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.

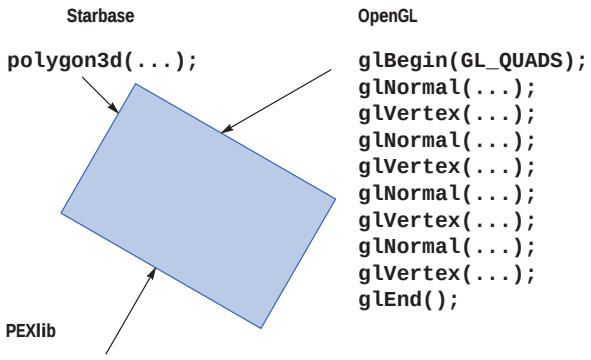
Since OpenGL is an industry-standard graphics API, much of the differentiating value HP delivers is in performance, quality, reliability, and time to market. The central goal of the HP implementation is to ship more performance and quality much sooner.

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

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

Figure 3
OpenGL pipeline.

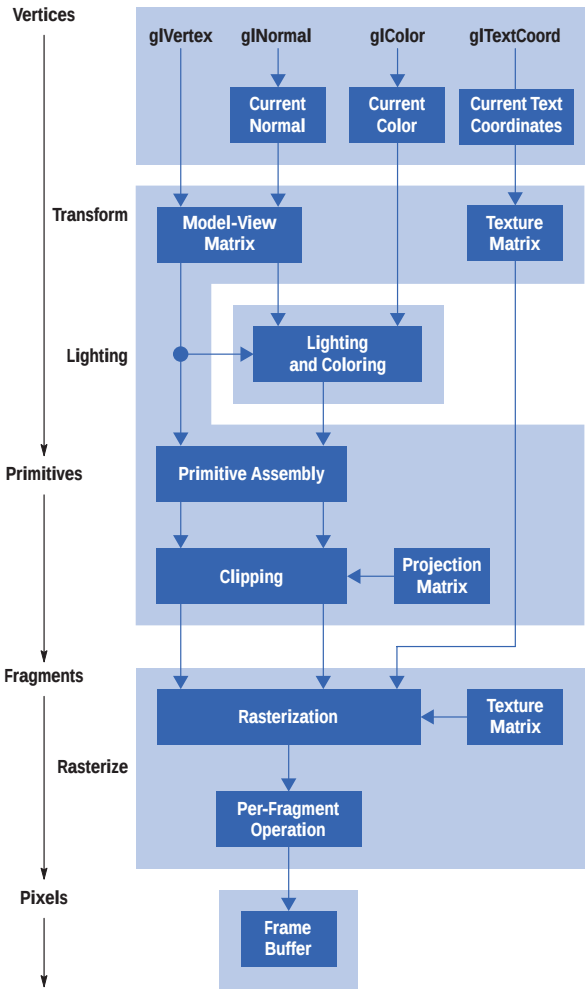
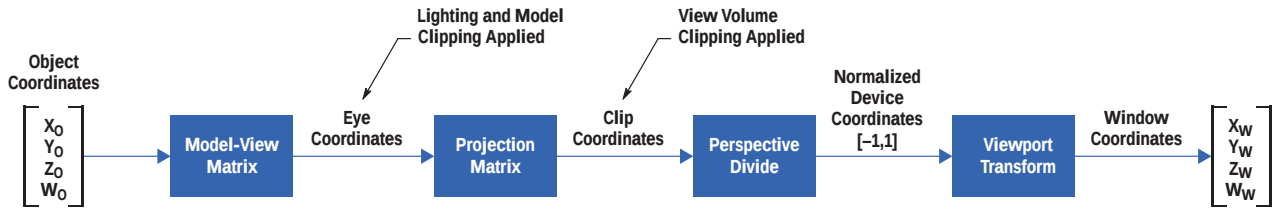


Figure 4

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

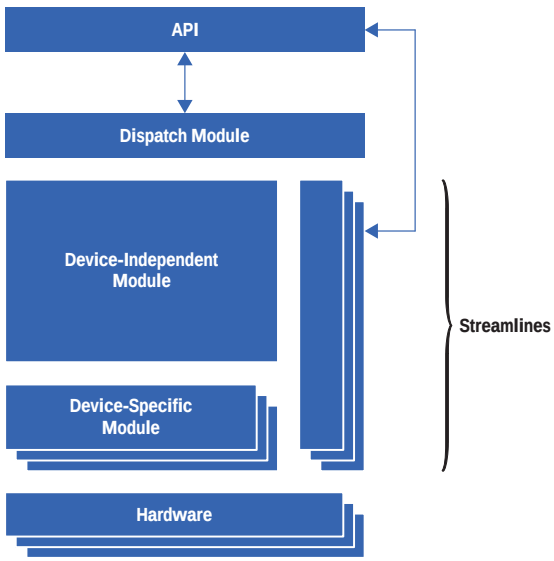
One of the main design goals for the HP OpenGL software architecture was to maximize performance where it would be most effective. For example, we decided to focus on reducing overhead to hardware-accelerated paths and to base design decisions on application use, minimizing the effort and cost required 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.

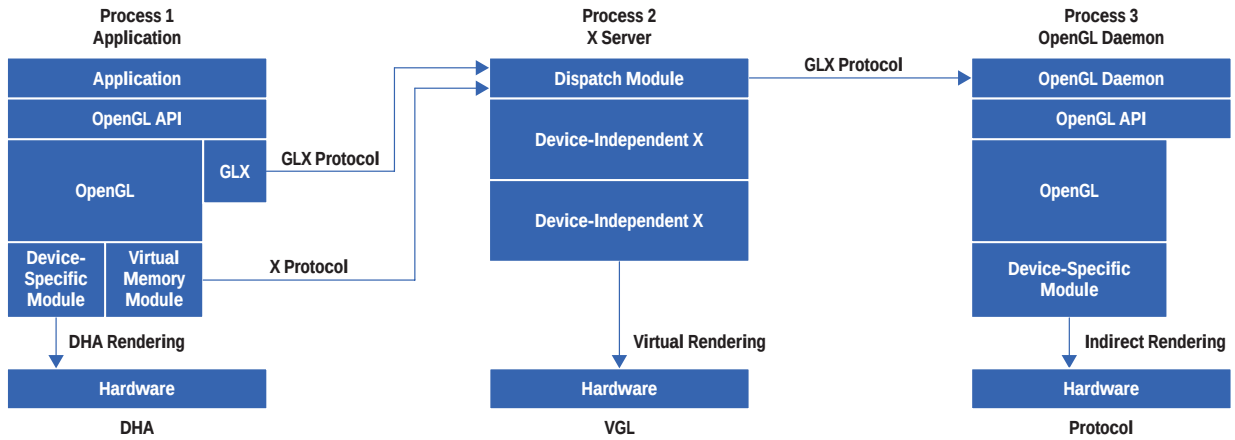
An important thing to understand about streamlines is that they can only be called when the current state is "clean" and the hardware supports the current rendering mode. An example of "not clean" is when the viewing matrix has been changed, and the hardware needs to be updated with the current transformation 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.

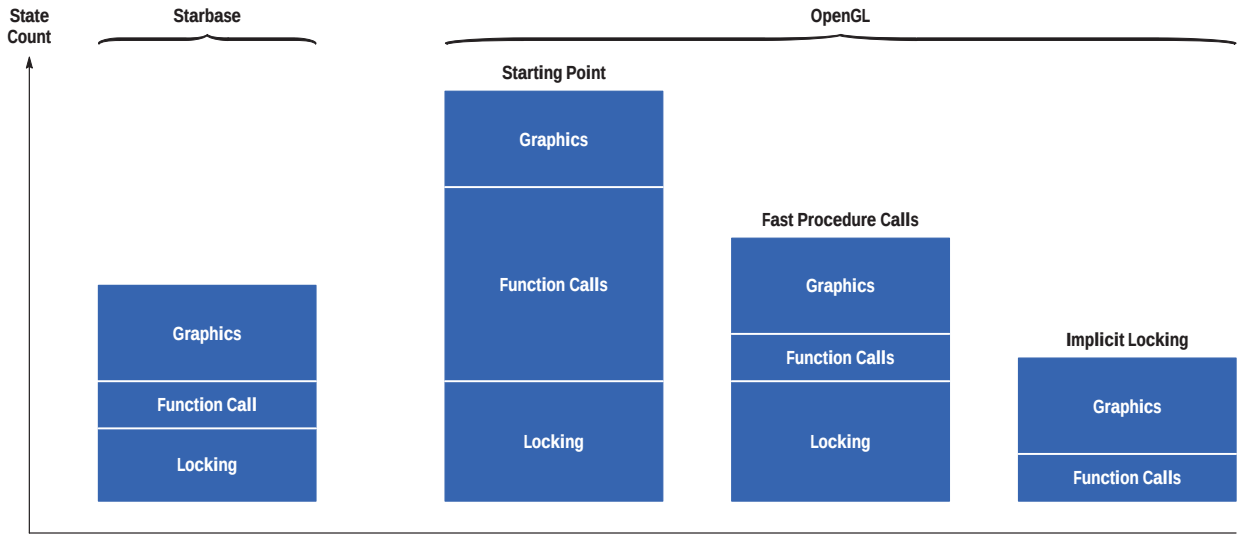
The graphics fault alerts the system that there is another process trying to access the graphics device. The kernel then makes sure that the graphics device context is saved, and the graphics context for the next process is restored. After the graphics context switch is complete, the new process is allowed 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.

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 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.

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

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.

▶ Go to Next Article

▶ Go to Journal Home Page