\, dA_j\, dA_i, \quad (13)$$

where the Boolean function HID takes on the value 1 or 0 depending on whether dA_i can see dA_j. This double area integral is difficult to solve analytically for general cases. An area integral, which is a double integral itself, can be transformed via Stokes' theorem into a single contour integral, which can then be evaluated numerically, but at considerable computational expense. Numerical approximation techniques can provide a more efficient means to compute form factors for general complex environments. The hemicube algorithm[8] employs such a numerical method and also addresses how to deal with the HID function.

Inner Integral Approximation

If the distance between the two patches i and j is large compared to their areas, and if they are not partially occluded from one another, the integrand of the inner integral of equation 13 remains almost constant over the area A_i. If we let K approximate the inner integral we have

$$F_{ij} = F_{A_i A_j}$$

$$\approx \frac{1}{A_i} \int_{A_i} K dA_i = K \frac{A_i}{A_i} = K = \int_{A_j} \frac{\cos\phi_i \cos\phi_j}{\pi r^2} dA_j. \quad (14)$$

Thus finding a solution for the inner integral K, the differential-area-to-finite-area form factor, equation 9, will provide a good approximation for the form factor from patch to patch. If the patches are close together relative to their size, or if there is partial occlusion, the patches must be subdivided into smaller patches until equation 9 provides

a good approximation.

The Nusselt Analog

To see how to evaluate the form factor integral numerically, Nusselt's geometric analog[9] to the form factor integral is helpful (Fig. 5).

Each differential area patch has its own view of the environment, which is the hemisphere of directions surrounding its normal. For a finite area, the form factor is equivalent to the fraction of the circle (which is the base of the hemisphere of directions) covered by projecting the area onto the hemisphere and then orthographically down onto the circle.

The easiest way to see that this analogy correctly describes the form factor integral is to think of it in terms of solid angles and projected areas. The area of plane A that is seen by or projected onto plane B is the area of A times the cosine of the angle between the normals of the two planes. It is equal to the area of the shadow that A would cast onto B. The solid angle can be thought of as a generalization of the planar angles with which we are familiar. Recall that a planar angle θ, measured in radians, is defined to be equal to the length of the arc subtended by the angle divided by the radius r of the circle containing the arc. Since the total circumference of a circle is $2\pi r$, there are 2π radians in a circle. Similarly, a solid angle ω, measured in steradians, is defined to be the area subtended by the solid angle divided by the square of the radius of the sphere containing the area. Since the total area of a sphere is $4\pi r^2$, there are 4π steradians in a sphere (and 2π steradians in a hemisphere). Stated another way, one steradian subtends a unit area of a unit sphere.

Now, returning to our inner integral approximation, equation 9, the solid angle $d\omega$ that subtends the infinitesimal area dA_j is (see Fig. 6):

$$\frac{S_{dA_j}}{r^2}. \tag{15}$$

where S_{dAj} is the portion of the area of the sphere with radius r that is projected by dA_j in the direction of r. Since dA_j is infinitesimally small, this projected area is planar, and is given by $\cos \phi_j dA_j$. Thus, $d\omega$ is

$$\frac{\cos\phi_j \, dA_j}{r^2}. \tag{16}$$

Now, returning to the unit hemisphere of the Nusselt analog, the area on the unit hemisphere projected by dA_j is

$$(\text{solid angle})(\text{radius}^2) = d\omega(1^2) = d\omega = \frac{\cos\phi_j dA_j}{r^2}. \tag{17}$$

The projection of this area onto the base of the unit hemisphere is

$$(\cos\phi_i)(d\omega) = \frac{\cos\phi_i \cos\phi_j dA_j}{r^2}. \tag{18}$$

Taking the ratio of this area to the total area of the base of the unit hemisphere (π) we have, as before, the differential form factor

$$F_{dA_i dA_j} = \frac{\cos\phi_i \cos\phi_j dA_j}{\pi r^2}. \tag{19}$$

Integrating these differential form factors over A_j and then taking the area average of this integral gives us the double area integral expressed in equation 10 for the form factor F_{ij}. Using the inner integral as an approximation is equivalent to using the center point of patch i to represent the average position of patch i, constructing a unit hemisphere around this point, and summing the differential

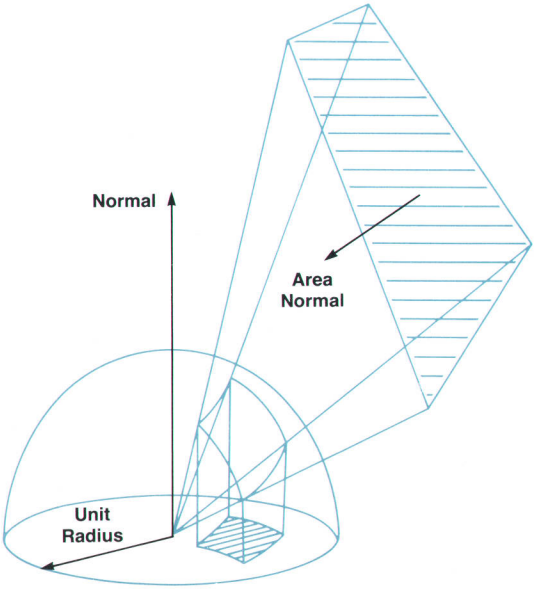


Fig. 5. *The Nusselt analog. The form factor is equal to the fraction of the base of the hemisphere covered by the projection. Copyright © 1985 by Cohen and Greenberg. Used with permission.*

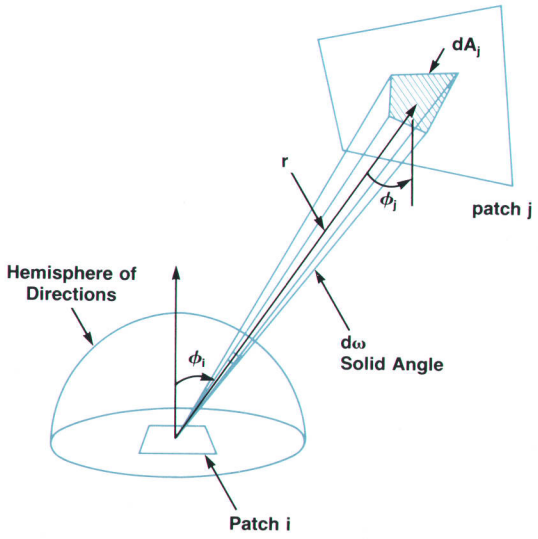


Fig. 6. *The area dAj is taken to be the area of patch j that is visible through the solid angle dω. Adapted from Wallace.[10] Used with permission.*

form factors.

Delta Form Factors

To approximate the inner integral, the hemisphere of directions can be divided into discrete solid angles $\Delta\omega$, and a delta form factor can then be calculated:

$$\Delta F_{ij} = \frac{\cos\phi_i \cos\phi_j \Delta A_j}{\pi r^2}. \quad (20)$$

The form factor F_{ij} can then be approximated by summing the delta form factors covered when projecting patch j onto the unit hemisphere surrounding the center of patch i. If all the patches in the environment are projected onto the hemisphere, discarding the projections of the more distant patches in the case of two or more patches with overlapping projections, the sums of the delta form factors covered by these projections give the form factors from all patches to the patch represented at the center of the hemisphere. This procedure intrinsically includes the effects of hidden surfaces.

To make this procedure practical, a convenient means of dividing the surface of the hemisphere into discrete areas (subtended by the discrete solid angles $\Delta\omega$) is needed. The delta form factors for each of these discrete areas could be precalculated and stored in a lookup table. An evaluation can then be made as to which patch projects onto a given discrete area. For a given patch j, the form factor calculation problem is reduced to determining through which of the discrete solid angles $\Delta\omega$ surface j is visible. Unfortunately, for a hemisphere, it is difficult to devise a method of creating equal discrete areas and a set of linear coordinates to describe the locations of these areas uniquely.

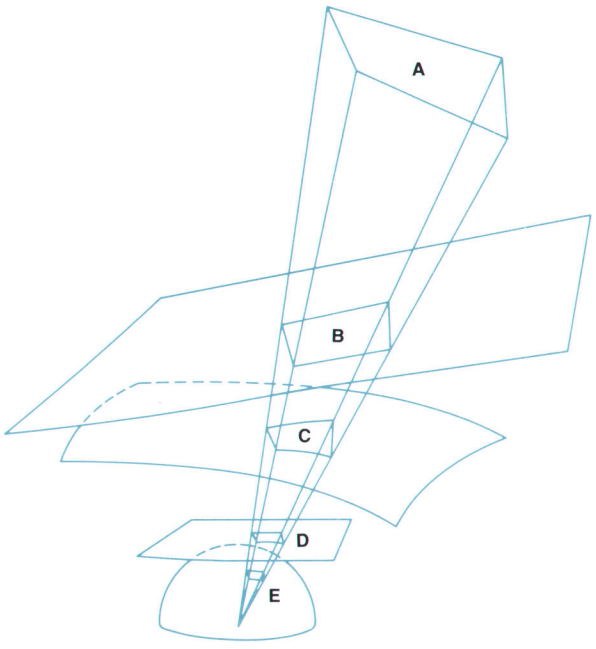


Fig. 7. *Areas with identical form factors. Areas A, B, C, D, and E all have the same form factor. Copyright © 1985 by Cohen and Greenberg. Used with permission.*

The Hemicube

It would be handy if we could choose a more convenient surface than a hemisphere to project the patches onto. From the Nusselt analog it can be seen that any two patches in the environment that project onto the same set of discrete areas of the hemisphere will have the same form factor value. Said another way, any two areas that are seen through the same set of delta solid angles will have the same form factor. In Fig. 7, E is the set of discrete areas and A, B, C, and D all have the same form factor. Consider area D. If we allow D to be part of the top part of a cube surrounding the patch i of interest, we can determine the form factor from patch i to the patch with area A by calculating the form factor to the patch on the cube with area D from patch i. Thus, instead of projecting directly onto the unit hemisphere, we can first project onto a "hemicube" and then calculate the form factor of the intermediate patch that has area equal to the projected area of the original patch.

More specifically, an imaginary cube is constructed around the center of the patch i of interest (Fig. 8). The environment is then transformed to set patch i's center at the origin (eye) with the patch's normal coincident with the positive Z axis (assuming a left-handed coordinate system). The cube is sized so that the perpendicular distance from the center of the patch to the surface of the cube is 1. In this orientation, the aforementioned unit hemisphere is surrounded by the upper-half surfaces of the cube, the lower half being below the horizon of the patch. One full face, facing in the $+Z$ direction, and four half faces, facing in the $\pm X$ and $\pm Y$ directions, replace the hemisphere. These faces are divided into square discrete areas (pixels) at some resolution, usually between 50×50 and 100×10 and the environment is then projected onto the five planar faces.

The beauty of this scheme is that the mathematics and algorithms involved in these projections are well-known: the same clipping, projection, and hidden surface removal techniques used for projection of an environment onto a raster display screen can be used here. (Hardware optimized for these operations can also be employed.) The view direction is set equal to each of the $+Z$, $+X$, $-X$, $+Y$, and $-Y$ axes, and every other patch in the environment is projected onto each of the five "screens," which are the faces of the hemicube perpendicular to each of these five directions. Each full face of the cube covers a 90° frustum as viewed from the center of the cube. This creates clipping planes of $Z = X$, $Z = -X$, $Z = Y$, and $Z = -Y$ that can be used in a simple Sutherland-Hodgman clipper[11] streamlined to handle 90° frustums. Each projected patch can then be scan converted or rasterized to determine which patch's projection covers a given pixel. If two patches project onto the same hemicube pixel, a Z-buffer algorithm can be used to decide which patch is seen in the discrete solid angle represented by the pixel. However, unlike the conventional Z-buffer algorithm used for image rendering, intensity data is not stored for each pixel. Instead, the frame buffer is used as an item buffer to store an integer identifying the patch that is seen by the pixel represented by the item buffer address.

After determining which patch j projects onto each

hemicube pixel, a summation of delta form factors for each pixel covered by patch j determines the form factor from patch i to patch j at the center of the hemicube. That is,

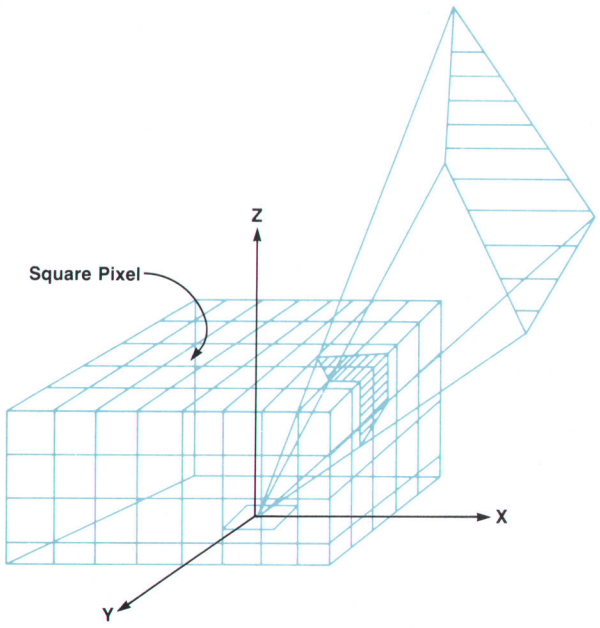
$$F_{ij} = \sum_{q=1}^{R} \Delta F_q, \tag{21}$$

where ΔF_q is the delta form factor associated with hemicube pixel q, and R is the number of hemicube pixels covered by projection of patch j onto the hemicube surrounding element i.

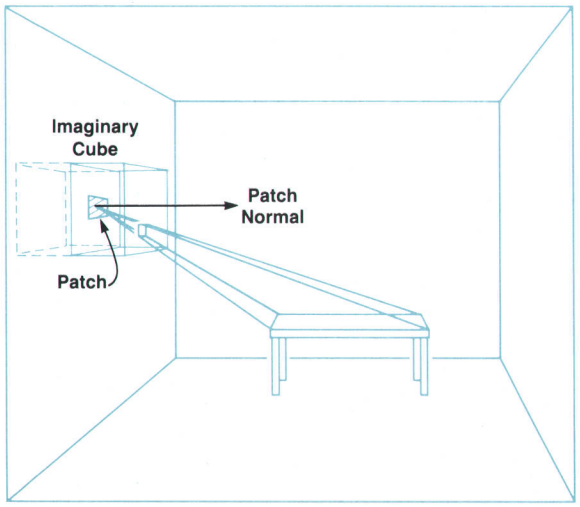
This summation is performed for each patch in the environment to form a complete row of N form factors. Then the hemicube is positioned around another patch i and the process is repeated.

The delta form factor values represented by each hemicube pixel are easily calculated from the delta form factor equation (20) and can be stored in a lookup table. Because of symmetry, this table need only contain values for one eighth of the top face and one half of a side face of the hemicube (Fig. 9).

In summary, the hemicube algorithm provides two main contributions. It provides a very practical method of numerically approximating the form factor integral, and provides a method of properly accounting for the effects of hidden and occluded surfaces at minimal additional expense.



(a)



An imaginary cube is created around the center of a patch. Every other patch in the environment is projected onto the cube.

(b)

Fig. 8. *(a) The hemicube. (b) Projection of the environment onto the hemicube. Copyright © 1985 by Cohen and Greenberg. Used with permission.*

Substructuring

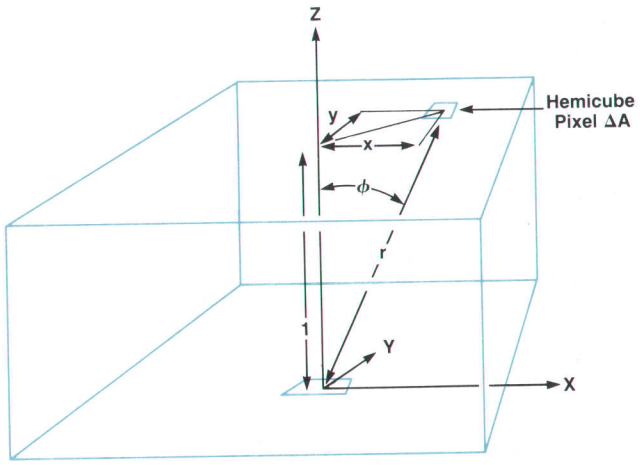
The hemicube algorithm, as presented above, has some problems. Areas in a scene with high intensity (radiosity) gradients (shadow boundaries and penumbra, for example) may be poorly represented, particularly when the patches are large relative to the area over which the radiosity gradient occurs. To remedy this, the areas of surfaces with high radiosity gradients must be subdivided into finer and finer grids of patches. This presents two problems: how to increase the number of patches without incurring significant additional computational cost, and how to decide which areas of the scene should be subdivided. These problems were addressed in a paper by Cohen and others.[12]

The solution of the radiosity simultaneous equilibrium equations using the Gauss-Seidel technique is $O(N^2)$, that is, the number of calculations required is of order N^2, where N is the number of patches used to describe the scene. The calculation of the form factors is also $O(N^2)$. If the first problem is not addressed and N is naively increased, the computational costs can be prohibitive.

To remedy this situation we borrow a concept from engineering mechanics known as substructuring, where the solution for local stress behavior is based on the global structure response to a coarse solution. Applying this notion to the radiosity problem, we subdivide the patches that are too large into a total of M elements (according to criteria to be discussed later), leaving K unsubdivided patches. It is assumed that each element has a constant radiosity, but that these element radiosities vary across the patch. Next, we would like to be able to find the radiosities of the elements using a solution for the radiosities of the original patches and avoiding a full $O((M+K)^2)$ solution, somehow applying the solution for the global patch radiosities to the elements.

Element Radiosities

To see how to do this, assume that patch i has been subdivided into R elements. We can then represent B_i, the radiosity of patch i, as the average over the area of the patch of the element radiosities:



Top of Hemicube: $r = \sqrt{x^2 + y^2 + 1}$

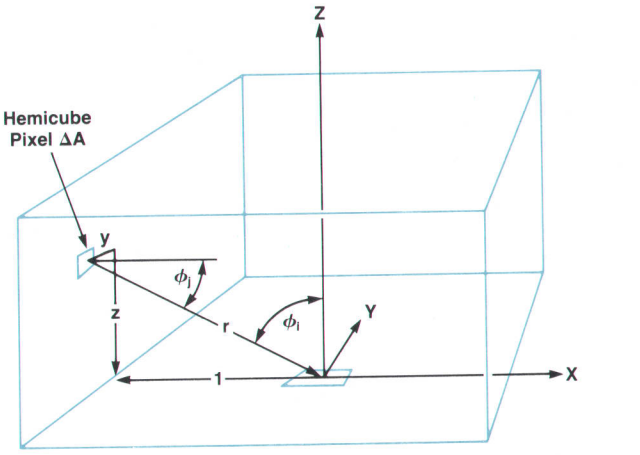
$$\cos\phi_i = \cos\phi_j$$

$$\cos\phi = \frac{1}{\sqrt{x^2 + y^2 + 1}}$$

$$\Delta\text{Form Factor} = \frac{\cos\phi_i \cos\phi_j}{\pi r^2}\Delta A$$

$$= \frac{1}{\pi(x^2 + y^2 + 1)^2}\Delta A$$

(a)



Side of Hemicube: $r = \sqrt{y^2 + z^2 + 1}$

$$\cos\phi_i = \frac{z}{\sqrt{y^2 + z^2 + 1}}$$

$$\cos\phi_j = \frac{1}{\sqrt{y^2 + z^2 + 1}}$$

$$\Delta\text{Form Factor} = \frac{\cos\phi_i \cos\phi_j}{\pi r^2}\Delta A$$

$$= \frac{z}{\pi(y^2 + z^2 + 1)^2}\Delta A$$

(b)

Fig. 9. *Derivation of delta form factors. Copyright © 1985 by Cohen and Greenberg. Used with permission.*

$$B_i = \frac{1}{A_i}\sum_{q=1}^{R} B_q A_q. \tag{22}$$

From the definition of the radiosity of an element given in equation 6 we know that

$$B_q = E_q + \rho_q \sum_{j=1}^{N} B_j F_{qj}. \tag{23}$$

Substituting equation 23 into equation 22, we have

$$B_i = \frac{1}{A_i}\sum_{q=1}^{R}\left(E_q + \rho_q \sum_{j=1}^{N} B_j F_{qj}\right) A_q. \tag{24a}$$

Distributing, we have

$$B_i = \frac{1}{A_i}\sum_{q=1}^{R} E_q A_q + \frac{1}{A_i}\sum_{q=1}^{R}\left(\rho_q \sum_{j=1}^{N} B_j F_{qj} A_q\right). \tag{24b}$$

If we assume the emission and reflectivity of the patch are constant, then $E_i = E_q$ and $\rho_i = \rho_q$. Also, if the global radiosities B_j are assumed constant for each element over the surface of each patch, we have

$$B_i = \frac{1}{A_i} E_i \sum_{q=1}^{R} A_q + \rho_i \sum_{j=1}^{N} B_j \left(\frac{1}{A_i}\sum_{q=1}^{R} F_{qj} A_q\right) \tag{24c}$$

and

$$B_i = E_i + \rho_i \sum_{j=1}^{N} B_j \left(\frac{1}{A_i}\sum_{q=1}^{R} F_{qj} A_q\right). \tag{24d}$$

By comparing equation 6, the quantity in parentheses above in equation 24d is easily seen to be the patch-to-patch form factor expressed as the area-weighted average of the element-to-patch form factors, where the elements are subdivisions of patch i. Thus

$$F_{ij} = \frac{1}{A_i}\sum_{q=1}^{R} F_{qj} A_q \ . \tag{25}$$

Each of the element-to-patch form factors F_{qj} can be found using the hemicube algorithm. Then the patch-to-patch form factors F_{ij} are calculated using equation 25. The standard system of simultaneous radiosity equations (7c) can then be solved to yield the patch radiosities in $O(N^2)$ time. The resulting patch radiosities are more accurate than those that would have been obtained without subdividing the patches into elements. This is because the expression for F_{ij} given by equation 25 represents a discrete numerical method for approximating the outer area integral of the form factor double area integral given in equation 13. Recall that in the original hemicube algorithm the outer integral was taken to be unity because we restricted the patches to be small relative to the distances that separate them. We now can remove that restriction by using equation 25, but placing the same restriction on the elements that make up a patch.

Once the patch radiosities have been calculated, the radiosity for each element q can be found using the basic equation for the radiosity of an element, equation 23.

In short, subdivision of patches into elements and use of the above equations provides two main advantages over naive use of a finer patch resolution. First, the local variations of intensity within a patch can be accurately approximated without having to solve the global radiosity equations on an element level. Second, the radiosity solution on the patch level is more accurate because it better approximates the patch-to-patch form factors.

Substructuring Algorithm

The following are the major steps involved in employing these optimizations in a rendering algorithm:

1. Form a hierarchical description of the environment consisting of surfaces, subsurfaces, patches, and elements.
2. For each element, find the form factor to each patch using the hemicube algorithm, and store the results in an $M \times N$ matrix (M = number of elements, N = number of patches).
3. Compress this matrix into an $N \times N$ patch form factor matrix using equation 25.
4. For each of the primary color bands—red, green, and blue—form and solve the set of N equations in N unknowns for the patch radiosities using the Gauss-Seidel iterative technique and equation 7c.
5. Compute the M element radiosities for each element q using equation 23, the patch radiosities, and the element-to-patch form factors computed in step 2.
6. Calculate element vertex radiosities from the radiosities of the elements adjacent to the vertex.
7. Linearly interpolate the vertex radiosities across the elements using Gouraud shading or object-space interpolation.

Adaptive Subdivision

We now address the criteria to be used in deciding how to partition the scene hierarchically down to the element level. Ideally, the element mesh should be densest in regions of high intensity gradients. Cohen's paper[12] says that a reasonable first guess must be provided by the user as to which areas are likely to have high intensity gradients. These areas include areas in shadow and areas near light sources. Then the intensities of adjacent vertices found in step 6 can be compared. If the change in intensity is greater than some threshold value, the elements adjacent to that vertex should be recursively subdivided until the intensity change is below the threshold. The algorithm is then recursively repeated, beginning at step 2.

Cohen used a simple binary subdivision, where each rectangular element is divided into four new elements. This preserves the original patch's geometry and allows most of the previously computed element-to-patch form factors to be reused. The only change that needs to be made to the original $M \times N$ element-to-patch form factor matrix to subdivide a particular element i is to remove row i from the matrix and insert four new element rows. (Of course, the hemicube algorithm must be used to calculate the elements of the new rows). This object-space subdivision technique is analogous to the Warnock algorithm,[13] which subdivides polygons to perform hidden surface removal.

Progressive Refinement

Perhaps the most significant improvement to the radiosity method is the algorithm based on the technique of progressive refinement devised by Cohen and his colleagues.[14] This algorithm has two main advantages over those we have described so far.

First, it provides renderings of the environment that are early approximations of the final energy-equilibrium solution. This has the advantage of allowing the user to see advance previews approximating the final correctly rendered scene without having to wait for the full $O(N^2)$ solution to equation 7c. At each step of the progressive refinement approach, the rendering of the scene gracefully converges to the full solution. The user can interactively stop this progression when the rendering looks good enough. In most cases, a useful image is produced in O(N) time.

The second advantage of the progressive refinement approach is a reduction in storage and start-up computational costs. The previous algorithms require that all form factors be precalculated before the Gauss-Seidel solution begins. This requires $O(N^2)$ storage. For reasonably complex environments, this cost can be significant. For example, an environment of 50,000 patches will require a gigabyte of storage.[14] In the progressive refinement algorithm, form factors are calculated on the fly to reduce the form factor storage requirements to O(N) and eliminate the associated startup computational costs.

The progressive refinement algorithm can be thought of as a restructuring of previous methods, and differs from them primarily in two ways. First, the radiosity of all patches is updated simultaneously instead of one at a time during each iteration. Second, patches are processed in sorted order according to their energy contribution to the environment.

To gain an insight into how this is possible, consider row i of equation 7c (i.e., equation 6). This equation may be thought of as one that determines the light leaving patch i by gathering in the light from the rest of the environment (Fig. 10).

A single term from the summation in equation 6 determines the contribution of patch j to the radiosity of patch i, that is,

$$\text{Contribution of } B_j \text{ to } \Delta B_i = \rho_i B_j F_{ij}. \qquad (26)$$

The progressive refinement method reverses this process by considering the contribution made by patch i to the radiosity of all other patches. The reciprocity relationship (equation 4) provides the basis for this reversal. The contribution of the radiosity from patch i to the radiosity of patch j is

$$\text{Contribution of } B_i \text{ to } \Delta B_j = \rho_j B_i F_{ij} \frac{A_i}{A_j}. \qquad (27)$$

The total contribution to the environment from the radiosity of patch i is determined by calculating the above equation for all patches j.

A key fact about this reformulation is that the radiosities of the patches j in the environment are updated using form factors calculated via a single hemicube placed at patch i.

Thus, each step of the iteration no longer requires that all of the form factors F_{ij} be known in advance. Each step of the solution now consists of placing a single hemicube around a patch i and adding the contribution from the radiosity of that patch to the radiosities of all other patches, calculating form factors as needed. In effect, we are shooting light from patch i out into the environment rather than gathering the light from the environment received at patch i (Fig. 10). For a more detailed description of this iterative shooting algorithm, consult the literature.[14]

To arrive at the final solution as quickly as possible, we capitalize on the fact that if the patches i with the largest contribution to the environment are processed first, the final value for the radiosity of patch j, which is the sum of these contributions, will be approached earlier. Stated intuitively, those patches radiating the most light energy should be treated first, since they have the greatest effect on the illumination of environment. This energy will tend to come from those patches having the largest product B_iA_i.

Accordingly, the progressive refinement algorithm is implemented by always shooting first from patches for which the difference ΔB_iA_i between the previous and current estimates of unshot radiosity is greatest. This usually results in most light sources being processed first, followed by the patches that receive the most light from the light sources, and so on. Thus, when solving in sorted order, the solution tends to proceed in nearly the same order as light would propagate through the environment. Solving in sorted order usually yields a useful estimate of the final solution in less than a single full iteration, substantially reducing computation costs.[14] Fig. 2 was rendered using a progressive refinement technique.

Summary and Conclusions

The radiosity global illumination method combats many of the deficiencies of ray tracing. Radiosity methods produce excellent penumbras, shadow envelopes, and color bleeding effects. Area light sources are accurately modeled. The radiosity model is "correct" in the sense that it is based on laws of physics (energy equilibrium because of conservation of energy). In the radiosity method, illumination modeling is decoupled from scan conversion and rendering. Finally, radiosity algorithms are view independent. This allows a high degree of interactivity for static geometry once the preprocessing is complete.

Despite these advantages, radiosity also has some disadvantages with respect to ray tracing. Rendering using the full radiosity solution is slow (although proponents claim it is faster than ray tracing because it is view independent). Also, specular reflections, transparency, and translucency are not modeled.

Radiosity and ray tracing are complementary methods. No one method models reality perfectly (although radiosity advocates point out that most natural environments are predominantly diffuse). Recent research involves combining aspects of both methods. For example, in a very recent paper by Wallace and his colleagues,[17] a ray tracing technique is used to compute form factors, instead of the hemicube algorithm. Also, techniques have been recently proposed for producing specular highlights along with global diffuse illumination components.[10,15,16]

There is still a fair amount of research that needs to be done before an interactive global model can be offered that models an environment perfectly without having to sacrifice diffuse components, as in ray tracing, or specular highlights, as in radiosity. However, the illumination models that have been developed to date are extremely useful and should be made available to users of graphics workstations. Accordingly, Hewlett-Packard chose to become the first workstation vendor to offer radiosity-based illumination modeling as well as the more traditional methods. In July 1989, HP released its Starbase Radiosity and Ray Tracing software, which integrates into the Starbase display list support for both radiosity and ray tracing on high-end Tur-

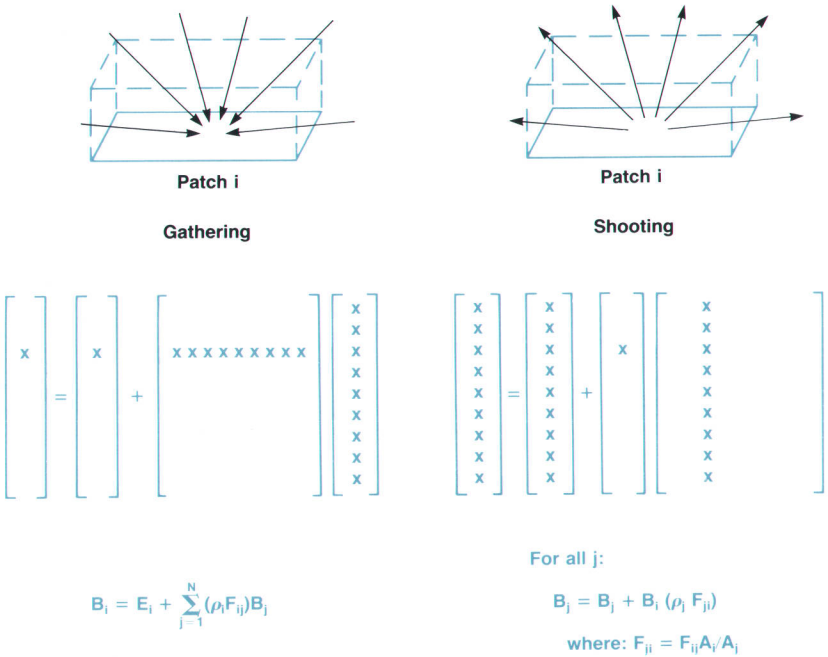


Fig. 10. *Gathering versus shooting. Gathering light through a hemicube allows one patch radiosity to be updated. Shooting light through a single hemicube allows the entire environment's radiosity values to be updated simultaneously. Copyright © 1989 by Cohen, Chan, Wallace, and Greenberg. Used with permission.*

boSRX workstations. This gives the graphics programmer a choice of three illumination methods: local illumination based on an enhanced Phong model, global illumination based on anti-aliased ray tracing, and global illumination using progressive refinement radiosity. Applications using the Starbase display list can now be written to provide the user with the widest possible variety of photorealistic rendering.

We have presented a tutorial summary of the theory and algorithms of the radiosity method that have appeared in the literature over the last few years. We have done so with the hope that the reader will gain an intuitive feel for the method, some of the improvements that have been made to it, and the advantages that may be gained from it. No attempt has been made to discuss the particulars of HP's implementation, which makes use of the most recent advances.[14,17]

Acknowledgments

The author would like to thank those who helped make possible the graduate study that produced this tutorial. Specifically, I would like to thank the Hewlett-Packard Company for providing financial support through the HP educational assistance program, and my advisor, Dr. Gary J. Herron, for teaching me the basics of computer graphics and motivating my study of radiosity methods. Thanks are also in order to Joan Bushek and Darel Emmot, who reviewed drafts of this paper, and to those members of the R&D Laboratories of HP's Graphics Technology Division who offered suggestions for the improvement of the oral presentation of this tutorial. Lastly, I would like to thank the ACM for sponsoring the SIGGRAPH conferences that introduced me to radiosity and have served to keep me excited about the field of computer graphics.

References

1. B.T. Phong, *Illumination for Computer Generated Images*, PhD Dissertation, University of Utah, 1973.
2. T. Whitted, "An Improved Illumination Model for Shaded Display," *Communications of the ACM*, June 1980.
3. D.P. Greenberg, "Ray Tracing and Radiosity," *State of the Art in Image Synthesis*, course notes, SIGGRAPH 1986.
4. D.P. Greenberg, M.F. Cohen, and K.E. Torrance, "Radiosity: A method for computing global illumination," *The Visual Computer*, Vol. 2, no. 5, September 1986.
5. C.M. Goral, K.E. Torrance, D.P. Greenberg, and B. Battaile, "Modeling the Interaction of Light Between Diffuse Surfaces," *ACM Computer Graphics (Proceedings of SIGGRAPH 1984)*.
6. C.F. Gerald, *Applied Numerical Analysis*, 2nd Edition, Addison-Wesley, 1978, pp. 116-117.
7. D.F., Rogers, *Procedural Elements for Computer Graphics*, McGraw-Hill, 1985, pp. 323-325.
8. M.F. Cohen and D.P. Greenberg, "The Hemi-Cube: A Radiosity Solution for Complex Environments," *ACM Computer Graphics (Proceedings of SIGGRAPH 1985)*, Vol. 19, no. 3, July 1985, pp. 31-40.
9. R. Shegel and J.R. Howell, *Thermal Radiation Heat Transfer*, Hemisphere Publishing Corporation, 1978.
10. J.R. Wallace, *A Two-Pass Solution to the Rendering Equation: A Synthesis of Ray Tracing and Radiosity Methods*, MS Thesis, Cornell University, January 1988.
11. D.F. Rogers, *op cit*, pp. 169-179.
12. M.F. Cohen, D.P. Greenberg, D.S. Immel, and P.J. Brock, "An Efficient Radiosity Approach for Realistic Image Synthesis," *IEEE Computer Graphics and Applications*, March 1986. pp. 26-35.
13. D.F. Rogers, *op cit*, pp. 240-259.
14. M.F. Cohen, S.E. Chen, J.R. Wallace, and D.P. Greenberg, "A Progressive Refinement Approach to Fast Radiosity Image Generation," *ACM Computer Graphics (Proceedings of SIGGRAPH 1988)*, Vol. 22, no. 4, August 1988, pp. 75-84.
15. D.S. Immel, M.F. Cohen, and D.P. Greenberg, "A Radiosity Method for Non-Diffuse Environments," *ACM Computer Graphics (Proceedings of SIGGRAPH 1986)*, Vol. 20, no. 4, August 1986, pp. 133-142.
16. J.R. Wallace, M.F. Cohen, and D.P. Greenberg, "A Two-Pass Solution to the Rendering Equation: A Synthesis of Ray Tracing and Radiosity Methods," *ACM Computer Graphics (Proceedings of SIGGRAPH 1987)*, Vol. 21, no. 4, July 1987, pp. 311-320.
17. J.R. Wallace, K.A. Elmquist, and E.A. Haines, "A Ray Tracing Algorithm for Progressive Radiosity," *ACM Computer Graphics (Proceedingjs of SIGGRAPH 1989)*.

Hewlett-Packard Company, 3200 Hillview Avenue, Palo Alto, California 94304

ADDRESS CORRECTION REQUESTED

Bulk Rate
U.S. Postage
Paid
Hewlett-Packard
Company

HEWLETT-PACKARD JOURNAL

December 1989 Volume 40 • Number 6

Technical Information from the Laboratories of Hewlett-Packard Company
Hewlett-Packard Company, 3200 Hillview Avenue
Palo Alto, California 94304 U.S.A.
Hewlett-Packard Marcom Operations Europe
P.O. Box 529
1180 AM Amstelveen, The Netherlands
Yokogawa-Hewlett-Packard Ltd., Suginami-Ku Tokyo 168 Japan
Hewlett-Packard (Canada) Ltd.
6877 Goreway Drive, Mississauga, Ontario L4V 1M8 Canada

CHANGE OF ADDRESS: To subscribe, change your address, or delete your name from our mailing list, send your request to Hewlett-Packard Journal, 3200 Hillview Avenue, Palo Alto, CA 94304 U.S.A. Include your old address label, if any. Allow 60 days.

5953-8578

HP Archive