HEWLETT-PACKARD

Journal

MAY 1998

Technical Information from the Hewlett-Packard Company

HEWLETT PACKARD

HEWLETT-PACKARD Journal

MAY 1998

Highlights

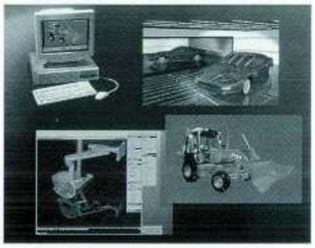
Technical computing today is increasingly dominated by design and analysis tasks that require high-performance workstation and software products. Some of the products described in this issue address the needs of this emerging market.

On the software side, we have the DirectModel 3D modeling toolkit and the HP implementation of the OpenGL® graphics standard. The toolkit provides application developers with the capability to develop applications that can construct 3D models containing millions or billions of polygons. DirectModel is built on top of the HP OpenGL product. OpenGL is a vendor-neutral, multiplatform, industry-standard application programming interface (API) for developing 2D and 3D visual applications.

For running these applications, we have the HP Kayak PC-based workstation running the Windows® NT operating system. HP Kayak provides world-leading 3D graphics performance typically found in high-end UNIX® workstations. Much of the hardware architecture for HP Kayak is based on the VISUALIZE fx4 graphics accelerator, which is designed to provide native acceleration for the OpenGL API.

A common theme underlying the development of all these products is the desire to shorten the time to market. Concurrent engineering was employed in the OpenGL project to achieve this goal. Processes done in serial were

THE COVER



Examples of 3D graphics images that can be rendered with HP workstations using the VISUALIZE fx graphics hardware. Renderings are courtesy of Dassault Systemes of Suresnes, France, Division of Bristol, England, and Parametric Technology Corporation of Waltham, Massachusetts.

Volume 49 • Number 2

The Hewlett-Packard Journal is published quarterly by the Hewlett-Packard Company to recognize technical contributions made by Hewlett-Packard personnel.

Printed on recycled paper.

modified to be done in parallel, shortening the product development cycle. Quality engineers at the HP Kobe Instrument Division reengineered their quality assurance process to deal with the time-to-market issue and still maintain high-quality released software.

We have two articles about HP-UX workstations. One describes a feature that allows multiple monitors to be configured as one contiguous viewing space, and the other discusses the challenges of adding the Peripheral Component Interconnect, or PCI, to HP B-class and C-class workstations.

Information is the fuel that drives today's enterprises. Thus, we have three articles that discuss the use of information to do such tasks as linking business manufacturing software to the factory floor, providing a knowledge database for support personnel, and forecasting component demand in material planning.

The article about HP VEE (Visual Engineering Environment) is an example of our new publishing paradigm of using the web to extend or complement what appears in the printed version of the Hewlett-Packard Journal.

C. L. Leath
Managing Editor

WHAT'S AHEAD

In August we will have articles about a 150-MHz-bandwidth membrane hydrophone, units measurement for optical instruments, and efforts to improve the reliability of ceramic pin grid array packaging and surface-mount LEDs. We will also have articles from the HP Design Technology Conference, the HP Compression Conference, and the HP Electronic and Assembly Conference.

M. Shahid Mujtaba, HP Laboratories, Palo Alto, California
Steven J. Narciso, VXI Systems Division, Loveland, Colorado
Danny J. Oldfield, Electronic Measurements Division, Colorado Springs, Colorado
Garry Orsolini, Software Technology Division, Roseville, California
Ken Poulton, HP Laboratories, Palo Alto, California
Günter Riebesell, Böblingen Instrument Division, Böblingen, Germany
Michael B. Saunders, Integrated Circuit Business Division, Corvallis, Oregon
Philip Stenton, HP Laboratories Bristol, Bristol, England
Stephen R. Undy, Systems Technology Division, Fort Collins, Colorado
Jim Willits, Network and System Management Division, Fort Collins, Colorado
Koichi Yanagawa, Kobe Instrument Division, Kobe, Japan
Barbara Zimmer, Corporate Engineering, Palo Alto, California

The Hewlett-Packard Journal Online

The Hewlett-Packard Journal is available online at:

http://www.hp.com/hpj/journal.html

A quick tour of our site:

Current Issue—see it before it reaches your mailbox.

Past Issues—review back to February 1994, complete with links to other HP sites.

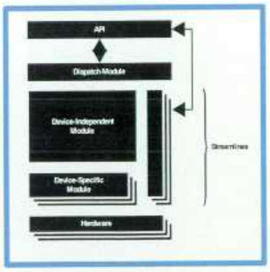
Index—search back to our first issue in 1949, complete with an **Order** button for issues or articles you'd like for your library.

Subscription Information—guidelines for U.S. and international subscriptions and a form you can fill out to receive an e-mail message about upcoming issues and changes to our Website.

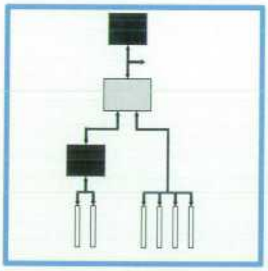
Previews—contains an early look at future articles.

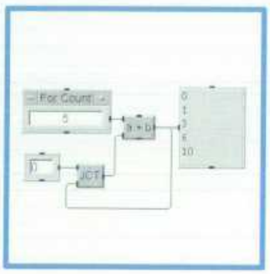
WWW

Articles



Dispatch Module
Device-Independent Module
Device-Specific Module
Hardware





The Hewlett-Packard Journal Online

http://www.hp.com/hpj/journal.html

What's new?

- The **Previews** section contains the following new articles:

Techniques for Higher-Performance Boolean Equivalence Verification

Theory and Design of CMOS HSTL I/O Pads

On-Chip Cross Talk Noise Model for Deep-Submicrometer ULSI Interconnect

Testing with the HP 9490 Mixed-Signal LSI Tester

A Low-Cost RF Multichip Module Packaging Family

Comparison of Finite-Difference and SPICE Tools for Thermal Modeling of the Effects of High-Power CPUs

E-Mail Registration

- Use **E-Mail Notification** to register your e-mail address so that you can be notified when new articles are published.

An API for Interfacing Interactive 3D Applications to High-Speed Graphics Hardware

Kevin T. Lefebvre

John M. Brown

The OpenGL® specification defines a software interface that can be implemented on a wide range of graphics devices ranging from simple frame buffers to fully hardware-accelerated geometry processors.

OpenGL is a specification for a software-to-hardware application programming interface, or API, that defines operations needed to produce interactive 3D applications. It is designed to be used on a wide range of graphics devices, including simple frame buffers and hardware-accelerated geometry processor systems. With design goals of efficiency and multiple platform support, certain functions, such as windowing and input support, have not been defined in OpenGL. These unsupported functions are included in support libraries outside the core OpenGL definition.

OpenGL is targeted for use on a range of new graphics devices for both UNIX® -based and Windows® NT-based operating system platforms. These systems differ in both capabilities and performance.

Early in the OpenGL program at HP, industry partnerships were established between the OpenGL R&D labs and key independent software vendors (ISVs) to ensure a high-quality, high-performance product that met the needs of these ISVs. These partnerships were also used to assist the ISVs in moving to the HP OpenGL product (see "The Fast Break Program" on page 8).

The various OpenGL articles in this issue describe the design philosophy and the implementation of the HP version of OpenGL and other graphics products associated with OpenGL.

Kevin T. Lefebvre
A senior engineer in the graphics products laboratory at the HP Workstation Systems Division, Kevin Lefebvre is responsible for the OpenGL architecture and its implementation and delivery. He came to HP in 1986 from the Apollo Systems Division. He has a BS degree in mathematics (1976) from Carnegie-Mellon University. He was born in Pittsfield, Massachusetts, is married and has two children. His hobbies include running, biking, and skiing.

John M. Brown
John Brown is a senior engineer in the graphics products laboratory of the HP Workstation Systems Division. He is responsible for graphics application performance. John came to HP in 1988. He holds a BSEE degree (1980) from the University of Kentucky.

History of OpenGL

OpenGL is a successor to Iris GL, a graphics library developed by Silicon Graphics® International (SGI). Major changes have been made to the Iris GL specification in defining OpenGL. These changes have been aimed at making OpenGL a cleaner, more extensible architecture.

With the goal of creating a single open graphics standard, the OpenGL Architecture Review Board (ARB) was formed to define the specification and promote OpenGL in terms of ISV use and availability of vendor implementations. The original ARB members were SGI, Intel, Microsoft®, Digital Equipment Corporation, and IBM. Evans & Sutherland, Intergraph, Sun, and HP were added more recently. For more information on current ARB members, OpenGL licensees, frequently-asked questions, and other ARB related information, visit the OpenGL web site at http://www.opengl.org.

The initial effort of the ARB was the 1.0 specification of OpenGL, which became available in 1992. Along with this specification was a series of conformance tests that licensees needed to pass before an implementation could be called OpenGL. Since then the ARB has added new features and released a 1.1 specification in 1995 (the HP implementation is based on 1.1). Work is currently being done to define a 1.2 revision of the specification.

HP Involvement in OpenGL

HP became an OpenGL licensee in 1995. We had the goal of delivering a native implementation of OpenGL that would run on hardware and software that would provide OpenGL performance leadership.

Shortly after licensing OpenGL, we established a relationship with a third party to provide an OpenGL implementation on our existing set of graphics hardware while we worked on a new generation of hardware that was better suited for OpenGL semantics. The OpenGL provided by the third party used the underlying graphics hardware acceleration where possible. However, it could not be considered an accelerated implementation of OpenGL because of features lacking in the hardware.

In August of 1996, we demonstrated our first native implementation of OpenGL at Siggraph 96. This implementation was fully functional and represented the software that would be shipped with the future OpenGL-based hardware. The implementation supported various device drivers including a software-based renderer. The OpenGL development effort culminated in the announcement and delivery of OpenGL-based systems in the fall of 1997.

Software Implementation

In our implementation, we focused on the hardware's ability to accelerate major portions of the rendering pipeline. For the software, we focused on its ability to ensure that the hardware could run at full performance. A fast graphics accelerator is not needed if the driving software cannot keep the hardware busy. The resulting software architecture and implementation was designed from a system viewpoint. Decisions were based on system requirements to avoid overoptimizing each individual component and still not achieve the desired results. An overview of the HP OpenGL software architecture is provided in the article on page 9. Another software-related issue is provided in the article on page 35, which discusses issues associated with porting a UNIX-based OpenGL implementation to Windows NT.

Hardware Systems

The new graphics systems are able to support OpenGL, Starbase, PHIGS, and PEX rendering semantics in hardware. Being able to support the OpenGL API means that there is hardware support for accelerating the full feature set of OpenGL instead of just having a simple frame buffer in which all or most of the OpenGL features are implemented in software. These systems are the VISUALIZE fx2, VISUALIZE fx4, and VISUALIZE fx6 graphics accelerator products. These systems differ in the amount of graphics acceleration they provide, the number of image planes, and the optional OpenGL features they provide. In addition to the base graphics boards, a texture mapping option is available for the fx4 and fx6 accelerators. The article on page 28 provides an overview of the new graphics hardware developed to support OpenGL.

Engineering Process

To meet the required delivery dates of OpenGL with a high level of confidence and quality, we used a new process to compress the time between first silicon and manufacturing release. The article on page 41 describes the

The Fast-Break Program

In basketball, a rapid offensive transition is called a fast-break. The fast-break program is about the transition game for OpenGL on HP systems. A key part of the HP transition to OpenGL is applications, because applications enable volume shipments of systems. Having the right applications is necessary for a successful OpenGL product, but it is also important that the applications run with outstanding performance and reliability. Fast-break is about both aspects—getting the applications on HP systems and ensuring that they have outstanding performance and reliability.

Fast-break began by working with application developers in the early stages of the OpenGL program to understand their requirements for the HP OpenGL product. These requirements helped to drive the initial OpenGL product definition.

As the program progressed, the Fast-break team developed a suite of tools that enabled detailed analysis of OpenGL applications. Analysis of key applications was used to further refine our OpenGL product performance and functionality. Analysis also yielded a set of synthetic API benchmarks that represented the behavior of key applications. These synthetic benchmarks enabled HP to perform early hands-on evaluation of the OpenGL product long before the actual applications were ported to HP.

Pre-porting laid the groundwork for the actual porting of applications to HP's implementation of OpenGL. The first phase of the porting took place during the OpenGL beta program. In this program, the HP fast-break team worked closely with selected application developers to initiate the porting effort. A software-only implementation of the OpenGL product was used, which enabled the beta program to take place even before hardware was available.

As hardware became available, the beta program was superseded by the early access program. This program included the original beta participants and additional selected developers. In both the beta and early access programs, HP found that the homework done earlier by the fast-break team paid big dividends. Most applications were ported to HP in just a few days and, in some cases, just a few hours!

Although not completely defect-free, these early versions of OpenGL were uniformly high-performance and high-quality products. By accelerating the application porting effort, HP was able to identify and resolve the few remaining issues before the product was officially released.

The ongoing involvement of the fast-break team with the OpenGL product development teams helped HP do it right the first time by delivering a high-quality, high-performance implementation of OpenGL and enabling rapid porting of key applications to the HP product.

engineering process we used to accelerate the time to market for OpenGL.

Graphics Middleware

A fast graphics API is not always enough. Leading edge CAD modelling problems far exceed the interactive capacity of graphical super workstations. For example, try spinning a complete CAD model of a Boeing 777 at 30 frames per second on any system.

What is needed is a new approach to solving the rendering problem of very large models. The goal is to trade off between frame rate, image quality, and system cost. HP has introduced a toolkit for use by CAD ISVs to assist them in solving this problem. The toolkit is called DirectModel and is described on page 19.

HP-UX Release 10.20 and later and HP-UX 11.00 and later (in both 32- and 64-bit configurations) on all HP 9000 computers are Open Group UNIX 95 branded products.

UNIX is a registered trademark of The Open Group.

X/Open is a registered trademark and the X device is a trademark of X/Open Company Limited in the UK and other countries.

Microsoft is a U.S. registered trademark of Microsoft Corporation.

Windows is a U.S. registered trademark of Microsoft Corporation.

Silicon Graphics and OpenGL are registered trademarks of Silicon Graphics Inc. in the United States and other countries.

An Overview of the HP OpenGL® Software Architecture

Kevin T. Lefebvre
Robert J. Casey
Michael J. Phelps
Courtney D. Goeltzenleuchter
Donley B. Hoffman

OpenGL is a hardware-independent specification of a 3D graphics programming interface. This specification has been implemented on many different vendors' platforms with different CPU types and graphics hardware, ranging from PC-based board solutions to high-performance workstations.

The OpenGL API defines an interface (to graphics hardware) that deals entirely with rendering 3D primitives (for example, lines and polygons). The HP implementation of the OpenGL standard does not provide a one-to-one mapping between API functions and hardware capabilities. Thus, the software component of the HP OpenGL product fills the gaps by mapping API functions to OpenGL-capable systems.

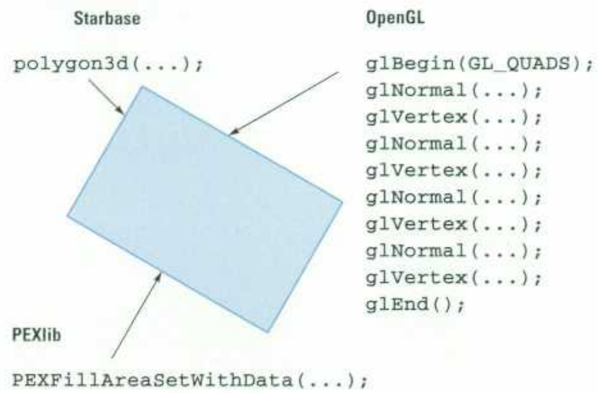
Since OpenGL is an industry-standard graphics API, much of the differentiating value HP delivers is in performance, quality, reliability, and time to market. The central goal of the HP implementation is to ship more performance and quality much sooner.

What is OpenGL?

OpenGL differs from other graphics APIs, such as Starbase, PHIGS, and PEX (PHIGS extension in X), in that it is vertex-based as opposed to primitive-based. This means that OpenGL provides an interface for supplying a single vertex, surface normal, color, or texture coordinate parameter in each call. Several of the calls between an OpenGL glBegin and glEnd pair define a primitive that is then rendered. **Figure 1** shows a comparison of the different API call formats used to render a rectangle. In PHIGS a single call could render a primitive by referencing multiple vertices and their associated data (such as normals and color) as parameters to the call. This difference in procedure calls per primitive (one versus eight for a shaded triangle) posed a performance challenge for our implementation.

Figure 1

Graphics API call comparison.



An OpenGL implementation consists of the following elements:

- A rendering library (GL) that implements the OpenGL specification (the rendering pipeline)
- A utility library (GLU) that implements useful utility functions that are layered on top of OpenGL (for example, surfaces, quadratics, and tessellation functions)
- An interface to the system's windowing package, including GLX for X Window Systems on the UNIX operating system and WGL for Microsoft Windows®.

Implementation Goals

The goals we defined for the OpenGL program that helped to shape our implementation were to:

- Achieve and sustain long term price/performance leadership for OpenGL applications running on HP platforms
- Develop a scalable architecture that supports OpenGL on a wide range of HP platforms and graphics devices.

The rest of this article will provide more details about our OpenGL implementation and show how these goals affected our system design.

OpenGL API

In general, OpenGL defines a traditional 3D pipeline for rendering 3D primitives. This pipeline takes 3D coordinates as input, transforms them based on orientation or viewpoint, lights the resulting coordinates, and then renders them to the frame buffer (**Figure 2**).

To implement and control this pipeline, the OpenGL API provides two classes of entry points. The first class is used to create 3D geometry as a combination of simple primitives such as lines, triangles, and quadrilaterals. The entry points that make up this class are referred to as the vertex API, or VAPI, functions. The second class, called the state class, manipulates the OpenGL state used in the different rendering pipeline stages to define how to operate (transform, clip, and so on) on the primitive data.

VAPI Class

OpenGL contains a series of entry points that when used together provide a powerful way to build primitives. This flexible interface allows an application to provide primitive data directly from its private data structures rather than requiring it to define structures in terms of what the API requires, which may not be the format the application requires.

Primitives are created from a sequence of vertices. These vertices can have associated data such as color, surface normal, and texture coordinates. These vertices can be grouped together and assigned a type, which defines how the vertices are connected and how to render the resulting primitive.

The VAPI functions available to define a primitive include glVertex (specify its coordinate), glNormal (define a surface normal at the coordinate), glColor (assign a color to the coordinate), and several others. Each function has several forms that indicate the data type of the parameter (for example, int, short, and float), whether the data is passed as a parameter or as a pointer to the data, and whether the data is one-, two-, three-, or four-dimensional. Altogether there are over 100 VAPI entry points that allow for maximum application flexibility in defining primitives.

The VAPI functions glBegin and glEnd are used to create groups of these vertices (and associated data). glBegin takes a type parameter that defines the primitive type and a count of vertices. The type can be point, line, triangle,

Figure 2

Graphics pipeline.

triangle strip, quadrilateral, or polygon. Based on the type and count, the vertices are assembled together as primitives and sent down the rendering pipeline.

For added efficiency and to reduce the number of procedure calls required to render a primitive, vertex arrays were added to revision 1.1 of the OpenGL specification. Vertex arrays allow an application to define a set of vertices and associated data before their use. After the vertex data is defined, one or more rendering calls can be issued that reference this data without the additional calls of glBegin, glEnd, or any of the other VAPI calls.

Finally, OpenGL provides several rendering routines that do not deal with 3D primitives, but rather with rectangular areas of pixels. From OpenGL, an application can read, copy, or draw pixels to or from any of the OpenGL image, depth, or texture buffers.

State Class

The state class of API functions manipulates the OpenGL state machine. The state machine defines how vertices are operated on as they pass through the rendering pipeline. There are over 100 functions in this class, each controlling a different aspect of the pipeline. In OpenGL most state information is orthogonal to the type of primitive being operated on. For example, there is a single primitive color rather than a specific line color, polygon color, or point color. These state manipulation routines can be grouped as:

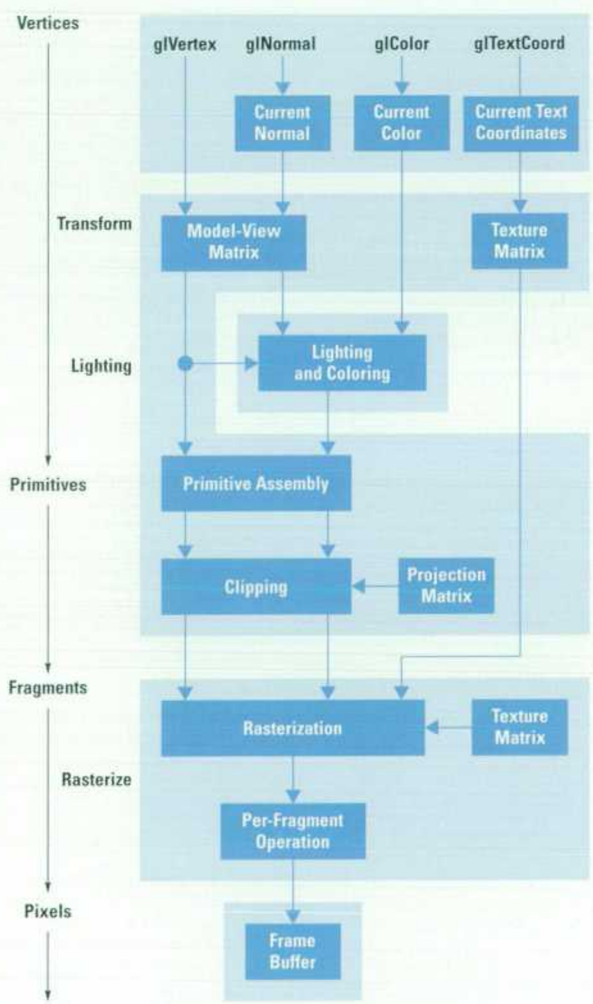
- Coordinate transformation
- Coloring and lighting
- Clipping
- Rasterization
- Texture mapping
- Fog
- Modes and execution.

Pipeline

Coordinate data (such as vertex, color, and surface normal) can come directly from the application, indirectly from the application through the use of evaluators,* or from a stored display list that the application had previously created. The coordinates flow into the pipeline as discrete points and are operated on (transformed) individually. At a certain point in the pipeline the vertices are assembled into primitives, and they are operated on at the primitive level (for example, clipping). Next, the primitives are rasterized into fragments in which operations like depth testing occur on each fragment. The final result is pixels that are written into the frame buffer. This more complex OpenGL pipeline is shown in **Figure 3**.

Conceptually, the transform stage takes application-specified object-space coordinates and transforms them to eye-space coordinates (the space that positions the object with respect to the viewer) with a model-view matrix. Next, the eye coordinates are projected with a

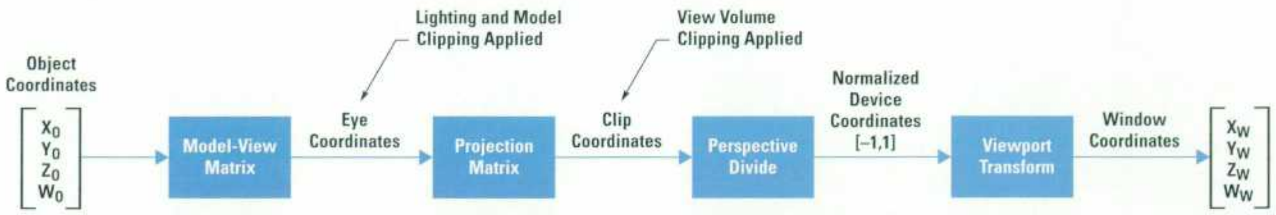
Figure 3
OpenGL pipeline.



* Evaluators are functions that derive coordinate information based on parametric curves or surfaces and basic functions.

Figure 4

Transformation from object-space to window coordinates.



projection matrix, divided by the perspective, and then transformed by the viewport matrix to get them to screen space (relative to a window). This process is summarized in **Figure 4**.

In the lighting stage, a color is computed for each vertex based on the lighting state. The lighting state consists of a number of lights, the type of each light (such as positional or spotlight), various parameters of each light (for example, position, pointing direction, or color), and the material properties of the object being lit. The calculation takes into consideration, among other things, the light state and the distance of the coordinate to each light, resulting in a single color for the vertex.

In rasterization, pixels are written based on the primitive type, and the pixel value to be written is based on various rasterization states (such as texture mapping enabled, or polygon stipple enabled). OpenGL refers to the resulting pixel value as a fragment because in addition to the pixel value, there is also coverage, depth, and other state information associated with the fragment. The depth value is used to determine the visibility of the pixel as it interacts with existing objects in the frame buffer. While the coverage, or alpha, value blends the pixel value with the existing value in the frame buffer.

Software Architecture

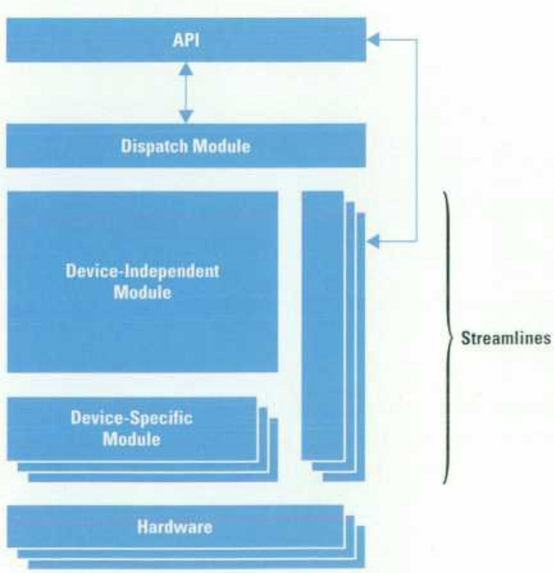
One of the main design goals for the HP OpenGL software architecture was to maximize performance where it would be most effective. For example, we decided to focus on reducing overhead to hardware-accelerated paths and to base design decisions on application use, minimizing the effort and cost required to support future system hardware. The resulting architecture is composed of two major components: a device-independent module and a device-specific module. A simple block diagram is shown in **Figure 5**.

The dispatch component is responsible for handling OpenGL API calls and sending them to the appropriate receiver. OpenGL can be in one of the following modes:

- Protocol mode in which API calls are packaged up and forwarded to a remote system for execution
- Display list creation mode in which API calls are stored in a display list for later execution
- Direct rendering mode in which API calls are intended for immediate rendering on the local screen.

Figure 5

OpenGL architecture.



The primary application path of any importance is the immediate rendering path. While in direct rendering mode the performance of all functions is important, but the performance of the VAPI calls is even more critical because of the increased frequency of rendering calls over other types of calls, like state setting. Any overhead in transferring application rendering commands to the hardware reduces overall performance significantly. See the "System Design Results" section in this article on page 14 for a discussion on some of these issues.

The device-independent module is the target for all the OpenGL state manipulation calls, and in some situations, for VAPI calls such as display list or protocol generation. This module contains state management, all system control logic, and a complete software implementation of the OpenGL rendering pipeline up to the rasterization stage, which is used in situations where the hardware does not support an OpenGL feature. The device independent module is made up of several submodules, including:

- GLX (OpenGL GLX support module) for handling window system dependent components, including context management, X Window System interactions, and protocol generation
- SUM (system utilities module) for handling system dependent components, including system interactions, global state management, and memory management
- OCM (OpenGL control module) for handling OpenGL state management, parameter checking, state inquiry support, and notification of state changes to the appropriate module
- PCM (pipeline control module) for handling graphics pipeline control, state validation, and the software rendering pipeline
- DLM (display list module) for handling display list creation and execution.

The device-specific module is basically an abstracted hardware interface that resides in a separate shared library. Based on what hardware is available, the device-independent code dynamically loads the appropriate device-specific module. In general the device-specific module is called only by the device-independent module, never by the API, and converts the requests to hardware-specific operations (register loads, operation execute). In addition to a device-specific module for the VISUALIZE fx series of graphics hardware, there is a virtual memory driver device-specific module for handling OpenGL operations on GLX pixmaps (virtual-memory-based image buffers) or for rendering to hardware that does not support OpenGL semantics.

The final key component of the architecture is streamlines. Streamlines are part of the device-specific module but are unique in that they are associated directly with the API. On geometry-accelerated devices like the VISUALIZE fx series, the hardware can support the full set of VAPI calls. To minimize overhead and maximize performance, the calls are targeted to optimized routines that communicate directly with the hardware. In many cases these routines are coded in PA RISC 1.1 or PA RISC 2.0 assembly language or C. At initialization time the appropriate routines are loaded in the dispatch table based on the system type and are dynamically selected at run time.

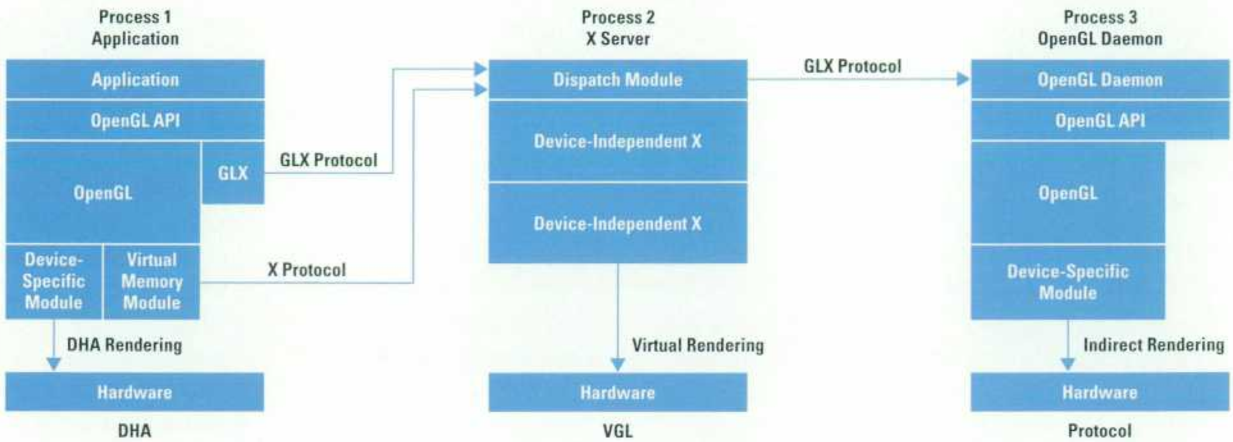
An important thing to understand about streamlines is that they can only be called when the current state is "clean" and the hardware supports the current rendering mode. An example of "not clean" is when the viewing matrix has been changed, and the hardware needs to be updated with the current transformation matrix. Because the application can make several different calls to manipulate the matrix, computing the state based on the viewing matrix and loading the hardware is deferred until it is actually needed. For example, when a primitive is to be rendered (initiated via a glBegin call), the state is made clean (validated) by the device-independent code and subsequent VAPI calls can be dispatched directly to the streamlines. Another situation in which streamlines cannot be called is when the hardware does not support a feature, such as texture mapping in the VISUALIZE fx^2 display hardware. In this situation the VAPI entry points do not target the streamlines but rather the device-independent code that implements what is called a general path, or in other terms, a software rendering pipeline.

Three-Process Model

Under the X Window System on the UNIX operating system, the OpenGL architecture uses a three-process model to support the direct and indirect semantics of OpenGL. In our implementation, we have leveraged our existing direct hardware access (DHA) technology to provide industry-leading local rendering performance. This has been

Figure 6

Three-process rendering model.



coupled with two distinct remote rendering modes, making our OpenGL implementation one of the most flexible implementations in the industry. These rendering modes are based upon the three-process rendering model shown in **Figure 6**. This model supports three rendering modes: direct, indirect, and virtual.

Direct Rendering. Direct rendering through DHA provides the highest level of OpenGL performance and is used whenever an OpenGL application is connected to a local X server running on a workstation with VISUALIZE fx graphics hardware. For all but a few operations, the application process communicates directly with the graphics hardware, bypassing the interprocess communication overhead between the application and the X server.

Indirect Rendering (Protocol). Indirect rendering is used primarily for remote operation when the target X server is running on a different workstation than the user application. In this mode, the OpenGL API library emits GLX protocol which is interpreted by a receiving X server that supports the GLX extension. The receiving server can be HP, Sun Microsystems, Silicon Graphics® International, or any other X server that supports the GLX server extension. In the HP OpenGL implementation, the receiving X server passes nearly all GLX protocol directly on to an OpenGL daemon process that uses DHA for maximum performance. Note that immediate mode rendering performance through protocol can be severely limited by the time it takes to send geometric data over the network. However, when display lists are used, geometric data is cached in the OpenGL daemon and remote OpenGL rendering can be as fast or sometimes even faster than local DHA rendering.

Virtual Rendering. As a value-added feature, HP OpenGL also provides a virtual GL rendering mode not available in other OpenGL implementations. Virtual rendering allows an OpenGL application to be displayed on any X server or X terminal even if the GLX extension is not supported on that server. This is accomplished by rendering through the virtual memory driver to local memory and then issuing the standard XPutImage protocol to display images on the target screen. Although flexible, virtual GL is typically the slowest of the OpenGL rendering modes. However, virtual GL rendering performance can be increased significantly by limiting the size of the output window

System Design Results

To deliver industry-leading OpenGL performance, we combined graphics hardware, libraries, and drivers. The hardware is the core enabler of performance. Although the excellence of each part is important, the overall system design is even more so. How well the operating system, compilers, libraries, drivers, and hardware fit together in the system design determines the overall result. We worked closely with teams in four HP R&D labs to optimize the system design, apply our design values to partitioning the system, balance performance bottlenecks, and simplify the overall architecture and interfaces. The following section describes some examples of applying our system design principles to the most important aspects of 3D graphics applications.

Improving OpenGL Application Performance

OpenGL required a radical change from the existing (legacy) HP graphics APIs. In analyzing the model for our legacy graphics APIs, we realized that the same model would have considerable overhead for OpenGL, which requires many more procedure calls. **Figure 1** compares the calls required to generate the same shaded quadrilateral.

To have a competitive OpenGL, we needed to reduce or eliminate function calls and locking overhead. We did this with two system design initiatives called *fast procedure calls* and *implicit device locking*.

Fast Procedure Calls. Two of our laboratories (the Graphics Systems Laboratory and the Cupertino Language Laboratory) worked together to create a specification for a new, faster calling convention for making calls to shared library components. This reduced the cost to one-fourth the cost of the previous mechanism.

OpenGL is a state machine. When the application calls an OpenGL function, different things happen depending on the current state. We also wanted to support different devices with varying degrees of support in the same OpenGL library. We needed a dynamic method of dispatching API function calls to the correct code to enable the appropriate functionality without compromising performance. Given this requirement, a naive implementation of OpenGL might define each of its API functions like the following:

```
void glVertex3fv (const GLfloat *v)
{
  switch (context.whichFunction)
  {
   case HW_STREAMLINE:
     HW_STREAMLINE_glVertex3fv(v);
     break;
   case GENERAL_PATH:
     GENERAL_PATH_glVertex3fv(v);
     break;
   case GLX_PROTOCOL:
     GLX_PROTOCOL_glVertex3fv(v);
     break;
   case diSPLAY_LIST:
     diSPLAY_LIST_glVertex3fv(v);
     break;
    ...
  }

}
```

However, this is a very impractical implementation in terms of both performance and software maintainability. We decided that the most efficient method of achieving this kind of dynamic dispatching was to retarget the API function calls at their source—the application code. Any call into a shared library is really a call through a pointer. The procedure name that the application calls is associated with a particular pointer. Conceptually, what we needed was a mechanism to manage the contents of those pointers. To accomplish this, we needed more assistance from the engineers in the compiler and linker groups.

In simplified terms, the OpenGL library maintains a procedure link table. Each entry in the procedure link table is associated with a particular function name and is composed of two pointers. One points to the code that is to be called, and the other, the link table pointer, points to the table used by shared library code (known as PIC, or position-independent code) to locate global data. When the compiler generates a call to an OpenGL function, it loads the appropriate registers with the two fields in the associated procedure link table entry and then branches to the function. Since OpenGL controls the contents of the procedure link table, it can change the contents of these fields during execution. This allows OpenGL to choose the appropriate code based on the OpenGL state dynamically.

For example, assume that we have a graphics device that, except for texture mapping, supports the OpenGL pipeline in hardware. In this case the scheduling code will find texture mapping enabled (meaning that the device cannot handle texture mapping) and choose the GENERAL_PATH_glVertex3fv code path, which performs software texture mapping. The HW_STREAMLINE_glVertex3fv code paths are taken if texture mapping is not enabled.

Implicit Device Locking. Graphics devices are a shared system resource. As such, there must be some control when an application has access to the graphics device so that two applications are not attempting to use the device at the same time. Normally the operating system manages such shared resources via standard operating system interfaces (open, close, read, write, and ioctl).

However, to get the maximum performance possible for graphics applications, a user process will access the graphics device directly through our 3D API libraries, rather than use the standard operating system interfaces.

This means that before OpenGL, the HP graphics libraries had to assume the task of managing shared access to the graphics device.

Before OpenGL, we used a relatively lightweight fast lock at the entry and exit of those library routines that actually accessed the device. With the high frequency of function calls in OpenGL, performing this lock and unlock step for each function call would exact a severe performance penalty, similar to the procedure call problem discussed earlier.

To solve this problem, HP engineers invented a technique called *implicit device locking*. When a process tries to access the graphics hardware and does not own the device, a virtual memory protection fault exception will be generated. The kernel must detect that this protection fault was an attempted graphics device access instead of a fault from trying to access something like an invalid address, a swapped out page, or from doing a copy on a write page.

The graphics fault alerts the system that there is another process trying to access the graphics device. The kernel then makes sure that the graphics device context is saved, and the graphics context for the next process is restored. After the graphics context switch is complete, the new process is allowed to continue with access to the device, and permission is taken away from all other processes. This allows the current process that owns the device to have zero overhead access.

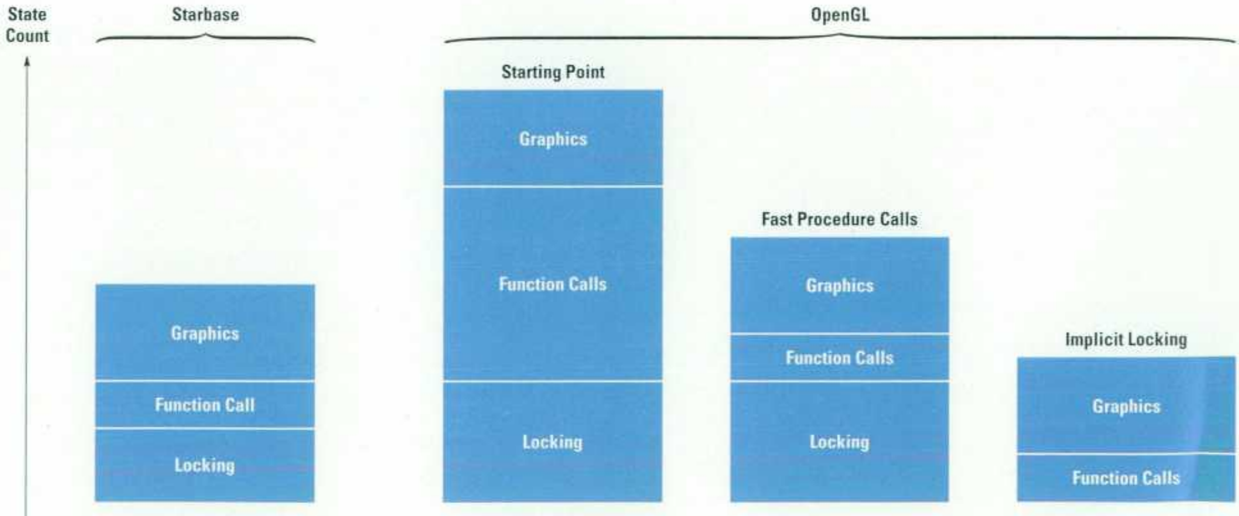
This method removes the requirement that the 3D graphics API library must explicitly lock the graphics device while accessing it. This means that the overhead associated with device locking, which was an order of magnitude more than with Starbase, is completely eliminated (see **Figure 7**).

This dramatic improvement in performance is made possible by improvements in the HP-UX* kernel and careful design of the graphics hardware. The basic idea is that when multiple graphics applications are running, the HP-UX kernel will ensure that each application gets its fair share of exclusive time to access the graphics device.

OpenGL was not the only API to benefit from implicit locking. The generality of the design allowed us to use the same mechanism to eliminate the locking code from Starbase as well. Keeping the whole system in mind while developing this technology allowed us to expand the benefit beyond the original problem—excessive overhead from locking for OpenGL.

Figure 7

State count comparison.



Hardware and Software Trade-offs

Keeping the whole picture in mind allowed us to make software and hardware trade-offs to simplify the system design. The criteria were based on performance criticality, frequency of use, system complexity, and factory cost.

For example, the hardware was designed to understand both OpenGL and Starbase windows. OpenGL requires the window origin to be in the lower left corner, whereas Starbase requires it to be in the upper left. Putting the intelligence in the hardware reduced the overall system complexity.

Nearly all OpenGL features are hardware accelerated. Of course, all vertex API formats and dimensions are streamlined and accelerated in hardware for maximum primitive performance. Similarly, all fragment pipeline operations had to be supported in hardware because fragment operations touch every pixel and software performance would not be sufficient. To maximize primitive performance, we also hardware-accelerated nearly every geometry pipeline feature. For example, all lighting modes, fog modes, and arbitrary clip planes are hardware-accelerated. Very few OpenGL features are not hardware-accelerated.

Based on infrequent use and the ability to reasonably accelerate in software, we implemented the following functions in software: RasterPos, Selection, Feedback, Indexed Lighting, and Indexed Fog. Infrequent use and factory cost also encouraged us to implement accumulation buffer support in software. (Accumulation is an operation that blends data between the frame buffer and the accumulation buffer, allowing effects like motion blur.)

State Change

Through systems design we achieved dramatic results in application performance by focusing on the design for OpenGL state change operations.

Application graphics performance is a function of both primitive and state change (attributes) performance. We have designed our OpenGL implementation to maximize primitive performance and minimize the costs of state changes.

State changes include all the function calls that modify the OpenGL modal state, including coordinate transformations, lighting state, clipping state, rasterization state, and texture state. State change does not include primitive calls, pixel operations, display list calls, or current state calls. Current state encompasses all the OpenGL calls that can occur either inside or outside glBegin() and glEnd() pairs (for example, glColor(), glNormal(), glVertex()).

There are two classes of state changes: fragment pipeline and geometry pipeline. Fragment pipeline state changes control the back end, or rasterization stage, of the graphics pipeline. This state includes the depth test enable (z-buffer hidden surface removal) and the line stipple definition (patterned lines such as dash or dot). Geometry pipeline state changes control the front end of the graphics pipeline. This state includes transformation matrices, lighting parameters, and front and back culling parameters. Fragment pipeline state changes are generally less costly than geometry pipeline state changes.

Our systems design focussed on several areas that resulted in large application performance gains. We realized that the performance of our state change implementation could significantly affect application performance. We decided that this was important enough to require a redesign of the state change modules and not just tuning. Applying these considerations led us to implement immediate and deferred validation schemes and provide redundancy checks at the beginning of each state change entry point.

Validation. We implemented different immediate and deferred validation schemes* for different classes of state changes. Geometry pipeline state changes are handled by deferred validation because they tend to be more complex, requiring massaging of the state. They are also more interlocked because changing one piece of state requires modifying another piece of state (for example, matrix changes cause changes to the light state). For us, deferred validation resulted in a simple design and increased performance, reliability, and maintainability. For fragment pipeline state changes, we chose immediate validation because this state is relatively simple and noninterlocked.

Redundancy Checks. Redundancy checks are done for all OpenGL API calls. Because our analysis showed that applications often call state changing routines with a redundant state (for example, new value==current value), we

* Validation is the mechanism that verifies that the current specified state is legal, computes derived information from the current state necessary for rendering (for example an inverse matrix for lighting based on the current model matrix), and loads the hardware with the new state.

wanted a design in which this case performs well. Therefore, our design includes redundancy checks at the beginning of each state change entry point, which allows a quick return without exercising the unnecessary validation code.

Results. For state-change intensive applications, these design decisions put us in a leadership position for OpenGL application performance, and we achieved greater than a 2× performance gain over our previous graphics libraries. Smaller application performance gains were achieved throughout our OpenGL implementation with the state-change design.

Conclusion

ISVs and customers indicate that we have met our application leadership price and performance goals that we set at the start of the program. We have also exceeded the performance metrics we committed to at the beginning of the project. For more information regarding our performance results, visit the web site:

http://www.spec.org/gpc/opc

For long-term sustainability of our price and performance leadership, we have continued working closely with our ISVs to tune our implementation in areas that improve application performance. In addition, new CPUs are planned that will allow our implementation to run faster without any effort on our part, and cost reductions are continuing in graphics hardware.

The goal to develop an implementation that can support a wide range of CPU or graphics devices has already been demonstrated. We support three graphics devices that have different performance levels (all based on the same hardware architecture) and a pure software implementation that supports simple frame buffer devices on UNIX and Windows NT systems.

Bibliography

1. M. Woo, J. Neider, and T. Davis, *OpenGL Programming Guide*, second edition, Addison Wesley, 1997.

2. *OpenGL Reference Manual*, second edition, OpenGL Architecture Review Board, 1997.

Kevin T. Lefebvre

This author's biography appears on page 6.

Robert J. Casey

This author's biography appears on page 41.

Michael J. Phelps

A graduate of the University of Connecticut in 1983 with a BSEE degree, Michael Phelps is now involved in current product engineering for the VISUALIZE fx family of graphics subsystems. He came to HP in 1994. He was born in Glen Cove, New York. He is married and enjoys hunting, fishing, and competitive shooting sports.

Courtney D. Goeltzenleuchter

Courtney Goeltzenleuchter is a software engineer at the HP Performance Desktop Computer Operation. With HP since 1995, he currently is responsible for design and development of graphics drivers and hardware and software interfaces for the HP 3D graphics accelerators. He graduated from the University of California at Berkeley in 1987 with a BA degree in computer science. Born in Tucson, Arizona, Courtney is married and has one child. He enjoys hiking, reading science fiction, and playing with his computer.

Donley B. Hoffman

Donley Hoffman is a software engineer at the Workstation Systems Division and is responsible for maintenance and support for current and future OpenGL products. He graduated from New Mexico State University in 1974 with a BS degree in computer science. He came to HP in 1985. Born in Alamogardo, New Mexico, Don is married and has three children. His outside interests include skiing, tennis, playing the oboe and piano, running, reading, hiking, and snorkling.

The DirectModel Toolkit: Meeting the 3D Graphics Needs of Technical Applications

Brian E. Cripe

Thomas A. Gaskins

The increasing use of 3D modeling for highly complex mechanical designs has led to a demand for systems that can provide smooth interactivity with 3D models containing millions or even billions of polygons.

DirectModel* is a toolkit for creating technical 3D graphics applications. Its primary objective is to provide the performance necessary for interactive rendering of large 3D geometry models containing millions of polygons. DirectModel is implemented on top of traditional 3D graphics applications programming interfaces (APIs), such as Starbase or OpenGL®. It provides the application developer with high-level 3D model management and advanced geometry culling and simplification techniques. **Figure 1** shows DirectModel's position within the architecture of a 3D graphics application.

This article discusses the role of 3D modeling in design engineering today, the challenges of implementing 3D modeling in mechanical design automation (MDA) systems, and the 3D modeling capabilities of the DirectModel toolkit.

Visualization in Technical Applications

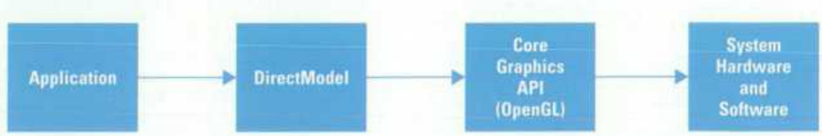
The Role of 3D Data

3D graphics is a diverse field that is enjoying rapid progress on many fronts. Significant advances have been made recently in photorealistic rendering, animation quality, low-cost game platforms, and state-of-the-art immersive

* DirectModel was jointly developed by Hewlett-Packard and Engineering Animation Incorporated of Ames, Iowa.

Figure 1

Application architecture.



Brian E. Cripe

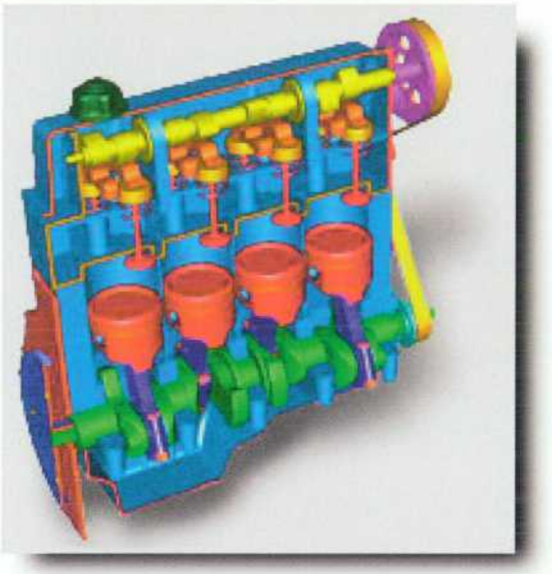
With HP since 1982, Brian Cripe is a project manager at the HP Corvallis Imaging Operation. He is responsible for DirectModel relationships with developers, Microsoft®, and Silicon Graphics®. He has worked on the HP ThinkJet and DeskJet printers and the Common Desktop Environment. He received a BSEE in 1982 from Rice University. Brian was born in Anapolis, Brazil, is married and has two daughters.

Thomas A. Gaskins

Thomas Gaskins was the project leader for the DirectModel project at the HP Corvallis Imaging Operation. With HP since 1995, he received a BS degree in mechanical engineering (1993) from the University of California at Santa Barbara. He specialized in numerical methods. His professional interests include 3D graphics and software architecture.

Figure 2

A low-resolution image of a 3D model of an engine consisting of 150,000 polygons.



virtual reality* applications. The Internet is populated with 3D virtual worlds and software catalogs are full of applications for creating them. An example of a 3D model is shown in **Figure 2**.

What do these developments mean for the users of technical applications (the scientists and engineers who pioneered the use of 3D graphics as a tool for solving complex problems)? In many ways this technical community is following the same trends as the developers and users of nontechnical applications such as 3D games and interactive virtual worlds. They are interested in finding less expensive systems for doing their work, their image quality standards are rising, and their patience with poor interactive performance is wearing thin.

However, there are other areas where the unique aspects of 3D data for technical applications create special requirements. In many applications the images created from the 3D data that are displayed to the user are the goal. For example, the player of a game or the pilot in a flight simulator cares a lot about the quality and interactivity of the images, but cares very little about the data used by the system to create those images. In contrast, many technical users of 3D graphics consider their data to be the most important component. The goal is to create, analyze, or improve the data, and 3D rendering is a useful means to that end.

* Immersive virtual reality is a technology that "immerses" the viewer into a virtual reality scene with head-mounted displays that change what is viewed as the user's head rotates and with gloves that sense where the user's hand is positioned and apply force feedback.

This key distinction between data that is the goal itself and data that is a means to an end leads to major differences in the architectures and techniques for working with those data sets.

3D Model Complexity

Understanding the very central role that data holds for the technical 3D graphics user immediately leads to the questions of what is that data and what are the significant trends over time? The short answer is that the size of the data is big and the amount and complexity of that data is increasing rapidly. For example, a mechanical engineer doing stress analysis may now be tackling problems modeled with millions of polygons instead of the thousands that sufficed a few years ago.

The trends in the mechanical design automation (MDA) industry are good examples of the factors causing this growth. In the not-too-distant past mechanical design was accomplished using paper and pencil to create part drawings, which were passed on to the model shop to create prototype parts, and then they were assembled into prototype products for testing. The first step in computerizing this process was the advent of 2D mechanical drafting applications that allowed the mechanical engineers to replace their drafting boards with computers. However, the task was still to produce a paper drawing to send to the model shop. The next step was to replace these 2D drafting applications with 3D solid modelers that could model the complete 3D geometry of a part and support tasks such as static and dynamic design analysis to find such things as the stress points when the parts move. This move to 3D solid modeling has had a big impact at many companies as a new technique for designing parts. However, in many cases it has not resulted in a fundamental change to the process for designing and manufacturing whole products.

Advances. In the last few years advances in the mechanical design automation industry have increasingly addressed virtual prototyping and other whole-product

Fahrenheit

Hewlett-Packard, Microsoft, and Silicon Graphics are collaborating on a project, code-named "Fahrenheit," that will define the future of graphics technologies. Based on the creation of a suite of APIs for DirectX on the Windows® and UNIX® operating systems, the Fahrenheit project will lead to a common, extensible architecture for capitalizing on the rapidly expanding marketplace for graphics.

Fahrenheit will incorporate the Microsoft Direct3D and DirectDraw APIs with complementary technologies from HP and Silicon Graphics. HP is contributing DirectModel to this effort and is working with Microsoft and Silicon Graphics to define the best integration of the individual technologies.

design issues. This desire to create new tools and processes that allow a design team to design, assemble, operate, and analyze an entire product in the computer is particularly strong at companies that manufacture large and complex products such as airplanes, automobiles, and large industrial plants. The leading-edge companies pioneering these changes are finding that computer-based virtual prototypes are much cheaper to create and easier to modify than traditional physical prototypes. In addition they support an unprecedented level of interaction among multiple design teams, component suppliers, and end users that are located at widely dispersed sites.

This move to computerized whole-product design is in turn leading to many new uses of the data. If the design engineers can interact online with their entire product, then each department involved in product development will want to be involved. For example, the marketing department wants to look at the evolving design while planning their marketing campaign, the manufacturing department wants to use the data to ensure the product's manufacturability, and the sales force wants to start showing it to customers to get their feedback.

These tasks all drive an increased demand for realistic models that are complete, detailed, and accurate. For example, mechanical engineers are demanding new levels of realism and interactivity to support tasks such as positioning the fasteners that hold piping and detecting interferences created when a redesigned part bumps into one of the fasteners. This is a standard of realism that is very different from the photorealistic rendering requirements of other applications and to the technical user, a higher priority.

Larger Models. These trends of more people using better tools to create more complete and complex data sets combine to produce very large 3D models. To understand this complexity, imagine a complete 3D model of everything you see under the hood of your car. A single part could require at least a thousand polygons for a detailed representation, and a product such as an automobile is assembled from thousands of parts. Even a small product such as an HP DeskJet printer that sits on the corner of a desk requires in excess of 300,000 triangles[1] for a detailed model. A car door with its smooth curves, collection of controls, electric motors, and wiring harness can require one million polygons for a detailed model—the car's power train can consist of 30 million polygons.[2]

These numbers are large, but they pale in comparison to the size of nonconsumer items. A Boeing 777 airplane contains approximately 132,500 unique parts and over 3,000,000 fasteners,[3] yielding a 3D model containing more than 500,000,000 polygons.[4] A study that examined the complexity of naval platforms determined that a submarine is approximately ten times more complex than an airplane, and an aircraft carrier is approximately ten times more complex than a submarine.[5] 3D models containing hundreds of millions or billions of polygons are real today.

As big as these numbers are, the problem does not stop there. Designers, manufacturers, and users of these complex products not only want to model and visualize the entire product, but they also want to do it in the context of the manufacturing process and in the context in which it is used. If the ship and the dry dock can be realistically modeled and combined, it will be far less expensive to find and correct problems before they are built.

Current System Limitations

If the task faced by technical users is to interact with very large 3D models, how are the currently available systems doing? In a word, badly. Clearly the graphics pipeline alone is not going to solve the problem even with hardware acceleration. Assuming that rendering performance for reasonable interactivity must be at least 10 frames per second, a pipeline capable of rendering 1,000,000 polygons per second has no hope of interactively rendering any model larger than 100,000 polygons per frame. Even the HP VISUALIZE fx^6, the world's fastest desktop graphics system, which is capable of rendering 4.6 million triangles per second, can barely provide 10 frames per second interactivity for a complete HP DeskJet printer model.

This is a sobering reality faced by many mechanical designers and other technical users today. Their systems work well for dealing with individual components but come up short when facing the complete problem.

Approaches to Solving the Problem

There are several approaches to solve the problem of rendering very complex 3D models with interactive performance. One approach is to increase the performance of the graphics hardware. Hewlett-Packard and other graphics hardware vendors are investing a lot of effort in this approach. However, increasing hardware performance alone is not sufficient because the complexity of many customers' problems is increasing faster than gains in hardware performance. A second approach that must also be explored involves using software algorithms to reduce the complexity of the 3D models that are rendered.

Complex Data Sets

To understand the general data complexity problem, we must examine it from the perspective of the application developer. If a developer is creating a game, then it is perfectly valid to search for ways to create the imagery while minimizing the amount of data behind it. This approach is served well by techniques such as extensive use of texture maps on a relatively small amount of geometry. However, for an application responsible for producing or analyzing technical data, it is rarely effective to improve the rendering performance by manually altering and reducing the data set. If the data set is huge, the application must be able to make the best of it during 3D rendering. Unfortunately, the problem of exponential growth in data complexity cannot be solved through incremental improvements to the performance of the current 3D graphics architectures—new approaches are required.

Pixels per Polygon

Although the problem of interactively rendering large 3D models on a typical engineering workstation is challenging, it is not intractable. If the workstation's graphics pipeline is capable of rendering a sustained 200,000 polygons per second (a conservative estimate), then each frame must be limited to 20,000 polygons to maintain 10 frames per second. A typical workstation with a 1280 by 1024 monitor provides 1,310,720 pixels. To cover this screen completely with 20,000 polygons, each polygon must have an average area of 66 pixels. A more realistic estimate is that the rendered image covers some subset of the screen, say 75 percent, and that several polygons, for example four, overlap on each pixel, which implies each polygon must cover an area of approximately 200 pixels.

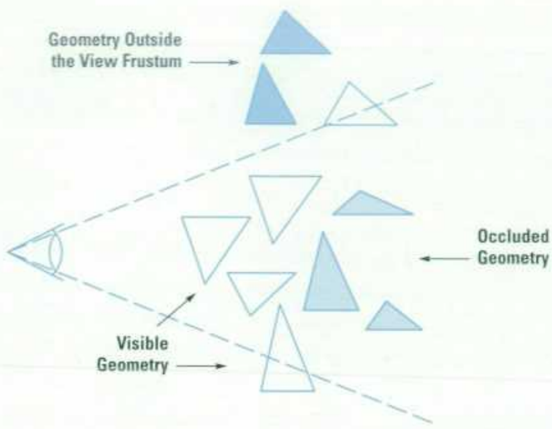
On a typical workstation monitor with a screen resolution of approximately 100 pixels per inch, these polygons are a bit more than 0.1-inch on a side. Polygons of this size will create a high enough quality image for most engineering tasks. This image quality is even more compelling when you consider that it is the resolution produced during interactive navigation. A much higher-quality image can be rendered within a few seconds when the user stops interacting with the model. Thus, today's 3D graphics workstations have enough rendering power to produce the fast, high-quality images required by the technical user.

Software Algorithms

The challenge of interactive large model rendering is sorting through the millions of polygons in the model and choosing (or creating) the best subset of those polygons

Figure 3

Geometry culling.



that can be rendered in the time allowed for the frame. Algorithms that perform this geometry reduction fall into two broad categories: *culling*, which eliminates unnecessary geometry, and *simplification*, which replaces some set of geometry with a simpler version.

Figure 3 illustrates two types of culling: *view frustum* culling (eliminating geometry that is outside of the user's field of view) and *occlusion* culling (eliminating geometry that is hidden behind some other geometry). The article on page 9 describes the implementation of occlusion culling in the VISUALIZE fx graphics accelerator.

Figures 4 and **5** show two types of simplification. **Figure 4** shows a form of geometry simplification called *tessellation*, which takes a mathematical specification of a smooth surface and creates a polygonal representation at the specified level of resolution.

The *decimation* simplification technique is shown in **Figure 5**. This technique reduces the number of polygons in a model by combining adjacent faces and edges.

The simplified geometry created by these algorithms is used by the level of detail selection algorithms, which choose the appropriate representation to render for each frame based on criteria such as the distance to the object.

Most 3D graphics pipelines render a model by rendering each primitive such as a polygon, line, or point individually. If the model contains a million polygons, then the polygon-rendering algorithm is executed a million times. In contrast, these geometry reduction algorithms must operate on the entire 3D model at once, or a significant portion of it, to achieve adequate gains. View frustum culling is a good example—the conventional 3D graphics pipeline will perform this operation on each individual polygon as it is rendered. However, to provide any significant benefit to the large model rendering problem, the culling algorithm must be applied globally to a large chunk of the model so that a significant amount of geometry can be eliminated with a single operation. Similarly, the geometry simplification algorithms can provide greatest gains when they are applied to a large portion of the model.

Desired Solution

The performance gap (often several orders of magnitude) between the needs of the technical user and the capabilities of a typical system puts developers of technical applications into an unfortunate bind. Developers are often experts in some technical domain that is the focus of their applications, perhaps stress analysis or piping layout. However, the 3D data sets that the applications manage are exceeding the graphics performance of the systems

Figure 4

Geometry tessellation.

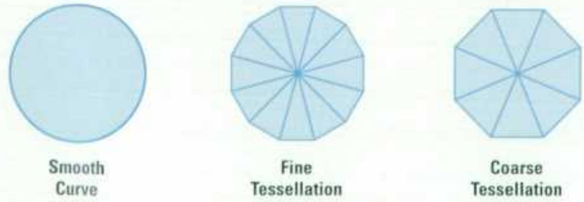


Figure 5

Geometry decimation.

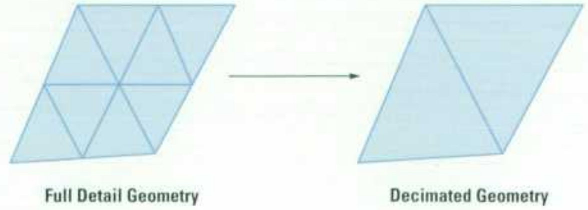
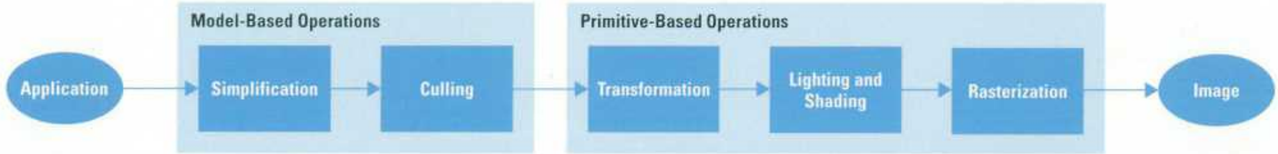


Figure 6

Extended graphics pipeline.



they run on. Developers are faced with the choice of obtaining the 3D graphics expertise to create a sophisticated rendering architecture for their applications, or seeing their applications lag far behind their customers' needs for large 3D modeling capacity and interactivity.

To develop applications with the performance demanded by their customers, developers need access to graphics systems that provide dramatic performance gains for their tasks and data. As shown in **Figure 6**, the graphics pipeline available to the applications must be extended to include model-based optimizations, such as culling and simplification, so that it can support interactive rendering of very large 3D models. When the graphics system provides this level of performance, application developers are free to focus on improving the functionality of their applications without concern about graphics performance. The article on page 9 describes the primitive-based operations of the pipeline shown in **Figure 6**.

DirectModel Capabilities

DirectModel is a toolkit for creating technical 3D graphics applications. The engineer or scientist who must create, visualize, and analyze massive amounts of 3D data does not interact directly with DirectModel. DirectModel provides high-level 3D model management of large 3D geometry models containing millions of polygons. It uses advanced geometry simplification and culling algorithms to support interactive rendering. **Figure 1** shows that DirectModel is implemented on top of traditional 3D graphics APIs such as Starbase or OpenGL. It extends, but does not replace, the current software and hardware 3D rendering pipeline.

Key aspects of the DirectModel toolkit include:

- A Focus on the needs of technical applications that deal with large volumes of 3D geometry data
- Capability for cross-platform support of a wide variety of technical systems
- Extensive support of MDA applications (for example, translators for common MDA data types).

Technical Data

As discussed above, the underlying data is often the most important item to the user of a technical application. For example, when designers select parts on the screen and ask for dimensions, they want to know the precise engineering dimension, not some inexact dimension that results when the data is passed through the graphics system for rendering. DirectModel provides the interfaces that allow the application to specify and query data with this level of technical precision.

Technical data often contains far more than graphical information. In fact, the metadata such as who created the model, what it is related to, and the results of analyzing it is often much larger than the graphical data that is rendered. Consequently DirectModel provides the interfaces that allow an application to create the links between the graphical data and the vast amount of related metadata.

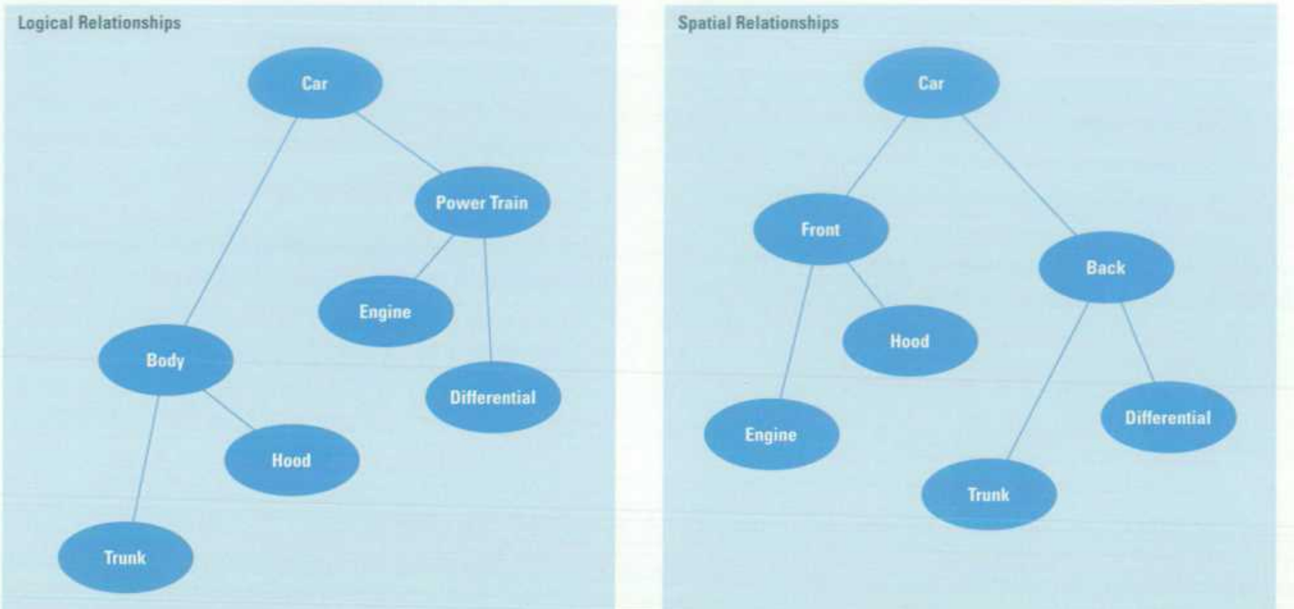
Components of large models are often created, owned, and managed by people or organizations that are loosely connected. For example, one design group might be responsible for the fuselage of an airplane while a separate group is responsible for the design of the engines. DirectModel supports this multiteam collaboration by allowing a 3D model to be assembled from several smaller 3D models that have been independently defined and optimized.

Multiple Representations of the Model

The 3D model is the central concept of DirectModel—the application defines the model and DirectModel is responsible for high-performance optimization and rendering of

Figure 7

Logical and spatial organization.



it. The 3D model is defined hierarchically by the model graph, which consists of a set of nodes linked together into a directed, acyclic graph. However, a common problem that occurs when creating a model graph is the conflict between the needs of the application needs and the graphics system. The application typically needs to organize the model based on the logical relationships between the components, whereas the graphics system needs to organize the model based on the spatial relationships so that it can be efficiently simplified, culled, and rendered. **Figure 7** shows two model graphs for a car, one organized logically and one spatially.

Graphics toolkits that use a single model graph for both the application's interaction with the model and for rendering the model force the application developer to optimize for one use while making the other use difficult. In contrast, DirectModel maintains multiple organizations of the model so that it can simultaneously be optimized for several different uses. The application is free to organize its model graph based on its functional requirements without consideration of DirectModel's rendering needs. DirectModel will create and maintain an additional spatial organization that is optimized for rendering. These multiple organizations do not significantly increase the memory or disk usage of DirectModel because the actual geometry, by far the largest component, is multiply referenced, not duplicated.

The Problem of Motion

Object motion, both predefined and interactive, is critical to many technical applications. In mechanical design, for example, users want to see suspension systems moving, engines rocking, and pistons and valves in motion. To use a virtual prototype for manufacturing planning, motion is mandatory. Assembly sequences can be verified only by observing the motion of each component as it moves into place along its prescribed path. Users also want to grab an object or subassembly and move it through space, while bumping and jostling the object as it interferes with other objects in its path. In short, motion is an essential component for creating the level of realism necessary for full use of digital prototypes.

DirectModel supports this demand for adding motion to 3D models in several ways. Because DirectModel does not force an application to create a model graph that is optimized for fast rendering, it can instead create one that is optimized for managing motion. Parts that are physically connected in real life can be connected in the model graph, allowing movement to cascade easily through all of the affected parts. In addition, the data structures and algorithms used by DirectModel to optimize the model graph for rendering are designed for easy incremental update when some portion of the application's model graph changes.

Models as Databases

3D models containing millions of polygons with a rich set of rendering attributes and metadata can easily require several gigabytes of data. Models of this size are frequently too big to be completely held in main memory, which makes it particularly challenging to support smooth interactivity.

DirectModel solves this problem by treating the model as a database that is held on disk and incrementally brought in and out of main memory as necessary. Elements of the model, including individual level-of-detail representations, must come from disk as they are needed and removed from main memory when they are not needed. In this way memory can be reserved for the geometric representations currently of interest. DirectModel's large model capability has as much to do with rapid and intelligent database interaction as with rendering optimization.

Interactive versus Batch-Mode Data Preparation

Applications that deal with large 3D models have a wide range of capabilities. One application may be simply an interactive viewer of large models that are assembled from existing data. Another application may be a 3D editor (for example, a solid modeler) that supports designing mechanical parts within the context of their full assembly. Consequently, an application may acquire and optimize a large amount of 3D geometry all at once, or the parts of the model may be created little by little.

DirectModel supports both of these scenarios by allowing model creation and optimization to occur either interactively or in batch mode. If an application has a great deal of raw geometry that must be rendered, it will typically choose to provide a batch-mode preprocessor that builds the model graph, invokes the sorting and simplification algorithms, and then saves the results. An interactive application can then load the optimized data and immediately allow the user to navigate through the data. However, if the application is creating or modifying the elements of the model at a slow rate, then it is reasonable to sort and optimize the data in real time. Hybrid scenarios are also possible where an interactive application performs incremental optimization of the model with any spare CPU cycles that are available.

The important thing to note in these scenarios is that DirectModel does not make a strong distinction between batch and interactive operations. All operations can be considered interactive and the application developer is free to employ them in a batch manner when appropriate.

Extensibility

Large 3D models used by technical applications have different characteristics. Some models are highly regular with geometry laid out on a fixed grid (for example, rectangular buildings with rectangular rooms) whereas others are highly irregular (for example, an automobile engine with curved parts located at many different orientations). Some models have a high degree of occlusion where entire parts or assemblies are hidden from many viewing perspectives. Other models have more holes through them allowing glimpses of otherwise hidden parts. Some models are spatially dense with many components packed into a tight space, whereas others are sparse with sizable gaps between the parts.

These vast differences impact the choice of effective optimization and rendering algorithms. For example, highly regular models such as buildings are amenable to preprocessing to determine regions of visibility (for example, rooms A through E are not visible from any point in room Z). However, this type of preprocessing is not very effective when applied to irregular models such as an engine. In addition, large model visualization is a vibrant field of research with innovative new algorithms appearing regularly. The algorithms that seem optimal today may appear very limiting tomorrow.

DirectModel's flexible architecture allows application developers to choose the right combination of techniques, including creating new algorithms to extend the system's capabilities. All of the DirectModel functions, such as its culling algorithms, representation generators, tessellators, and picking operators, are extensible in this way. Extensions fit seamlessly into the algorithms they extend, indistinguishable from the default capabilities inherent to the toolkit.

In addition, DirectModel supports mixed-mode rendering in which an application uses DirectModel for some of its rendering needs and calls the underlying core graphics

API directly for other rendering operations. Although DirectModel can fulfill the complete graphics needs of many applications, it does not require that it be used exclusively.

Multiplatform Support

A variety of systems are commonly used for today's technical 3D graphics applications, ranging from high-end personal computers through various UNIX-based workstations and supercomputers. In addition, several 3D graphics APIs and architectures are either established or emerging as appropriate foundations for technical applications. Most developers of technical applications support a variety of existing systems and must be able to migrate their applications onto new hardware architectures as the market evolves.

DirectModel has been carefully designed and implemented for optimum rendering performance on multiple platforms and operating systems. It presumes no particular graphics API and is designed to select at run time the graphics API best suited to the platform or specified by the application. In addition, its core rendering algorithms dynamically adapt themselves to the performance requirements of the underlying graphics pipeline.

Conclusion

The increasing use of 3D graphics as a powerful tool for solving technical problems has led to an explosion in the complexity of problems being addressed, resulting in 3D models containing millions or even billions of polygons. Unfortunately, many of the applications and 3D graphics systems in use today are built on architectures designed to handle only a few thousands polygons efficiently. These architectures are incapable of providing interactivity with today's large technical data sets.

This problem has created a strong demand for new graphics architectures and products that are designed for interactive rendering of large models on affordable systems. Hewlett-Packard is meeting this demand with DirectModel, a cross-platform toolkit that enables interaction with large, complex, 3D models.

References

1. Data obtained from design engineers at the Hewlett-Packard Vancouver Division.

2. Estimates provided by automotive design engineers.

3. S.H. Shokralla, "The 21st Century Jet: The Boeing 777 Multimedia Case Study,"
http://pawn.berkely.edu/~shad/b777/main.html

4. E. Brechner, "Interactive Walkthrough of Large Geometric Databases," *SIGGRAPH tutorial*, 1995.

5. J.S. Lombardo, E. Mihalak, and S.R. Osborne, "Collaborative Virtual Prototype, *John Hopkins APL Technical Digest*, Vol. 17, no. 3, 1996.

An Overview of the VISUALIZE fx Graphics Accelerator Hardware

Noel D. Scott

Daniel M. Olsen

Ethan W. Gannett

Three graphics accelerator products with different levels of performance are based on varying combinations of five custom integrated circuits. In addition, these products are the first ones from Hewlett-Packard to provide native acceleration for the OpenGL® API.

The VISUALIZE fx family of graphics subsystems consists of three products, fx^6, fx^4, and fx^2, and an optional hardware texture mapping module. These products are built around a common architecture using the same custom integrated circuits. The primary difference between these controllers is the number of custom chips used in each product (see **Table I**).

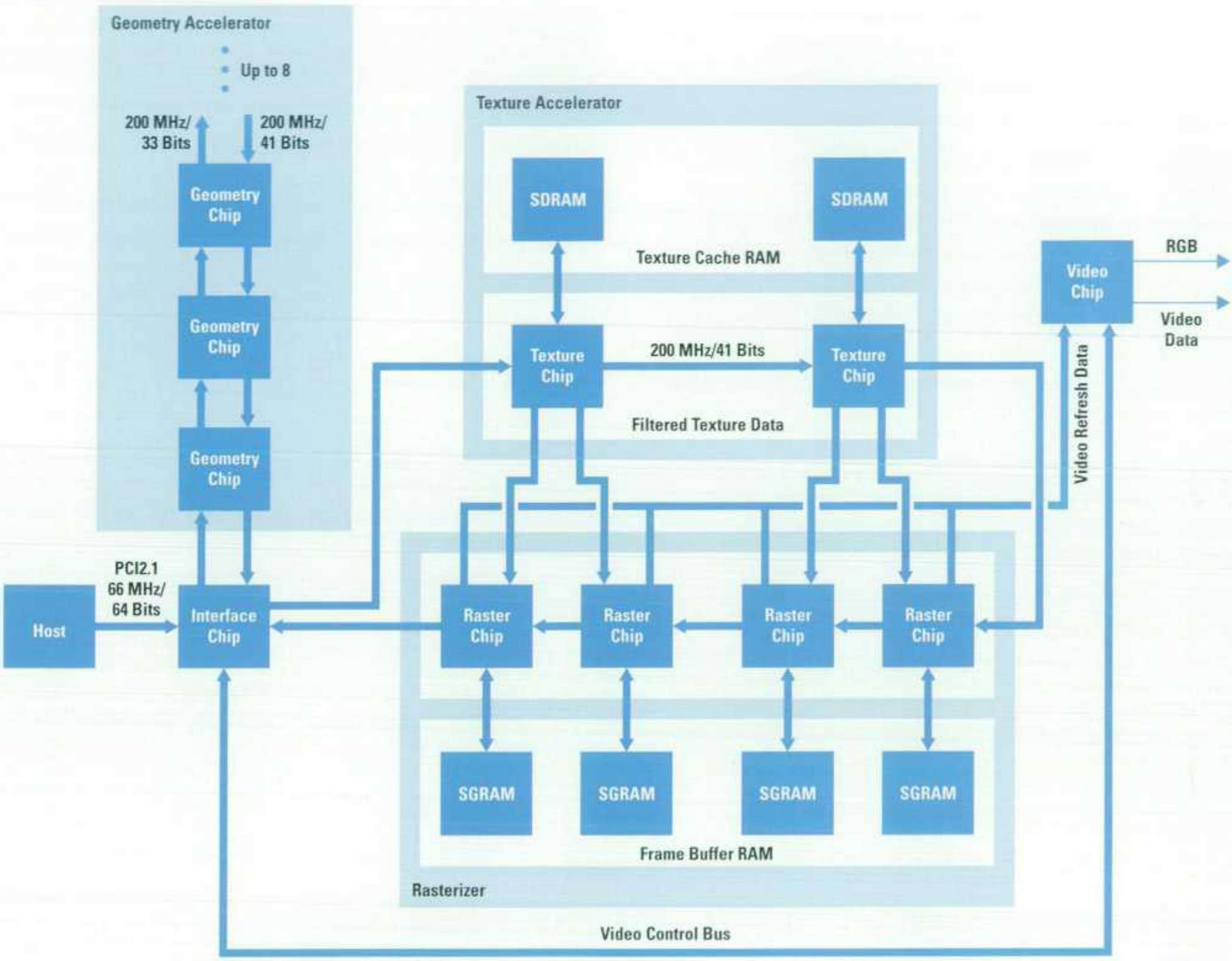
Table I
Number of custom chips in the different VISUALIZE fx products

Product	Texture Chip	Geometry Chip	Raster Chip
fx^2	—	1	2
fx^4	1	2	2
fx^6	2	3	4

A chip-level block diagram of the VISUALIZE fx^6 product is shown in **Figure 1**. This is the most complex configuration and also the one with the highest performance in the product line. The VISUALIZE fx^4 and the VISUALIZE fx^2 products use subsets of the chips used in the fx^6. The fx^6 and fx^4 subsystems have support for the optional hardware-accelerated texture map module, which contains a local texture cache for storage of texture map images. If the texture accelerator is not present, the bus between the interface chip and the first raster chip is directly connected.

Figure 1

A chip-level diagram of the VISUALIZE fx[6] product.



Geometry Chip
- 3D Geometry and Lighting Acceleration

Texture Chip
- Texture Rasterization
- Texture Map Cache Controller
- Texture Memory Control
- Texture Interpolation

Interface Chip
- I/O Buffering
- 3D Geometry Workload Distribution and Concentration
- 2D and 3D Data Path Arbitration
- 2D Acceleration
- YUV to RGB Conversion Support
- Pixel Level Pan and Zoom
- Pixel Level Image Rotations

Raster Chip
- Fragment Processing
- Frame Buffer Control Functions

Video Chip
- Color Lookup Tables
- Video Timing
- Digital-to-Analog Conversion
- Video-Out Data

Interface Chip

The interface chip provides a PCI 2.1 (also referred to as PCI 2X) compliant interface.* It operates at up to 66 MHz in 64-bit mode. Special efforts have been made in the design of the buffering and the interface to the PCI. As a result, the driver is able to sustain writes of 3D geometry commands to the PCI at almost the theoretical maximum rates that could be computed for the PCI. The article on page 51 discusses PCI capability.

* PCI = Peripheral Component Interconnect.

Occlusion Culling

The HP fast-break program (page 8) enabled us to understand customer requirements by analyzing what is important in OpenGL graphics today. As a result, we developed a technology called *occlusion culling* as an extension to OpenGL and implemented it in the VISUALIZE fx graphics hardware.

We found that the data sets many graphics workstation customers are trying to visualize are very complex. These data sets have large numbers of small, complex components that are not always visible in the final images. For instance, when rendering an airplane, all of the MCAD parts are present in the data set represented by potentially millions of polygons that must be processed. However, when this airplane is viewed from the outside only the outer surfaces are visible, not the fan blades of the engine or the seats or bulkheads in the interior.

In a traditional 3D z-buffered graphics system, all polygons in a scene must be processed by the graphics pipeline because it is not known a priori which polygons will be visible and which ones will be occluded (not visible). The notion of occlusion culling, or removal of occluded objects, has been talked about in the research community for several years. However, implementations tend to be in software where the performance is not at a satisfactory level.

In the VISUALIZE fx series of graphics devices, HP developed a very efficient algorithm that tests objects for visibility. An application program can very quickly use the occlusion culling visibility test to determine if a simple bounding box representation of a more complex part is visible. Since a bounding box, or more generally a bounding volume, completely encloses the more complex part, it is possible to know a priori that if the bounding volume is not visible then the complex part it encloses is not visible. Thus, the part that is not visible does not need to be processed through the graphics pipeline. The real benefit of occlusion culling comes when a very complex part consisting of many vertices can be rejected, avoiding the expenditure of valuable time to process it.

For very complex data sets, such as the airplane mentioned above or an automobile, a tremendous performance increase can be realized by using the HP occlusion culling technology. To date, several ISVs have begun using occlusion culling in their applications and are seeing a 25 to 100 percent increase in graphics performance. This magnitude of performance benefit typically costs a customer several thousand dollars for the extra computational horsepower. HP includes this technology as standard in all VISUALIZE fx series graphics accelerators, giving even better price and performance results to our customers.

The future of 3D graphics will continue toward visualizing ever more complex objects and environments. Occlusion culling together with HP's DirectModel technology (page 19) are well positioned to be industry leaders in providing the technology for 3D modeling applications.

The primary responsibility of the interface chip is to separate the streams of data that arrive from the host SPU into three paths and arbitrate access among those paths.

3D Path. Typically data from the host CPU looks very much like the OpenGL API functions themselves. Data following this first path is routed to the geometry chips. The geometry chips process the data and return the results to the interface chip. These results are then sent on to the texture chips or directly to the raster chips if the texture mapping subsystem is not installed. In either case the data is transmitted to and through all the texture and raster chips in the system.

Unbuffered Path. This path passes data directly through the interface chip to the texture and raster chips. This provides a bypass method that allows traffic to get around other pending operations. An example would be a texture cache download that is required to complete a primitive that is currently being rasterized, a situation that would lead to deadlock without the unbuffered path.

2D Path. This path runs directly through the interface chip to the texture and raster chips. The 2D path differs from the unbuffered path in the way its priority is handled. The interface chip manages priority among the three paths as they all converge on the same set of wires between the interface chip and the first texture chip. The unbuffered path goes directly through the interface chip to those wires and has priority over the other two paths. Data targeting the 2D path is held off until all preceding 3D work in the geometry chip has been flushed through to the first texture chip.

There is also special circuitry in the interface chip that is used to accelerate many operations commonly done by X11 or other 2D APIs.

Buses

The three primary buses in the system are each run at 200 MHz, allowing sustainable transfer rates of more than 800 Mbytes per second. To control the loading on the interconnections for these buses, they are built as point-to-point connections from one chip to the next.

Each chip receives the signals and then retransmits them to the next chip in the sequence. This requires more pins on each part, but limits the number of loads on each wire to a single receiver as well as limiting the wiring length that signals must traverse. This allows for reliable communications despite the high frequency of the buses.

The first of these three buses distributes work to the geometry chips. This bus starts at the interface chip and runs through all the geometry chips in the system. Each geometry chip monitors the data stream as it flows through the bus and picks off work to operate upon based on an algorithm that selects the least busy geometry chip.

The second of these buses starts at the last geometry chip and passes through the others back to the interface chip. The results of the work done by the geometry chips is placed on this bus in the same sequence as it was moved along the first bus. This strict ordering control prevents certain artifacts from showing up in the final image.

The third bus ties the interface chip to the texture and frame buffer subsystems. It is wired in a loop that goes back to the interface chip from the last chip in the chain. 3D operations typically flow from the interface chip to the chips along this bus, and when they eventually get back to the end of the loop, they are thrown away.

For 2D operations, such as moving blocks of pixels around the frame buffer, the operation of the third bus is somewhat different. The movement of pixel data operates as a sequence of reads followed by a sequence of writes. The reads cause data to be dumped from the frame buffer locations onto the bus and the results travel back to the interface chip. This data is then associated with new addresses and sent as writes back down the bus, ending up back at the frame buffer but in different locations.

Besides the three primary buses mentioned above, there are three secondary buses in the system. The first bus connects the interface chip to the video chip. This provides video control, download of color maps, and cursor control. The second bus is a connection from each raster chip to the video chip. This path is used to provide video refresh data to display frame buffer contents. The final secondary bus is a connection from each texture chip to two of the raster chips. This path allows the flow of filtered texture data into the raster chips for combination with nontexture fragment data.

Geometry Chip

The geometry and lighting chips are responsible for taking in geometric primitives (points, lines, triangles, and quadrilaterals) and executing all the operations associated with the transform stage of the graphics pipeline (see the article on page 9 for more about the graphics pipeline). These operations include:

- Transformation of the coordinates from model space to eye space
- Computing a vertex color based on the lighting state, which consists of up to eight directional or positional light sources
- Texture map calculations that include:
  - Environment map calculations for texture mapping
  - Texture coordinate transformation
  - Linear texture coordinate generation
  - Texture projection
- View volume clipping and clipping against six arbitrary application-specified planes to determine whether a primitive is completely visible, rejected because it is completely outside the view area, or needs to be reduced into its visible components
- Perspective projection transformation to cause primitives to look smaller the further away from the eye they are
- Setup calculations for rasterization in the raster chip.

There were some interesting problems to solve in the design of the distribution and coalescing of work up and down the geometry chip daisy chain. For example, load balancing, maintaining strict order in the output stream, and ensuring that operations, such as binding of colors and normals to vertices, perform as required by OpenGL.

Fast Virtual Texturing

Texture mapping, which is wrapping a picture over a three dimensional object, has been used over the years as a key feature to enhance photorealism, reduce data set sizes, perform visual analysis, and aid in simulations (see **Figure 1**). Since texturing calculations are computationally expensive and memory access for large textures can be prohibitively slow, various workstation graphics vendors have provided hardware-accelerated texture mapping as a key differentiator for their product.

A primary drawback of these attempts at hardware acceleration is that dedicated local hardware texture memory is limited in size and is expensive. To take advantage of the performance boost, graphics applications were constrained to textures that fit in the local hardware texture memory. In other words, the application was responsible for managing this hardware resource.

Figure 1

A 3D textured skull. The VISUALIZE fx^4 and fx^6 subsystems support a texture map acceleration option. Pictured here is the use of 3D texture mapping OpenGL extensions with this option. This feature allows visualization of 3D data sets such as MRI images.

Noticing this obvious artificial application limitation in texturing functionality, performance, and portability, Hewlett-Packard introduced, in the VISUALIZE-48, a new concept in hardware texture mapping called *virtual texture mapping*. Virtual texture mapping uses the dedicated local hardware texture memory as a true texture cache, swapping in and out of the cache the portions of textures that are needed for rendering a 3D image. Thus, for texturing applications, these limitations were eliminated. The application could define and use a texture map of any size (up to a theoretical limit of 32K texels × 32K texels*) that would be hardware accelerated, eliminating the need for the application to be responsible for managing local texture memory.

Using the local hardware texture memory as a cache also means that this memory uses only the portions of the texture maps needed to render the image. This efficiency translates to more and larger texture maps being hardware accelerated at the same time. Applications that previously could not run because of texture size limits can now run because of the unlimited virtual texture size. Also, with only the used portions of the texture map being downloaded to the cache, far less graphics bus traffic occurs.

The system design of virtual texture mapping involved changes in the HP-UX operating system to support graphics interrupts, onboard firmware support for these interrupts, the introduction of an asynchronous texture interrupt managing daemon process, and the associated texturing hardware described in this

*A texel is one element of a texture.

The output of the geometry chip's daisy chain is passed back through the interface chip. Generally, for triangle based primitives, the output takes the form of plane equations. As these floating-point plane equations are returned from the geometry chip to the interface chip and passed on to the texture chips, certain addressed locations in the interface chip will result in the floating-point values being converted to fixed-point values as they pass through. These fixed-point values are in a form the raster chips need to rasterize the primitive.

The daisy-chain design allows up to eight of the geometry chips to be used although only three are applied in the case of the VISUALIZE fx^6 product at this time.

article. Having a centralized daemon process manage the cache allows for cache efficiency, parallel handling of texture downloads while 3D graphics rendering is occurring, and sharing textures among graphics contexts.

The VISUALIZE fx^4 and VISUALIZE fx^6 texture mapping options incorporate the second generation advances in virtual texture mapping. Full OpenGL 1.1 texture map hardware support has brought about dramatic improvements in texture map download performance and switching between texture maps and new extended features such as 3D texture mapping, shadows (**Figure 2**), and proper specular lighting on textures (**Figure 3**). These features have made these products very appealing systems for texturing applications on workstation graphics.

The texture mapping performance on these systems is very competitive. The VISUALIZE fx^6 texture fill rate is about twice that of the VISUALIZE fx^4 texture option. However, fill rates alone do not describe how these systems perform in a true application environment. Aggressive texture mapping application performance comparisons show two to three times performance superiority over similarly priced graphics workstation products.

Figure 2

A shadow texture image.

Figure 3

A specular lit texture image. Correct specular lighting of textured images can be achieved with VISUALIZE fx^4 and fx^6 texture mapping options.

Texture Chip

The texture chip is responsible for accelerating texture mapping operations. Towards this end, it performs three basic functions:

- Maintains a cache of texture map data, requesting cache updates for texture values required by current rendering operations as needed (see "Fast Virtual Texturing" on page 32)
- Generates perspective corrected texture coordinates from plane equations representing triangles, points, or lines
- Fetches and filters the texture data as specified by the application based on whether the texture needs to be magnified or minimized to fit the geometry it is being mapped to and passes the result on to the raster chips.

Raster Chip

The raster chip rasterizes the geometry into the frame buffer. This means it determines which pixels are to be potentially modified and, if so, whether they should be modified based on various current state values (including the contents of the z buffer). The raster chip also controls access to the various buffers that make up the frame buffer. This includes the image buffer for storing the image displayed on the screen (potentially two buffers if double buffering is in effect), an overlay buffer that contains images that overlay the image buffer, the depth or z buffer for hidden surface removal, the stencil buffer,* and an alpha buffer** on the VISUALIZE fx[6]. To accomplish its work the raster chip performs four basic functions:

- Rasterize primitives described as points, lines, or triangles
- Apply fragment operations as defined by OpenGL (such as blending and raster operations)
- Control of and access to buffer memory, including all the buffers described earlier
- Refresh the data stream for the video chip, including handling windows and overlays.

Video Chip

The video chip provides video functions for controlling the data flow from the frame buffer to the display and mapping data from values to color. The features of the video chip include:

- Data mapping to colors:
  - Two independent 4096-by-24-bit lookup tables
  - Four independent 256-by-3-by-8-bit lookup tables for image planes
  - A bypass path for 24-bit true color data
  - Two independent 256-by-8-bit lookup tables for overlay planes
- Digital-to-analog conversion
- Video timing
- Video output.

* A stencil buffer is per pixel data that can be updated when pixel data is written and used to restrict the modification of the pixel.

** An alpha buffer contains per pixel data that describes coverage information about the pixel and can be used when blending new pixel values with the current pixel value.

Conclusion

The VISUALIZE fx family of products currently has a substantial lead in not only price/performance measurements, but it also leads in performance independent of cost.

For information regarding how these systems compare against the competition, visit the SPEC (an industry standard body of benchmarks) web page at:

http://www.spec.org/gpc

Acknowledgments

We would like to thank Paul Martz for the shadow texture image (**Figure 2** on page 33).

Noel D. Scott

Noel Scott is a senior engineer at the HP Workstation Systems Division. He is responsible for product definition, performance projections, and modeling. He designed the I/O bus for the geometry chip described in the article. He came to HP in 1981 after receiving a BS degree in computer engineering from the University of Kansas.

Daniel M. Olsen

A software engineer in the graphics products laboratory at the HP Workstation Systems Division, Daniel Olsen is responsible for the development of new 3D products for HP workstations. He has been with HP since 1994. He has a BSEE degree (1991) from North Dakota State University and an MS degree in computer engineering (1997) from Iowa State University. Daniel was born in Des Moines, Iowa, is married and has two daughters. His leisure time activities include skiing, home projects, scuba diving, and aviation.

Ethan W. Gannett

Ethan Gannett is a lead engineer for graphics software development at the HP Workstation Systems Division. He came to HP in 1988 after receiving an MS degree in computer science from Iowa State University. He also holds a BS degree in physics (1983) and a BS degree in astronomy (1983) from the University of Iowa. Born in Davenport, Iowa, he is married and has one daughter. He enjoys kayaking, backcountry camping, telemarking, and hiking.

HP Kayak: A PC Workstation with Advanced Graphics Performance

Ross A. Cunniff

World-leading 3D graphics performance, normally only found in a UNIX® workstation, is provided in a PC workstation platform running the Windows NT® operating system. This system was put together with a time to market of less than one year from project initiation to shipment.

Computer graphics workstations are powerful desktop computers used by a variety of technical professionals to perform their day-to-day work. Traditionally, such computers have run with a version of the UNIX operating system. In the past year, however, workstations featuring Intel processors such as the Pentium™ Pro and Pentium II and running the Microsoft® Windows NT operating system have begun to gain ground in both capability and market share. Hewlett-Packard has historically been a leader in the UNIX workstation business. In February, 1997, Hewlett-Packard began a project to put its high-performance workstation graphics into a PC workstation platform.

Technical Challenges

Fitting HP workstation graphics into a Windows NT platform was not an easy task. The task was made more exciting with the addition of schedule pressure. The schedule gave us only four months to reach functional completion and only two months after that to finish the quality assurance process. This schedule was made even more challenging because the hardware was not yet complete. It was difficult at times to distinguish software defects from hardware defects. This article describes how we overcame some of the challenges we encountered while implementing this project.

The Hardware

The hardware for the HP Kayak workstation (**Figure 1**) is based on the VISUALIZE fx[4] graphics subsystem for real-time 3D modeling (see the article on page 28). However, a couple of changes were necessary. First, to achieve

Ross A. Cunniff

A senior software engineer at the HP Performance Desktop Computing Operation, Ross Cunniff has been with HP since 1985. He was the lead software engineer for the 3D device driver used in the HP Kayak workstation. He continues to be the lead 3D device driver engineer for high-end graphics products. He received a BS degree in mathematics and a BS degree in computer science in 1985 from the University of New Mexico. His professional interests include computer graphics, particularly 3D hardware acceleration.

Figure 1

An HP Kayak XW workstation.

the performance available in the graphics hardware, the bus interface had to be changed from the standard Peripheral Component Interconnect (PCI) to the accelerated graphics port (AGP),† since no commodity PC chipset supported PCI 2X. With normal industry-standard PCI, we would have been limited to 132 Mbytes/s for I/O, which would have hurt our performance on several important benchmarks. With the accelerated graphics port, the available I/O bandwidth increased to 262 Mbytes/s.

The second change necessary to the hardware was the addition of industry-standard VGA graphics. During the boot process of Windows NT, and at occasional intervals after that, the computer will access VGA graphics registers directly. To achieve this, a VGA daughtercard was created that displays its graphics through the video feature connector created for the UNIX video solution. The main graphics board was modified slightly, making it possible to dynamically switch between VGA graphics and VISUALIZE fx[4] graphics. **Figure 2** shows a hardware block diagram for an HP Kayak workstation.

† AGP is a bus that transfers data to and from a graphics accelerator.

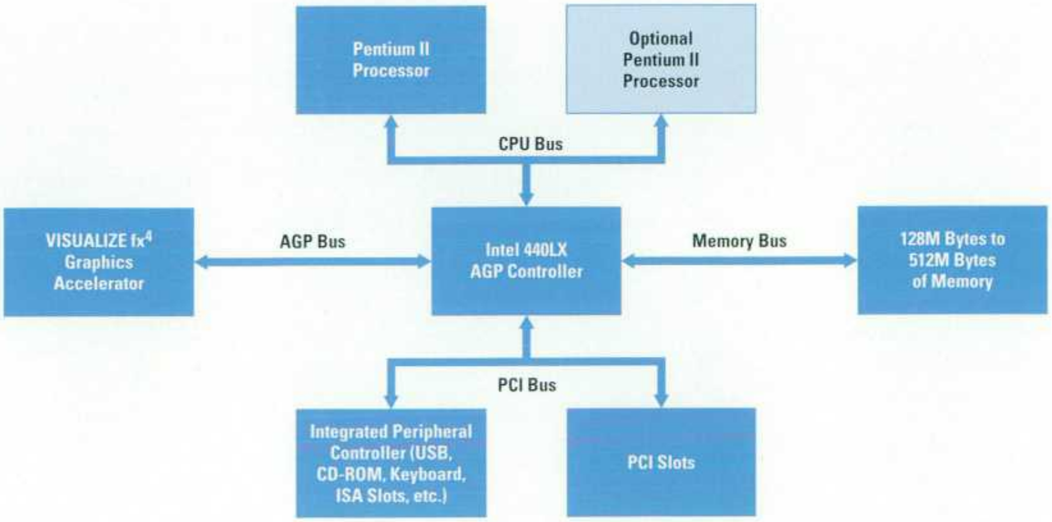
Windows NT Driver Architecture

The fact that the hardware for the HP Kayak workstation is similar to the VISUALIZE fx[4] hardware, which runs the UNIX operating system, made the software effort much easier. However, many significant hurdles had to be overcome to get the software running under Windows NT.

The first challenge was the Windows NT device driver architecture (**Figure 3**). On HP-UX*, graphics device drivers have a large amount of kernel support, allowing them to access the graphics hardware directly from user-level code without having to execute any special locking routines. This direct hardware access (DHA) method is not present on Windows NT. Instead, all accesses to the hardware must be performed from the kernel (ring 0 in **Figure 3**).

Figure 2

A hardware block diagram for an HP Kayak workstation.

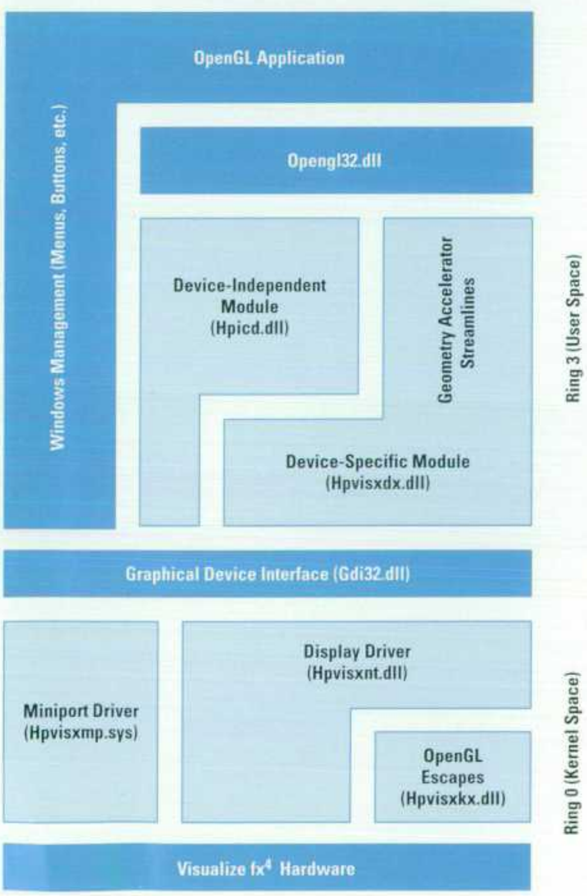


Fortunately, the VISUALIZE fx[4] architecture specifies a buffered form of communication in which graphical commands are placed into command data packets in a large buffer in the hardware. It was a simple task to modify the HP-UX drivers to access a software allocated command data packet buffer instead. When one of these software buffers gets full, it is passed to the ring 0 driver that forwards the buffer to the hardware.

The lighter-shaded modules in **Figure 3** represent the libraries that were delivered by HP to support the VISUALIZE fx[4] hardware. The libraries in ring 3 (Hpicd.dll and Hpvisxdx.dll) were fairly straightforward ports of the corresponding UNIX libraries libGL.sl and libddvisxgl.sl. The libraries in ring 0 (Hpvisxmp.sys, Hpvisxnt.dll, and Hpvisxkx.dll) had to be created from scratch to support the Windows NT driver model. These modules make up about 30 percent of the size of the ring 3 modules.

Figure 3

The Windows NT device driver architecture.



Integration with 2D Windows NT Graphics

The second challenge was to integrate the 3D OpenGL graphics support with the standard Windows NT graphical device interface. Microsoft specifies two methods that can be used to do this. The first, called a *miniclient driver*, is a rasterization-level OpenGL driver that uses the Microsoft OpenGL software pipeline for lighting and transformation. This driver would have been easy to create, but it would not have allowed us to take advantage of the hardware transformation and lighting provided by VISUALIZE fx[4].

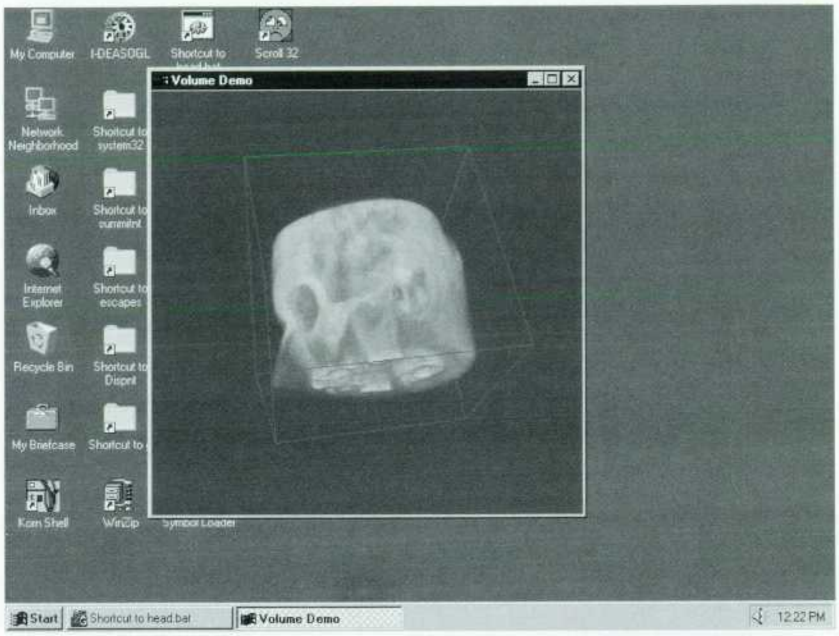
The second method, called an *installable client driver*, is a geometry-level OpenGL driver that leaves implementation of the lighting and transformation pipeline up to the driver writer. The driver allows us full access to all OpenGL API routines. This is the route we chose because we already had a full implementation of OpenGL, which we had created to run on the HP-UX operating system. This implementation was ported to the installable client driver model over a span of several weeks, while we added support for Windows NT multithreading. The bulk of the VISUALIZE fx[4] graphical device interface driver was written by a separate team of experts without much consideration for 3D graphics acceleration. This enabled them to get the Windows NT display driver running in a short amount of time and allowed them to continue enhancing 2D performance without severely impacting the 3D device driver team. Some of the results of these efforts are shown in **Figure 4**.

Integrating the Windows NT Driver with Ring 0

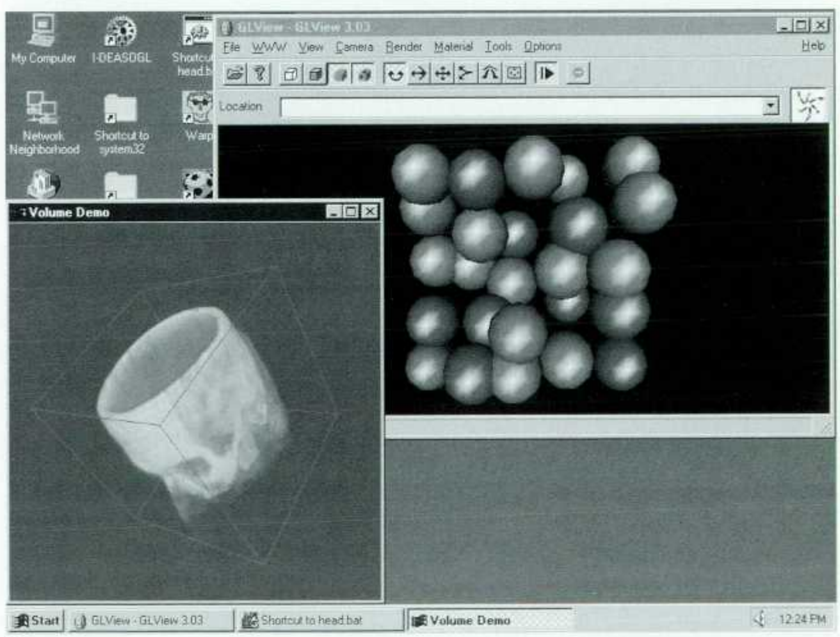
A third challenge was to integrate the Windows NT driver with the ring 0 portion of the OpenGL driver while maintaining separate code bases for the different teams. We decided to make our ring 0 driver a separately loadable library. This decision kept the source code separate. It enabled much faster edit-compile-debug cycles, since it allowed us to replace a portion of the ring 0 driver without having to reboot the computer. However, the separation added extra complexity because we had two very different drivers accessing the same piece of hardware. To solve this problem, we created a variable called a *hardware access token*. Each driver has a special token

Figure 4

(a) A 3D image in a 2D environment. (b) Several 3D programs in a 2D environment.



(a)



(b)

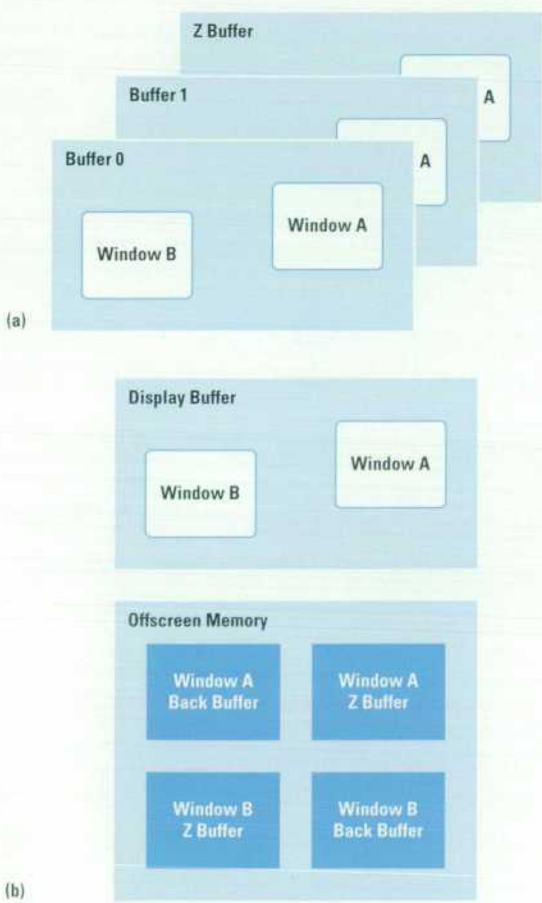
that it places in the hardware access token to indicate that it was the last driver to access the hardware. When a driver detects that the token is not its own, it executes procedures known as *context save* and *context restore*. The context save reads all applicable hardware state information from the device into software buffers. The context restore places the previously saved state back into the hardware. This same mechanism is used to mediate hardware accesses between different processes running OpenGL.

Integration of VISUALIZE fx^4 Architecture

A fourth challenge for the team was the integration of the VISUALIZE fx^4 stacked planes architecture (**Figure 5a**) into the Windows NT environment. Workstations traditionally have very deep pixels, each pixel having up to 90 bits of information. This information includes support for such things as transparent overlays, double buffering, hidden surface removal, and clipping. Windows NT expects a slightly different model, in which the extra per pixel information is allocated in offscreen storage when a 3D rendering context is created (**Figure 5b**). What this means is that when the window state is changed (for example, when a window is moved on the desktop), Windows NT does not make any special calls to the device driver. This presented a problem, since our stacked planes architecture needs to keep all of the extra information directly associated with the correct visible screen pixels.

Figure 5

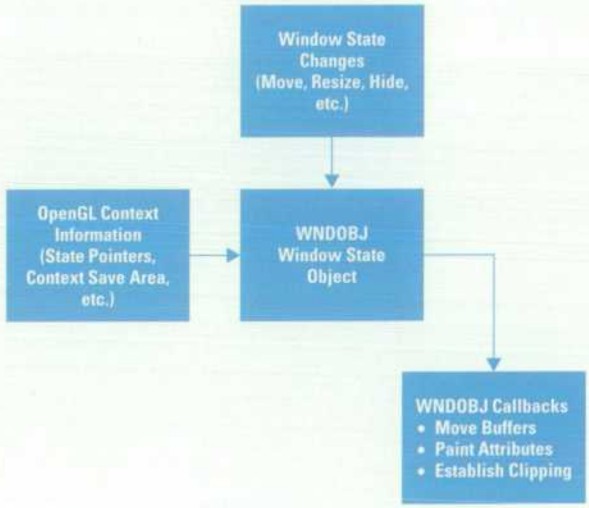
(a) VISUALIZE fx^4 stacked frame buffer model. (b) Windows NT offscreen frame buffer model.



To fix this problem, we used a Windows mechanism called a *window object* (**Figure 6**). The window object tracks a window state and executes callbacks into our driver when a window state is modified. This added an unfortunate amount of complexity into our driver, since the window state is asynchronous to all other hardware accesses and not all of the window state information we need was directly available to us. In addition, applications expect to be able to mix Windows NT graphical device interface rendering and 3D OpenGL rendering in the same window. These two problems required us to add a double

Figure 6

The components of a window object.



buffering mechanism that actually copies the physical back buffer bits into the displayed front buffer. This is significantly slower than the native per pixel double buffering of VISUALIZE fx^4. However, it fits better into the Windows NT model and enables all applications to run. We still enable the native method for applications and benchmarks that work correctly with it, since it is significantly faster.

Performance

A fifth challenge for the team was performance. In the graphics workstation market, performance is usually the main differentiator. The most popular single measure of performance in the PC graphics market is the OPC Viewperf benchmark known as CDRS-03.[1] By July, 1997, we had achieved a CDRS-03 rating of 74—a performance level that exceeded all known competitors. This met our goals set at the beginning of the project. However, we were aware that the hardware was capable of supporting much higher performance. With a goal in mind of a SIGGRAPH 97 announcement in August, we redesigned the device driver. The redesign optimized certain paths through the driver, enabling much higher performance for this benchmark and for important applications such as Unigraphics and Structural Dynamics Research Corporation (SDRC). As a result, we were able to announce a CDRS-03 rating of over 100 at SIGGRAPH 97.

In addition to benchmark performance, the team focused on application performance because it is typically this measure that determines whether a customer will buy the product. We obtained a variety of in-house applications and built up expertise in running the applications. We also obtained data sets that represented typical customer workloads and adjusted various performance parameters (such as display list size) to maximize performance for the benchmark. Using this technique, the performance with some data sets was up to 100 times faster.

Conclusion

With VISUALIZE fx^4, Hewlett-Packard has the fastest Windows NT graphics on the market.[1,2,3] Integrated into the HP Kayak XW platform, the graphics device and its successors will help Hewlett-Packard maintain its market leadership.

References

1. *CDRS (CDRS-03) Results*, OpenGL Performance Characterization Project:
http://www.specbench.org/gpc/opc/opc.cdrs.html

2. *Advanced Visualizer (AWadvs-01) Results*, OpenGL Performance Characterization Project:
http://www.specbench.org/gpc/opc/opc.AWadvs.html

3. *Data Explorer (DX-03) Results*, OpenGL Performance Characterization Project:
http://www.specbench.org/gpc/opc/opc.dx.html

Concurrent Engineering in OpenGL® Product Development

Robert J. Casey

L. Leonard Lindstone

Time to market was reduced when tasks that had been traditionally serialized were completed in parallel.

Concurrent engineering is the convergence, in time and purpose, of interdependent engineering tasks. The benefits of concurrent engineering versus traditional serial dependency are shown in **Figure 1**. Careful planning and management of the concurrent engineering process result in:

- Faster time to market
- Lower engineering expenses
- Improved schedule predictability.

This article discusses the use of concurrent engineering for OpenGL product development at the HP Workstation Systems Division.

Robert J. Casey
A senior engineer in the graphics products laboratory at the HP Workstation Systems Division, Robert Casey was the chief software architect for the OpenGL product. Currently, he leads the efforts on Direct 3D® technology in the graphics products laboratory. He came to HP in 1987 after receiving a BS degree in computer engineering from Ohio State University. He was born in Columbus, Ohio, is married and has two children. His outside interests include skiing, soccer, and woodworking.

L. Leonard Lindstone
Leonard Lindstone is a project manager at the HP Workstation Systems Division. He is responsible for software drivers for new graphics hardware. He joined HP in 1976 at the Calculator Products Division after earning a BSEE degree from the University of Colorado. He also has an MS degree in computer science from Colorado State University. Leonard is married and has three children. He enjoys music of all kinds and historical fiction.

OpenGL Concurrent Engineering

We applied concurrent engineering concepts in the development of our OpenGL product in a number of ways, including:

- Closely coupled system design with partner laboratories
- Software architecture and design verification
- Real-use hardware verification
- Hardware simulation
- Milestones and communication
- Joint hardware and software design reviews
- Test programs written in parallel.

Cultural Enablers

In addition to these technical tactics, the OpenGL team enjoyed the benefits of several cultural enablers that have been nurtured over many years to encourage concurrent engineering. These include early concurrent staffing, an environment that invites, expects, and supports bottoms-up ideas to improve time to market, and the use of a focused program team to use expertise and gain acceptance from all functional areas and partners.

System Design with Partner Labs

We worked closely with the compiler and operating system laboratories to design new features to greatly improve our performance (see the "System Design Results" section in the article on page 9). Our early system design revealed that OpenGL inherently requires approximately ten times more procedure calls and graphics device accesses than our previous graphics libraries. This large increase in system use meant we had to minimize these costs we previously had been able to amortize over a complete primitive.

We worked closely with our partner laboratories to ensure success. Our management secured partner acceptance, funding, and staffing, and the engineers worked on the joint system design. Changes of this magnitude in the kernel and the compiler take time, and we could not afford to wait until we had graphics hardware and software running for problems to occur. Rather, we used careful system performance models and competitive performance projections to create processor state count budgets for procedure calls and device access. These performance goals guided our design. In fact, our first design to improve procedure call overhead missed by a few states per call, so we had to get more creative with our design to arrive at an industry-leading solution. We managed these dependencies throughout the project with frequent communication and interim milestones.

Software Architecture and Design Verification

We designed and followed a risk-driven life cycle. To support the concurrent engineering model, we needed a life cycle that avoided the big bang approach of integrating all

Figure 1

The benefits of concurrent engineering.

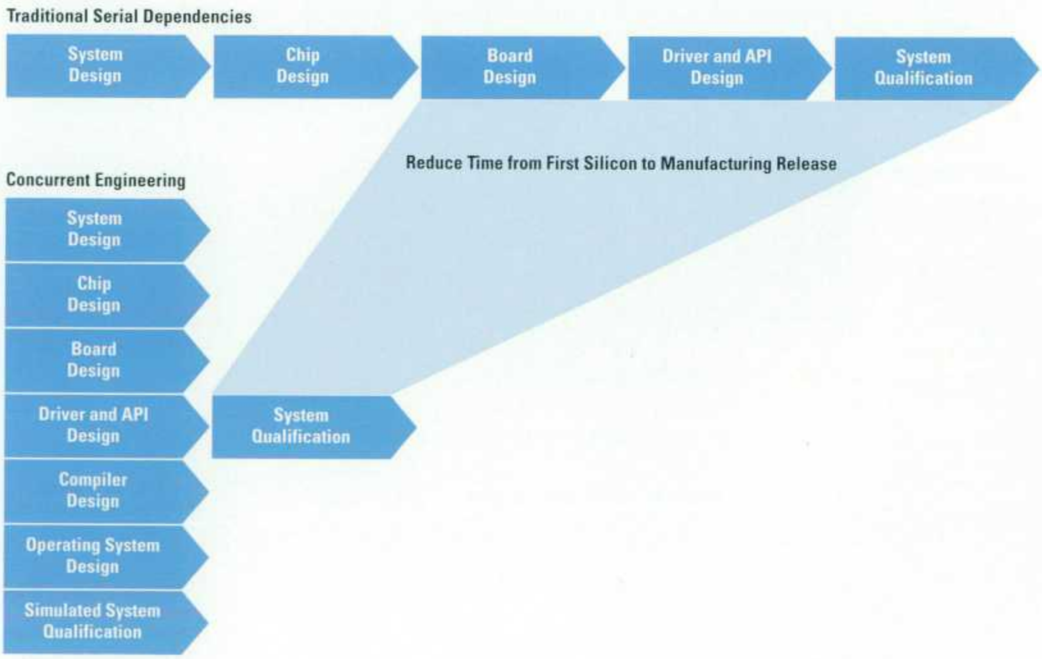
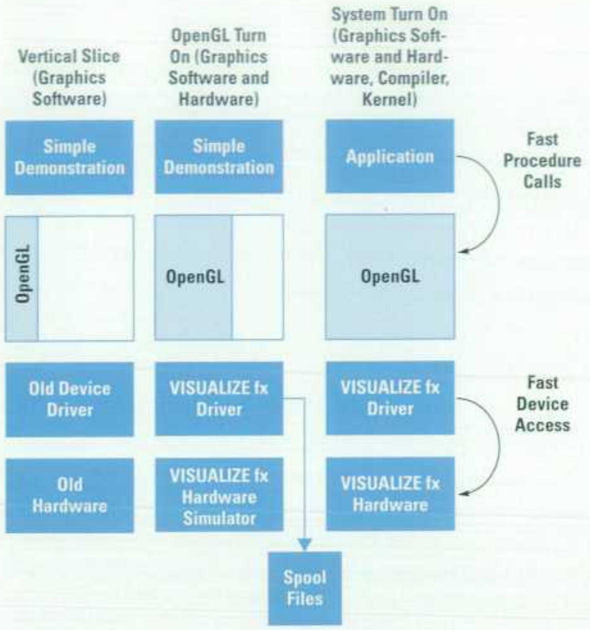


Figure 2

OpenGL concurrent engineering techniques.



the pieces at the end. This would result in a longer and less predictable time to market. Instead, we created a prototyping environment. This environment was initially created to test the software architecture and early design decisions. The life cycle included a number of checkpoints focused on interface specification, design, and prototyping.

One key prototyping checkpoint in this environment is what we called our "vertical slice," which represented a thin, tall slice through the early OpenGL architecture (see **Figure 2**). Thin because it supports a small subset of the full OpenGL functionality, and tall because it exercises all portions of the software architecture, from the API to the device driver-level interface. With this milestone, we had a simple OpenGL demonstration running on our software prototype.

The objectives of this vertical slice were to verify the OpenGL software architecture and design, create a prototyping design environment, and rally the team around this key deliverable.

Hardware Verification

Before we had completed verification of the software architecture, it became evident that this same environment needed to be quickly adapted and evolved to handle the demands of hardware verification. OpenGL features and performance represented the biggest challenge for the new VISUALIZE fx hardware. Although this hardware would also support our legacy APIs (Starbase, PHIGS, PEX), most of the newness and therefore risk was contained in our support of OpenGL. By evolving our prototyping environment for use as the hardware verification vehicle, we were able to exercise the hardware model in real-use scenarios (albeit considerably slower than full performance).

Evolving this environment for hardware verification required us to take the prototyping further than we would have for software verification alone. We had to add more functionality to more fully test the OpenGL features in hardware. We also had to do so quickly to avoid delaying the hardware tape release.

This led to our second key prototyping checkpoint, which we called "OpenGL turn on." This milestone included the same OpenGL demonstration running on the VISUALIZE fx hardware simulator. We also added functionality breadth to the vertical slice (see **Figure 2**). Doing all this for a new OpenGL API represented a new level of concurrent engineering, in that we were running OpenGL programs on a prototype OpenGL library and driver and displaying pictures on simulated VISUALIZE fx hardware, all more than a year before shipments.

The key objective of this milestone was to verify system design across the API, driver, operating system, and hardware. The system generated pictures and, more importantly, spool files (command and data streams that cross the hardware and software interface). These spool files are then run against the hardware models to verify hardware design under real OpenGL use scenarios.

This prototyping environment has the following advantages:

- Reduces risk for system design and component design
  - Resolve integration issues early

- Identify holes and design or architecture flaws
- Enable prototyping to evaluate design alternatives

- Enables key deliverables (hardware verification spool files)
- Creates exciting focal points for developers
- Fosters teamwork
- Enables joint development
- Provides a means to monitor progress
- Provides a jump start to our code development phase.

This environment also has potential downsides. We felt there was a risk that developers would feel that the need or desire to prototype (for system turn on and hardware verification) could overshadow the importance of product design. We did not want to leave engineers with the model: write some code, give it a try, and ship it if it works.

Thus, to keep the benefits of this environment and mitigate these potential downsides, we made a conscious decision to switch gears from system turn on and prototype mode to product code development mode. This point came after we had delivered the spool files required for hardware verification and before we had reached our design complete checkpoint. From that point on, we prototyped only for design purposes, not for enabling more system functionality. We also created explicit checkpoints for replacing previously prototyped code with designed product code. This was an important shift to avoid shipping prototype code. All product code had to be designed and reviewed.

Hardware Simulation

One key factor in our concurrent engineering process is hardware simulation. A detailed discussion of the hardware simulation techniques used in our project are beyond the scope of this article. Briefly, we use three levels of hardware simulation:

- A behavioral model (written in C)
- A register transfer level model (RTL)
- A gate model, which models the gate design and implementation.

The advantages of the behavioral model are that it can be done well before the RTL and gate model so we can use it with other components and prototypes. The behavioral model is also significantly faster than the other models (though still about 100 times slower than the real product), allowing us to run many simple real programs on it. The RTL model runs in Verilog and runs about one million times slower than the real product. This limits the number and size of test cases that can be run. The gate model is even slower. Even so, we kept over 30 workstations busy around the clock for months running these models. Often a simulation run will use C models for all but one of the new chips, with the one chip being simulated at the gate level.

Milestones and Communication

We set up a number of R&D milestones to guide and track our progress. The vertical slice and OpenGL turn on were two such key milestones. OpenGL developer meetings were held monthly to make sure that everyone had a clear understanding of where we were headed and how each of the developers' contributions helped us get there.

Software and Hardware Design Reviews

The hardware and software engineers also held joint design reviews. The value of design reviews is to minimize defects by enabling all the engineers to have the same model of the system and to catch design flaws early and correct them while defect finding and fixing is still inexpensive in terms of schedule and dollars.

On the software side, the review process focused heavily on up-front design reviews (where changes are cheaper) to get the design right. We maintained the importance of doing inspections but reduced the inspection coverage from 100 percent to a smaller representative subset of code, as determined by the review team. We also increased the number of reviewers at the design reviews and reduced the participation as we moved to code reviews. We maintained a consistent core set of reviewers who followed the component from design to code review.

Tests Written in Parallel

To bring more parallelism to the development process, we had an outside organization develop our OpenGL test programs. By doing so, we were able to begin nightly regression testing simultaneous with the code completion checkpoint because the test programs were immediately available. Historically, the developers have written the tests following design and coding. This translates into a lull between the code completion checkpoint and the beginning of the testing phase.

Parallel development of the tests with the design and implementation of the system was a key success factor in our ability to ship a high-quality, software-only beta version of our OpenGL product. No severe defects were found in this beta product—our first OpenGL customer deliverable.

One thing we learned from using an outside organization to help with test writing was that writing test plans is more a part of design than of testing. The developers, with intimate knowledge of the API and the design, were able to write much more comprehensive test plans than the outside organization.

Conclusion

We achieved several positive results through the use of concurrent engineering on our OpenGL product. Ultimately, we reduced time to market by several months. Along the way, we made performance and reliability improvements in our software and hardware architectures and implementations, and we likely prevented a chip turn or two, which would have cost significant time to market.

Silicon Graphics and OpenGL are registered trademarks of Silicon Graphics Inc. in the United States and other countries.

Direct 3D is a U.S. registered trademark of Microsoft Corporation.

Microsoft is a U.S. registered trademark of Microsoft Corporation.

Advanced Display Technologies on HP-UX Workstations

Todd M. Spencer

Paul M. Anderson

David J. Sweetser

Multiple monitors can be configured as a contiguous viewing space to provide more screen space so that users can see most, if not all, of their applications without any special window manipulations.

In today's computing environment, screen space is at a premium. The entire screen can be easily consumed when primary work-specific applications are used together with browsers, schedulers, mailers, and editors. This forces the user to continuously shuffle windows, which is both distracting and unproductive.

The advanced display technologies described here allow users to increase productivity by reducing the time spent manipulating windows. Three technologies are discussed:

- Multiscreen
- Single logical screen (SLS)
- SLSclone.

Implementation details and procedures for configuring HP-UX workstations to use the SLS technology are described in references 1 and 2.

Multiscreen

When considering the problem of limited screen space, one solution that comes to mind is to use a bigger monitor with a higher resolution. Unfortunately, it is often impractical to add a monitor with a resolution high enough to accommodate all the data a user wants to view. Although demand has increased for monitors of higher resolution, such as 2K by 2K pixels, they are still too expensive for companies to place on every desktop. In addition, these large monitors are cumbersome and heavy. There are also safety considerations: the monitor must be stable and properly supported.

A more practical, cost-effective solution is to use additional standalone monitors to increase the amount of visible screen space. The X Window System (X11) standard incorporates a feature known as multiscreen, which provides this type of environment. In multiscreen configurations, a single X server is used to control more than one graphics device and monitor simultaneously. These types of configurations are only possible on systems containing multiple graphics devices.

In these multiscreen scenarios, a single mouse and keyboard are shared between screens. This allows the pointer but not the windows to move between screens. Each application must be directed to a specific screen to display its windows. This is done by either using the –display command line argument or by setting the DISPLAY environment variable.

Figure 1 shows a two-monitor multiscreen configuration. Both monitors are connected to the same workstation and are controlled by the same X server. This type of configuration effectively doubles the visible workspace. For example, users could have their alternate applications, such as web browsers, mailers, and schedulers on the left-hand monitor and their primary applications on the right-hand monitor. Since the X server controls both screens, the pointer can move between screens and be used with any application.

Multiscreen offers the advantage that it will work with any graphics device. There are no constraints that the graphics devices be identical or have the same properties.

Figure 1

A multiscreen configuration.

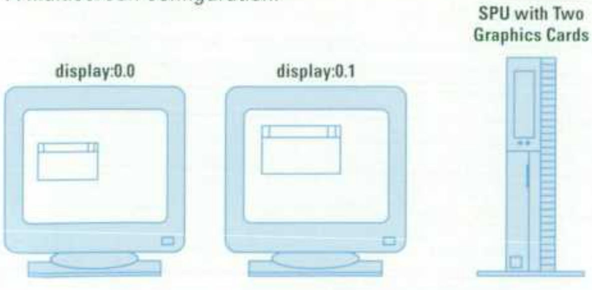
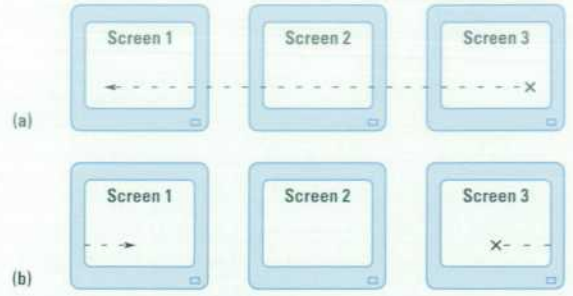


Figure 2

Cursor wraparound in a multiscreen configuration.



For example, on an HP 9000 Model 715 workstation containing an HCRX24 display (a 24-plane device) and an internal color graphics display (an 8-plane device), the user can still create a multiscreen configuration. Of course, those applications directed to the HCRX24 will have access to 24 planes while those contained on the other are limited to 8 planes. Currently, the HP-UX X server allows a maximum of four graphics devices to be used in a multiscreen configuration.

The HP-UX X server also provides several enhancements to simplify the use of a multiscreen configuration. If a user has a 1-by-3 configuration (**Figure 2a**), there may be a need to move the pointer from screen 3 to screen 1. This requires moving the pointer from screen 3 to screen 2 to screen 1. By specifying an X server configuration option, the user can move the pointer off the right edge of screen 3, and the pointer will wrap to screen 1 (**Figure 2b**). The same screen wrapping functionality can be provided if the user has configured the screens in a column. Finally, a 2-by-2 configuration can contain both horizontal and vertical screen wrapping.

Although multiscreen is convenient, it has shortcomings. Namely, the monitors function as separate entities, rather than as a contiguous space. The different screens within a multiscreen configuration cannot communicate with one another with respect to window placement. This means that windows cannot be moved between monitors. Once a window is created, it is bound to the monitor where it is created. Although some third-party solutions are available to help alleviate this problem, they are costly, inconvenient (sometimes requiring the application to make code changes), and lack performance.

The lack of communication between screens with respect to window placement forces users to direct their applications towards a specific screen at application start time. After a screen has been selected all additional subwindows will be confined to that screen. With today's larger applications, it is possible to find that certain screens still get overcrowded, resulting in the original predicament of having to iconify and raise windows.

Single Logical Screen

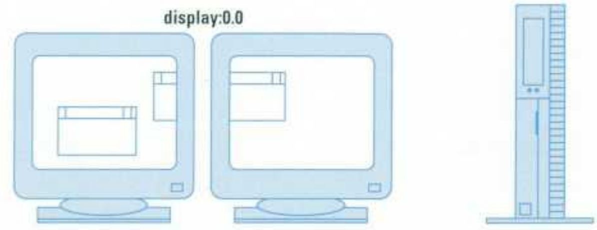
To remedy the shortfall of the multiscreen configuration, HP developed a technology called *single logical screen* (SLS).[3] SLS has been incorporated into the HP standard X server product and allows multiple monitors to act as a single, larger, contiguous screen. As a result, windows can move across physical screen boundaries, and they can span more than one physical monitor. In addition, SLS functionality has been implemented in an application-transparent manner. This means that any application currently running on HP-UX workstations will run, without modification, under SLS. Therefore, SLS is not an API that application writers need to program to or that an application needs to be aware of. The application simply sees a large screen. This ease-of-use lets end users take advantage of a large workspace without requiring applications to be rewritten or recompiled.

Many of electronic design automation (EDA) and computer-aided design applications can benefit from SLS. Some of these applications, by themselves, can easily occupy an entire screen while only showing a fraction of the desired information. For example, with more screen real estate, an EDA application can simultaneously display waveforms, schematics, editors, and other data without having any of this information obscured. To do this on a workstation with only a single monitor would require displaying the waveforms, schematics, and other items in such small areas as to be unreadable.

On HP-UX Workstations, a single logical screen actually represents a collection of homogeneous graphics devices whose output has been combined into a single screen. **Figure 3**, shows an example of a 1-by-2 SLS configuration. Most HP-UX workstations are not limited to only two graphics devices. Some models support up to four devices. When using these graphics devices to create an SLS environment, any rectangular configuration is allowed.

Figure 3

A 1-by-2 SLS configuration.



SLSclone

SLSclone is similar to the SLS configuration. The difference is that the contents from a selected monitor are replicated on all other monitors in the configuration (see **Figure 4**). A user can dynamically switch between SLS and SLSclone using an applet being shipped with the HP-UX 10.20 patch PHSS_12462 or later.

This functionality is useful in an educational or instructional environment. Instead of crowding many users around a single monitor to view its contents, SLSclone can be used to pipe these contents to neighboring monitors. As with SLS, SLSclone currently supports up to four physical monitors, depending on the workstation model.

SLSclone functionality easily lends itself to a collaborative work environment. If additional people enter a user's office to debug some software source code, for example, the user can quickly switch the SLS configuration into an SLSclone configuration, and the debugging screen will be displayed on all monitors. Also, the additional monitor can easily be adjusted to the correct height and tilt without affecting the original user's view of the display.

Figure 4

An example of a 1-by-2 SLSclone configuration.

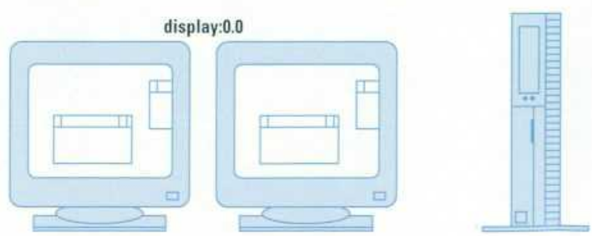
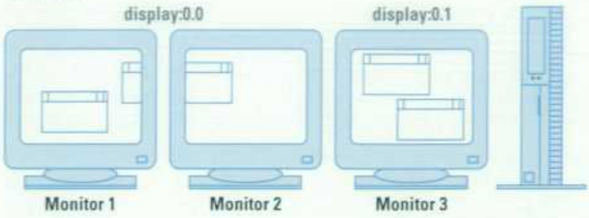


Figure 5

A hybrid configuration consisting of a 1-by-2 SLS with multiscreen.



SLS and Multiscreen

Even with the benefits of SLS, there may be cases in which a user will want to use SLS and multiscreen at the same time. For example, a user could have a 1-by-2 SLS configuration acting as one screen, and a third monitor acting as a second screen. A depiction of this is shown in **Figure 5**.

In this type of configuration, a user can move windows between physical monitors 1 and 2 but not drag a window from monitor 2 to monitor 3. The pointer, however, can move between all monitors. This type of hybrid configuration can be useful in a software development environment. All of the necessary editors, compilers, and debuggers can be used on monitors 1 and 2, and the application can be run and tested on monitor 3.

If a workstation supports four graphics devices, another possible hybrid configuration is to use two screens, each of which consists of a two-screen SLS configuration (**Figure 6**).

In this configuration, windows can be moved between monitors 1 and 2 or between monitors 3 and 4. However, a window cannot be moved between monitors 2 and 3. As with all multiscreen configurations, the pointer can move across all four monitors. These two screens could also be placed vertically, resulting in a 2-by-2 monitor arrangement and a 2-by-1 multiscreen configuration.

Conclusion

Advanced display configurations can be used to increase productivity. The increase in screen space facilitates collaboration and communication of information. We have also found that these configurations are very useful for independent software vendors (ISVs) who demonstrate their applications on HP-UX workstations. They appreciate the additional screen space because they are able to display more information and rapidly describe their products without losing their customers' attention.

Finally, the configuration of an advanced display is accomplished in an easy and straightforward manner through the HP-UX System Administration Manager (SAM). Additional information on advanced display configurations and other exciting X server features are available at: http://www.hp.com/go/xwindow

HP-UX Release 10.20 and later and HP-UX 11.00 and later (in both 32- and 64-bit configurations) on all HP 9000 computers are Open Group UNIX 95 branded products.

References

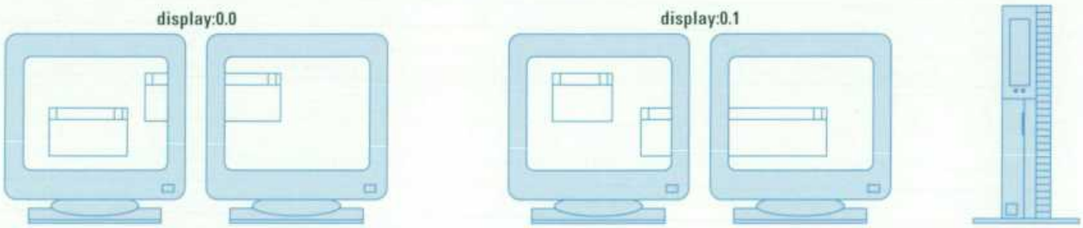
1. T. Spencer and P. Anderson, "Implementation of Advanced Display Technologies on HP-UX Workstations," *Hewlett-Packard Journal*, Vol. 49, no. 2, May 1998 (available online only).
http://www.hp.com/hpj/98may/ma98a7a.htm

2. R. MacDonald, "Hewlett-Packard's Approach to Dynamic Loading within the X Server," *Hewlett-Packard Journal*, Vol. 49, no. 2, May 1998 (available online only).
http://www.hp.com/hpj/98may/ma98a7b.htm

3. M. Allison, P. Anderson, and J. Walls, "Single Logical Screen," *InterWorks '97 Proceedings*, April 1997, pp. 366 - 376.

Figure 6

Two 1-by-2 SLS configurations combined via multiscreen.



Todd M. Spencer

A software engineer at the HP Workstation Systems Division, Todd Spencer was responsible for development of the the SAM component that allows users to set up multiscreen and single logical screen configurations. He came to HP in 1989 after receiving an MS degree in computer science from the University of Utah. Todd was born in Utah, is married and has four children. His outside interests include tropical fish, camping, woodworking, piano (playing classical music), and jogging.

Paul M. Anderson

Paul Anderson is a software engineer at the HP Workstation Systems Division. He joined HP in 1996 after receiving a BS degree in computer science from the University of Minnesota. He is currently working on device drivers for new peripheral technologies. His professional interests include I/O drivers, operating systems, and networking. Paul was born in Edina, Minnesota. His outside interests include hiking, music, and mountain biking.

David J. Sweetser

With HP since 1977, David Sweetser is a project manager at the HP Workstation Systems Division. He is responsible for the X server and some of the client-side X libraries. He received a BSEE degree and an MSEE degree from Harvey Mudd College in 1975. He was born in Woodland, California, is married and has two children. His outside interests include mountain biking, hiking, snowshoeing, cross-country skiing, and white-water rafting.

Delivering PCI in HP B-Class and C-Class Workstations: A Case Study in the Challenges of Interfacing with Industry Standards

Ric L. Lewis

Erin A. Handgen

Nicholas J. Ingegneri

Glen T. Robinson

In the highly competitive workstation market, customers demand a wide range of cost-effective, high-performance I/O solutions. An industry-standard I/O subsystem allows HP workstations to support the latest I/O technology.

Industry-standard I/O buses like the Peripheral Component Interconnect (PCI) allow systems to provide a wide variety of cost-effective I/O functionality. The desire to include more industry-standard interfaces in computer systems continues to increase. This article points out some of the specific methodologies used to implement and verify the PCI interface in HP workstations and describes some of the challenges associated with interfacing with industry-standard I/O buses.

PCI for Workstations

One of the greatest challenges in designing a workstation system is determining the best way to differentiate the design from competing products. This decision determines where the design team will focus their efforts and have the greatest opportunity to innovate. In the computer workstation industry, the focus is typically on processor performance coupled with high-bandwidth, low-latency memory connections to feed powerful graphics devices. The performance of nongraphics I/O devices in workstations is increasing in importance, but the availability of cost-effective solutions is still the chief concern in designing an I/O subsystem. Rather than providing a select few exotic high-performance I/O solutions, it is better to make sure that there is a wide range of cost-effective solutions to provide the I/O functionality that each customer requires. Since I/O performance is not a primary means of differentiation and since maximum flexibility with appropriate price and performance is desired, using an industry-standard I/O bus that operates with high-volume cards from multiple vendors is a good choice.

The PCI bus is a recently established standard that has achieved wide acceptance in the PC industry. Most new general-purpose I/O cards intended for use in PCs and workstations are now being designed for PCI. The PCI bus was developed by the PCI Special Interest Group (PCI SIG), which was founded by Intel and now consists of many computer vendors. PCI is designed to meet today's I/O performance needs and is scalable to meet future needs. Having PCI in workstation systems allows the use of competitively priced cards already available for use in the high-volume PC business. It also allows workstations to keep pace with new I/O functionality as it becomes available, since new devices are typically designed for the industry-standard bus first and only later (if at all) ported to other standards. For these reasons, the PCI bus has been implemented in the HP B-class and C-class workstations.

PCI Integration Effort

Integrating PCI into our workstation products required a great deal of work by both the hardware and software teams. The hardware effort included designing a bus interface ASIC (application-specific integrated circuit) to connect to the PCI bus and then performing functional and electrical testing to make sure that the implementation would work properly. The software effort included writing firmware to initialize and control the bus interface ASIC and PCI cards and writing device drivers to allow the HP-UX* operating system to make use of the PCI cards.

The goals of the effort to bring PCI to HP workstation products were to:

- Provide our systems with fully compatible PCI to allow the support of a wide variety of I/O cards and functionality
- Achieve an acceptable performance in a cost-effective manner for cards plugged into the PCI bus
- Create a solution that does not cause performance degradation in the CPU-memory-graphics path or in any of the other I/O devices on other buses in the system
- Ship the first PCI-enabled workstations: the Hewlett-Packard B132, B160, C160, and C180 systems.

Challenges

Implementing an industry-standard I/O bus might seem to be a straightforward endeavor. The PCI interface has a thorough specification, developed and influenced by many experts in the field of I/O bus architectures. There is momentum in the industry to make sure the standard succeeds. This momentum includes card vendors working to design I/O cards, system vendors working through the design issues of the specification, and test and measurement firms developing technologies to test the design once it exists. Many of these elements did not yet exist and were challenges for earlier Hewlett-Packard proprietary I/O interface projects.

Although there were many elements in the team's favor that did not exist in the past, there were still some significant tasks in integrating this industry-standard bus. These tasks included:

- Designing the architecture for the bus interface ASIC, which provides a high-performance interface between the internal proprietary workstation buses and PCI
- Verifying that the bus interface ASIC does what it is intended to do, both in compliance with PCI and in performance goals defined by the team
- Providing the necessary system support, primarily in the form of firmware and system software to allow cards plugged into the slots on the bus interface ASIC to work with the HP-UX operating system.

With these design tasks identified, there still remained some formidable challenges for the bus interface ASIC design and verification and the software development teams. These challenges included ambiguities in the PCI specification, difficulties in determining migration plans, differences in the way PCI cards can operate within the PCI specification, and the unavailability of PCI cards with the necessary HP-UX drivers.

Architecture

The Bus Interface ASIC

The role of the bus interface ASIC is to bridge the HP proprietary I/O bus, called the general system connect (GSC) bus, to the PCI bus in the HP B-class and C-class workstations. **Figures 1** and **2** show the B-class and C-class workstation system block diagrams with the bus interface ASIC bridging the GSC bus to the PCI bus. The Runway bus shown in **Figure 2** is a high-speed processor-to-memory bus.[1]

The bus interface ASIC maps portions of the GSC bus address space onto the PCI bus address space and vice versa. System firmware allocates addresses to map between the GSC and PCI buses and programs this information into configuration registers in the bus interface ASIC. Once programmed, the bus interface ASIC performs the following tasks:

- Forward writes transactions from the GSC bus to the PCI bus. Since the write originates in the processor, this task is called a processor I/O write.
- Forward reads requests from the GSC bus to the PCI bus, waits for a PCI device to respond, and returns the read data from the PCI bus back to the GSC bus. Since the read originates in the processor, this task is called a processor I/O read.
- Forward writes transactions from the PCI bus to the GSC bus. Since the destination of the write transaction is main memory, this task is called a direct memory access (DMA) write.
- Forward reads requests from the PCI bus to the GSC bus, waits for the GSC host to respond, and returns the read data from the GSC bus to the PCI bus. Since the source of the read data is main memory, this task is called a DMA read.

Figure 3 shows a block diagram of the internal architecture of the bus interface ASIC. The bus interface ASIC uses five asynchronous FIFOs to send address, data, and transaction information between the GSC and PCI buses.

Figure 1

HP B-class workstation block diagram.

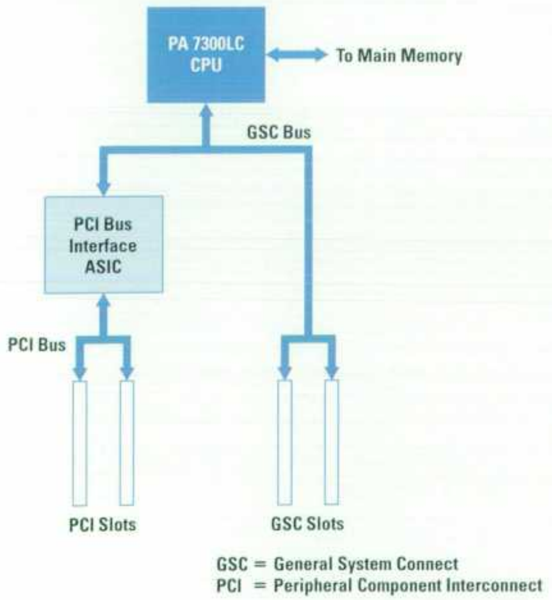


Figure 2

HP C-class workstation block diagram.

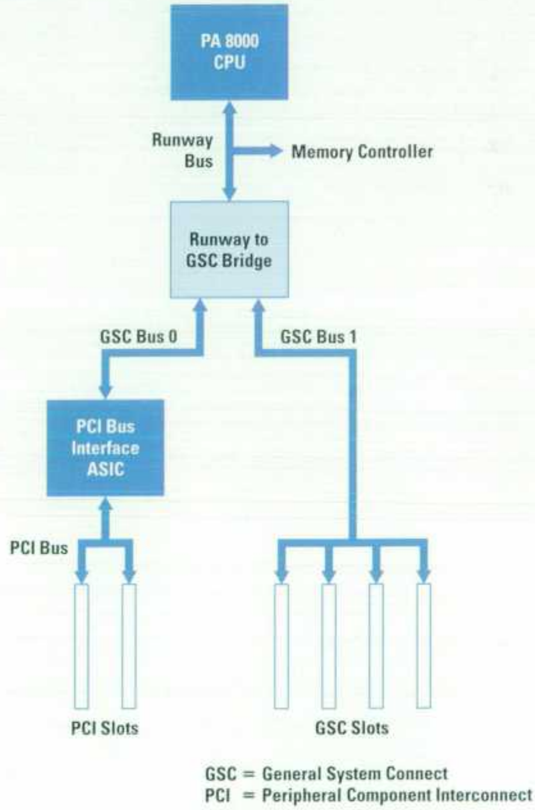
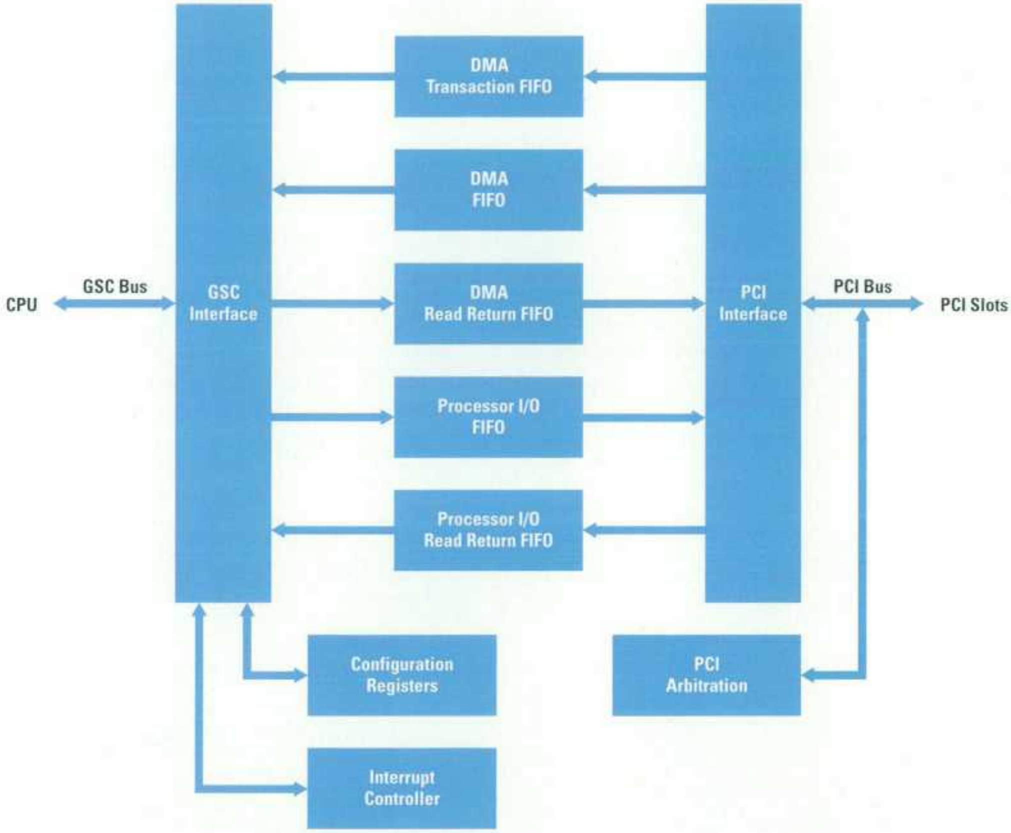


Figure 3

A block diagram of the architecture for the bus interface ASIC.



A FIFO is a memory device that has a port for writing data into the FIFO and a separate port for reading data out of the FIFO. Data is read from the FIFO in the same order that it was written into the FIFO. The GSC bus clock is asynchronous to the PCI bus clock. For this reason, the FIFOs need to be asynchronous. An asynchronous FIFO allows the data to be written into the FIFO with a clock that is asynchronous to the clock used to read data from the FIFO.

Data flows through the bus interface ASIC are as follows:

- Processor I/O write:
  - The GSC interface receives both the address and the data for the processor I/O write from the GSC bus and loads them into the processor I/O FIFO.
  - The PCI interface arbitrates for the PCI bus.
  - The PCI interface unloads the address and data from the processor I/O FIFO and masters the write on the PCI bus.
- Processor I/O read:
  - The GSC interface receives the address for the processor I/O read from the GSC bus and loads it into the processor I/O FIFO.
  - The PCI interface arbitrates for the PCI bus.
  - The PCI interface unloads the address from the processor I/O FIFO and masters a read on the PCI bus.
  - The PCI interface waits for the read data to return and loads the data into the processor I/O read return FIFO.
  - The GSC interface unloads the processor I/O read return FIFO and places the read data on the GSC bus.

- DMA Write:
  - The PCI interface receives both the address and the data for the DMA write from the PCI bus and loads them into the DMA FIFO.
  - The PCI interface loads control information for the write into the DMA transaction FIFO.
  - The GSC interface arbitrates for the GSC bus.
  - The GSC interface unloads the write command from the DMA transaction FIFO, unloads the address and data from the DMA FIFO, and masters the write on the GSC bus.
- DMA Read:
  - The PCI interface receives the address for the DMA read from the PCI bus and loads it into the DMA FIFO.
  - The GSC interface arbitrates for the GSC bus.
  - The GSC interface unloads the address from the DMA FIFO and masters a read on the GSC bus
  - The GSC interface then waits for the read data to return and loads the data into the DMA read return FIFO.
  - The PCI interface unloads the DMA read return FIFO and places the read data on the PCI bus.

Architectural Challenges

One of the difficulties of joining two dissimilar I/O buses is achieving peak I/O bus performance despite the fact that the transaction structures are different for both I/O buses. For example, transactions on the GSC bus are fixed length with not more than eight words per transaction while transactions on the PCI bus are of arbitrary length. It is critical to create long PCI transactions to reach peak bandwidth on the PCI bus. For better performance and whenever possible, the bus interface ASIC coalesces multiple processor I/O write transactions from the GSC bus into a single processor I/O write transaction on the PCI bus. For DMA writes, the bus interface ASIC needs to determine the optimal method of breaking variable-size PCI transactions into one-, two-, four-, or eight-word GSC transactions. The PCI interface breaks DMA writes into packets and communicates the transaction size to the GSC interface through the DMA transaction FIFO.

Another difficulty of joining two dissimilar I/O buses is avoiding deadlock conditions. Deadlock conditions can occur when a transaction begins on both the GSC and PCI buses simultaneously. For example, if a processor I/O read begins on the GSC bus at the same time a DMA read begins on the PCI bus, then the processor I/O read will wait for the DMA read to be completed before it can master its read on the PCI bus. Meanwhile, the DMA read will wait for the processor I/O read to be completed before it can master its read on the GSC bus. Since both reads are waiting for the other to be completed, we have a deadlock case. One solution to this problem is to detect the deadlock case and retry or split one of the transactions to break the deadlock. In general, the bus interface ASIC uses the GSC split protocol to divide a GSC transaction and allow a PCI transaction to make forward progress whenever it detects a potential deadlock condition.

Unfortunately, the bus interface ASIC adds more latency to the round trip of DMA reads. This extra latency can have a negative affect on DMA read performance. The C-class workstation has a greater latency on DMA reads than the B-class workstation. This is due primarily to the extra layer of bus bridges that the DMA read must traverse to get to memory and back (refer to **Figures 1** and **2**). The performance of DMA reads is important to outbound DMA devices such as network cards and disk controllers. The extra read latency is hidden by prefetching consecutive data words from main memory with the expectation that the I/O device needs a block of data and not just a word or two.

Open Standard Challenges

The PCI bus specification, like most specifications, is not perfect. There are areas where the specification is vague and open to interpretation. Ideally, when we find a vague area of a specification, we investigate how other designers have interpreted the specification and follow the trend. With a proprietary bus this often means simply contacting our partners within HP and resolving the issue. With an industry-standard bus, our partners are not within the company, so resolving the issue is more difficult. The PCI mail reflector, which is run by the PCI SIG at www.pcsig.com, is sometimes helpful for resolving such issues. Monitoring the PCI mail reflector also gives the benefit of seeing what parts of the PCI specification appear vague to others. Simply put, engineers designing to a standard need a forum for communicating with others using that standard. When designing to an industry standard, that forum must by necessity include wide representation from the industry.

The PCI specification has guidelines and migration plans that PCI card vendors are encouraged to follow. In practice, PCI card vendors are slow to move from legacy standards to follow guidelines or migration plans. For example, the PCI bus supports a legacy I/O* address space that is small and fragmented. The PCI bus also has a memory address space that is large and has higher write bandwidth than the I/O address space. For obvious reasons, the PCI specification recommends that all PCI cards map their registers to the PCI I/O address space and the PCI memory address space so systems will have the most flexibility in allocating base addresses to I/O cards. In practice, most PCI cards still only support the PCI address I/O space. Since we believed that the PCI I/O address space would almost never be used, trade-offs were made in the design of the bus interface ASIC that compromised the performance of transactions to the PCI I/O address space.

Another example in which the PCI card vendors follow legacy standards rather than PCI specification guidelines is in the area of PCI migration from 5 volts to 3.3 volts. The PCI specification defines two types of PCI slots: one for a 5-volt signaling environment and one for a 3.3-volt signaling environment. The specification also defines three possible types of I/O cards: 5-volt only, 3.3-volt only, or universal. As their names imply, 5-volt-only and 3.3-volt-only cards only work in 5-volt and 3.3-volt slots respectively. Universal cards can work in either a 5-volt or 3.3-volt slot. The PCI specification recommends that PCI card vendors only develop universal cards. Even though it costs no more to manufacture a universal card than a 5-volt card, PCI card vendors are slow to migrate to universal cards until volume platforms (that is, Intel-based PC platforms) begin to have 3.3-volt slots.

Verification

Verification Methodology and Goals

The purpose of verification is to ensure that the bus interface ASIC correctly meets the requirements described in the design specification. In our VLSI development process this verification effort is broken into two distinct parts called phase-1 and phase-2. Both parts have the intent of proving that the design is correct, but each uses different tools and methods to do so. Phase-1 verification is carried out on a software-based simulator using a model of the bus interface ASIC. Phase-2 verification is carried out on real chips in real systems.

Phase-1. The primary goals of phase-1 verification can be summarized as correctness, performance, and compliance. Proving correctness entails showing that the Verilog model of the design properly produces the behavior detailed in the specification. This is done by studying the design specification, enumerating a function list of operations and behaviors that the design is required to exhibit, and generating a suite of tests that verify all items on that function list. Creating sets of randomly generated transaction combinations enhances the test coverage by exposing the design to numerous corner cases.

Performance verification is then carried out to prove that the design meets or exceeds all important performance criteria. This is verified by first identifying the important performance cases, such as key bandwidths and latencies, and then generating tests that produce simulated loads for performance measurement.

Finally, compliance testing is used to prove that the bus protocols implemented in the design will work correctly with other devices using the same protocol. For a design such as the bus interface ASIC that implements an industry-standard protocol, special consideration was given to ensure that the design would be compatible with a spectrum of outside designs.

Phase-2. This verification phase begins with the receipt of the fabricated parts. The effort during this phase is primarily focused on testing the physical components, with simulation techniques restricted to the supporting role of duplicating and better understanding phenomenon seen on the bench. The goals of phase-2 verification can be summarized as compliance, performance, and compatibility. Therefore, part of phase-2 is spent proving that the physical device behaves on the bench the same as it did in simulation. The heart of phase-2, however, is that the design is finally tested for compatibility with the actual devices that it will be connected to in a production system.

* Legacy refers to the Intel I/O port space.

Verification Challenges

From the point of view of a verification engineer, there are benefits and difficulties in verifying the implementation of an industry-standard bus as compared to a proprietary bus. One benefit is that since PCI is an industry standard, there are plenty of off-the-shelf simulation and verification tools available. The use of these tools greatly reduces the engineering effort required for verification, but at the cost of a loss of control over the debugging and feature set of the tools.

The major verification challenge (particularly in phase-1) was proving compliance with the PCI standard. When verifying compliance with a proprietary standard there are typically only a few chips that have to be compatible with one another. The design teams involved can resolve any ambiguity in the bus specification. This activity tends to involve only a small and well-defined set of individuals. In contrast, when verifying compliance with an open standard there is usually no canonical source that can provide the correct interpretation of the specification. Therefore, it is impossible to know ahead of time where devices will differ in their implementation of the specification. This made it somewhat difficult for us to determine the specific tests required to ensure compliance with the PCI standard. In the end, it matters not only how faithfully the specification is followed, but also whether or not the design is compatible with whatever interpretation becomes dominant.

The most significant challenge in phase-2 testing came in getting the strategy to become a reality. The strategy depended heavily on real cards with real drivers to demonstrate PCI compliance. However, the HP systems with PCI slots were shipped before any PCI cards with drivers were supported on HP workstations. Creative solutions were found to develop a core set of drivers to complete the testing. However, this approach contributed to having to debug problems closer to shipment than would have been optimal. Similarly, 3.3-volt slots were to be supported at first shipment. The general unavailability of 3.3-volt or universal (supporting 5 volts and 3.3 volts) cards hampered this testing. These are examples of the potential dangers of "preenabling" systems with new hardware capability before software and cards to use the capability are ready.

An interesting compliance issue was uncovered late in phase-2. One characteristic of the PA 8000 C-class system is that when the system is heavily loaded, the bus interface ASIC can respond to PCI requests with either long read latencies (over 1 μs before acknowledging the transaction) or many (over 50) sequential PCI retry cycles. Both behaviors are legal with regard to the PCI 2.0 bus specification, and both of them are appropriate given the circumstances. However, neither of these behaviors is exhibited by Intel's PCI chipsets, which are the dominant implementation of the PCI bus in the PC industry. Several PCI cards worked fine in a PC, but failed in a busy C-class system. The PCI card vendors had no intention of designing cards that were not PCI compliant, but since they only tested their cards in Intel-based systems, they never found the problem. Fortunately, the card vendors agreed to fix this issue on each of their PCI cards. If there is a dominant implementation of an industry standard, then deviating from that implementation adds risk.

Firmware

Firmware is the low-level software that acts as the interface between the operating system and the hardware. Firmware is typically executed from nonvolatile memory at startup by the workstation. We added the following extensions to the system firmware to support PCI:

- A bus walk to identify and map all devices on the PCI bus
- A reverse bus walk to configure PCI devices
- Routines to provide boot capability through specified PCI cards.

The firmware bus walk identifies all PCI devices connected to the PCI bus and records memory requirements in a resource request map. When necessary, the firmware bus walk will traverse PCI-to-PCI bridges.* If a PCI device has Built-in Self Test (BIST), the BIST is run, and if it fails, the PCI device is disabled and taken out of the resource request map. As the bus walk unwinds, it initializes bridges and allocates resources for all of the downstream PCI devices.

Firmware also supports booting the HP-UX operating system from two built-in PCI devices. Specifically, firmware can load the HP-UX operating system from either a disk attached to a built-in PCI SCSI chip or from a file server attached to a built-in PCI 100BT LAN chip.

* A PCI-to-PCI bridge connects two PCI buses, forwarding transactions from one to the other.

Firmware Challenges

The first challenge in firmware was the result of another ambiguity in the PCI specification. The specification does not define how soon devices on the PCI bus must be ready to receive their first transaction after the PCI bus exits from reset. Several PCI cards failed when they were accessed shortly after PCI reset went away. These cards need to download code from an attached nonvolatile memory before they will work correctly. The cards begin the download after PCI reset goes away, and it can take hundreds of milliseconds to complete the download. Intel platforms delay one second after reset before using the PCI bus. This informal compliance requirement meant that firmware needed to add a routine to delay the first access after the PCI bus exits reset.

Interfacing with other ASICs implementing varying levels of the PCI specification creates additional challenges. Compliance with PCI 2.0 (PCI-to-PCI) bridges resulted in two issues for firmware. First, the bridges added latency to processor I/O reads. This extra latency stressed a busy system and caused some processor I/O reads to timeout in the processor and bring down the system. The firmware was changed so that it would reprogram the processor timeout value to allow for this extra delay. The second issue occurs when PCI 2.0 bridges are stacked two or more layers deep. It is possible to configure the bridges such that the right combination of processor I/O reads and DMA reads will cause the bridges to retry each others transactions and cause a deadlock or starve one of the two reads. Our system firmware fixes this problem by supporting no more than two layers of PCI-to-PCI bridges and configuring the upstream bridge with different retry parameters than the downstream bridge.

Operating System Support

The HP-UX operating system contains routines provided for PCI-based kernel drivers called *PCI services*. The first HP-UX release that provides PCI support is the 10.20 release. An infrastructure exists in the HP-UX operating system for kernel-level drivers, but the PCI bus introduced several new requirements. The four main areas of direct impact include context dependent I/O, driver attachment, interrupt service routines (ISR), and endian issues. Each area requires special routines in the kernel's PCI services.

Context Dependent I/O

In the HP-UX operating system, a centralized I/O services context dependent I/O (CDIO) module supplies support for drivers that conform to its model and consume its services. Workstations such as the C-class and B-class machines use the workstation I/O services CDIO (WSIO CDIO) for this abstraction layer. The WSIO CDIO provides general I/O services to bus-specific CDIOs such as EISA and PCI. Drivers that are written for the WSIO environment are referred to as WSIO drivers. The services provided by WSIO CDIO include system mapping, cache coherency management, and interrupt service linkage. In cases where WSIO CDIO does need to interface to the I/O bus, WSIO CDIO translates the call to the appropriate bus CDIO. For the PCI bus, WSIO CDIO relies on services in PCI CDIO to carry out bus-specific code.

Ideally, all PCI CDIO services should be accessed only through WSIO CDIO services. However, there are a number of PCI-specific calls that cannot be hidden with a generic WSIO CDIO interface. These functions include PCI register operations and PCI bus tuning operations.

Driver Attachment

The PCI CDIO is also responsible for attaching drivers to PCI devices. The PCI CDIO completes a PCI bus walk, identifying attached cards that had been set up by firmware. The PCI CDIO initializes data structures, such as the interface select code (ISC) structure, and maps the card memory base register. Next, the PCI CDIO calls the list of PCI drivers that have linked themselves to the PCI attach chain.

The PCI driver is called with two parameters: a pointer to an ISC structure (which contains mapping information and is used in most subsequent PCI services calls) and an integer containing the PCI device's vendor and device IDs. If the vendor and device IDs match the driver's interface, the driver attach routine can do one more check to verify its ownership of the device by reading the PCI subsystem vendor ID and subsystem ID registers in the configuration space. If the driver does own this PCI device, it typically initializes data structures, optionally links in an interrupt service routine, initializes and claims the interface, and then calls the next driver in the PCI attach chain.

Interrupt Service Routines

The PCI bus uses level-sensitive, shared interrupts. PCI drivers that use interrupts use a WSIO routine to register their interrupt service routine with the PCI CDIO. When a PCI interface card asserts an interrupt, the operating system calls the PCI CDIO to do the initial handling. The PCI CDIO determines which PCI interrupt line is asserted and then calls each driver associated with that interrupt line.

The PCI CDIO loops, calling drivers for an interrupt line until the interrupt line is deasserted. When all interrupt lines are deasserted, the PCI CDIO reenables interrupts and returns control to the operating system. To prevent deadlock, the PCI CDIO has a finite (although large) number of times it can loop through an interrupt level before it will give up and leave the interrupt line disabled. Once disabled, the only way to reenable the interrupt is to reboot the system.

PCI Endian Issues

PCI drivers need to be cognizant of endian issues.* The PCI bus is inherently little endian while the rest of the workstation hardware is big endian. This is only an issue with card register access when the register is accessed in quantities other than a byte. Typically there are no endian issues associated with data payload since data payload is usually byte-oriented. For example, network data tends to be a stream of byte data. The PCI CDIO provides one method for handling register endian issues. Another method lies in the capability of some PCI interface chips to configure their registers to be big or little endian.

Operating System Support Challenges

We ran into a problem when third-party card developers were porting their drivers to the HP-UX operating system. Their drivers only looked at device and vendor identifiers and claimed the built-in LAN inappropriately. Many PCI interface cards use an industry-standard bus interface chip as a front end and therefore share the same device and vendor IDs. For example, several vendors use the Digital 2114X family of PCI-to-10/100 Mbits/s Ethernet LAN controllers, with each vendor customizing other parts of the network interface card with perhaps different physical layer entities. It is possible that a workstation could be configured with multiple LAN interfaces having the same vendor and device ID with different subsystem IDs controlled by separate drivers. A final driver attachment step was added to verify the driver's ownership of the device. This consisted of reading the PCI subsystem vendor ID and subsystem ID registers in the configuration space.

The HP-UX operating system does not have the ability to allocate contiguous physical pages of memory. Several PCI cards (for example, SCSI and Fibre Channel) require contiguous physical pages of memory for bus master task lists. The C-class implementation, which allows virtual DMA through TLB (translation lookaside buffer) entries, is capable of supplying 32K bytes of contiguous memory space. In the case of the B-class workstation, which does not support virtual DMA, the team had to develop a workaround that consisted of preallocating contiguous pages of memory to enable this class of devices.

Conclusion

PCI and Interoperability. We set out to integrate PCI into the HP workstations. The goal was to provide our systems with access to a wide variety of industry-standard I/O cards and functionality. The delivery of this capability required the creation and verification of a bus interface ASIC and development of the appropriate software support in firmware and in the HP-UX operating system.

Challenges of Interfacing with Industry Standards. There are many advantages to interfacing with an industry standard, but it also comes with many challenges. In defining and implementing an I/O bus architecture, performance is a primary concern. Interfacing proprietary and industry-standard buses and achieving acceptable performance is difficult. Usually the two buses are designed with different goals for different systems, and determining the correct optimizations requires a great deal of testing and redesign.

Maintaining compliance with an industry standard is another major challenge. It is often like shooting at a moving target. If another vendor ships enough large volumes of a nonstandard feature, then that feature becomes a de facto part of the standard. It is also very difficult to prove that the specification is met. In the end, the best verification techniques for us involved simply testing the bus interface ASIC against as many devices as possible to find where the interface broke down or performed poorly.

* Little endian and big endian are conventions that define how byte addresses are assigned to data that is two or more bytes long. The little endian convention places bytes with lower significance at lower byte addresses. (The word is stored "little-end-first.") The big endian convention places bytes with greater significance at lower byte addresses. (The word is stored "big-end-first.")

Finally, it is difficult to drive development and verification unless the functionality is critical to the product being shipped. The issues found late in the development cycle for the bus interface ASIC could have been found earlier if the system had required specific PCI I/O functionality for initial shipments. The strategy of preenabling the system to be PCI compatible before a large number of devices became available made it difficult to achieve the appropriate level of testing before the systems were shipped.

Successes. The integration of PCI into the HP workstations through design and verification of the bus interface ASIC and the development of the necessary software components has been quite successful. The goals of the PCI integration effort were to provide fully compatible, high-performance PCI capability in a cost-effective and timely manner. The design meets or exceeds all of these goals. The bandwidth available to PCI cards is within 98 percent of the bandwidth available to native GSC cards. The solution was ready in time to be shipped in the first PCI-enabled HP workstations B132, B160, C160, and C180.

The bus-bridge ASIC and associated software have since been enhanced for two new uses in the second generation of PCI on HP workstations. The first enhancement provides support for the GSC-to-PCI adapter to enable specific PCI functionality on HP server GSC I/O cards. The second is a version of the bus interface supporting GSC-2x (higher bandwidth GSC) and 64-bit PCI for increased bandwidth to PCI graphics devices on HP C200 and C240 workstations.

Acknowledgments

This article is a summary of some of the challenges experienced by numerous team members involved in the integration of PCI into workstations. We would like to specifically thank several of those team members who helped contribute to and review this article. George Letey, Frank Lettang, and Jim Peterson assisted with the architecture section. Vicky Hansen, Dave Klink, and J.L. Marsh provided firmware details. Matt Dumm and Chris Hyser reviewed the operating system information.

References

1. W. Bryg, K. Chan, and N. Fiduccia, "A High-Performance, Low-Cost Multiprocessor Bus for Workstations and Midrange Servers," *Hewlett-Packard Journal*, Vol. 47, no. 1, February 1996, p. 18.

Ric L. Lewis

A project manager at the HP Workstation Systems Division, Ric Lewis was responsible for managing the development of the PCI bus interface ASIC. He came to HP in 1987 after receiving a BSEE degree from Utah State University. He also has an MSEE degree (1993) from Stanford University and an MBA (1992) from Santa Clara University. Ric was born in Twin Falls, Idaho, and he is married and has one son. His outside interests include basketball, mountain biking, and skiing.

Erin A. Handgen

Erin Handgen is a technical contributor at the HP Workstation Systems Division working on ASICs for HP workstations. He was the lead engineer for the PCI bus interface ASIC during the shipment phase. He has a BS degree in computer and electrical engineering (1986) and an MSEE degree (1988) from Purdue University. He joined HP in 1988. Born in Warsaw, Indiana, he is married and has three children. His outside interests include astronomy, camping, and hiking.

Nicholas J. Ingegneri

A hardware design engineer at the HP Fort Collins Systems Laboratory, Nicholas Ingegneri was the lead verification engineer for the PCI bus interface ASIC. He has a BSEE degree (1989) from the University of Nevada at Reno and an MSEE degree (1994) from Stanford University. He came to HP in 1990. His outside interests include travel and camping.

Glen T. Robinson

Glen Robinson is a technical contributor at the HP Workstation Systems Division. He came to HP in 1979 and was responsible for the subsystem test for the PCI network and kernel drivers. He has an MBA degree (1979) from Chapman College. Born in Santa Monica, California, he is married and has two grown children. He enjoys biking and AKC hunting tests with Labrador retrievers.

Linking Enterprise Business Systems to the Factory Floor

Kenn S. Jennyc

Information is the fuel that drives today's business enterprises. The ability to link different components in the enterprise together in a user-friendly and transparent manner increases the effectiveness of companies involved in manufacturing and production.

Computers have had a profound effect on how companies conduct business. They are used to run enterprise business software and to automate factory-floor production. While this has been a great benefit, the level of coordination between computers running unrelated application software is usually limited. This is because such data transfers are difficult to implement, often requiring manual intervention or customized software. Until recently, off-the-shelf data transfer solutions were not available.

HP Enterprise Link is a middleware software product that increases the effectiveness of companies involved in manufacturing and production. It allows business management software running at the enterprise level, such as SAP's R/3 product, to exchange information (via electronic transfer) with software applications running on the factory floor. It also allows software applications running on the factory floor to exchange information with each other.

HP Enterprise Link is available for HP 9000 computers running the HP-UX* operating system and PC platforms running Microsoft's Windows® NT operating system.

This article will discuss the evolution of the link between business software systems and factory automation systems, and the functionality provided in HP Enterprise Link to enable these two environments to communicate.

Kenn S. Jennyc

Kenn Jennyc is a software engineer at the HP Lake Stevens Division. He worked on the software design, development, and quality assurance for the HP Enterprise Link. Before that he worked on software design and development for the RTAP (real-time application platform) product. He received a BSEE degree from the University of Calgary in 1983 and came to HP in 1989. Kenn was born in Calgary, Alberta, Canada, is married, and has two children. In his spare time he likes to fly his home-built aircraft and dabble in analog electronics.

Background

Initially, only large corporations could afford computers. They ran batch-oriented enterprise business software to do payroll, scheduling, and inventory.

As the cost of computing dropped, smaller companies began using computers to run business software, and companies involved in manufacturing began using them to automate factory-floor production.

Although factory-floor automation led to improved efficiency and productivity, it was usually conducted on a piecemeal basis. Different portions of an assembly line were often automated at different times and often with different computer equipment, depending on the capabilities of computer equipment available at the time of purchase. As a result, today's factory-floor computers are usually isolated hosts, dedicated to automating selected steps in production. While various factory-floor functions are automated, they do not necessarily communicate with one another. They are isolated in "islands of automation." To make matters worse, the development of programmable logic controllers (PLCs) and other dedicated "smart" factory-floor devices has increased the number of isolated computers, making the goal of integrated factory-floor computation that much harder to achieve.

While production software was generally used for smaller, more isolated problems, business software was used to solve larger company-wide problems. Furthermore, while production software was more real-time oriented, business software was more transaction and batch oriented. These differing needs caused business systems to evolve with little concern for the kind of computing found on the factory floor. Similarly, production systems evolved with little concern for the kind of computing found at the enterprise level. As a result, many enterprise-level business systems and factory-floor computers are not able to intercommunicate. **Figure 1** shows an example of the components that make up a typical enterprise and factory-floor environment.

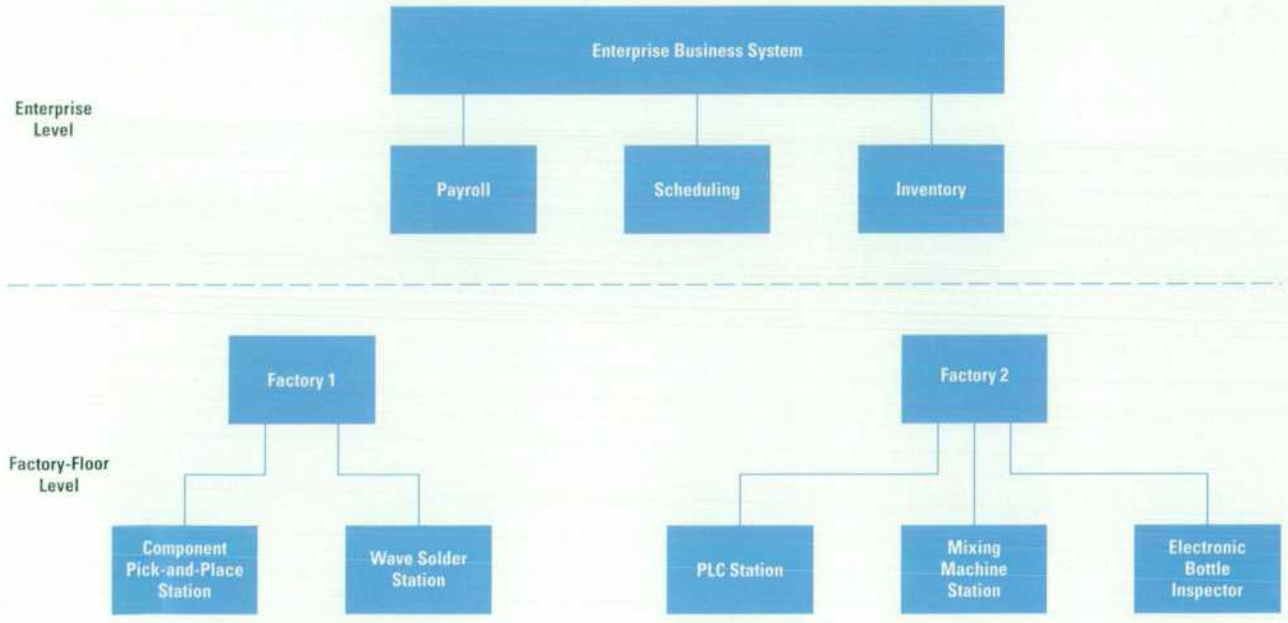
The net effect is that today companies find it difficult and expensive to integrate factory-floor systems with each other and with business software running at the enterprise level. This is unfortunate because the dynamic nature of the marketplace and the desire to reduce inventory levels have made the need for such integration very high.

Marketplace Dynamics

Over the last decade, the marketplace has become increasingly dynamic, forcing businesses to adapt ever more quickly to changing market conditions. Computer systems now experience a continuous stream of modifications and

Figure 1

Computing at the enterprise and factory-floor levels.



upgrades. Generally, this has forced business systems to adopt more real-time behaviors and production systems to become more flexible. It has also increased the frequency and volume of data transferred between business and production systems and between the many production systems.

There has always been a requirement to transfer information between computers in an organization, both horizontally between computers at the same functional level, and vertically between computers at different functional levels. In the past, manual data entry was an often-used approach. Hard-copy printouts generated by business management systems would be provided to operators who manually entered the information into one or more production systems. Although this was an acceptable approach in the past, such an approach is not sufficiently responsive in today's dynamic business environment. As a result, the need for electronic data transfer capability between the various business management and production level computers is now very high.

Electronic Data Transfers

Integrated business software with built-in support for data transfers between components is sometimes used at the business management level. While this minimizes the effort required to exchange data between the various components of enterprise business systems, it is often inflexible and restrictive with regard to what can be exchanged and when exchanges occur.

Organizations that use a variety of business software packages, rather than a single integrated package, have typically developed custom software for electronic data transfers between packages. Unfortunately, marketplace dynamics require custom software to be constantly reworked. This ongoing rework forces companies to either maintain in-house programming expertise or repeatedly hire software consultants to implement needed changes. As a result, custom data transfer software is not only expensive to develop but also costly to maintain—especially if changes must be implemented on short notice.

On the factory floor, software programmers have been employed to develop custom data transfer solutions that allow the different islands of automation to communicate. As previously noted, this approach is difficult to implement and expensive to maintain. In addition, this approach is often inflexible since the resulting software is usually developed assuming that the configuration of factory-floor systems is largely static.

When new equipment and application software are to be integrated into the overall system, software programmers don't just prepare additional custom software. They must also modify the existing custom software for all applications involved. For this reason, custom software is often avoided, and electronic data transfer capability is frequently confined to transfers between equipment and software supplied by the same manufacturer.

Differences in hardware (and associated operating systems) and differences in the software applications themselves cause numerous application integration problems. Here are a few examples:

- Data from applications running on computers that have proprietary hardware architectures and operating systems is often not usable on other systems.
- Different applications use different data types according to their specific needs.
- Incompatible data structures often result because of the different groupings of data elements by software applications. For example, an element with a common logical definition in two applications may still be stored with two different physical representations.
- Applications written in different languages sometimes interpret their data values differently. For example C and COBOL interpret binary numeric data values differently.

What is needed, therefore, is an off-the-shelf product that is specifically designed to interconnect applications that were not originally designed to work together. That product must automatically, quickly, efficiently, and cost-effectively integrate applications having incompatible programming interfaces at the same or different functional levels of an organization. HP Enterprise Link is such a product.

HP Enterprise Link is an interactive point-and-click software product that is used to connect software applications (such as business planning and execution systems) to control supervisory systems found on the factory floor. HP Enterprise Link greatly reduces the cost and effort required to interconnect such systems while eliminating the need for custom software.

The Data Transfer Problem

The problem of transferring data from one software application to another is conceptually simple: just fetch the data from one system and place it in another. In practice the problem is more complex. The following issues arise when trying to implement electronic data transfer solutions:

- There must be a way to obtain data from the software application serving as the data source. Such access, for example, might be provided by a library of callable C functions.
- There must be a way to forward data to the software application serving as the data destination. For example, data might be placed in messages that are sent to the destination application.
- There must be a specification of exactly what to fetch from the source application and exactly where to place it in the destination application.
- The data being transferred must be translated from the format provided by the data source to the format required by the data destination.
- There must be a specification of the circumstances under which data should be transferred and a way to detect when these circumstances occur.

All of these issues are addressed in HP Enterprise Link.

HP Enterprise Link

HP Enterprise Link product consists of the three components shown in **Figure 2**:

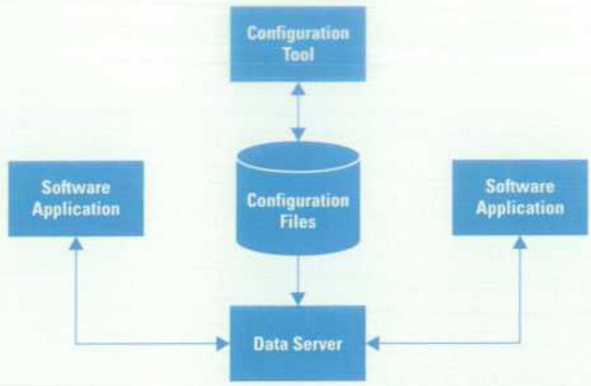
- An interactive configuration tool. This interactive window-based application allows users to direct the movement of data between two software applications.
- A data server. This noninteractive process runs in the background. It moves data in accordance with the directives that the user specified with the configuration tool.
- Configuration files. This is the set of mappings and trigger criteria created by users. The data is stored in configuration files. These files are created and modified by the configuration tool and read by the data server.

Linking Components

The HP Enterprise Link components listed above have the common goal of enabling users to create middleware that

Figure 2

The components of HP Enterprise Link.



maps components with different interfaces together for data transfer.

In HP Enterprise Link, the combination of a single source address and a single destination address is called a *mapping*. A unit of data at the specified source address is said to be mapped to the specified destination address. In other words, it can be read from the specified source address and written to the specified destination address.

Although a mapping deals with the transfer of a single unit of data, real-world situations usually require the transfer of many units of data simultaneously. Therefore, HP Enterprise Link collects mappings into groups called methods. A method contains one or more mappings.

Mappings describe what to transfer and where to transfer it, whereas triggers describe exactly when to do the transfer. Data is actually transferred whenever a specified trigger condition is satisfied. This condition is called the trigger criterion. There are many possible trigger criteria such as:

- Whenever a unit of data at a specified source address changes value
- Whenever a unit of data at a specified source address is set to a specified value
- Whenever the source data becomes available—such as arriving in a message
- At a preset time of the day or a preset day of the week.

HP Enterprise Link considers trigger criteria to be part of the definition of a method. All the mappings for a single method share the same trigger criteria. Whenever the trigger criteria are met, HP Enterprise Link transfers—in unison—all the data specified by the method's mappings.

Multiple methods can simultaneously exist in HP Enterprise Link. For example, a user can create one method to transfer a particular production recipe from a business enterprise system down to a factory-floor control system. Conversely, raw-material consumption information for the recipe currently in production could be transferred periodically from the factory-floor control system up to the business enterprise system, using a second method.

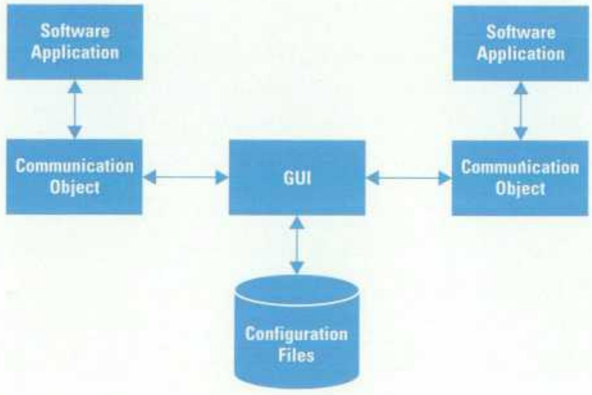
The Configuration Tool

The HP Enterprise Link configuration tool provides users with a view of each software application's name space, and the tool graphically depicts what data to transfer and under what circumstances such transfers should occur (**Figure 3**).

The HP Enterprise Link configuration tool is composed of communication objects and a graphical user interface (GUI). Communication objects are used to obtain namespace data that is unique to each application and to provide application-specific windows. The configuration tool provides the user with an easy-to-use point-and-click style GUI.

Figure 3

The HP Enterprise Link configuration tool.



All dependencies on particular software applications are encapsulated in communication objects. The configuration tool's communication objects provide the following functionality:

- They fetch namespace information from communicating software applications for presentation to the user.
- They provide routines to create and manage application dependent control panel widgets, such as those used to specify triggers unique to a particular software application.
- They provide routines to tell the GUI exactly what functionality is supported by a communication object. For example, can the application software serve only as a data source (supply data values), or can it serve as both a data source and a data destination (both supply and use data values)?

There are three important windows in the configuration tool's GUI: the Edit Method window, the Edit Mapping window, and the Trigger Configuration window.

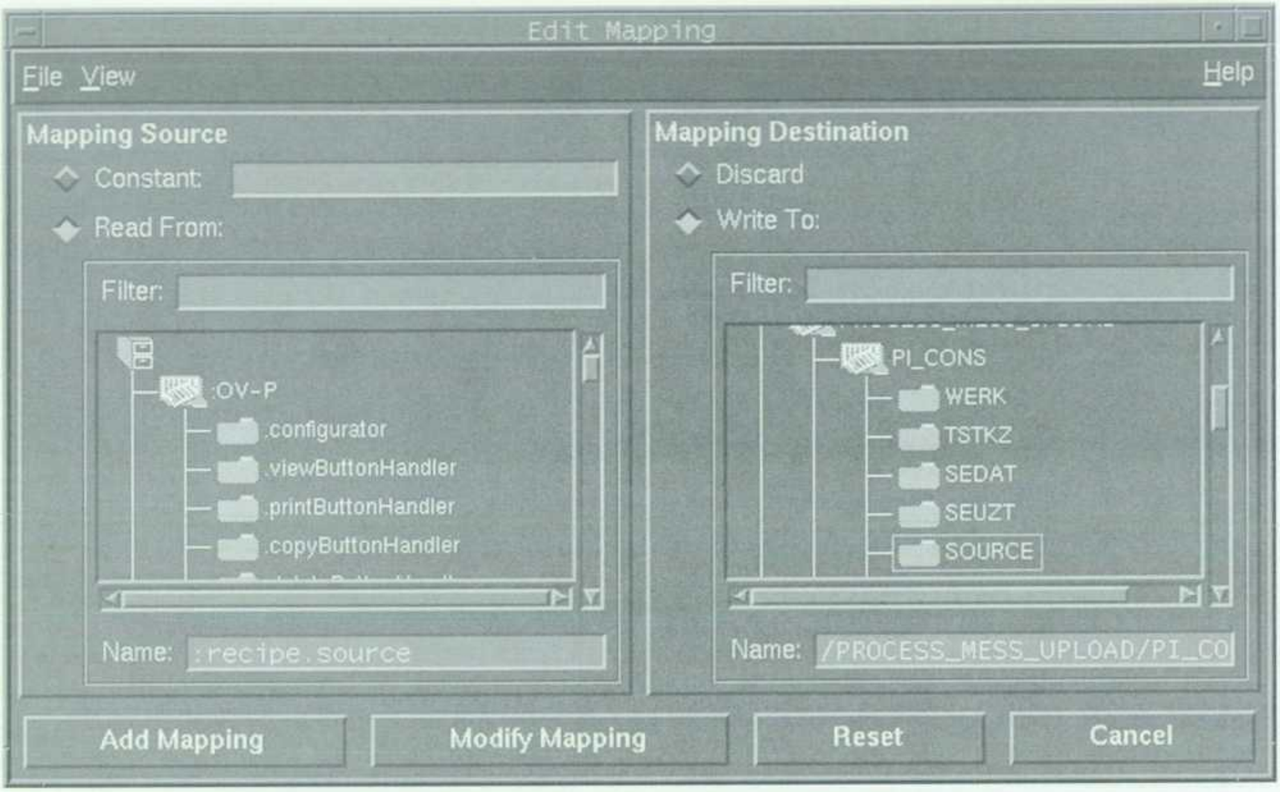
Edit Mapping. The Edit Mapping window is used to create new mappings (**Figure 4**). The namespaces of both the source software application and the destination software application are shown. They are graphically displayed as tree diagrams. This makes it easy for users to specify which data to move where. They don't have to remember the names of data sources or data destinations. Instead they just choose from the displayed list of possibilities. The side-by-side display of application namespaces makes it much easier to integrate the applications.

Tree diagrams are used because they make large namespaces manageable. A linear namespace display was rejected early in the design of HP Enterprise Link because a flat list representation would only be manageable with software applications having a small namespace. Another advantage of tree diagrams is that most users are already familiar with them from file selector windows found in many software applications.

To create a new mapping the user selects an item from the Mapping Source tree diagram and an item from the Mapping Destination tree diagram, and then clicks the Add Mapping button. A new mapping is added to the mapping table displayed on the Edit Method window (**Figure 5**).

Figure 4

The Edit Mapping window.



Multiple static mappings can be created in a single step using branch assignments. This requires that the last component of the source and destination addresses be identical (so that appropriate mappings can be automatically created). Mappings can also be automatically created at the time methods are triggered. This is called dynamic mapping and requires the user to specify algorithms that can select source addresses and transform these addresses to valid destination addresses.

Edit Method. The Edit Method window (**Figure 5**) displays a method's mappings as a two-column table titled Mappings. Source addresses appear in the left column and destination addresses appear in the right. The data server transfers mapped data from source addresses to destination addresses in the same order as the mappings are listed in this table. The Mappings table makes mappings both explicit and intuitive to the user.

This window allows the user to specify in which direction to transfer data. All of a method's mappings specify data transfers in one direction—from one software application to another. The Edit Method window also allows the user to specify how to respond to errors that occur during data transfers. This will be described later in more detail.

Trigger Configuration. The Trigger Configuration window is used to define trigger criteria (**Figure 6**). This window displays all possible triggers to the user, as well as the currently configured trigger criteria. The Trigger Configuration window is designed to make setting up trigger criteria explicit and intuitive for the user.

The Trigger Configuration window is split into three groups: time triggers, triggers unique to the source application, and triggers unique to the destination application. Time triggers allow the user to specify that data mapping start

Figure 5

The Edit Method window.

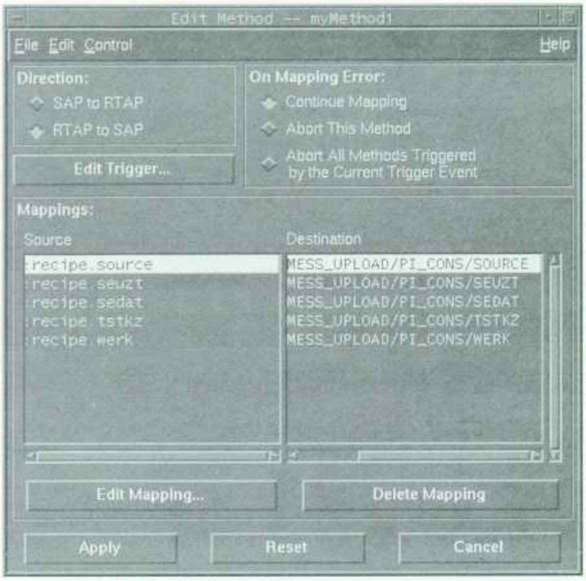
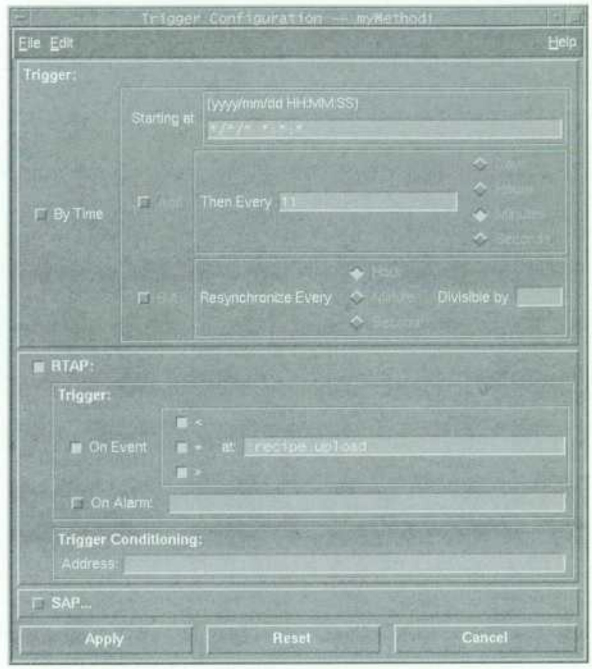


Figure 6

The Trigger Configuration window.



at some specified time and repeat at a specified time interval, but be synchronized to a specified hour/minute/second of the day/hour/minute.

Triggers unique to the source application, such as the RTAP (real-time application platform) triggers shown in **Figure 6**, allow data to be mapped when something interesting happens in the source application. For the RTAP triggers in **Figure 6** interesting events include a database value change or the occurrence of an RTAP database alarm. Data can also be mapped when something interesting happens in the destination application.

Thus, triggers allow data transfers to be pushed from the source application, pulled from the destination application, or scheduled by time.

Summary. Using the windows just described, users can create methods with the configuration tool. These methods specify one or more mappings and associated trigger criteria. This information is saved in one or more configuration files. The data server then reads these configuration files to implement the user's methods.

The Data Server

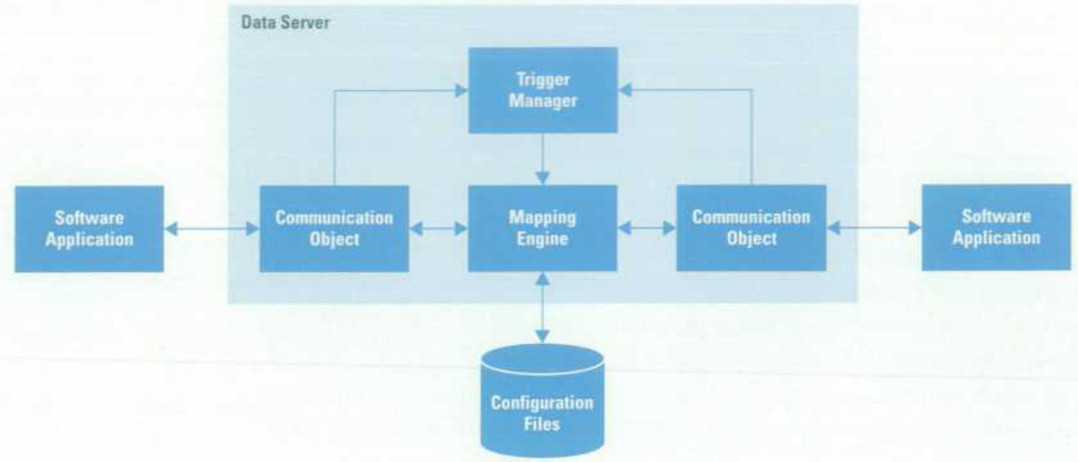
The HP Enterprise Link data server is composed of communication objects, a trigger manager, and a mapping engine (**Figure 7**). Communication objects deal with the problems of generating triggers and getting data into and out of software applications. The trigger manager deals with dispersing Trigger Configuration data, coordinating trigger events, and notifying the mapping engine of trigger events. The mapping engine deals with the problems of reading configuration files, responding to triggers, mapping source addresses to destination addresses, and transforming the data as it is being mapped.

All software-application dependencies are encapsulated in communication objects. Communication objects serve as translators between external software applications and the data server's mapping engine—they translate the software application's native application program interface (API) to the interface used by the mapping engine.

The interface between a communication object and the mapping engine is standardized, with all communication objects using the same interface. For data that is being transferred, the interface consists solely of address-value pairs, where the address is from the application software's namespace, and the value is encoded in a neutral form. Thus a communication object only needs to be

Figure 7

The components of the HP Enterprise Link data server.



aware of its own namespace and how to convert between the software application's proprietary data formats and the neutral HP Enterprise Link data format. For triggers, the interface consists of well-documented interactions between the trigger manager and the communication objects.

Communication objects are usually distributed. They are split into two parts that are interconnected by a communication channel such as a TCP/IP socket. One part of the object is incorporated into the HP Enterprise Link data server process, while the other runs on the same machine as the corresponding software application. When a communication object is not split into two parts, the object, the data server, and the software application must run on the same machine.

Communication objects communicate with their corresponding software applications through whatever mechanism is available. For example, this could be through a serial port, shared memory, shared files, TCP/IP sockets, or an application program interface (API).

When a communication object transfers data, it translates data between the format used by the source software application and the neutral format required by the mapping engine. For example, for numeric values, a communication object may have to translate between binary IEEE-754 floating-point format and the mapping engine's neutral format.

In practice, not all data transfer attempts will be successful. For example, a particular source address might have been deleted, or a destination address may no longer exist. The configuration tool is used to specify what the mapping engine should do in this situation, and the data server must detect the condition and deal with it appropriately. When data transfer attempts fail, the user can have the data server do any one of the following:

- Continue mapping data (ignoring the error)
- Abort all subsequent mappings associated with the current method
- Abort all subsequent mappings and all subsequent methods triggered by the current trigger event (if multiple methods were simultaneously triggered).

The interface between the communication object and the mapping engine is designed to support transaction-oriented data transfers, using commit and rollback. This functionality comes into play when mapping attempts fail. It allows the data server to undo (roll back) all data transfers done in all currently processed mappings associated with the method's current trigger event.

The Running Data Server

When the HP Enterprise Link data server starts up, it reads the configuration files that the user created with the configuration tool. It then prepares to deal with the specified trigger criteria, usually by notifying the appropriate communication object to detect it. Finally, it enters an event-driven mode, waiting for the trigger criteria of any configured method to be satisfied.

When either a source or destination communication object in the data server detects that a method's trigger criteria have been satisfied, the object informs the data server trigger manager that a method has been triggered. This starts the mapping engine. Alternatively, if the data server trigger manager detects that a method's time-based trigger criteria have been satisfied, the mapping engine starts.

When triggered, the mapping engine requests that the source communication object provide the current data values at the method's configured source addresses. The source communication object obtains these values from the software application, translates the format of all fetched data values to a neutral format, and passes the result to the mapping engine as address-value pairs, with one such pair for each of the method's defined mappings.

The data server mapping engine looks up the destination address that corresponds to each source address. This lookup results in a new list of address-value pairs, with the address now being the destination address, and the value unchanged (and still expressed in the mapping engine's neutral format). To minimize the impact on performance, this lookup is implemented using a hash table.

The mapping engine sends the new list of address-value pairs to the destination communication object. The destination communication object converts the received values into the format required by the destination software application, and writes the converted result to the specified addresses in the destination software application.

Communication Objects and Software Applications

There are two fundamental ways for software applications to provide communication objects access to their data: the *request-reply* method and the *spontaneous-message* method.

In the request-reply method, the communication object sends a software application the address of a wanted data unit in a request and receives its current value in a reply. With this method the communication object controls the data transfer. It determines which unit of data to read and when to read it. Structured Query Language (SQL) and real-time databases are two examples of software applications that employ the request-reply method.

In the spontaneous-message method, communication objects receive data, usually as messages, from the software application whenever the application chooses to send it. With this method the software application controls the data transfer. It determines which data to provide and when to provide it. SAP's R/3 product is an example of a software application using the spontaneous-message method.

The method that a software application employs to provide external data access determines the trigger criteria that are possible for that application's communication object. The request-reply method allows event, value, and time-based trigger criteria since the communication object controls the data transfer. The spontaneous message method is limited to value-based triggering (essentially filtering) because the software application providing the data controls the data transfer.

Spooling

The HP Enterprise Link data server's communication objects must cope with communication failures. This means that outgoing data must be locally buffered until a communication object verifies that the application software, when acting as a destination, has successfully received it. It also means that incoming data must either be safely transferred through the mapping engine or locally buffered when a communication object accepts data from the source application software.

Spooling is especially important if the source application is separated from the HP Enterprise Link data server by a wide area network (WAN). WANs are considerably less reliable than local area networks, and thus are more likely to lose data.

In a typical HP Enterprise Link installation the data server runs on a machine located near or on the factory floor. Production orders are downloaded from the enterprise level to HP Enterprise Link as soon as they are available. The downloaded data is buffered at the factory until it is needed. Using HP Enterprise Link in this way reduces the probability that the factory would lack unprocessed production orders if the WAN is down for a prolonged period of time.

Buffered data must be preserved even if the HP Enterprise Link host machine is shut down or crashes. To do this, HP Enterprise Link stores buffered data in disk-resident spool files.

The amount of storage used to hold buffered data must be restricted to protect the host computer from failure caused by insufficient resources. HP Enterprise Link can limit the size of spool files by controlling:

- The maximum size of spool storage
- The maximum number of messages buffered
- The age of the oldest message buffered.

The user can set any one or all of these limits, using the HP Enterprise Link configuration tool.

Tracing

HP Enterprise Link allows the data being transferred to be monitored by the user. The monitoring is called *tracing*. Tracing is useful for creating an audit trail of the transferred data and for debugging and testing methods. Tracing does not affect the data being transferred.

The configuration tool is used to enable and disable tracing, but it is the data server that generates trace messages when tracing is enabled.

Data can be traced at a number of different internal locations within the data server (see **Figure 8**). Some of the forms in which trace results can be expressed include:

- Data as received by a data server communication object from a source software application. This trace data is expressed using the source software application's native data format and includes the source address, the value received or read, and the time of transfer.
- Data as sent by a data server communication object to the destination software application. This trace data is expressed using the destination software application's native data format and includes the destination address, the value sent or written, and the time of transfer.
- Data being mapped by the mapping engine. This trace data is expressed using the data server mapping engine's neutral data format and includes the source address, the destination address, the value transferred, and the time of transfer.

Error messages reported by the mapping engine or by communication objects can also be included in the trace output. This ability ensures that the relative sequencing of data transfer messages and error messages is preserved, which greatly aids the user when trying to troubleshoot mapping problems.

Server Data Flow

HP Enterprise Link allows the flow of data in the data server to be interrupted at a number of different internal points (see **Figure 9**). This is useful for isolating the effects of data mappings during debugging and testing. When an information flow is interrupted, data does not pass the point of interruption; instead, the data is discarded.

The flow of information being transferred from a communication object to a software application can be interrupted. Interrupting the flow here allows the data server

Figure 8

Tracing data that is transferred between applications.

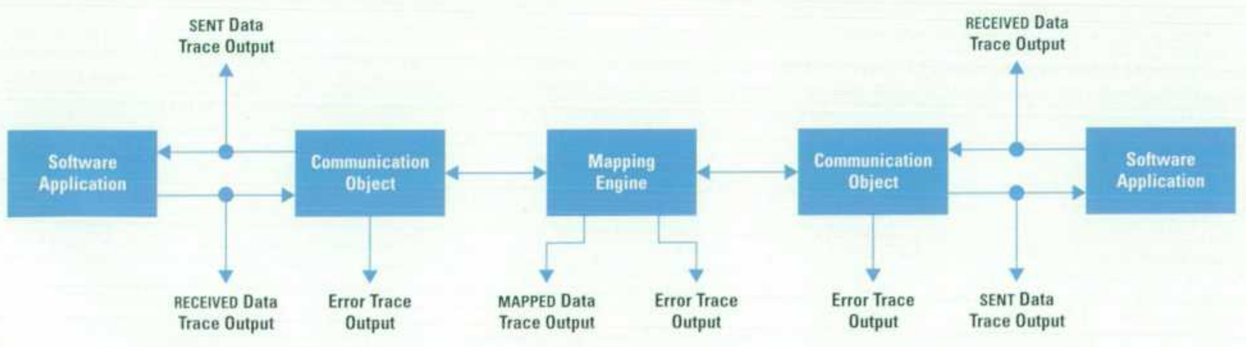


Figure 9

Interrupt locations in the data server.

to read from mapped source addresses, map to new destination addresses, and then discard the data just before it would have been written to the destination software application.

The flow of information being transferred from a software application to a communication object can also be independently interrupted. Interrupting the flow here allows the data server to ignore all data sent to the communication object by the source software application.

Data Integrity

The HP Enterprise Link data server is carefully designed to preserve the integrity of the data being mapped and to map the data exactly once for each trigger event. The design was influenced by considering how to react to communication channel failures and data server process terminations. The circumstances that could cause the data server process to terminate are the following:

- If a person or software process explicitly kills the data server process
- If the host machine suffers a hardware or software failure, loses power, or is manually turned off.

Communication channel failures must be handled carefully. If the communication channel connecting a communication object to its software application fails, the data being mapped at the time of failure must not be lost or duplicated. Also, after normal operation of the communication channel is restored, communication between the communication object and its application must be automatically established again and all interrupted data transfers restarted.

The following steps are taken to ensure data integrity when communication channels fail:

- For data received from the source software application, the communication object never acknowledges receipt of the data until the data has safely been saved to a disk-resident receive-spool file.
- Data received by the communication object from the source software application is not removed from the receive-spool file until the data has successfully passed through the mapping engine and been forwarded to the communication object responsible for sending it to the destination software application.
- The communication object that sends data to the destination software application only notifies the mapping engine that it successfully received the data after the data has been safely saved to a disk-resident transmit-spool file. Also, it only removes data from the transmit-spool file when the destination software application has acknowledged successful receipt of the data.

Conclusion

The HP Enterprise Link product greatly reduces the cost and effort required to interconnect business management systems (such as SAP's R/3 product) and measurement and control systems (such as Hewlett-Packard's RTAP/Plus product). HP Enterprise Link is an off-the-shelf product that allows users to connect software applications using an easy-to-use point and click graphical user interface.

With HP Enterprise Link, companies can minimize the costs associated with changes made to computer systems and adapt more quickly to changing market conditions.

Acknowledgments

The author wishes to thank Andrew Ginter and Andy Mah for their significant contributions to the design and development of the HP Enterprise Link product, John Burnell for his comments during the design of the product, and Steve Heckbert for his valuable feedback.

Online Information

For more information about HP Enterprise Link, take a look at the information located at the following URLs:

- http://www.tmo.hp.com/tmo/pia/Vantera/Index/English/Index.html
- http://www.tmo.hp.com/tmo/pia/Vantera/Index/English/Products.html
- http://www.tmo.hp.com/tmo/pia/Vantera/Index/English/ELink.html

HP-UX 9. and 10.0 for HP 9000 Series 700 and 800 computers are X/Open Company UNIX 93 branded products.*

UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited.

X/Open is a registered trademark and the X device is a trademark of X/Open Company Limited in the UK and other countries.

Microsoft is a U.S. registered trademark of Microsoft Corporation.

Windows is a U.S. registered trademark of Microsoft Corporation.

Knowledge Harvesting, Articulation, and Delivery

Kemal A. Delic

Dominique Lahaix

Harnessing expert knowledge and automating this knowledge to help solve problems have been the goals of researchers and software practitioners since the early days of artificial intelligence. A tool is described that offers a semiautomated way for software support personnel to use the vast knowledge and experience of experts to provide support to customers.

A consequence of the global shift toward networked desktops is visible in customer technical support centers. Support personnel are overwhelmed with telephone calls from customers who are experiencing a steady increase in the number of problems with intricate software products on various platforms.

Support centers are staffed with less knowledgeable (and less experienced) first-line agents answering the simple questions and solving common problems. Expert (and more expensive) technicians resolve more complex problems and execute troubleshooting procedures. The work of both (the first-line agents and the technicians) is supported by various technical tools, but they always have to use their brains and experience to handle effectively the stream of problems they encounter. This knowledge is seen as the key ingredient for the efficient functioning of support centers.[1]

Kemal A. Delic

Kemal Delic is a technical consultant at HP's Software Services Division in Grenoble, France. He is responsible for knowledge technologies. He received a Dipl.El.Ing. degree from the Faculty of Electrical Engineering at the University of Sarajevo in Bosnia. Before joining HP in 1996, he was a senior scientific consultant with CPR Consortium in Pisa, Italy. He is married and has two children. In his free time he enjoys reading medieval history.

Dominique Lahaix

Dominique Lahaix is the knowledge and electronic services manager at HP's Software Services Division. He came to HP at the Grenoble Division in 1988. He received an engineering degree in computer science in 1985 from the Institut National des Sciences Appliquées de Lyon. Born in Burgundy, he is married and has three children. In his spare time he enjoys playing the saxophone, reading philosophy, and outdoor activities such as skiing, soccer, and running.

The number of calls and their complexity have both increased. At the same time, support solution efficiency has decreased as the cost for providing those solutions has increased. As a result, there is a need for a knowledge sharing solution in which the first-line agents will be able to solve the majority of problems and escalate to the technicians only the complex problems. To enable such a solution, we have to:

- Find efficient knowledge extraction methods
- Create compact, efficient knowledge representation models
- Use extracted knowledge directly in the customer support operations.

This article describes the HP approach to providing customer support in the Windows®-Intel business segment. This segment includes networked desktop environments known for their high total cost of ownership. Help-desk services for this segment are supposed to solve the majority of problems with software applications, local area networks, and interconnections.

The system described here, called WiseWare,* is a knowledge harvesting and delivery system specifically designed to provide partially automated help for HP customer support centers in their problem solving chores.

Partial automation of help-desk support is seen as a suitable, cost-effective solution that will:

- Shorten the time spent per call
- Decrease the number of incoming calls (because of proactive mechanisms)
- Decrease the number of calls forwarded to the next support level
- Decrease the overall labor costs.

The objective is to reduce dramatically the support costs per seat per year.

Where Is Knowledge?

To find the most efficient knowledge extraction methods, we must first answer the question, "Where is the knowledge?" Books, technical articles, journals, technical notes, reports, and product documentation are all classical resources that rely on a human being's ability to extract, evaluate, and apply knowledge. Mechanized efforts still can't replace these human attributes.

Current support solutions usually are based on electronic collections in a free-text format, in which the important concepts are expressed using natural human language. The latest release of WiseWare uses technical notes, frequently asked questions, help files, call log extracts, and user submissions as the primary raw material. According to the knowledge resource, different knowledge representations and extraction methods are used.

Extensive research in the field of artificial intelligence has created several knowledge representation and extraction paradigms in which the final use for knowledge determines the characteristics of the representation scheme. The earliest knowledge extraction efforts, known as *information retrieval*, initially had small industrial impact. However, recent interest in the Internet and in electronic book collections has revived the business interest in information retrieval. Some of the hottest products on the market today are search engines. Different search methods (by keywords or by concepts) are being used and other search methods (by examples and by natural language phrases) are being investigated. Recent synergy with artificial intelligence methods has created a promising subfield known as intelligent information retrieval.[2] The majority of today's customer support solutions can be classified as enriched information retrieval systems.

Electronic Document Libraries

Developments in the information retrieval field have transformed free-text collections into more refined collections known as electronic document libraries. Electronic document libraries have an articulated structure (author, subject, abstract, and keywords), enabling efficient searches and classification. They combine advanced technological methods (such as hypertext and multimedia) to fit users' information retrieval needs. Some of the best support solutions today are in a digital library class and represent sophisticated document management systems.

Case-Based Retrieval

Early hardware support documentation contained troubleshooting diagrams that made it possible for service technicians to troubleshoot equipment consistently by following the diagrams and performing the appropriate tests and measurements. The recent revival of these diagrams is

* WiseWare is an internal tool and not an HP product.

Glossary

Cluster. Natural association of similar concepts, words, and things.

Concept. Group of words conveying semantic content. It can be described graphically as relationships between words having different attributes (and in some cases as numerical measurements of strength).

Data Mining. Collective name for the field of research dealing with data analysis in large data depositories. It includes statistics, machine learning, clustering, classification, visualization, inductive learning, rule discovery, neural networks, Bayesian statistics, and Bayesian belief networks.

Information Retrieval. Identification of documents or information from the collection that is relevant for the particular information need.

Keyword. Characteristic word that may enable efficient retrieval of relevant documents. Two criteria used to assess the value of a keyword are the number of documents retrieved and the number of useful documents (recall and precision)

Knowledge. Group of interrelated concepts used to describe a certain domain of interest. Complex structures formed by emulating human behavior in certain activities (for example, assessment, problem solving, diagnosing, reasoning, and inducing). Different schemes are used to enable knowledge representation such as rules, conceptual graphs, probability networks, and decision trees. Knowledge is found in large text collections and is biologically resident in human brains.

Knowledge Map. Graphical display of interrelated concepts.

Knowledge Base. Complex entity typically containing a database, application programs, search and retrieval engines, multimedia tools, expert system knowledge, question and answer systems, decision trees, case databases, probability models, causal models, and other resources.

Metrics. Group of measurement methods and techniques introduced to enable quantification of processes, tools, and products

Natural Language Processing. Activity related to concept extraction from, formalization of, and methods deployment in a problem area.

Paradigm. A theoretical framework of a discipline within which theories, generalizations, and supporting experiments are formulated.

Problem Domain. Area of interest defined by terminology, concepts, and related knowledge.

Search. Activity guided by a find and match cycle in which a search space is usually explored with an appropriate choice of search words (keywords). Advanced search is done by concepts.

seen in interactive troubleshooting systems that enable PC hardware technicians to solve hardware problems. So far, such systems are implemented as *case-based* retrieval (or reasoning) systems. The majority of these systems provide only retrieval; just a few include the reasoning component. The case-based retrieval paradigm is based on the human ability to solve problems by remembering previously solved problems. The support system plays the role of an electronic case database in which the knowledge consists of documented experience (cases). Creation and maintenance of the cases is an expensive and nontrivial process. Currently, these activities are performed by humans and are used mainly for hardware support. Such systems cannot deal efficiently with large, complex, and dynamic problem areas.

Rule-Based Systems

Some support centers have tried to use expert systems based on rules, but they have discovered that the rule-based systems are difficult to create, maintain, and keep consistent. Crafting a collection of rules is a complex chore. It is not clear if this technology will have a role in future knowledge representation and extraction development.

Model-Based Systems

A model-based paradigm in which various statistical, causal, probability, and behavioral models are used is another example of knowledge representation for customer support. The knowledge here is expressed by the fault/failure model that contains quantified relationships between causes, symptoms, and consequences. Basic decision making is enabled with such models. Although some limited experiments with this highly sophisticated knowledge representation paradigm have been done, no system is in operational use in support centers.

New Research

The newest research in the field of data mining and knowledge discovery[3] may offer some potentially effective knowledge representation methods for deployment in customer support centers. This research aims at the extraction of previously unknown patterns (insights) from the existing data repositories. Research in artificial intelligence has identified the initial assembly of a low-cost knowledge base as a potential "engineering bottleneck." The knowledge authoring environment discussion later in this article addresses that issue. Because most of the knowledge for WiseWare comes from text sources, we will focus our attention here on the knowledge extraction process.

WiseWare and Knowledge Refinement

Knowledge is a fluid, hard-to-define but essential ingredient for all human intellectual activities. It is difficult to extract, articulate, and deploy. The prevailing quantity of knowledge is encoded in the form of text (90 percent) expressed in natural language and is articulated as a web of interrelated concepts. A goal of research in natural language is to enable automatic and semiautomatic extraction of knowledge. Content analysis must be automated to efficiently provide suggestions and solutions for users. As we have already seen, several knowledge representation paradigms are being invented and investigated (for example, semantic nets, rules, cases, and decision trees). Additionally, we can deploy various techniques to extract concepts (symbolic knowledge) and numerical quantities (numerical and statistical knowledge).

Refinement Process

Human experts use spreadsheets, outline processors, and some vendor-specific tools to refine source text, but have not yet developed systematic, efficient processing methods. In the future, we would like to automate some phases of this process, leading toward more efficient and effective deployment.

Knowledge refinement is seen as a process for converting raw text into coherent, compact, and effective knowledge forms suitable for software problem solving and assistance (for example, decision trees, rules, probability models, and semantic nets). The basic raw material (the knowledge in its primary form) remains accessible. This preserves previous investments in knowledge and enables integration into future, more sophisticated solutions.

We can describe the knowledge refinement process as efforts made to transform raw text to a compact representation and then to operational knowledge. Associated costs increase as raw text moves through the refinement process to become operational knowledge.

Currently WiseWare content is partitioned into three conceptual categories: fixes, step notes, and technical notes. The first two contain shallow, specific knowledge and the third contains complex technical concepts. A fix is a simple, short document that describes with fewer than 100 words a known and recurring problem with a known solution, the fix (see **Figure 1a**). A fix often helps the customer out of the immediate problem but does not provide a long-term solution. It is essentially a "quick fix."

A step note usually walks the user through a procedure that prevents the problem from occurring in the future (see **Figure 1b**). The step note requires more of the user's time to solve the immediate problem than the fix does, but it saves time in the future.

Both fixes and step notes offer additional references. Those references contain keywords providing links to technical notes that explain the most relevant related subjects in depth. Technical notes require deep technical knowledge to be properly understood and applied.

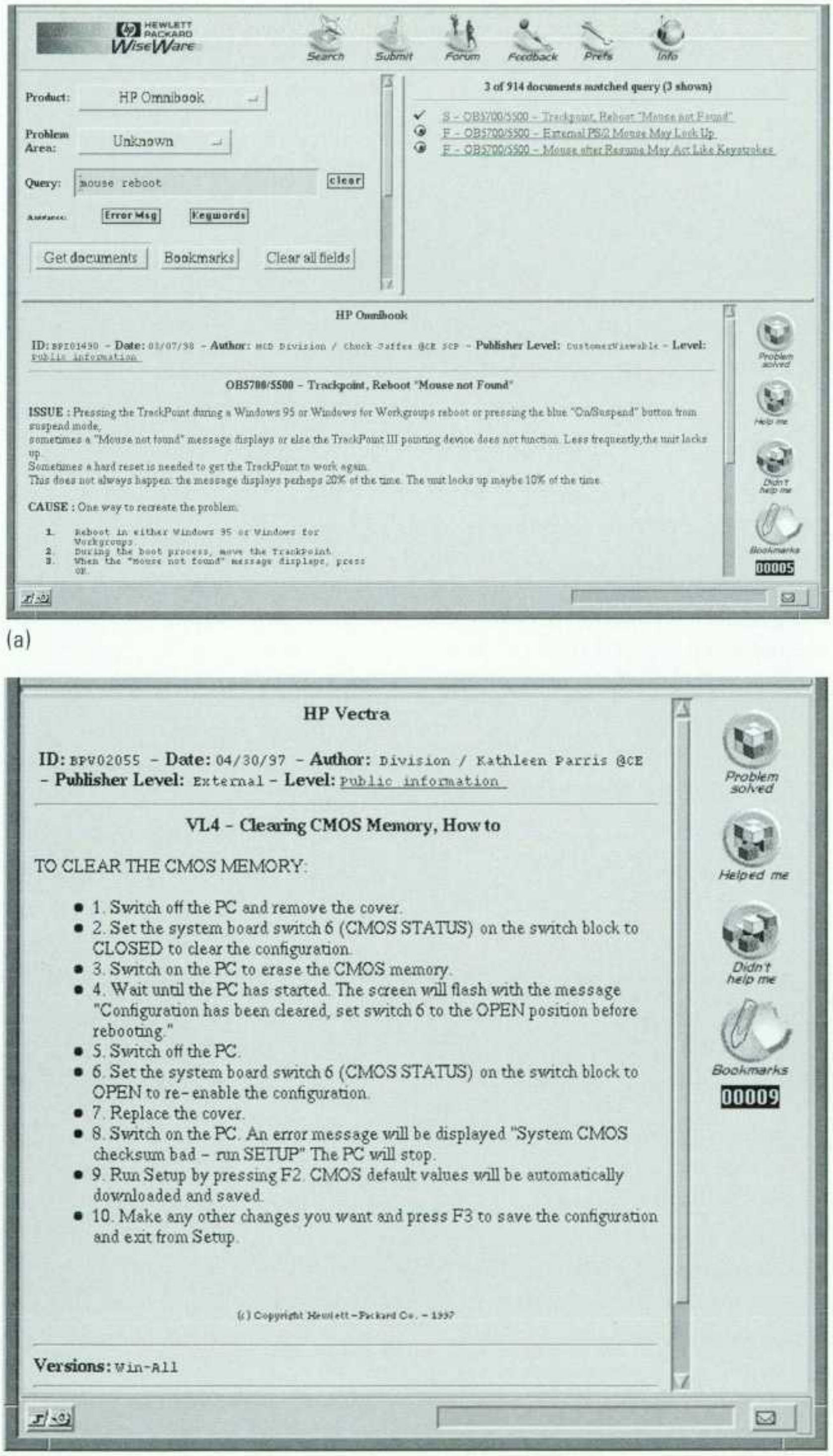
The whole collection of fixes, step notes, and technical notes is tagged to associate the content of each document with the proper problem classes. Consequently, WiseWare content is perceived by the user as a repository of advice and solutions for given problems (quick fixes, step-by-step procedures, and technical theory).

Some generic activities in the refinement process can be denoted as:

- Assessment
- Extraction
- Filtering
- Summarization
- Clustering
- Classification.

Figure 1

Two WiseWare screens: (a) WiseWare fix screen, (b) WiseWare step note screen.



(a)

(b)

We can describe the evolution of WiseWare as going from answering questions to giving advice and finally to problem solving and troubleshooting. The support costs in this evolution have escalated as the problems have become more complex.

Knowledge Authoring Environment

Since a critical mass of knowledge can be reached only if multiple authors contribute to the knowledge base, the knowledge authoring environment must be able to deal with multiauthor issues effectively. Additionally, because the knowledge authoring environment is deployed on a worldwide basis, the issue of different languages is relevant as well. Finally, deployment in different time zones requires very high reliability and availability of the knowledge authoring environment.

The quality of the knowledge is constantly monitored and refined. Areas for improvement are pinpointed by analyzing results reported on the knowledge base logs. As weak points are identified and strengthened, better system performance will help to optimize return on investment figures. Even user satisfaction can be assessed from the various logs and usage traces that will reflect a combined measure of system quality and usefulness.

Future worldwide cooperation among support centers to share knowledge is our objective. Ideally, each center will deploy and create the necessary knowledge locally. Centers operate in different time zones, have different cultural and social contexts, and have the ability to manipulate huge amounts of data, information, and knowledge. Coordinating the knowledge bases for all support centers pose several challenging problems. The complexity of these problems is reduced by careful engineering and incremental deployment. The result is a low-cost, knowledge-based support, adding new value to the support business.

In a very advanced situation, and from a long-term perspective, extracted knowledge will become the crucial ingredient for the next development phase. In this phase, human mediation in problem solving could be removed. Support could be delivered electronically without human intervention. For example, imagine intelligent agents traveling over the network to the troubled system to fix a problem.[4] Current viruses on the Internet are doing exactly the opposite task. What if the trend were reversed? Support knowledge could be adapted so that healing viruses could travel through a system, delivering remote fixes. To understand how this could become a reality, let's review the history of WiseWare.

WiseWare Architecture

In November of 1995, the first challenge was posed to the WiseWare team when the French call center decided to outsource low-end software support services. Their support personnel were without computer technology background and demonstrated poor English language skills. The knowledge department in HP's Software Services Division in Europe responded to the challenge and delivered the first operational WiseWare solution in April of 1996. Since then, new releases are issued every two months with steady improvements.

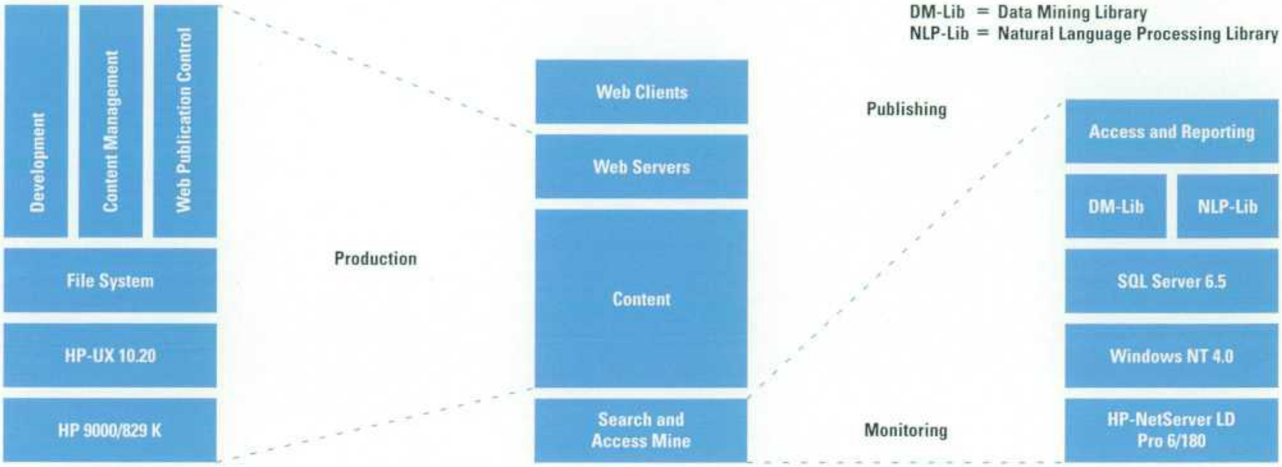
In the WiseWare release 4.1, mirroring intranet servers (Europe and the United States) cover three super regions. The number and quality of accessible documents is constantly improved, while use of the system is closely monitored from access and search logs. We have established close links with software vendors who allow us privileged access to their documents. (The legal framework for cooperation and alliances is defined as well.) All activities and services undergo quality assurance scrutiny in preparation for ISO-9000 certification.

WiseWare provides approximately 80,000 documents to 13 call centers worldwide. The average problem resolution assistance rate is over 30 percent. More than 40 products are covered in the various types of documents offering quick fixes for agents and in-depth technical knowledge for advanced WiseWare users.

WiseWare is a distributed system with three major parts: production, publishing, and monitoring (see **Figure 2**). They are implemented on UNIX® and Windows NT platforms, with intranet technology providing the necessary glue for client/server solutions. It is a nonstop, highly available system. The key advantage of the WiseWare system lies in the tight loop between the monitoring and production areas in which the principal objective is to provide users with highly adaptable documents for everyday problem-solving chores. Data mining and natural language processing modules dynamically create user, problem, and document profiles that will then drive the

Figure 2

WiseWare system architecture.



production side, enabling technical and business insights to be gleaned from large and extensive access and search logs.

At this time, customers call the express hubs and explain their problems to support personnel, using natural language constructs that sometimes blur the real nature of the problem. According to their understanding, support personnel create and launch a search phrase. It is a Boolean construct containing relevant keywords or free-text phrases that roughly represent the problem. Different search, hit, and presentation strategies are currently used, but formation of the effective search query and reduction of the number of relevant replies are largely still unresolved. A mixture of artificial intelligence techniques and traditional information retrieval and database methods is being offered as potential solutions.

Table I shows how one, two, and three words in a typical search phrase can influence the number of relevant documents returned with current version of WiseWare. A well-formed phrase helps to quickly pinpoint relevant documents while retaining necessary coverage of the problem area. Notice the quick decrease in the number of relevant documents returned as the phrase becomes longer.

Support center personnel work under time-pressured, stressful circumstances. As a result, the whole human-computer interaction issue must be carefully considered. Efficiently delivering advice and problem-solving assistance can depend on the smallest detail. Besides the quality of the material in the supporting knowledge base, questions regarding query formulation and presentation of the retrieved information will influence final acceptance from the users. Support activities can be treated as symbiotic human-machine problem solving in a bidirectional learning paradigm. The user learns how to manipulate the system (facilitated by language features such as localization and query wizards). At the same time, the system adapts to the user's methods of accessing the knowledge base. The WiseWare system learns user behavior from access and language patterns. Interaction with the system customizes the environment to suit the specific user's profile. The reasoning activity is still done by humans and is supported by refined electronic collections. Good synergy and efficient functioning of such human-computer systems are the current objectives.

Because the support centers are located in different geographical, cultural, and language areas, the natural language layer is seen as crucial for search and presentation.

Table I

Search phrases and number of documents returned

Application/Documents	Search Phrases/Documents Returned: Change	Change+User	Change+User+ Password	Create	Create+New	Create+New+ Message
WinNT/6046	1371	9	3	–	–	–
Win3.x/4397	1224	1	0	–	–	–
CC:Mail/2600	–	–	–	727	12	2
MS:Mail/3963	–	–	–	878	21	2

Technological advances in visual search and delivery combined with audio and video techniques may improve the quality and efficiency of the system. Better architecture combined with object-oriented (multimedia) databases will add another dimension to the delivery phase. These improvements will be made over time and will be accelerated by technological developments in related fields.

Conclusion

Accessible knowledge is the essential ingredient for successfully dealing with the rising quantity and complexity of customer support calls. A semiautomated system with refined knowledge in reusable forms can enable users to share knowledge among different, geographically dispersed customer support centers. The overall objective of HP's WiseWare server is to provide low-cost, effective customer support. This is a simple objective but one that is difficult to achieve, especially when significant effort and investment are required to achieve technological breakthroughs in the problem-solving field.[5]

In the short term, incremental deployment of advanced methods such as data mining and natural language processing techniques will improve system quality and usage. In the long run, it is very likely that most of the client-hub telephone voice communication will be gradually replaced by computer-computer communication. Several layers of the present problem- solving architecture will disappear or will be replaced by some new elements. The problem-solving knowledge along with search and access log collections being developed now will serve as the fundamental basis for future electronic support.

Acknowledgments

We would like to thank Markus Baer, Markus Brandes, and Jean-Claude Foray from HP's support labs for their role in WiseWare development. Its development would not have been possible without the understanding and support of Jim Schrempp and Alain Moreau. Finally, special thanks to all WiseWare team members.

References

1. http://www.HelpDeskInst.com/
2. http://ciir.cs.umass.edu/
3. http://www.kdnuggets.com/
4. http://retriever.cs.umbc.edu/agents/
5. http://www.gartner.com/

A Theoretical Derivation of Relationships between Forecast Errors

Jerry Z. Shan

This paper studies errors in forecasting the demand for a component used by several products. Because data for the component demand (both actual demand and forecast demand) at the aggregate product level is easier to obtain than at the individual product level, the study focuses on the theoretical relationships between forecast errors at these two levels.

With a sound theoretical foundation for understanding forecast errors, a much more confident job can be done in forecasting and in related planning work, even under uncertain business conditions.[1]

In a typical material planning process, planners are constantly challenged by forecast inaccuracies or errors. For example, should a component forecast error be measured for each platform for which it may be needed, or should its forecast accuracy be measured at the aggregate level, across platforms? What is the relation between the two accuracy measures?

This paper describes a theoretical study of forecast errors. First, we formally define forecast errors with different rationales, derive several relationships among them, and prove a heuristic formula proposed by Mark Sower.[1] Then we study the effects of a systematic bias on the forecast errors. Finally, we extend our study to the situations where correlations across product demands and time effects in demand and forecast are taken into account. Definitions and theorems are presented first, and proofs of the theorems are given at the end of the paper.

Jerry Z. Shan

Jerry Shan is a software engineer at HP Laboratories, responsible for statistical analysis and experimental design for applications of HP Laboratories' Enterprise Modeling and Simulation system. He received his PhD degree in statistics from Stanford University and joined HP in 1994. Born in China's Jiangsu province, he taught statistics at China Textile University before coming to the Unites States. He is married, has two children, and enjoys soccer, photography, and swimming.

Basic Concepts

Consider the case of a component that can be used for the manufacture of n different products, or *platforms*. For platform i ($1 \leq i \leq n$), denote by F_i the forecast demand for the component, and by D_i the actual demand. In the treatment of forecast and actual, we propose in this paper the following framework: *Regard forecast demand as deterministic, or predetermined, and actual demand as stochastic.* By stochastic, we mean that given the same operating environment or experimental conditions, the actual demand can be different from one operation run to another. Thus, we can postulate a probability distribution for it.

For a generic case, denote by D the actual demand and by F the forecast. We call the forecast *unbiased* if $E(D) = F$, where $E(D)$ denotes the expectation, or expected value, of D with respect to its probability distribution. Practically speaking, this unbiased requirement means that over many runs under the same operating conditions, the average of the realized demand is the same as the forecast. If there is a deterministic quantity $b \neq 0$ such that $E(D) = F + b$, then we say the forecast is *biased*, and the bias is b. In practice, this means that there is a systematic departure of the average realized demand from the forecast.

Throughout the paper, we often make the normality assumption on the demand, that is, for unbiased forecasts, we assume that the demand D has a normal (Gaussian) distribution with mean F and standard deviation σ, that is, $D \sim N(F, \sigma^2)$. Is this a reasonable assumption in reality? The answer is yes. First of all, this assumption is technically equivalent to assuming that the difference $\varepsilon = D - F$ between the actual demand D and the forecast F is normally distributed: $\varepsilon \sim N(0, \sigma^2)$. The validity of this latter assumption is based on the fact that the difference between the actual demand and a good forecast is some aggregation of many small random errors, and on the central limit theorem, which states that the aggregation of many small random errors has a limiting normal distribution.

Unbiased Forecast Case

In this section, we assume unbiased forecasting at all platforms.[1] Statistically, $E(D_i) = F_i$, where F_i is the forecast for the common component at platform i, and D_i is the actual demand of the component at platform i.

Definition 1: (Same Weight Mean Based) Define $E_\pi = E(\varepsilon_\pi)$ to be the forecast error at the mean (average) platform level, and $E_a = E(\varepsilon_a)$ to be the forecast error at the aggregate platform level, where:

$$\varepsilon_\pi = \frac{1}{n}\sum_{i=1}^{n}\frac{|D_i - F_i|}{F_i}, \qquad (1a)$$

and

$$\varepsilon_a = \frac{\left|\sum_{i=1}^{n} D_i - \sum_{i=1}^{n} F_i\right|}{\sum_{i=1}^{n} F_i}. \qquad (1b)$$

The rationale of defining the forecast error at the mean level and at the aggregate level is as follows. Let $\varepsilon_i = |D_i - F_i|/F_i$. Then ε_i measures, in terms of the relative difference, the forecast error at a single platform i. Accordingly, ε_a measures the forecast error, also in terms of the relative difference, at the aggregate level from all platforms, and ε_π provides an estimate for the forecast error at any individual platform since it is the average of the forecast errors over all individual platforms. Because all the quantities in equation 1 are stochastic, we take expectations to get their deterministic means. Now, a natural question is: What is the relation between the errors at the two different levels?

Theorem 1: Based on definition 1, and assuming that $D_i \sim N(F_i, \sigma^2)$, $i = 1, 2, ..., n$, and that the D_i are uncorrelated (strictly speaking, we also need the joint normality assumption, which in general can be satisfied), we have:

1. $E_\pi = \sqrt{n}E_aC_n$, where:

$$C_n = \left(\frac{1}{n}\sum_{i=1}^{n}\frac{1}{F_i}\right)\left(\frac{1}{n}\sum_{i=1}^{n}F_i\right). \qquad (2)$$

2. It is always true that $C_n \geq 1$, and $C_n = 1$ if and only if the forecasts across all the platforms are the same.

We note that in the definition for ε_π, we used the same weight, 1/n, for all platforms. If instead we use a weight proportional to the forecast at the platform, then we have the following:

Definition 2: (Weighted Mean Based) Define $E_\pi = E(\varepsilon_\pi)$ and $E_a = E(\varepsilon_a)$, where:

$$\varepsilon_\pi = \sum_{i=1}^{n}\frac{F_i}{\sum_{j=1}^{n}F_j}\frac{|D_i - F_i|}{F_i} = \frac{\sum_{i=1}^{n}|D_i - F_i|}{\sum_{i=1}^{n}F_i} \qquad (3a)$$

and

$$\varepsilon_a = \frac{\left|\sum_{i=1}^{n} D_i - \sum_{i=1}^{n} F_i\right|}{\sum_{i=1}^{n} F_i}. \tag{3b}$$

Theorem 2: Based on definition 2 and with the same assumptions as in theorem 1, we have:

$$E_\pi = \sqrt{n}\, E_a. \tag{4}$$

Mark Sower[1] proposed this heuristic formula. Theorem 2 says that under suitable conditions, equation 4 holds exactly.

Other researchers have addressed a similar problem from the perspective of demand variability. In measuring the relative errors of the forecast at the individual platforms, it was assumed that σ_i/μ_i $(i = 1, 2, ..., n)$ are the same, where σ_i is a measure of demand variability and μ_i is the mean demand at platform i. The advantage here is we do not need to make such a strong assumption. In fact, our measure of the forecast error at the individual platform level can be interpreted as the forecast error at an averaged individual platform.

The following definition of error is based on this observation in practice. The standard deviation of a random variable can be very large if the values this random variable takes on are very large. A more sensible error measure of such a random variable would be the relative error rather than the absolute error. So, given a random variable X, we can measure its error by the coefficient of variance $cv(X) = \sigma(X)/E(X)$ rather than by its standard deviation $\sigma(X)$.

With the unbiased forecast assumption, the forecast error at platform i can be measured by $cv(D_i)$. The average of these coefficients over all platforms is a good measure of the forecast error at the individual platform level. On the other hand, $\sum_{i=1}^{n} D_i$ is the demand from all platforms, and $\sum_{i=1}^{n} F_i$ is the corresponding forecast, so $cv\left(\sum_{i=1}^{n} D_i\right)$ is a good measure of the forecast error at the aggregated platform level.

Definition 3: (CV Based) Define:

$$E_\pi = \sum_{i=1}^{n} cv(D_i)/n \text{ and } E_a = cv\left(\sum_{i=1}^{n} D_i\right).$$

Theorem 3: Based on definition 3, and assuming that the D_i are uncorrelated, we have

$$E_\pi = \sqrt{n}\, E_a C_n, \tag{5}$$

where C_n is defined in equation 2. For theorem 3, we do not have to assume normality to get the relevant results. This is also true for theorem 4.

General Case: The Effect of Bias

We assume here that forecasts are consistently biased. This is expressed as $E(D_i) = F_i + b$, where b denotes the common forecast bias. This indicates that F_i overestimates demand when $b < 0$ and underestimates demand when $b > 0$.

Can we extend the use of definition 3 for the forecast errors to this general case? The answer is no. This is because the standard deviation is independent of bias, and therefore one could erroneously conclude that the forecast error is small when the standard deviation is small, even though the bias b is very significant. Instead, the forecast error now should be measured by the functional:

$$e(D, F) = \sqrt{E([D - F]^2)}\,/F, \tag{6}$$

rather than by the cv, which is $\sqrt{E([D - E(D)]^2)}\,/E(D)$. Hence, in parallel with definition 3, we have the following definition.

Definition 4: (e-Functional Based) Define:

$$E_c = e\left(\sum_{i=1}^{n} D_i, \sum_{i=1}^{n} F_i\right) \text{ and } E_\pi = \sum_{i=1}^{n} e(D_i, F_i)/n,$$

where the functional e is defined in equation 6.

If the bias $b = 0$, then the functional e in equation 6 is the same as the cv, and hence definitions 3 and 4 are equivalent.

Theorem 4: Based on definition 4, and assuming that $D_i \sim (F_i + b, \sigma^2)$,* $i = 1, 2, ..., n$ and that the D_i are uncorrelated, we then have:

$$E_\pi = \sqrt{n}\, E_a C_n \sqrt{\frac{\sigma^2 + b^2}{\sigma^2 + nb^2}}, \qquad (7)$$

where C_n is given in equation 2.

Since definition 1 considers the relative difference between the forecast and the actual, any bias in the forecast will be retained in the difference, so there is no problem in using this definition even if there is bias. However, the relation between the two errors has changed.

Theorem 5: Based on definition 1 and the assumption that $D_i \sim N(F_i + b, \sigma^2)$, $i = 1, 2, ..., n$ and that the D_i are uncorrelated, we have:

$$E_\pi = \sqrt{n}\, E_a C_n \frac{\sqrt{\frac{2}{\pi}}\sigma e^{-b^2/2\sigma^2} + b[2\Phi(b/\sigma) - 1]}{\sqrt{\frac{2}{\pi}}\sigma e^{-b^2/2\sigma^2} + \sqrt{n}\, b[2\Phi(\sqrt{n}\, b/\sigma) - 1]}, \qquad (8)$$

where C_n is defined in equation 2, and $\Phi(x)$ is the cumulative distribution function of the standard normal distribution $N(0, 1)$ at x.

If there is no bias in the forecasting, the relationships between the errors at the two levels are exactly the same for definitions 1 and 3: $E_\pi = \sqrt{n}\, E_a C_n$. This formula, with the introduction of the constant C_n, is slightly different from the hypothesized equation 4. As noted in theorem 1, it is always true that $C_n \geq 1$. If we use definition 2, then equation 4 holds exactly.

If there is bias in the forecasting, then in each relationship formula (equation 7 or equation 8), there is another multiplying factor that reflects the effect of the bias. One can easily find that both of these multiplying factors are less than or equal to 1. This implies that, compared to the error at the component level, the error at the platform-component level when forecast bias exists is less than when the forecast bias does not exist.

If bias does exist, as it does in reality, it seems that the multiplying factor resulting from bias in either equation 7 or equation 8 should be taken into consideration, with suitable estimation of the parameters involved.

* The notation $X \sim (\mu, \sigma^2)$ means that X has mean μ and standard deviation σ but is not necessarily normally distributed.

Correlated Demands

It is reasonable to assume that demand for a component for one platform affects demand for this component for another platform. Also, for a given platform, there is usually a strong correlation between the current demand and the historical demands. The forecast is usually made based on the historical demands. In this section, we first propose a correlated multivariate normal distribution model for the demand stream when the platform is indexed, and then propose a time-series model for the demand and forecast streams when time is indexed. Our goal is to expand our study of the relationship between the two layers of forecast errors in the presence of correlations. Throughout this section, we assume unbiased forecasts, and use the weighted average definition (definition 3) for the forecast error.

Correlated Normal Distribution Model at a Time Point. In this subsection we consider the case where there is correlation across platform demands, but we still assume that time does not affect demand. Suppose that the demand stream D_i, $i = 1, 2, ..., n$ can be modeled by a correlated normal distribution such that $D_i \sim N(F_i, \sigma^2)$ for $i = 1, 2, ..., n$ and that there is a correlation between different D_i expressed as $Cov(D_i, D_j) = \sigma^2 \rho_{ij}$ for $1 \leq i \neq j \leq n$. With this assumption on the demand stream, we have the following result.

Theorem 6: Based on definition 2 and the above correlated normal distribution modeling for the demand stream, we have:

$$E_\pi = \frac{n}{\sqrt{\sum_{1 \leq i \neq j \leq n} \rho_{ij} + n}} E_a. \qquad (9)$$

In particular, if $\rho_{ij} = \rho$ for all $1 \leq i \neq j \leq n$, then we get:

$$E_\pi = \frac{\sqrt{n}}{\sqrt{(n-1)\rho + 1}} E_a. \qquad (10)$$

When the common correlation coefficient ρ is 0 or near 0, we see that equation 4 holds exactly or approximately.

Autoregressive Time Series Model. Now we take into consideration the time effect in the product demand. For platform i, $i = 1, 2, ..., n$ at time t, $t = 1, 2, ...,$ denote by $D_i^{(t)}$

the demand and $F_i^{(t)}$ the forecast. Suppose that the demand stream over time at each platform can be modeled by an autoregressive model AR(p). At platform i, the autoregressive model assumes that the demand at the current time t is a linear function of the past demands plus a random disturbance, that is:

$$D_i^{(t)} = \sum_{j=1}^{p} a_{i,j} D_i^{(t-j)} + \varepsilon_i^{(t)},$$

where the $a_{i,j}$ are constant coefficients. Further, suppose that the forecast $F_i^{(t)}$ is optimal given the historical demand profile $\mathcal{F}_i^{(t-1)} = \sigma\left(D_i^{(1)},\ D_i^{(2)},\ \ldots,\ D_i^{(t-1)}\right)$. That is, with $D_i^{(0)},\ D_i^{(-1)},\ \ldots,\ D_i^{(-(p-1))}$ properly initialized, for $t \geq 1$:

$$D_i^{(t)} = a_{i,1} D_i^{(t-1)} + a_{i,2} D_i^{(t-2)} + \cdots + a_{i,p} D_i^{(t-p)} + \varepsilon_i^{(t)},$$

and

$$F_i^{(t)} = E\left(D_i^{(t)} \middle| \mathcal{F}_i^{(t-1)}\right)$$
$$= a_{i,1} D_i^{(t-1)} + a_{i,2} D_i^{(t-2)} + \cdots + a_{i,p} D_i^{(t-p)},$$

where $\varepsilon_i^{(1)},\ \varepsilon_i^{(2)},\ \ldots,\ \varepsilon_i^{(t)},\ \ldots$ are independently and identically distributed as $N(0, \sigma^2)$ and the random disturbance at time t, that is, $\varepsilon_i^{(t)}$, is independent of the demand stream before time t, that is, $\{D_i^{(t-1)},\ D_i^{(t-2)},\ \ldots\}$. Also, we assume independence across platforms. With the above modeling of the demand and forecast, what can we say about the relationship between the two layers of forecast errors?

Theorem 7: Based on definition 1 or 2 and the above time-series modeling for the demand stream and forecast stream, and assuming that the variances at all platforms are the same, then at any time point, if definition 1 is used:

$$E_\pi^{(t)} = \sqrt{n}\, E_a^{(t)} \tilde{C}_n, \tag{11}$$

where

$$\tilde{C}_n = \frac{E\left(\frac{1}{n}\sum_{i=1}^{n} \frac{1}{F_i^{(t)}}\right)}{E\left[\frac{n}{\sum_{i=1}^{n} F_i^{(t)}}\right]}.$$

and if definition 2 is used, then:

$$E_\pi^{(t)} = \sqrt{n}\, E_a^{(t)}. \tag{12}$$

Rewriting C_n in equation 2 as

$$C_n = \frac{\frac{1}{n}\sum_{i=1}^{n} \frac{1}{F_i^{(t)}}}{\frac{n}{\sum_{i=1}^{n} F_i^{(t)}}}$$

and taking expectations for the numerator and denominator separately in the expression leads to $\tilde{C}_n$. Hence, it is always true that $\tilde{C}_n \geq 1$.

Proofs

Theorem 1 is a special case of theorem 5. Theorem 3 is a special case of theorem 4. The proof for theorem 6 is similar to that for theorem 5, with an application of lemma 1.

Lemma 1: If $X \sim N(b, \sigma^2)$, then:

$$E|X| = \sqrt{\frac{2}{\pi}}\sigma e^{-b^2/2\sigma^2} + b[2\Phi(b/\sigma) - 1] \equiv H(b, \sigma). \tag{13}$$

Proof of Lemma 1: Without loss of generality, we can assume that $\sigma = 1$, since otherwise we can make a simple transformation $Y = X/\sigma$.

$$E|X| = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} |x| e^{-(x-b)^2/2} dx$$

$$= \frac{1}{\sqrt{2\pi}} \int_{0}^{\infty} |x| e^{-(x-b)^2/2} dx + \frac{1}{\sqrt{2\pi}} \int_{0}^{\infty} |y| e^{-(y+b)^2/2} dy$$

$$= I(b) + I(-b), \text{ where}$$

$$I(b) \equiv \frac{1}{\sqrt{2\pi}} \int_0^\infty x e^{-(x-b)^2/2} dx$$

$$= \frac{1}{\sqrt{2\pi}} \int_{-b}^\infty (y+b) e^{-y^2/2} dy$$

$$= \frac{1}{\sqrt{2\pi}} e^{-b^2/2} + b\Phi(b), \text{ and hence}$$

$$E|X| = \frac{2}{\sqrt{2\pi}} e^{-b^2/2} + b\Phi(b) + (-b)\Phi(-b)$$

$$= \sqrt{\frac{2}{\pi}} e^{-b^2/2} + b[2\Phi(b) - 1].$$

Proof of Theorem 1 Parts 2 and 3. First note that function $\varphi(x) = 1/x$ is convex over $(0, \infty)$. Let random variable X have a uniform distribution on the set $\{F_i: 1 \le i \le n\}$, that is, $P(X = F_i) = 1/n$. An application of the Jensen inequality[2] $E\varphi(X) \ge \varphi(EX)$ leads to the desired inequality. The second part is based on the condition for the Jensen inequality to become an equality.

Proof of Theorem 4:

$$e(D_i, F_i) = \frac{\sqrt{E([D_i - F_i]^2)}}{F_i} = \frac{\sqrt{\sigma^2 + b^2}}{F_i},$$

$$E_\pi = \frac{1}{n}\sum_{i=1}^{n} e(D_i, F_i) = \frac{1}{n}\sum_{i=1}^{n} \frac{\sqrt{\sigma^2 + b^2}}{F_i},$$

$$E_a = e\left(\sum_{i=1}^{n} D_i, \sum_{i=1}^{n} F_i\right) = \frac{\sqrt{n\sigma^2 + (nb)^2}}{\sum_{i=1}^{n} F_i}.$$

Hence we have:

$$\frac{E_\pi}{E_a} = \sqrt{n}\sqrt{\frac{\sigma^2 + b^2}{\sigma^2 + nb^2}}C_n.$$

Proof of Theorem 5: Noting that:

$$D_i - F_i \sim N(b, \sigma^2) \text{ and } \sum_{i=1}^{n}(D_i - F_i) \sim N(nb, n\sigma^2),$$

then we have:

$$\frac{E_\pi}{E_a} = \frac{E(e_\pi)}{E(e_a)}$$

$$= \frac{\frac{1}{n}\sum_{i=1}^{n}\frac{1}{F_i}}{\frac{1}{\sum_{i=1}^{n} F_i}} \frac{H(b, \sigma)}{H(nb, \sqrt{n}\sigma)} \text{ (by lemma 1)}$$

$$= \sqrt{n}C_n \frac{\sqrt{\frac{2}{\pi}}\sigma e^{-b^2/2\sigma^2} + b[2\Phi(b/\sigma) - 1]}{\sqrt{\frac{2}{\pi}}\sigma e^{-b^2/2\sigma^2} + \sqrt{n}b[2\Phi(\sqrt{n}b/\sigma) - 1]}.$$

Proof of Theorem 7: The proofs for equations 11 and 12 are similar. We give a proof for equation 11 only. First notice that $D_i^{(t)} - F_i^{(t)} = \varepsilon_i^{(t)} \sim N(0, \sigma_i^2)$. At any given time t, by the definitions for $E_\pi^{(t)}$ and $E_a^{(t)}$, we have:

$$E_\pi^{(t)} = \frac{1}{n}\sum_{i=1}^{n} E\left(\frac{|\varepsilon_i^{(t)}|}{F_i^{(t)}}\right)$$

$$= \frac{1}{n}\sum_{i=1}^{n} E\left(|\varepsilon_i^{(t)}|\right)E\left(\frac{1}{F_i^{(t)}}\right)$$

$$= \frac{1}{n}\sum_{i=1}^{n}\sqrt{\frac{2}{\pi}}\sigma E\left(\frac{1}{F_i^{(t)}}\right).$$

This second step follows from the fact that $\varepsilon_i^{(t)}$ is independent of demands before time t, and hence independent of the optimal forecast at time t, $F_i^{(t)}$. The last step follows from lemma 1 and the same variance assumption across platforms.

$$E_a^{(t)} = E\left(\frac{\left|\sum_{i=1}^{n}\varepsilon_i^{(t)}\right|}{\sum_{i=1}^{n} F_i^{(t)}}\right)$$

$$= E\left(\left|\sum_{i=1}^{n}\varepsilon_i^{(t)}\right|\right)E\left(\frac{1}{\sum_{i=1}^{n} F_i^{(t)}}\right)$$

$$= \sqrt{\frac{2}{\pi}\sum_{i=1}^{n}\sigma_i^2}\, E\left[\frac{1}{\sum_{i=1}^{n} F_i^{(t)}}\right]$$

$$= \sqrt{n}\,\sigma \sqrt{\frac{2}{\pi}}\sigma E\left[\frac{1}{\sum_{i=1}^{n} F_i^{(t)}}\right].$$

The reasoning is the same as for proving $E_\pi^{(t)}$ above.

Conclusion

Forecast errors increase the complexity and difficulty of the production planning process. This results in excessive inventory costs and reduces on-time delivery. In this paper we have studied the forecast errors for the case of several products using the same component. Because data for the component demand (both actual demand and forecast demand) is easier to obtain at the aggregate product level than at the individual product level, we focused on the theoretical relationships between forecast errors at these two levels.

Our first task was to propose formal definitions for measuring forecast errors under different rationales and technical assumptions. The second task was to formally derive relationships between forecast errors at the two levels. As part of our work we proved the validity of a heuristic formula proposed by Mark Sower of the business operations planning department at the HP Roseville, California site.

In addition to analyzing the two-level problem, we derived a theoretical basis for relaxing the usual assumptions concerning correlations in the data across products and over time.

Acknowledgments

I offer my first thanks to Pano Santos for introducing me to a related material planning problem, and to Mark Sower for his marvelous intuition. Special thanks go to Farid Aitsahlia for his careful technical reading and editorial assistance. Thanks also go to Shahid Mujtaba, Alex Zhang (University of Southern California), and Dirk Beyer for their valuable comments and suggestions, and to Bob Ritter, Shailendra Jain, and Paul Williams for their management support and encouragement. In particular, Paul Williams helped greatly in writing the conclusion.

References

1. P. Santos, J. Shan, M. Sower, and A. Zhang, *Material Planning in Configure-To-Order Environment with Product Demand Uncertainty and Component Shortage Conditions*, HP Laboratories Technical Report HPL-97-32, 1997 (HP Internal Only).

2. E.B. Manoukian, *Modern Concepts and Theorems of Mathematical Statistics*, 1986.

Strengthening Software Quality Assurance

Mutsuhiko Asada

Pong Mang Yan

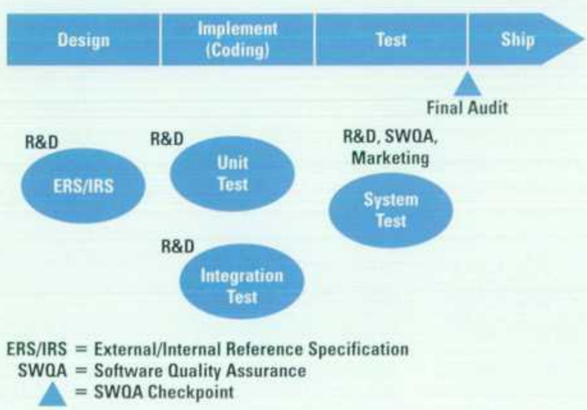
Increasing time-to-market pressures in recent years have resulted in a deterioration of the quality of software entering the system test phase. At HP's Kobe Instrument Division, the software quality assurance process was reengineered to ensure that released software is as defect-free as possible.

The Hewlett-Packard Kobe Instrument Division (KID) develops measurement instruments. Our main products are LCR meters and network, spectrum, and impedance analyzers. Most of our software is built into these instruments as firmware. Our usual development language is C. **Figure 1** shows our typical development process.

Given adequate development time, we are able to include sufficient software quality assurance activities (such as unit test, system test, and so on) to provide high-quality software to the marketplace. However, several years ago, time-to-market pressure began to increase and is now very strong. There is no longer enough development time for our conventional process. In this article, we describe our perceived problems, analyze the causes, describe countermeasures that we have adopted, and present the results of our changes.

Figure 1

Hewlett-Packard Kobe Instrument Division software development process before improvement.



Mutsuhiko Asada

Mutsuhiko Asada is a software quality assurance engineer at HP's Kobe Instrument Division. He received a Master's degree in nuclear engineering from Tohoku University in 1986 and joined HP the same year. Born in Japan's Miyagi prefecture, he is married, has two children, and enjoys mountain climbing and photography.

Pong Mang Yan

Bryan Pong is an R&D engineer with HP's Kobe Instrument Division, working mainly on firmware. He received Master's degrees in electronics and computer engineering from Yokohama National University in 1996. He was born in Hong Kong and likes travel and swimming.

Existing Development Process

The software development process that we have had in place since 1986 includes the following elements:

- Improvement in the design phase. We use structured design methods such as modular decomposition, we use defined coding conventions, and we perform design reviews for each software module.
- Product series strategy. The concept of the product series is shown in **Figure 2**. First, we develop a platform product that consists of newly developed digital hardware and software. We prudently design the platform to facilitate efficient development of the next and succeeding products. We then develop extension products that reuse the digital hardware and software of the platform product. Increasing the reuse rate of the software in this way contributes to high software quality.
- Monitoring the defect curve. The defect curve is a plot of the cumulative number of defects versus testing time (**Figure 3**). We monitor this curve from the beginning of system test and make the decision to exit from the system test phase when the curve shows sufficient convergence.

As a result of the above activities, our products' defect density (the number of defects within one year after shipment per thousand noncomment source statements) had been decreasing. In one product, less than five defects were discovered in customer use.

Perceived Problems

Strong time-to-market pressure, mainly from consumers and competitors, has made our development period and the interval between projects shorter. As a result, we have recognized two significant problems in our products

Figure 2

The product series concept increases the software reuse rate, thereby increasing software quality.

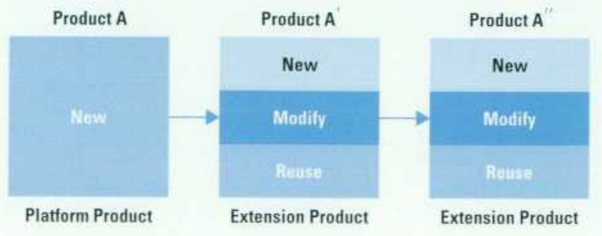
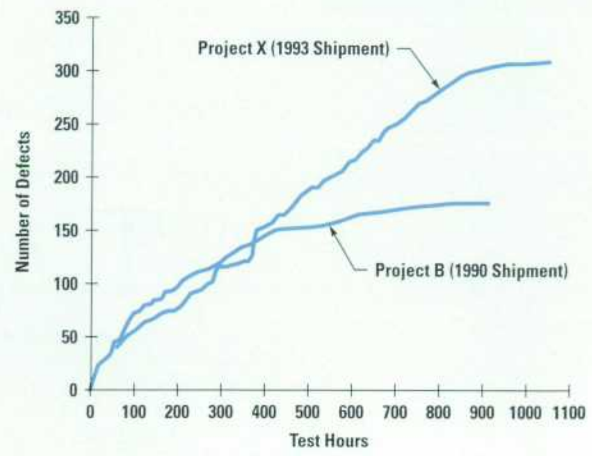


Figure 3

Typical defect curves.



and process: a deterioration of software quality and an increase in maintenance and enhancement costs.

Deterioration of software quality. In recent years (1995 to 1997), software quality has apparently been deteriorating before the system test phase. In our analysis, this phenomenon is caused by a decrease in the coverage of unit and integration testing. In previous years, R&D engineers independently executed unit and integration testing of the functions that they implemented before the system test phase. At present, those tests are not executed sufficiently because of the shortness of the implementation phase under high time-to-market pressure. Because of the decrease in test coverage, many single-function defects (defects within the range of a function, as opposed to combination-function defects) remain in the software at the start of system test (**Figure 4**). Also, our system test periods are no longer as long. We nearly exhaust our testing time to detect single-function defects in shallow software areas, and we often don't reach the combination-function defects deep within the software. This makes it less likely that we will get convergence of the defect curve in the limited system test phase (**Figure 5**).

Increase of maintenance and enhancement costs. For our measurement instruments, we need to enhance the functionality continuously to satisfy customers' requirements even after shipment. In recent products, the enhancement and maintenance cost is increasing (**Figure 6**). This cost consists of work for the addition of

Figure 4

Change in the proportion of single-function defects found in the system test phase.

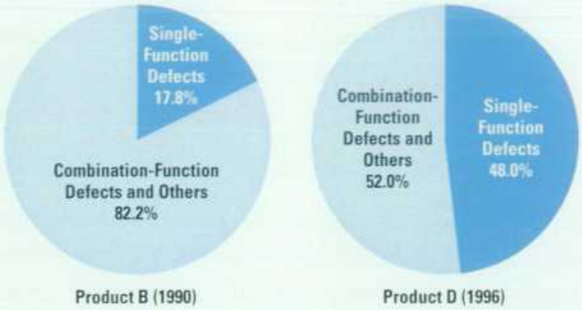
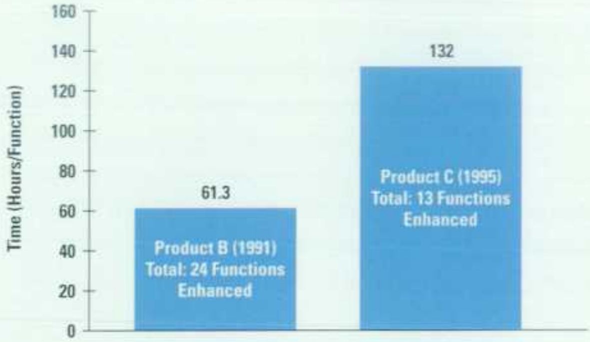


Figure 6

Increase in the cost per function of enhancement and maintenance. The first enhancements for Product B occurred in 1991.



new functions, the testing of new modified functions, and so on. In our analysis, this phenomenon occurs for the following reasons. First, we often begin to implement functions when the detailed specifications are still vague and the relationships of functions are still not clear. Second, specifications can change to satisfy customer needs even in the implementation phase. Thus, we may have to implement functions that are only slightly different from already existing functions, thereby increasing the number of functions and pushing the cost up. **Figure 7** shows that the number of functions increases from one product to another even though the two products are almost the same.

Often the internal software structure is not suitable for a particular enhancement. This can result from vague function definition in the design phase, which can make the software structure inconsistent and not strictly defined. In the case of our combination network and spectrum analyzers, we didn't always examine all the relationships among analyzer modes and the measurement and analyzer functions (e.g., different display formats for network and spectrum measurement modes).

Figure 5

Defect curves for post-1995 products.

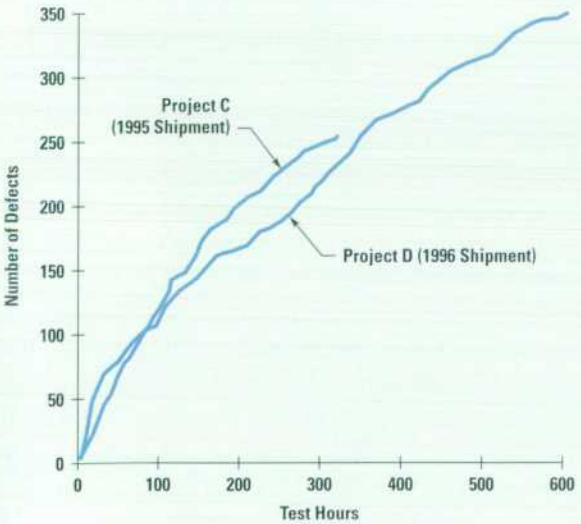
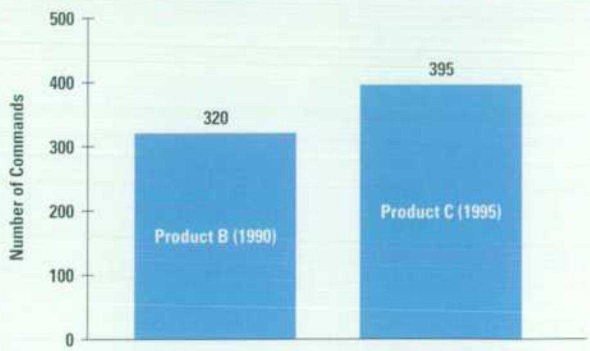


Figure 7

Increase in the number of commands in two similar analyzers as a result of changing customer needs.



Naturally, the enhancement process intensely disturbs software internal structures, which forces us to go through the same processes repeatedly and detect and fix many additional side-effect defects.

Countermeasures[1,2]

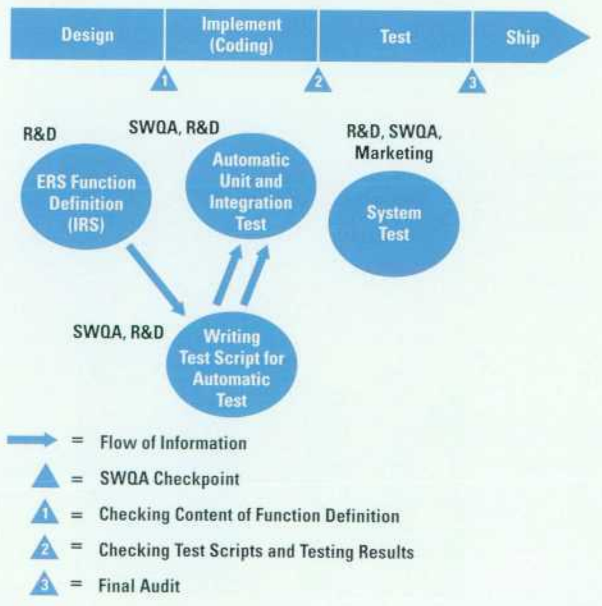
If we had enough development time, our problems would be solved. However, long development periods are no longer possible in our competitive marketplace. Therefore, we have improved the development process upstream to handle these problems. We have set up two new checkpoints in the development process schedule to make sure that improvement is steady (**Figure 8**). In this section we describe the improvements.

We plan to apply these improvement activities in actual projects over a three-year span. The software quality assurance department (SWQA) will appropriately revise this plan and improve it based on experience with actual projects.

Design Phase—Improvement of Function Definition. We have improved function definition to ensure sufficient investigation of functions and sufficient testing to remove single-function defects early in the development phase.

Figure 8

Improved software development process.



We concisely describe each function's effects, range of parameters, minimum argument resolution, related functions, and so on in the function definition (**Figure 9**). Using this function definition, we can prevent duplicate or similar functions and design the relationships of the measurement modes and functions precisely. In addition, we can clearly define functions corresponding to the product specifications and clearly check the subordinate functions, so that we can design a simple and consistent internal software structure. We can also easily write the test scripts for the automatic tests, since all of the necessary information is in the function definitions.

SWQA, not R&D, has ownership of the template for function definition. SWQA manages and standardizes this template to prevent quality deterioration and ensure that improvements that have good effects are carried on to future projects.

Checkpoint at the End of the Design Phase. The first new checkpoint in the development process is at the end of the design phase. SWQA confirms that all necessary information is contained in the function definitions. SWQA approves the function definitions before the project goes on to the implementation phase.

Implementation Phase—Automatic Test Execution. In this phase, SWQA mainly writes test scripts based on the function definitions for automatic tests to detect single-function defects. We use equivalence partitioning and boundary value analysis to design test scripts. As for combination-function defects, since the number of combinations is almost infinite, we write test scripts based only on the content of the function definitions. When we implement the functions, we immediately execute the automatic tests by using the scripts corresponding to these functions. Thus, we confirm the quality of the software as soon as possible. For functions already tested, we re-execute the automatic tests periodically and check for side effects caused by new function implementations. As a result of these improvements, we obtain software with no single-function defects before the system test phase, thereby keeping the software quality high in spite of the short development period. The test scripts are also used in regression testing after shipment to confirm the quality of modified software in the enhancement process. In this way, we can reduce maintenance and enhancement costs.

Figure 9

An example of the improvement in function definition. (a) Before improvement. (b) After improvement.

```
[SENSe Subsystem]

SENSe
        :FREQuency
            :CENTer              <numeric>|DMARKer|MARKer          (Parameter changed
                 STEP
                 [:INCRement]    <numeric>|DMARKer|MARKer          (Parameter changed)
                   :AUTO         ON|OFF
            :MODE                FIXed|LIST|SWEep
            :SPAN                <numeric>|DMARKer|MZAPerture      (Parameter changed)
                 :FULL                                  [no query]
            :STARt               <numeric>|MARKer                  (Parameter changed)
            :STOP                <numeric>|MARKer                  (Parameter changed)
```

(a)

Module	Command	Range	Initial Value	Attribution						Description
Swp	POIN <val>	2 to 801 (int type)	201 (NA, ZA), 801 (SA)	S		2C	SL	4		In SWPT $\neq$ LIST, set MEAS POINT of sweep. (In SA, it's available when SPAN = 0.)
Swp	SPAN <val> (Hz)	0 to 510 M(Hz)	500 MHz	S	S	2C	SL	4	I	In SWPT $\neq$ LIST, set SPAN value of sweep parameter.
Swp	STAR <val> (Hz)	0 to Max Val (Hz)	0 Hz	S	S	2C	SL	4	I	In SWPT $\neq$ LIST, set START value of sweep parameter. Min and max value of START, STOP, CENTER depend of RBW.
Swp	STOP <val> (Hz)	Min Val to 510 M(Hz)	500 MHz	S	S	2C	SL	4	I	In SWPT $\neq$ LIST, set STOP value of sweep parameter. Min and max value of START, STOP, CENTER depend on RBW.
Swp	SPAN <val> (Hz/dBm)	0 to 510 M(Hz), 0 to 20 (dBm)	500 MHz, 20 dBm	S	NZ	2C	SL	4	I	In SWPT $\neq$ LIST, set SPAN value of sweep parameter.
Swp	STAR <val> (Hz/dBm)	0 to 510 M(Hz), − 50 to 15 (dBm)	0 Hz, −50 dBm	S	NZ	2C	SL	4	I	In SWPT $\neq$ LIST, set START value of sweep parameter.
Swp	STOP <val> (Hz/dBm)	0 to 510 M(Hz), − 50 to 15 dBm	500 MHz, 30 dBm	S	NZ	2C	SL	4	I	In SWPT $\neq$ LIST, set STOP value of sweep parameter.
Swp	SWET <val> (s)	0 (Min Meas Time) to 99:59:59 s	Min Meas Time	S		2C	SL	4		Turn sweep time auto setting off and set arbitrary value to sweep time. (In SA, query only.)
Swp	SWETAUTO <bool>	Off (0)/On (1)	On	S	NZ	2C	SL	4	D	Select Auto and Manual of Sweep Time (In SA, Auto only.)
Swp	SWPT <enum>	LINF/LOGF/LIST/POWE	LINF	S	NZ	2C	SL	6	H	Select Sweep Type.
Swp	SWPT <enum>	LINF/LIST	LINF	S	S	2C	SL	6	H	Select Sweep Type.

(b)

Checkpoint at the End of the Implementation Phase. At the second new checkpoint in the development process, SWQA confirms that the test scripts reflect all the content of the function definitions, and that there are no significant problems in the test results. The project cannot go on to the system test phase without this confirmation.

System Test Phase—Redefinition of System Testing. In an ideal testing process, we can finish system testing when we have executed all of the test items in the test cases we have written. However, if many single-function defects are left in the software at the start of system test, we will detect single-function and combination-function defects simultaneously, and the end of testing will become unclear. Therefore we use statistical methods, such as convergence of the defect curve, to decide when to end the system test phase.

In our improved process, we can start the system test phase with high-quality code that includes only a few single-function defects. Thus, we can redefine the testing method to get more efficiency in detecting the remaining defects. We divide the system test items into two test groups. The first group uses black box testing. We write these test cases based on the instrument characteristics as a system and on common failures that have already been detected in the preceding series products. The second group is measurement application testing, which is known as white box testing. The R&D designers, who clearly know the measurement sequence, test the measurement applications according to each instrument's specifications. We try to decide the end of system test based on the completion of test items in the test cases written by R&D and SWQA. We try not to depend on statistical methods.

Checkpoint at the End of the System Test Phase. We use this checkpoint as in the previous process, as an audit point to exit the system test phase. SWQA confirms the execution of all test items and results.

A Feasibility Study of Automatic Test

Before implementing the improved development process described above, we wanted to understand what kind of function is most likely to cause defects and which parts we can't test automatically. Therefore, we analyzed and summarized the defect reports from a previous product series (five products). We found that the front-panel keys, the HP-IB remote control functions, and the Instrument BASIC language are most likely to cause defects. We also observed that the front-panel keys and the display are difficult to test automatically. Based on this study, we knew which parts of the functions needed to be written clearly on the function definitions, and we edited the test items and checklist to make the system test more efficient.

Application of the Improvement Process

Project Y. Product Y is an extension and revision of Product X, a combination network, spectrum, and impedance analyzer. The main purpose of Project Y was to change the CRT display to a TFT (thin-film transistor) display and the HP-IB printer driver to a parallel printer driver. Most of the functions of the analyzer were not changed.

Since Product Y is a revision product, we didn't have to write new function definitions for the HP-IB commands. Instead, we used the function reference manual, which has the closest information to a function definition. The main purpose of the test script was to confirm that each command worked without fail. We also tested some combination cases (e.g., testing each command with different channels). The test script required seven weeks to write. The total number of lines is 20,141.

For the automatic tests, we analyzed the defect reports from five similar products and selected the ones related to the functions that are also available in Product Y (391 defect reports in the system phase). Then we identified the ones that could be tested automatically. The result was 140 reports, which is about 40% of the total. The whole process took three weeks to finish and the test script contains 1972 lines. The rest of the defect reports were checked manually after the end of system test. It took about seven hours to finish this check.

Both of the above test scripts were written for an in-house testing tool developed by the HP Santa Clara Division.[3] An external controller (workstation) transfers the command to the instrument in ASCII form, receives the response, and decides if the test result passes or fails.

Instrument BASIC (IBASIC), the internal instrument control language, has many different functions. It comes with a suite of 295 test programs, which we executed automatically using a workstation. The workstation downloaded each test program to the instrument, ran the program, and saved the result. When all the programs finished running, we checked if the result was pass or fail.

For all of the automatic testing, we used the UNIX® make command to manage the test scripts. The make command let each test program execute sequentially.

Using the test scripts, we needed only half a day to test all of the HP-IB commands and one day to test the IBASIC. Since Product Y is a revision product, we also used the test scripts to test its predecessor, Product X, to confirm that Product Y is compatible with Product X. The test items in the Product X checklist were easily modified to test Product Y.

Project Z. Product Z belongs to the same product series as Product Y (a combination network, spectrum, and impedance analyzer). The reuse rate of source code is 77% of Product Y.

One R&D engineer took one month to finish the first draft of the function definitions. To test the individual HP-IB commands, since the necessary function definition information existed, we easily modified the test script for Product Y to test Product Z. We employed a third-party engineer to write the test scripts. This took five weeks.

Since Product Z is in the same series as Product Y, we are reusing the test scripts for Product Y and adding the new test scripts corresponding to the new defects that were detected in Product Y to test Product Z.

The IBASIC is the same as Product Y's, so we use the same test program for Product Z. The automatic test environment is also the same as for Product Y.

Since Product Z is still under development, we don't have the final results yet. We use the test scripts to confirm the individual HP-IB commands periodically. This ensures that the quality of the instrument's software doesn't degrade as new functions are added. At this writing, we haven't started system test, but we plan to reuse the same product series checklist to test Product Z.

Results

Project Y. In this project, we found 22 mistakes in the manual, 66 defects in Product X while preparing the test scripts, and 53 defects in Product Y during system test. The following table lists the total time spent on testing and the numbers of defects that were detected in Product X in Project X and Project Y.

Table I
Defects found in Product X

	Project X	Project Y
Testing Time (hours)	1049	200
Number of Defects	309	88

According to this data, using the test scripts based on the function reference manual, we detected 88 defects in Product X during Project Y, even though we had already invested more than 1000 test hours in Project X and the defect curve had already converged (**Figure 3**). We conclude that testing the software with a test script increases the ability to detect defects. Also we see that a function definition is indispensable for writing a good test script.

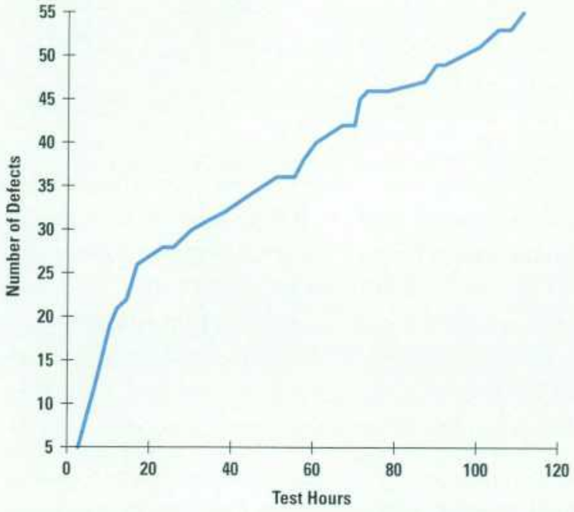
Since the automatic test is executed periodically during the implementation phase, we can assume that no single-function defects remained in Product Y's firmware before system test. Since Product Y is a revision product, there were only a few software modifications, and we could assume that the test items for the system testing covered all the modified cases. Therefore, we could make a decision to stop the system test when all the test items were completed, even though the defect curve had not converged (**Figure 10**). However, for a platform product or an extension product that has many software modifications and much new code, the test items of the system test are probably not complete enough to make this decision, and we will still have to use the convergence of the defect curve to decide the end of the system test. Nevertheless, it will always be our goal to make the test items of the system test complete enough that we can make decisions in the future as we did in Project Y.

The test script is being used for regression testing during enhancement of Product Y to prevent the side effects caused by software modifications.

In **Figure 11**, we compare the test time and the average defect detection time for these two projects. Because Product Y is an extension of Product X, the results are not exactly comparable, but using the test script appears to be better because it didn't take as much time to detect the average defect.

Figure 10

Defect curve for Project Y.



We needed time to write the test scripts, but the system test phase became shorter, so the total development time was shorter for Project Y. The enhancement cost will be lower because we can reuse the same test script for regression testing.

Project Z. We expect that the quality of Product Z will be high before system test because we test Product Z periodically in the implementation phase and confirm the result before entering system test.

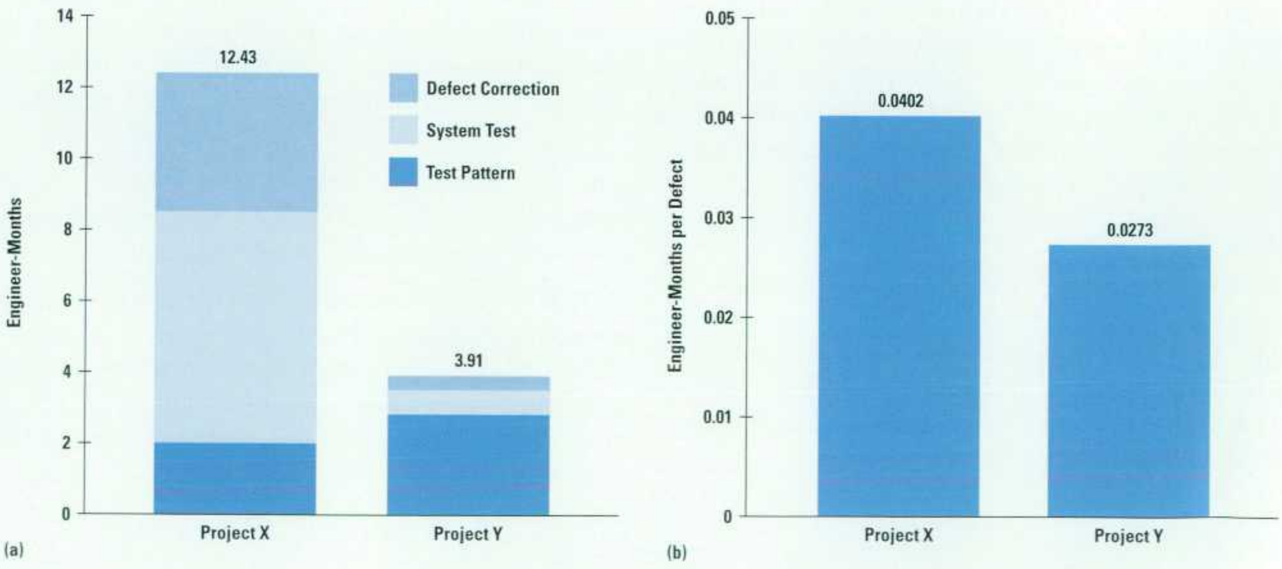
The additional work of the improvement process is to write formal function definitions and test scripts. Since this project is the first to require a formal function definition, it took the R&D engineer one month to finish the first draft. For the next project, we expect that the function definition can be mostly reused, so the time needed to write it will be shorter.

The test scripts are written during the implementation phase and do not affect the progress of the project. Therefore, we only need to wait about a month for writing the function definition before starting the implementation phase, and since the time needed for system test will be shorter, the whole development process will be faster.

Since we are reusing the test scripts of Product Y, the time for writing test scripts for Product Z is two weeks shorter than for Product Y. Thus, for a series product, we can reuse the test scripts to make the process faster. Also, making test scripts is not a complicated job, so a third-party engineer can do it properly.

Figure 11

Cost of software testing for Projects X and Y. (a) Engineer-months spent on software testing. (b) Engineer-months per defect.



Conclusion

We analyzed the software (firmware) development problems of the Hewlett-Packard Kobe Instrument Division and decided on an improvement process to solve these problems. This improvement process has been applied to two projects: Project Y and Project Z. The results show that we can expect the new process to keep the software quality high with a short development period. The main problems—deteriorating software quality and increasing enhancement cost—have been reduced.

This improvement process will be standardized and applied to other new projects. It will also make our software development process conform to the key process areas of CMM (Capability Maturity Model) level 2 and some part of level 3.[1,2]

Acknowledgments

We cannot overstate the importance of the work of Mitsuo Matsumoto on the automatic IBASIC tests. We would like to thank Akira Nukiyama and several of our colleagues for reviewing this paper and giving valuable comments.

References

1. M.C. Paulk, et al, *Capability Maturity Model for Software, Version 1.1*, Carnegie Mellon University, SEI-93-TR-024.

2. M.C. Paulk, et al, *Key Practices of the Capability Maturity Model, Version 1.1*, Carnegie Mellon University, SEI-93-TR-025.

3. B. Haines, *UNIX-Based HP-IB Test Tool (Ttool) Operation Manual*, 1991.

A Compiler for HP VEE

Steven Greenbaum

Stanley Jefferson

With the addition of a compiler, HP VEE programs can now benefit from improved execution speed and still provide the advantages of an interactive interpreter.

This article presents the major algorithmic aspects of a compiler for the Hewlett-Packard Visual Engineering Environment (HP VEE). HP VEE is a powerful visual programming language that simplifies the development of engineering test-and-measurement software. In the HP VEE development environment, engineers design programs by linking visual objects (also called devices) into block diagrams. A simple example is shown in **Figure 1**. Features provided in HP VEE include:

- Support for engineering math and graphics
- Instrument control
- Concurrency
- Data management
- GUI support
- Test sequencing
- Interactive development and debugging environment.

Beginning with release 4.0, HP VEE uses a compiler to improve the execution speed of programs. The compiler translates an HP VEE program into byte-code that is executed by an efficient interpreter embedded in HP VEE. By analyzing the control structures and data type use of an HP VEE program, the compiler determines the evaluation order of devices, eliminates unnecessary run-time decisions, and uses appropriate data structures.

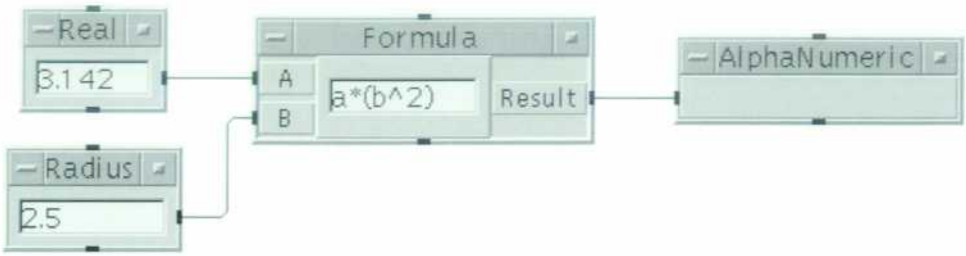
The HP VEE 4.0 compiler increases the performance of computation-intensive programs by about 40 times over previous versions of HP VEE. In applications

Steven Greenbaum
A member of the technical staff at HP Laboratories since 1989, Steve Greenbaum is currently researching "hardware-in-the-loop" systems and programming for distributed systems. He has a PhD degree in computer science (1986) from the University of Illinois at Urbana-Champaign and a BS degree in computer science (1980) from Syracuse University. Steve was born in New York City, is married and has two children. In his leisure time he enjoys playing guitar and taking field trips with his family.

Stanley Jefferson
Stanley Jefferson is a member of the technical staff at HP Laboratories, where he began his career at HP in 1990. He is currently doing research in the area of "hardware-in-the-loop" systems. He has a PhD degree in computer science (1988) from the University of Illinois at Urbana-Champaign. He received BS (1977) and MA (1979) degrees in mathematics from the University of California at Davis. Stan was born in Oakland, California, is married and has two children. He enjoys playing piano and day trips to the beach with his family.

Figure 1

A simple HP VEE program to compute the area of a circle.



where execution speed is constrained by instruments, file input and output, or display update, performance typically increases by 150 to 400 percent.

The compiler described in this article is a prototype developed by HP Laboratories to compile HP VEE 3.2 programs. The compiler in HP VEE 4.0 differs in some details.The HP VEE prototype compiler consists of five components:

- Graph Transformation. Transformations are performed on a graph representation of the HP VEE program. The transformations facilitate future compilation phases.
- Device Scheduling. An execution ordering of devices is obtained. The ordering may have hierarchical elements, such as iterators, that are recursively ordered. The ordering preserves the data flow and control flow relationships among devices in the HP VEE program. Scheduling does not, however, represent the run-time flow branching behavior of special devices such as If/Then/Else.
- Guard Assignment. The structure produced by scheduling is extended with constructs that represent run-time flow branching. Each device is annotated with boolean guards that represent conditions that must be satisfied at run time for the device to run. Adjacent devices with similar guards are grouped together to decrease redundancy of run-time guard processing. Guards can result from explicit HP VEE branching constructs such as If/Then/Else, or they can result from implicit properties of other devices, such as guards that indicate whether an iterator has run at least once.
- Type Annotation. Devices are annotated with type information that gives a conservative analysis of what types of data are input to, and output from, a device. The annotations can be used to generate type-specific code.
- Code Generation. The data structures maintained by the compiler are traversed to generate target code. The prototype compiler can generate C code and byte-code. However, code generation is relatively straightforward to implement for most target languages.

These components combine to implement the semantics explicitly and implicitly specified in an HP VEE program.

Online Information

This complete article can be found at:
http://www.hp.com/hpj/98may/ma98a13.htm

More information about HP VEE can be found at:
http://www.hp.com/go/HPVEE

The Hewlett-Packard Journal

The Hewlett-Packard Journal is published 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 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.

To receive an **HP employee subscription** send an e-mail message indicating your HP entity, employee number, and mailstop to: **ldc_litpro@hp0000.hp.com**

Qualified non-HP individuals, libraries, and educational institutions in the **U.S.** can request a subscription by going to our website and following the directions to subscribe.

To request an **International subscription** locate your nearest country representative listed on our website and contact them directly for a subscription. Free subscriptions may not be available in all countries.

Back issues of the Hewlett-Packard Journal can be ordered through our website.

Our Website

Current and recent issues are available online at **http://www.hp.com/hpj/journal.html**

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. Before doing any work on an article, please contact the editor by sending an e-mail message to: **hp_journal@hp.com**

Copyright

Inquiries

Please address inquiries, submissions, and requests to:

Editor
Hewlett-Packard Journal
3000 Hanover Street, Mail Stop 20BH
Palo Alto, CA 94304-1185 U.S.A.

HEWLETT-PACKARD
Journal

MAY 1998 • Volume 49, Number 2

Technical Information from the Hewlett-Packard Company

FR: HEWLETT-PACKARD LITPRO BLDG 190
ENT: 0000 M/S: 20BBA
TO: KAREN R LEWIS
CORPORATE OFFICES

SEL: JOURNAL - USA #36992

5966-2130E