

Software OpenGL: Architecture and Implementation

Chandrasekhar Narayanaswami, Daniel Brokenshire, Bruce D'Amora, Suzy Deffeyes, Gordon Fossum, James Miller, Barry Minor, Christopher Mussack, John Spitzer, and Linas Vepstas

Due to improved processor architectures and higher clock rates, recent workstation performance has doubled approximately every 18 months. Meanwhile, the price of these machines has continued to decrease. The decreasing cost of computational power has opened doors to many applications which had been considered too computationally intensive. One such case is entry three-dimensional (3D) graphics – an area previously dominated by hardware accelerated graphics adapters. Graphics algorithms, which were previously enabled in dedicated hardware, can now be implemented in a high-level language on the host processor, with an attractive price-performance ratio. This enables 3D graphics application programming interfaces (APIs) such as OpenGL [1], PEX [2], and graPHIGS [3] to perform well on low-cost machines without the addition of costly 3D graphics adapters.

The software approach has many short-term and long-term benefits:

- Complete graphics functionality is immediately available over a wide range of products, from the low end to the high end on the POWER, POWER2 and PowerPC implementations.
- Graphics performance is directly proportional to processor performance and

will continue to increase as processors improve.

- Implementation in high-level programming languages on the host processor enables quick response to changes and enhancements in graphics APIs. For example, software implementation, testing, and maintenance are simpler for general purpose processors than for custom processors (ASICs). In addition, development on general purpose processors generally includes use of better debuggers and several other productivity enhancement tools.

- The software solution also inherits powerful features of a computer system such as virtual memory, pointers, fast context switching, and direct access to main memory. As a result, faster responses to graphics queries are possible with this approach than with heavily pipelined onboard graphics accelerators. This advantage is significant; users are becoming increasingly aware that fast graphical rendering is not the only important aspect of graphics applications.

The following sections detail the design and implementation points which were used to fully optimize OpenGL. The rendering software (renderer) described in this article was implemented within AIXwindows 1.2.5 (X Window System [4] under AIX). Software renderers for graPHIGS and PEX also exist in

AIXwindows but are beyond the scope of this article.

OpenGL: Introduction

OpenGL is a network transparent 3D graphics rendering interface. Users can generate high quality pictures from user defined graphical objects through a simple low-level modular interface.

The OpenGL API is licensed from Silicon Graphics, Inc. and is governed by the OpenGL Architecture Review Board. Current members of the board include IBM, Intel, Microsoft, Digital Equipment Corporation, and Silicon Graphics. The similarity between OpenGL and IRIS Graphics Library, a proprietary programming standard, bodes well for the success of OpenGL as an open 3D programming standard.

OpenGL supports 3D transformation, clipping and culling, Phong lighting, as well as Gouraud shaded, texture-mapped, anti-aliased point, line, and surface primitives. Utilization of stencil and accumulation buffers enables the efficient implementation of constructive solid geometry, shadows, full scene anti-aliasing, and motion blur effects [1].

In a typical scenario (see Figure 1), the OpenGL application must first create a **drawable** (window or off-screen image) and

an OpenGL rendering context. This context must then be associated with the drawable. The required graphics buffers (depth buffer, alpha buffer, stencil buffer, and accumulation buffer) are created when a context is first associated with a drawable. The application may then issue OpenGL commands to be interpreted and drawn into the drawable. A detailed specification of OpenGL exists [1] and will not be discussed here.

Architecture Overview

To facilitate development work and to allow future optimization and expansion, the complete system was segmented into three well-defined components, the window system component, the geometry pipeline, and the rasterizer, as shown in Figure 1.

Window system dependent component

This component encodes commands from the application and sends them (across the network if necessary) to the renderer. It also performs operations upon windows that will host OpenGL rendering (creation, movement, resizing, clipping and destruction). Management of rendering contexts is performed in this component as well.

OpenGL does not explicitly handle user input and interaction, but relies upon the window system to do so.

Geometry pipeline

This component accepts primitives such as vertex coordinates, colors, normal vectors, and texture coordinates. It then performs a subset of the following operations depending upon the current OpenGL state: transformation, clipping, culling, lighting, fog

calculation, and texture coordinate generation. The processed primitives are then passed to the rasterizer. In addition, this component manages state variables such as light and material properties.

Rasterizer

This component accepts primitives (from the geometry pipeline) such as points, lines, and triangles and computes which pixels they cover on the drawable. The generated pixels, called fragments in OpenGL, are then processed according to the current OpenGL state. These operations may include stippling, scissoring, texture mapping, fogging, anti-aliasing, alpha testing, stencil testing and updating, depth testing and updating, blending, logical operation, masking, and frame-buffer writing. The output of the rasterizer goes to the graphics buffers shown in Figure 1. In addition, this component manages all state variables pertaining to rasterization.

The partitioning described above makes it easier to migrate from the X Window System to other window systems (such as Presentation Manager), to build a wide range of graphics accelerators which may implement a subset of the above three components, and to port to multiprocessor systems. The partitioning also allows concurrent software development to proceed in groups across multiple sites. Minimizing component interaction ensures low overhead, resulting in higher performance. Each of the components will now be discussed in greater detail.

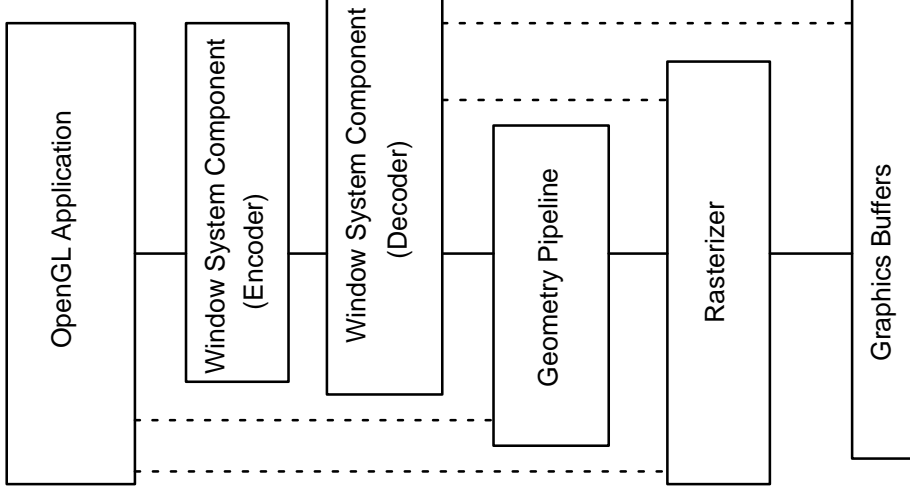


Figure 1: Components of OpenGL. Includes an encoder/decoder, a geometry pipeline, and a rasterizer. Lines connecting components indicate bidirectional communication links. Dashed lines indicate communication links that may not be currently enabled.

Window System Dependent Component

This component has two subcomponents as shown in Figure 1: one in the OpenGL application side (Encoder), the other in the OpenGL extension or library (Decoder). Note that in the current implementation (OpenGL 1.0.0 on AIX 3.2.5) the case where the application communicates directly with the geometry pipeline is not enabled. Several steps were taken to ensure reasonable performance while retaining the advantages of network independence. These steps are described below. The Encoder accepts each application command and creates a command packet which includes an opcode for the command followed by the data. The fast and extensive string manipulation features [5] of the POWER, POWER2, and POWERPC implementations were exploited for encoding and moving data. The command packets are stored in command buffers. These buffers are then sent to the Decoder whenever the user explicitly makes a request or when the buffer is full.

Minimizing communication overhead between the Encoder and Decoder is crucial. To address this issue, shared memory transport, which is faster than the standard socket transport, is used when the Encoder and the Decoder are on the same machine. The Decoder accepts the command packets and stores them into command buffers. These command buffers are then processed in a round-robin fashion. The Decoder is capable of ensuring fairness across clients, so that no client dominates the processor with long sequences of time-consuming atomic

commands. The Decoder also manages the graphics buffers, contexts, and shared display lists (a set of OpenGL commands which are stored and executed as a single unit). Other commands, such as those received from the mouse and keyboard input, are handled by the core window system mechanisms such as those in X [4]. In the current implementation, all graphics buffers are in system memory. Thus, the color buffer in system memory is copied to the frame buffer whenever necessary. Keeping the color buffers in system memory allows us to support OpenGL on most AIX-based POWER, POWER2, and POWERPC workstations.

Geometry Pipeline

The geometry pipeline accepts vertex data from the application (only via the Encoder and Decoder in the current implementation) and stores it in local buffers. The vertices in the buffer are then transformed and processed before being sent to the rasterizer.

The following requirements influenced the overall design and organization of this component:

- Fast switching among various rendering contexts in the client/server model.
- Window system independence for maximum component reuse.
- Geometry pipeline access from either the Decoder or the application directly; allows the application to execute in a client/server or a direct rendering mode [1].
- Optimization based upon the current rendering state.

- Connection of the geometry pipeline component to rasterizers with varying degrees of hardware support. In some situations, the hardware renderer may have insufficient functionality and may require a switch to a conformant software renderer.
- Ability to switch quickly between rendering modes: immediate mode, feedback mode, selection mode, and display list compilation and execution mode.
- Flexibility to accommodate future OpenGL enhancements.

These criteria led to a rendering function hierarchy consisting of procedure tables for internal and external interfaces which satisfied the requirements without sacrificing performance.

At the highest level, the external interface to the pipeline is a procedure table attached to the graphics context. Besides context management functions, the only interface into the pipeline is through this procedure table. There is a mapping between OpenGL

rendering calls and entries in the procedure table. The Decoder dispatches rendering calls through this table. Since the table is anchored to the rendering context, it is automatically swapped when a context switch occurs. This table is explicitly changed when the rendering mode changes. The ability to change tables enables trace facilities, error checking, and rendering mode checking with little or no cost to the user. Because the state is implicit in the procedure table used, no extra branches are required in each call.

Internal function calls are handled similarly. A **methods** table consists of pointers to internal functions. These pointers are switched when

state changes are made. An example of a pointer would be a **methods** table element which is responsible for lighting. Often optimizations can be made for lighting calculations — this makes some calculations far less computationally intensive than the general calculations. If the current state dictates the use of the cheaper function, the lighting entry will be set appropriately instead of pointing to the general function.

Three critical considerations in our design from the standpoint of the processor were:

1) In several situations, the POWER architecture can achieve zero-cycle branch penalties for floating-point compares. Graphics algorithms have several branches and the above property is very useful. However, some sequences of code such as a compare followed by a conditional branch which is further followed by a unconditional branch could cause as much as a 10-cycle delay in the floating-point unit (FPU) and a 7-cycle delay in the fixed-point unit (FXU) [6]. Reordering code allows the compiler to schedule compares well in advance of the branch, which can reduce this penalty.

2) Our target system has a fast on-chip cache. To make optimal use of the cache, the processing-intensive areas or **hot spots** of the geometry pipeline code should be small enough to fit inside the cache. To minimize instruction cache misses, each of the pipeline operations is performed upon a buffered set of vertices before advancing to the next operation in the geometry pipeline. In a processor with a unified

data/instruction cache, this solution offers more room for vertex data at the cost of a small amount of loop overhead.

3) Deviating from the logical sequence of instructions required to implement a given algorithm can greatly increase performance by removing pipeline bubbles so as to increase the utilization of the processor's multiple computation units. Instruction scheduling within a single computation unit is also critical [6] since the FXU [7,8] and FPU are each pipelined. These factors altered the code sequences in several situations (e.g., the transformation phase).

Now we discuss the buffering, transformation, and validation phases of the geometry pipeline.

Vertex Buffering

This phase accepts vertices received from the Decoder and stores them into the vertex buffer along with the current color, normal vector, and texture coordinates.

Such buffering allows amortization of function call and other overheads over several vertices, thereby improving overall performance. Buffering the vertex and color data in contiguous memory locations reduces data stride and therefore improves data cache utilization. The buffer is passed to the transformation stage when the primitive is finished or the buffer is full.

An important consideration is how many vertices should be buffered before they are actually processed. Rendering could be performed as soon as enough vertices to

define the geometric primitive are collected (e.g., after every vertex for points, after two vertices for lines, after three for triangles). Instead, vertices for an entire OpenGL primitive are buffered (up to a maximum of 256). The maximum vertex buffer size was chosen to be 256 vertices since many IRIS Graphics Library applications use that as a limit. (This limit for each primitive aggregate originates from early Silicon Graphics hardware.)

Transformation

This phase transforms the coordinates of the vertices in the vertex buffer from the application-defined coordinate system to the normalized-device coordinate system.

The transformation portion of the pipeline demands a large percentage of processor cycles when processing high-bandwidth primitives such as lines and points. This code is a prime candidate for optimization. This section will describe the different parts of the transformation process and show how it has been specifically tuned for the POWER architecture.

The transformation code performs three major functions:

- Modeling/projection matrix transformation of each vertex.
- Clip code generation and storage for each vertex.
- Normalized device coordinate (NDC) calculation and storage for each vertex.

These three functions exploit superscalar implementations effectively since they have both integer and floating-point computations. First, all vertices are multiplied by the current modeling/projection matrix to transform them into eye space (also called clipping space). **Clip codes** are then computed using vertex values held in registers. Clip codes determine whether a vertex is inside the viewing region. Such codes are helpful in determining whether primitives are trivially accepted (all vertices of the primitive are in the viewing region), trivially rejected (the primitive is not in the viewing region), or need to be clipped (some part of the primitive is in the viewing region). If the vertex's clip code determines that it is within the window, the NDC values are then computed. This three stage process has been tuned to take advantage of the POWER implementation's superscalar features, single-cycle throughput for floating-point operations, and large register files.

The POWER FPU's ability to execute pipelined floating-point multiply-add (FMA) instructions each cycle and to concurrently perform operand loads is used extensively in the transformation code. The entire 16-element transformation matrix is moved into the FPRs (FPU's register file) and then a tight loop performs the transformation, clip code generation, NDC calculation, and storage for the entire primitive buffer. Many of the cycles which calculate clip codes are overlapped on the FXU during the transformation, resulting in nearly free clip code generation. This is followed by calculation of the NDC coordinates from the

intermediate results stored in the FPRs. Lastly, the resulting NDC values and clip codes are stored if needed. Branch logic throughout the entire process is overlapped on the POWER implementation's Instruction Cache Unit (ICU), resulting in zero or single cycle conditional branches and loop control. The zero or single cycle delay is made possible by reordering instructions so that the condition code field of a conditional branch is set far enough before the branch that the BPU can determine the validity of the branch before it is executed, thereby effectively executing it in zero cycles.

The above tuning results in the following approximate cycle counts, for the above three step process, determined from assembler listings for the POWER architecture (thus ignoring cache miss penalties):

Trivially rejected points:
32 cycles/vertex.

Trivially accepted parallel projected points:
38 cycles/vertex.

Trivially accepted perspective projected points:
60 cycles/vertex.

Lines and triangles differ from points in that the concept of trivial rejection is extended to include a deferred clip of the primitive in the next stage of the pipeline. The following are the approximate cycle counts, determined from assembler listings on the POWER architecture, for lines and triangles for this phase of the pipeline:

Primitives that require clipping:

37 cycles/vertex. Trivially accepted parallel projected:

39 cycles/vertex. Trivially accepted perspective projected:
59 cycles/vertex.

Because these performance numbers are for one stage of the complete rendering process, the data should not be extrapolated for rendering performance numbers.

Vertex Buffer Validation

This phase examines the vertex buffer created at the end of the transformation process and determines the set of operations that need to be performed based on the currently specified options. Validation routines contain sections for transforming normal vectors, calculating lighting, calculating fog factors, and generating texture coordinates. The vertices output from the validation routine are then sent to the rasterizer.

Since the validation phase operates on several vertices and makes several passes, the size of the vertex buffer should be such that the buffer stays in cache between the passes. Smaller buffers would flush too frequently and result in high function-pointer and loop-overhead costs. The 256-element buffer satisfied the above requirements.

The first stage of validation determines whether all the vertices in the buffer are trivially rejected. If so, the buffer is discarded. Thus no intensive computations are performed in this case. If all the vertices are trivially accepted, none of the primitives are clipped and the buffer is passed to the culling

stage. If some primitives are clipped, then the buffer is passed to the clipping stage, where new vertices are generated as needed, and then passed to the culling stage. After culling, the vertices are sent to the final stage of validation. The final stage of validation is performed by using a function table that is indexed through the current state. This reduces the number of branches required per vertex.

Lighting is a typical example of the final stage of validation. This phase evaluates the light intensity at the vertices of the objects. Since it is an important and time-consuming part of validation, we explain it in greater detail. OpenGL supports up to eight lights with specular, diffuse, and ambient lighting components. Each light can be local or infinite. Local lights can also be specified as spotlights with cones of a specified width. The user can also select two-sided lighting and can independently define front and back material properties. Additionally, attenuation coefficients (constant, linear and quadratic) are defined on a per-light basis. The **methods** table holds the lighting function for the current lighting state. When the vertex buffer is ready to be lit, it is passed to the correct function through one of three lighting function pointers: front-faced lighting, back-faced lighting, or two-sided lighting.

Certain commonly used lighting conditions have fast paths, where redundant or unnecessary calculations are avoided. For example, if an enabled light is located at infinity, then the light's direction is constant across all vertices. Moreover, most lights are

not spotlights and that simplifies many of the lighting equations.

For specular lighting and spotlights, a power function is needed. To increase speed, tables of values based on the exponent are used to linearly interpolate between evenly spaced values. These tables represent the slope and intercept of the line that approximates the function between two points. The appropriate tables are calculated when the user sets the specular exponent or the spotlight exponent. An input threshold is first computed. Any input below the threshold will produce a zero output. The range from the threshold to 1.0 is divided into a constant number of segments, so no resolution is lost if large exponents are used. A larger exponent will have a higher threshold, therefore a smaller range and closer interpolation points.

These techniques have reduced the performance penalty usually associated with lighting, especially adding multiple lights.

Similar techniques have been applied to the other stages such as fog factor computation and texture coordinate generation.

Rasterizer

The rasterizer accepts the transformed and validated vertex buffers and determines which pixels in the frame-buffer need to be updated and performs all pixel processing operations. The rasterizer also accepts pertinent state transition commands, but we will not discuss it here, since it is rather straightforward.

The rasterizer has a broad primitive level interface. It has native entry points for points,

independent lines, connected lines, line loops, independent triangles, triangle strips, independent quadrilaterals, quadrilateral strips, polygons, bit maps and pixel maps. Three processing steps may be applied to the incoming data:

- 1) Edge generation. For line and surface primitives, sub-pixel accurate slopes and offsets are generated for color, depth, alpha, fog and texture coordinates.
- 2) Fragment generation. Edge walking and span interpolation is performed for surface primitives; interpolation along a line is performed directly. The resulting interpolated data is referred to as "fragments" to distinguishing them from pixels, as they are not yet ready to be placed in the frame buffer.
- 3) Fragment processing. Fragments are converted into pixels by applying stippling, scissor test, texture mapping, fog, anti-aliasing coverage, alpha test, stencil test and update, depth test and update, alpha blending, dithering, logical operations, and masking before the frame buffer contents are updated. The fragment processing also supports rendering into multiple color buffers.

Our implementation of the rasterizer is designed to maximize code reuse, utilize common algorithms, and provide optimum performance. It currently supports three APIs: PEX, glPHIGS, and OpenGL. Common rasterization algorithms are used for all three APIs whenever possible to maintain rendering consistency across all APIs and to reduce development cycles.

The biggest challenge to the rasterizer development is finding an architecture that is compact, yet optimizes performance. The broad primitive interface to the rasterizer and the immense number of possible fragment processor paths (there are at least 31 bits of state, leading to at least 2^{31} possible paths), it is impractical to write a tuned renderer for each path. Writing a single renderer with a conditional branch for each bit of state would have been simple, but would have resulted in poor performance.

A two-pronged approach is used. A renderer was written for each basic primitive, for each combination of interpolant. There are roughly 200 renderers, of which the depth-buffered, true-color, texture-mapped aliased polygon renderer is a typical example. The multiple execution units of the RISC System/6000 implementation [5] were used to maximize interpolation performance by performing depth and texture coordinate interpolation in floating point, and other data in fixed point.

Each renderer comes in one of two varieties. In one case, the fragment processing path deemed most likely to occur was coded and "in-lined" with the renderer. This allows the compiler to schedule instructions in an optimal fashion. For all other fragment processing paths, the renderers place fragments into a fragment buffer, which is handed over to a fragment processing specialist.

As stated earlier, it is impractical to write code for each of the fragment processors. So code for a particular fragment processor is assembled dynamically, when needed.

Assembly code snippets are written for each stage of the fragment processing pipeline. As an OpenGL application changes the state of the rasterizer, appropriate snippets are chained together to create a custom fragment processor. The resulting fragment processor is flushed from the data cache (where it was built) into system memory, where it is invoked as a call through a function pointer. Thus, any previous fragment processor is flushed from the instruction cache automatically. To avoid repeated regeneration of fragment processing specialists, the cache of the most recently used specialists is maintained.

Wherever possible, the fragment processing pipeline stages are reordered so that expensive stages occur after all testing stages. For example, texture mapping and fogging can generally be performed after the testing stages. By doing so, fragments that are not visible do not get texture mapped or fogged. Commonly executed fragment processor code is designed so that variables which are constant for all fragments (for example, buffer pointers and reference values) are loaded into registers once for an entire fragment buffer. The POWER architecture's reasonably large register file helps avoid register spillage in the fragment specialists in all but some cases where sophisticated texture filters are used.

Buffer clearing (including color, depth, stencil, and accumulation buffer) is significantly accelerated by utilizing the cache instructions provided by the POWER architecture. When clearing a buffer to zero, an entire cache line/block is cleared in a single instruction.

When clearing a buffer to a nonzero value, the cache line is cleared to zero using these same instructions, then the nonzero values are written into the cache line. Clearing the cache line before writing eliminates the cache loading overhead. The writing of nonzero values is further improved by using 64-bit double-precision floating-point stores.

Improving Application Performance

To obtain better performance, the application developer can adopt several techniques, including:

1. Judicious use of display lists so that the communication overhead between the Encoder and Decoder is minimized.
2. Judicious choice of rendering primitives based on the underlying architecture. The cost of communication overhead and computation speed must be considered. If communication is fast and the architecture has significant parallelism, using primitives with few vertices will increase overall efficiency. On the other hand, if communication overhead is large and rendering is slow, using primitives with many vertices is more effective.

Similarly, using the appropriate option for rendering primitives improves performance. For example, if the set of polygons to be drawn are all triangles, rendering them using the specific OpenGL disjoint triangle option (treats several disjoint triangles as one primitive) provides better performance than rendering them each as an independent polygon.

3. Avoiding redundant state transitions. For example, modification of rasterizer state in feedback and selection mode may not be necessary. Also, inappropriate application of object-oriented concepts to graphics rendering can result in redundant state changing commands. For example, it is common to define a "rendering method" for an object that sends down several state changing commands for each object. Many of these state change requests will be redundant when rendering several objects that possess similar attributes.

4. Positioning lights at infinity if that satisfies the application's requirements.

Several other tips are provided in [1].

Conclusions

This article described some of the salient features of the current software implementation of OpenGL on IBM's workstations based on the POWER, POWER2 and PowerPC family of processors. Several optimizations have been made to improve speed while retaining sufficient flexibility to provide even faster solutions in the future.

As parallel systems, such as symmetric multiprocessor systems and workstation clusters, become increasingly powerful and available [9], general software solutions will be even more interesting.

Acknowledgments

We would like to thank a few people who have significantly contributed to the development of OpenGL for the POWER,

POWER2, and PowerPC systems. They are: Abraham Mammen, for high level design of the rasterizer, Brian Horton and John Readey for their contribution to the rasterizer; and Bill Luken and John Gerth of IBM Research for their valuable input. Several others have also contributed their time and ideas to the effort.

References

1. OpenGL Architecture Review Board, "OpenGL Reference Manual," Addison-Wesley, 19928.
2. R. J. Rost, J. D. Friedberg and P. L. Nishimoto, "PEX: A Network-Transparent 3D Graphics System," *IEEE Computer Graphics and Applications*, Vol. 9., No. 4, July 1989, pp. 14-26
3. graPHIGS Programming Interface, Subroutine Reference Manual, Document No. SC33-8194, IBM Corporation.
4. R. W. Scheifler and J. Gettys, "The X Window System," *ACM Transactions on Graphics*, Vol. 5, No. 10, pp. 79-109, 1986.
5. R. Groves and R. Oehler, "RISC System/6000 Processor Architecture," *IBM RISC System/6000 Technology*, SA23-2619, IBM Corporation, 1990, pp. 16-23.
6. Predicting Execution Time on the IBM RISC System/6000, Document No. GC24-3711-00, IBM Corporation, pp. 14.
7. B. Olson, R. Montoye, P. Markstein, and M. NguyenPhu, "RISC System/6000 Floating-Point Unit," *IBM RISC System/6000 Technology*, SA23-2619, IBM Corporation, 1990, pp. 34-43.
8. R. K. Montoye, E. Hokenek and S. L. Runyon, "Design of the IBM RISC System floating-point execution unit," *IBM Journal of Research and Development*, Vol. 34, No. 1, January 1990, pp. 59-70.
9. IBM Shared Memory System Power/4 User's Guide and Technical Reference, IBM Corporation.

© IBM Corp. 1993

International Business
Machines Corporation
11400 Burnet Road
Austin, Texas 78758-3493