The rendering architecture of the DN10000VS

Conference Paper *in* ACM SIGGRAPH Computer Graphics · September 1990

DOI: 10.1145/97880.97912 · Source: DBLP

2 authors:

David B. Kirk
NVIDIA
112 PUBLICATIONS **7,173** CITATIONS
SEE PROFILE

Douglas Voorhies
11 PUBLICATIONS **241** CITATIONS
SEE PROFILE

The Rendering Architecture of the DN10000VS

David Kirk†* and Douglas Voorhies*

*Apollo Systems Division of Hewlett-Packard
300 Apollo Drive
Chelmsford, MA 01824

†California Institute of Technology
Computer Science 256-80
Pasadena, CA 91125

ABSTRACT

The Apollo DN10000VS treats graphics as an integral part of the system architecture. Graphics requirements influence the entire system design. All floating-point computations for graphics are performed by the CPU(s), while rasterizing is handled by simplified hardware having no microcode. We decided to support alpha buffering, quadratic interpolation, and texture mapping directly in hardware. This partitioning reduces the cost of a high-end workstation, without sacrificing high rendering quality and performance. This paper describes some of the design trade-offs which led to the final system design.

CR Categories and Subject Descriptors: C.1.2 [**Processor Architectures**]: Multiprocessors - parallel processors; C.1.3 [Processor Architectures]: Other Architecture Styles; I.3.1 [**Computer Graphics**]: Hardware Architecture - raster display devices; I.3.2 [Computer Graphics]: Graphics Systems; I.3.3 [**Computer Graphics**]: Picture/Image Generation; I.3.7 [**Computer Graphics**]: Three-Dimensional Graphics and Realism

General Terms: Architecture, Algorithms, Graphics, Interpolation, Systems

Additional Key Words and Phrases: quadratic, shading, alpha buffering, texture mapping, Mach band

INTRODUCTION

High-performance graphics hardware for workstations has been evolving into massive, complex, autonomous, and highly-specialized subsystems. While this focus improves cost/performance ratios, especially those measured by narrow rendering performance metrics (e.g., vectors/sec. and polygons/sec.), and leaves the CPU free for other tasks, the hardware is generally idle while the system executes the applications' non-graphics parts, such as modeling, simulation, and database manipulation.

An alternative approach is to invest in the CPU, memory, and bus structure, where performance improvements benefit all application activity. System balance, for example between modeling and rendering, is shifted from the presence of various hardware options to CPU process scheduling. Such a system can dynamically respond to the applications' changing demands from millisecond to millisecond. We will describe here a commercial system architecture which offers high-performance graphics while having a minimum of specialized hardware. We will discuss our goals, its rendering features, and where we were successful and unsuccessful in minimizing the hardware.

PREVIOUS WORK

Several commercial architectures, most notably Silicon Graphics [Akeley88] and Hewlett-Packard [Rhoden89] [Swanson86], incorporated straightforward VLSI rendering pipelines. The process of polygon rendering is well understood; each step was cast in various mixes of hardware and microcode. Not unexpectedly, the targeted cases enjoy impressive cost/performance efficiencies. However, scalability and configurability, which allow a product to match and continue matching a user's evolving needs, were sacrificed. Moreover, the burden of microcode development and maintenance is significant, and often coding complexity is not faced until after the hardware is unalterable. The trend to add increasing amounts of hardware to improve polygon performance has continued [Deering88][Potmesil89]. Again, we see these systems excelling at the specialized rendering for which they were designed, and again they are difficult to modify for new rendering algorithms. Some advanced research implementations go even further [Gharachorloo88][Fuchs89], but currently these designs are too costly for a balanced workstation product.

There has been work towards solving the scalability problem. A good example is Torborg's design, which configures 1 to 8 floating point processors to handle transformation, clipping, and lighting [Torborg87]. Unfortunately, the unit of work is coarse, so the system balance can be altered only by increasing the graphics power in large, fixed steps. While Torborg's dedicated graphics hardware potentially provides very high graphics performance, it does so at a commensurately high cost.

A step in the integration direction was taken in the Stellar GS1000, which leverages the CPU for part of the graphics setup [Apgar88]. However, there is still considerable graphics hardware, including a microcoded VLSI setup processor and a massive microcoded drawing engine. The Intel i860 also has an integrated architecture, since it appends some rasterizing instructions to a RISC instruction set. But graphics needs do not appear to have influenced the overall CPU design. It provides only moderate graphics performance, while placing heavy demands on its memory subsystem.

Brute force graphics hardware has been widely explored and commercially exploited, but little work has been focused on the gains possible through finesse and integration. Clearly, applying graphics techniques to solve real-world problems involves far more than just ultra-high polygon rendering rates. We wanted to explore the integration of a broad range of graphics functionality into the system as a whole.

GOALS

Our first goal was a balanced system, in which most of the hardware was usable most of the time. This is an elusive goal. The extreme variability in the system workload of a high-end interactive workstation led us away from devoting hardware to specialized tasks and toward techniques which improve system versatility and agility.

Although our performance goals were aggressive and unyielding, we explicitly minimized the graphics hardware. We chose to try to modify the system as a whole to be more graphics-capable. Fortunately, the DN10000's PRISM™ instruction set was being designed simultaneously. We were able to influence its evolution, greatly improving the performance of our transform and lighting pipeline. Also, this software was recoded to run on multiple CPUs in parallel. This work allowed us to replace the traditional micro-programmable graphics floating-point engine with powerful CPUs.

Nonetheless, the unforgiving rasterizing performance requirements demanded some dedicated hardware. This is a critical point. Myer and Sutherland [Myer68] forcefully argue against specialized hardware and for beefing up the main system to handle the job. We believe both extremes should be avoided. For example, device controllers, modems, and video back-ends all off-load the main CPU effectively and without loss of generality. But there is a risk in casting too much functionality in hardware.

The fast-changing marketplace punishes the profligate implementation of questionable features in hardware as newer techniques quickly supercede older ones. Moreover, as object procedures replace display lists, general-purpose CPUs will become more critical to rendering. Together these argue for a model where very fast general-purpose CPUs are central, and for reexamining the graphics pipeline's hardware/software boundary.

We chose to push that boundary quite low to keep the graphics hardware as simple and inexpensive as possible. For example, we were able to eliminate the need for it to have microcode. This simplification mandated careful changes above the boundary to maintain performance.

CPU MODIFICATIONS

We worked with the instruction set and CPU designers to bolster our performance. For better or worse, we targeted coordinate transformations as the "typical" application to be optimized to exhaustion. Most instruction set enhancements which improve the transformation loop positively affect matrix and vector operations as well.

We completely examined the operations for the perspective transformation, clipping, and drawing of 3-D vectors. An in-depth attack on this one algorithm proved very effective in pushing our performance beyond our 1 million (connected, 10-pixel, 3-D, transformed) vectors per second rendering target. The CPU cycle time was 55 ns. This translates to 19 or fewer CPU cycles available per vector. For each vector, we must read the next model space point, perform the 4x3 matrix multiplication, check the clip limits, perform perspective divisions, and send the drawing command to the hardware. This results in the following 59 basic operations:

Data fetch:	3 loads
4x3 transform:	12 loads
	9 multiplies
	9 adds
Perspective:	1 load
	1 divide
	2 multiplies
Clip-check:	6 loads
	6 compares
	6 branches
Draw command:	2 fp-int converts
	1 mask-merge
	1 store

To perform these 59 operations in 19 cycles requires both innovative instructions and instruction-level parallelism. To meet these as well as other needs, the PRISM™ CPU evolved into a super-scalar RISC architecture [Apollo88]. The instruction set permits parallel dispatch of two instructions simultaneously: one for the integer processor (IP) and one for the floating point processor (FP). Furthermore, the pipelined FP can initiate both an ALU and a multiplier operation in each cycle. The combination of parallel integer and floating point dispatch and 5-operand floating point opcodes allows the completion of 3 operations *every* cycle.

One quickly notices that operating at these rates, the FP would be starved for data. For this reason, the IP was modified to directly access the floating point register file. By using a 7-port FP register file, the IP can decode and execute floating-point loads or stores (64-bits/cycle) while the FP is executing a 5-operand double operation. Although this does not provide fresh data at the maximum FP consumption rate, there is no starvation if more than one operation is performed on the data. Certainly this is the case in coordinate transforms and other matrix operations. Additional buffering between memory and FP operations is provided by increasing the number of FP single-precision registers to 64. Consequently the compilers can be far more effective at loop unrolling and software pipelining. Memory and bus bandwidth were further increased by using a larger 64-byte FP cache-fill line size.

The clip check comparisons and associated branch conditions are operations that do not commonly occur in applications other than graphics. In a typical CPU with compare and test operations, 3-D vector clip checks require 12 comparisons in a hierarchy of test and jump operations. The PRISM™ architecture includes a clip condition code register, holding the most recent 12 comparison results as two 6-bit fields representing the 3-D clip volume comparisons for two endpoints. Branch predicates are provided for trivial accept and trivial reject. For connected vectors, these condition codes and branch predicates reduce the clip test to six FP compares and one branch for each polyline segment trivially accepted.

Enlarging the FP register file to contain 32 double precision values or 64 single precision values reduces the latency of the transform and clip loop and reduces the number of instructions per iteration. The need to reload the transform matrix, view distance for perspective, and clipping boundaries on subsequent loop traversals was eliminated. Taken together, these changes reduce the instruction count to 18 cycles as follows:

	Total Cycles		
	IP	**FP-ALU**	**FP-MPY**
Data fetch:			
3 loads	3		
4x3 transform:			
9 multiplies, 9 adds		9	9
Perspective:			
1 divide			7
2 multiplies			2
Clip-check:			
6 compares, 1 branch	1	6	
Draw command:			
2 fp-int, 1 merge, 1 store	2	2	

With these improvements, the multiplier is busy for 18 cycles, the ALU is busy for 17 cycles, and the IP is busy for 6 cycles, so the multiplier limits the transform rate. Unfortunately, although the cycle count is 18, the operations are sequential and the associated latencies (3 cycles for multiply/add, 7 cycles for divide) prevent processing of a single vertex in 18 cycles. After a single loop is unrolled to calculate 2 points at once and packed as tightly as possible, however, the resulting code does execute at 18 cycles per vertex.

Transform and clip was an effective focus for influencing the architecture and implementation, because it is simple and readily analyzed. Nonetheless, it is a narrow goal, and not the only operation which must be optimized. Shaded polygon rendering is the obvious next graphics algorithm to examine, but its outer loop is over 30 times as large, making it more difficult to analyze. We surveyed it at a high level, to see if it contains operations that are missing from the transform loop. We found it to be a fairly typical floating point mix. Consequently, the main impact of examining polygon rendering was not specific floating point performance or instruction set modifications but the drive for multiprocessing.

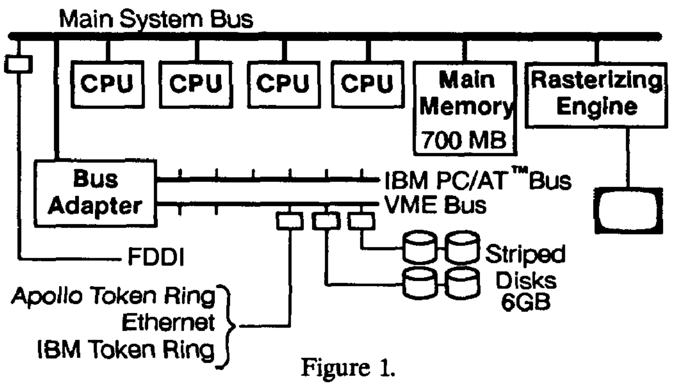


Figure 1.

Software Architecture for Multiprocessing

In order to render shaded 3-D polygon images, the CPU must perform coordinate transformations, clipping, lighting calculations, and differential setup for interpolation of color and depth information. One DN10000 CPU can do this at 32K triangles/sec. (for one light source, perspective, unconnected triangles), far short of the 108K 24-bit RGB Z-buffered 100-pixel triangle/sec. hardware rasterizing rate. To balance the hardware and software, and to make performance scalable, we need to exploit multiple CPUs, as shown in Figure 1.

One might think that an effective way to harness multiple CPUs for a rendering pipeline is to emulate a hardware pipeline in software. While this is a direct approach, it has some straightforward drawbacks. In a software graphics pipeline, the data would flow from the display list through each of the processors in turn and then to the drawing engine hardware. This means that the data, in various forms, would cross the bus 8 times on a 4 processor system, passing in and out of the caches of each of the processors. This is a waste of bus resources, and would swamp the DN10000's 150 Mbyte/sec bus. Finally, such a pipeline runs at the rate of its slowest stage, and rendering does not readily split into equal-load pipeline stages, especially if the number of stages may vary from one to four CPUs.

We chose a different and more universal approach: each CPU does all the computing associated with a unit of work (e.g., a triangle). The CPUs stagger their activity so that access to each critical resource never overlaps. Simple round-robin spin locks enforce synchronization.

Round-robin synchronization has very low overhead and works well when the tasks require equal time. The tasks naturally delay each other in a sequential rhythm; each lock is freed by the previous requestor just before being tested by the next. When the tasks are unequal, some spinning on locks occurs. Another important feature of the round-robin approach is that primitives are rendered in display list order, guaranteeing that a multiprocessor-rendered image is identical to the single-processor image at object intersections.

Two interlocks are used for graphics: one protecting the "next drawing primitive" fetch pointer and one protecting the drawing hardware itself. Since these resources are used at the beginning and the end of each processing loop respectively, they could apparently be merged into a single critical region. Separating them proves worthwhile, however, since they have different exception conditions.

All four processes must have access rights to the graphics hardware in rapid sequence. Since access is controlled by virtual memory mapping of the drawing engine input FIFO [Voorhies88], we must bypass this normally exclusive mechanism and map access into the address space of all four processes concurrently. The multiprocessing is symmetrical for normal drawing. Non-drawing tasks, including attribute update and instancing, are handled by a single processor. This asymmetry arises from the need to broadcast the results of such tasks to the other processors.

For this approach to multiprocessing to be successful, access to critical resources for N CPUs must be concentrated in 1/Nth (chronological) of the code. For example, the basic fetch, transform, clip-test, light, and hardware setup loop takes about 600 CPU cycles. Updating the pointer to the next triangle only takes about 40 cycles, and the 25 stores to the drawing hardware are scattered among 50 more cycles. But adding overhead from bus contention, interprocessor communication locks, and cache coherency makes for a very tight fit. Because of these effects, the multiprocessor speedup is slightly less than linear, especially with four processors. An image which averages 32K triangles/sec. on one CPU averages 98K on four CPUs.

Taken together, enhancing the floating point CPU performance and structuring the setup task to span multiple processors allow our general purpose CPUs to handle the rendering process up to pixel drawing. Since our performance goals required pixels to be drawn faster than the CPU instruction rate, dedicated drawing hardware is necessary. Our success was mixed in keeping that hardware simple, due to our desire to add advanced rendering features.

GRAPHICS HARDWARE

Address Synthesis

We found it straightforward to keep the control logic and address synthesis logic simple, but our desire for high-quality rendering forced us to make the color synthesis data path highly parallel and complex. See Figure 2. We chose the simple control logic approach of generating one pixel at a time [Voorhies89]. For address synthesis we chose scanline-aligned trapezoids as the fundamental hardware area primitive. The same hardware easily handles rectangles and vectors. Trapezoids are an effective primitive for composing polygons, yet do not require much hardware to traverse. A screen-aligned trapezoid can be traversed using only two vector generators, plus a horizontal traversal counter and a scanline counter, as in Figure 3.

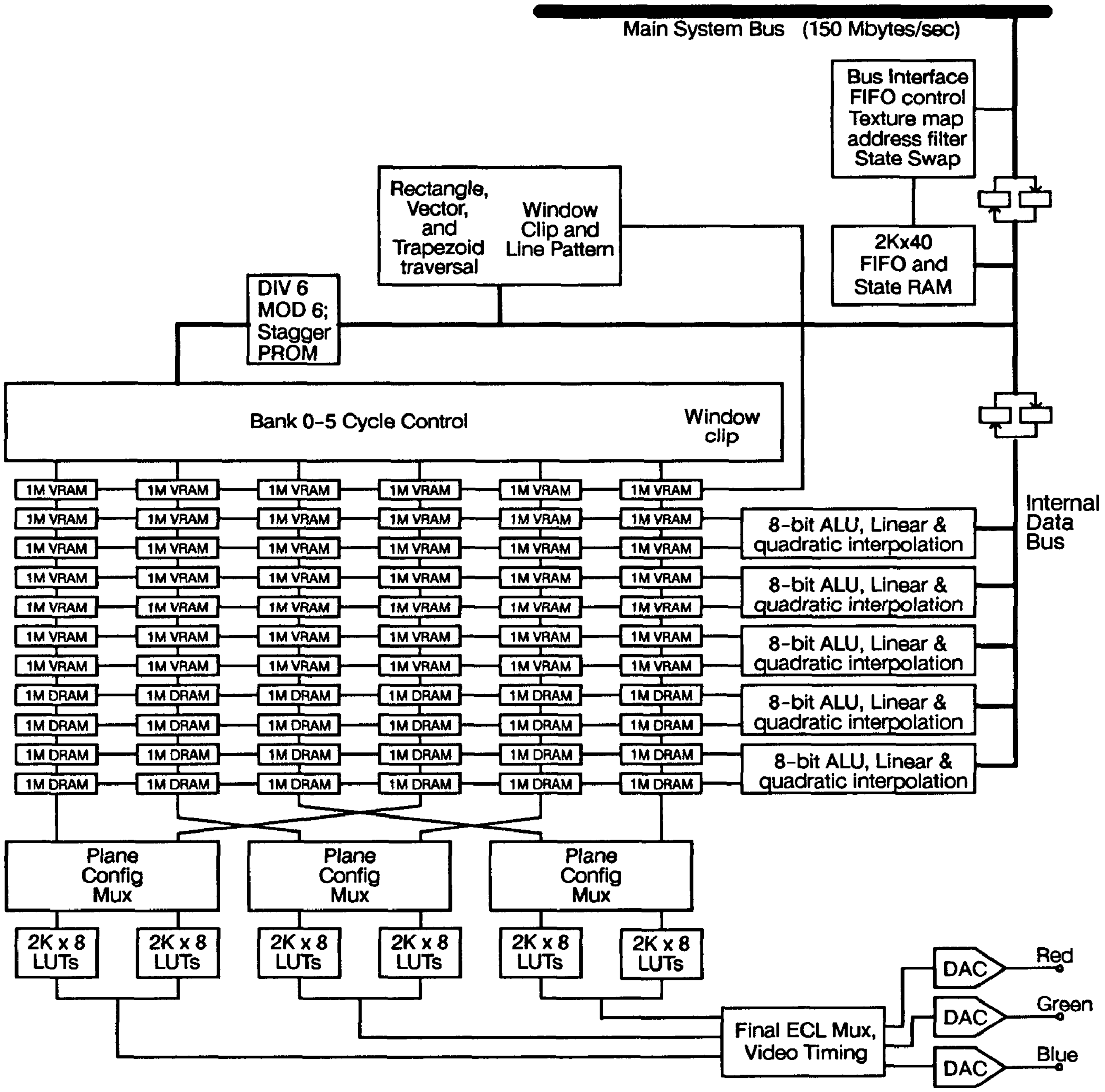

Main System Bus (150 Mbytes/sec)
Bus Interface FIFO control Texture map address filter State Swap
2Kx40 FIFO and State RAM
Rectangle, Vector, and Trapezoid traversal
Window Clip and Line Pattern
DIV 6 MOD 6; Stagger PROM
Bank 0-5 Cycle Control
Window clip
1M VRAM
1M DRAM
8-bit ALU, Linear & quadratic interpolation
Internal Data Bus
Plane Config Mux
2K x 8 LUTs
Final ECL Mux, Video Timing
DAC
Red
Green
Blue

Figure 2.

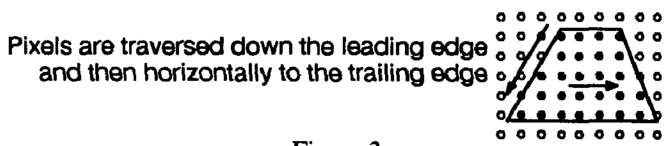

Pixels are traversed down the leading edge
and then horizontally to the trailing edge

Figure 3.

Sub-pixel Vertex Positioning

Faster CPUs allow delaying the conversion from floating point space to fixed point space, aiding precision. Bresenham vector and polygon edge-walking algorithms [Bresenham65] initialized using integer vertexes do not solve the crucially important consistency problem of double-hitting or missing pixels along the shared boundary of abutting triangles [Lathrop90], and cause visually distressing temporal aliasing in animated vectors. We chose to initialize these algorithms using floating-point arithmetic, effectively positioning vertexes between pixels.

For consistency between collinear vectors and edges, we derive the two Bresenham increments from the floating point slope alone, with final precision equivalent to the ratio of two 16-bit integers. We normalize all Bresenham constants values to preserve precision in the integer hardware. As all Bresenham vector generators are founded upon DIV and MOD arithmetic, setting the two hardware increment registers is a simple matter. However, setting the initial error term is not as straightforward, and choosing the correct starting pixel is problematic.

Three principles guided our thinking:

Fidelity Rule: Collinear line segments which overlap select the same pixels (within that overlap). Note that pixel selection being independent of traversal direction is an instance of the Fidelity Rule.

Connected Rule: Start and end pixels of consecutive vectors, or shared polygon vertex pixels, are identical or adjacent. So what is connected in floating point space remains visually connected in screen space.

Extent Rule: When chosen pixels are projected onto either axis, all lie within the projection of the true line segment.

Unfortunately, it is impossible to satisfy all three rules simultaneously, or even to choose the same rules for vectors and triangle edges. For polygons, we satisfy the Connected rule only in the Y dimension. For vectors, on the other hand, we were unwilling to sacrifice the Connected Rule, lest we have unsightly gaps. Instead, we give up the Extent Rule, as necessary, to connect polyline segments.

Once we have identified the first pixel, we are free to calculate the initial data values for that pixel center, such as color/Z/alpha for triangles or the initial error term for vectors. We note in passing that our solution for vectors is appropriate for anti-aliased vectors. It largely solves the problems of abutting triangles and quivering or ratchety vector motion, producing more aesthetically pleasing results. The complexity of this method lies in the floating point software initialization; the hardware is only a common 16-bit integer Bresenham vector and edge walker.

COLOR/Z/ALPHA SYNTHESIS FEATURES

We were less successful in minimizing the hardware to synthesize data values. The decision to synthesize one pixel at a time, coupled with our performance goals, forced a pixel drawing rate of 27.5 ns. To complete pixels at this rate required parallelism. The difficulty was compounded by each rendering feature needing its own idiosyncratic hardware.

Color Synthesis

Most 3-D rendering employs linear interpolation of color and Z values across an area. Our linear interpolation uses a 28-bit data path. For colors and alpha values, the 28 bits are divided into 1 clamping bit, 8 color bits, and 19 color fraction bits. A full-screen trapezoid requires only 12 fraction bits to remain accurate (within .5) during linear interpolation. The 28-bit width was chosen to support 16-bit Z buffering and quadratic interpolation. Of course, this precision is only of value when the color and depth information are calculated accurately for the initial pixel center. A procedure for this is presented in [Lathrop90].

Pixel addresses and data, although generated one-at-a-time, are pipelined. Trapezoid traversal requires three different color increments, one for horizontal and two for the Bresenham edge-walking. For linear interpolation, we keep only two color derivative registers, since the edge increments differ by exactly a horizontal increment. Since all derivative and initial value calculation is done in software, the hardware need handle only the interpolation itself.

Z Synthesis

For Z buffering, Z values of 16, 24, and 32 bits can be kept in the frame buffer. The 28-bit interpolator provides 16-bit Z values with 12 fraction bits. By duplicating the computation in two chips, one per Z byte, no carries need propagate between them. 24 and 32-bit Z-buffering use 44-bit arithmetic; propagating carries make them slightly slower.

Quadratic Interpolation

Originally we implemented quadratic interpolation to reduce the perceived Mach bands at polygon boundaries which are caused by discontinuous color derivatives [Mach59]. For a more current and detailed discussion, as well as reprints of English translations, see [Ratliff65]. We expected quadratic to come quite a bit closer than linear to preventing this problem. We saw it as between Gouraud and true Phong shading, equivalent in computational complexity to "Fast Phong" shading [Bishop86]. Indeed, by combining quadratic interpolation with an exponentiation table as a texture map, we can directly implement Bishop's "Fast Phong" algorithm.

Quadratic hardware is also useful in accelerating the product of two linear polygon functions. Examples include depth-cued shading (as in the PHIGS model) where a linear shading intensity is multiplied by a linear depth function, or variable transparency, where the linear shading intensity is multiplied by a linear transparency function. This approach succeeds because the linear functions are combined as part of the setup done with high precision in the CPU.

Additionally, initial experiments suggested second-order interpolation would reduce highlight aliasing compared to linear interpolation. Figure 4 shows a curved surface rendered with no color interpolation, linear color interpolation, and quadratic interpolation. Note with quadratic interpolation the boundaries of the highlights are curved.

Quadratic interpolation is set up by fitting a second-order surface to six points: a triangle's vertexes and side midpoints. Six lighting calculations are performed, and the simultaneous equations are solved to give the color derivatives. The hardware then

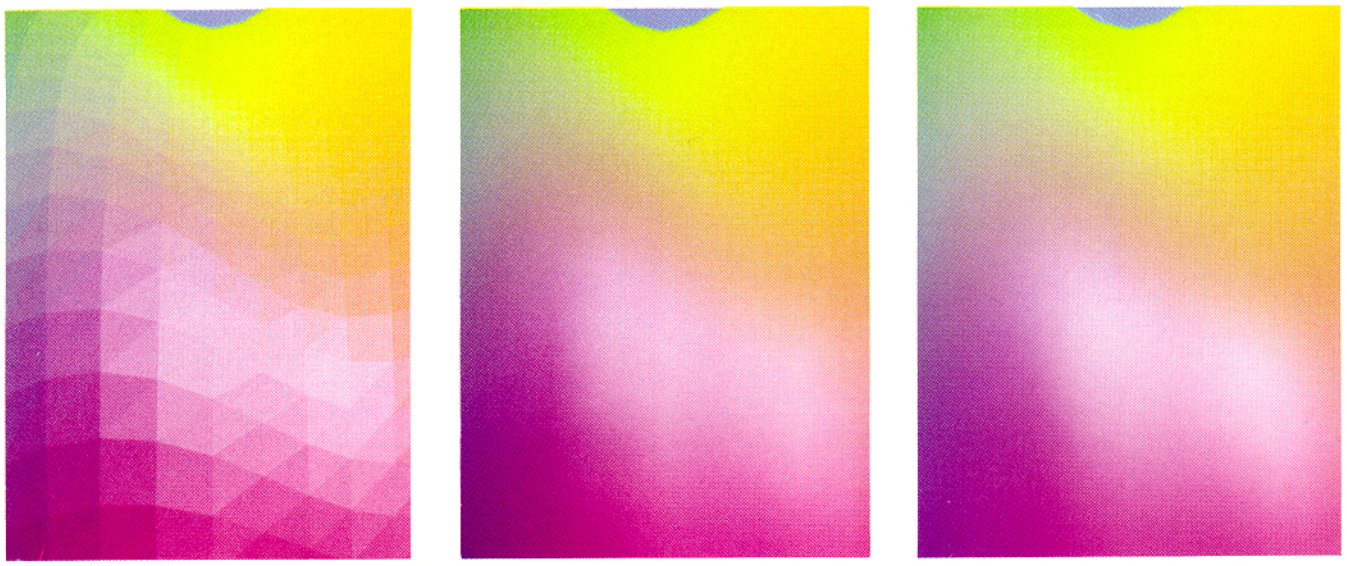


Figure 4.

interpolates by forward differences, requiring three second-derivative registers, each 28-bits wide [Kirk88]. 28 bits is not sufficient precision for full-screen traversal, so software limits the size of quadratic patches. Care must be taken during setup to ensure that overflow is minimized during interpolation. Software limits the second derivative values, although final color values can be clamped to 0 and 255 rather than wrapping.

Alpha Buffering

Alpha buffering is available in all drawing operations for the appearance of transparency or for anti-aliased matting. 8-bit alpha fractions, encoding opacity in units of 1/255, can be multiplied with interpolated or frame buffer pixel values for blending into the frame buffer. In most cases, the blending formula is fixed during the entire drawing operation. One case, the "Z-buffer" blend, selects dynamically between OVER and REVERSE-OVER [Porter84] based on the Z comparison result at each pixel. The hardware cost was four parallel modulo-255 multipliers, one each for alpha, red, green, and blue, plus their data paths.

Texture Mapping

Texture mapping patterns polygons by mapping images onto them. In our implementation, the image must reside in the frame buffer. We interpolate the frame buffer addresses as we traverse the destination polygon. We were able to reuse the color interpolators to synthesize texture map indexes, so little additional hardware is needed. Thus S and T, the coordinate offsets in the texture map, are treated by both software and hardware just as the other triangle attributes: color, alpha, and Z. Alternating independent reads and writes make poor use of the interleaved frame buffer, so texture mapping is much slower than interpolated color. This tradeoff was accepted to keep texture mapping an inexpensive addition.

Texture Address Synthesis

S and T are synthesized as 10-bit offsets into a 1024x1024 texture map. Fast-changing S or T can undersample the map image and cause severe aliasing artifacts. We avoid this by storing a pyramid of square texture maps [Williams83]. Pre-processing software is used to prepare texture maps of any power-of-two size from 1024x1024 to 1x1. If the S and T interpolation is quadratic, the hardware uses their first derivatives to dynamically select which pyramid level to use:

$$\text{Biggest} = \max\left(\left|\frac{\partial S}{\partial X}\right|, \left|\frac{\partial S}{\partial Y}\right|, \left|\frac{\partial T}{\partial X}\right|, \left|\frac{\partial T}{\partial Y}\right|\right)$$

$$\text{Level} = \text{TRUNC}\left(\log_2\left(\text{Biggest}\right)\right)$$

The size range can also be clamped to limits, for use with smaller texture maps. Smaller and more filtered maps are chosen for object horizons or distant objects. For simplicity, we do not interpolate either between pixels or between maps, and the maps are always square. Consequently, horizons blur more than necessary.

Texture Perspective Approximation

In addition to dynamically selecting filtered maps for anti-aliasing, quadratic interpolation serves another purpose. Linear interpolation of texture indexes does not account for the foreshortening caused by perspective division [Heckbert86]. While some static views may appear reasonable, animation immediately exposes the error. Heckbert points out that for proper appearance it is necessary to perform a divide per pixel to properly generate the S and T values. We have combined a quadratic perspective compensation composed with the linear texture mapping function.

A linear perspective approximation is a poor fit for the inverse of this rational function; a quadratic approximation is much more successful, giving a reasonable appearance except in extreme perspective. Figure 5 shows a graph comparing the perspective warping of a textured span at 45 degrees to the line-of-sight using linear, quadratic, and the correct function. The span extents from the origin to $X=40$, $Z=40$, and the eyepoint is back at $X=0$, $Z=-40$. The span's perspective screen projection covers 20 pixels. In this fairly severe test, quadratic is accurate to 2.6% (less than half a pixel), whereas linear deviates by 17.1%.

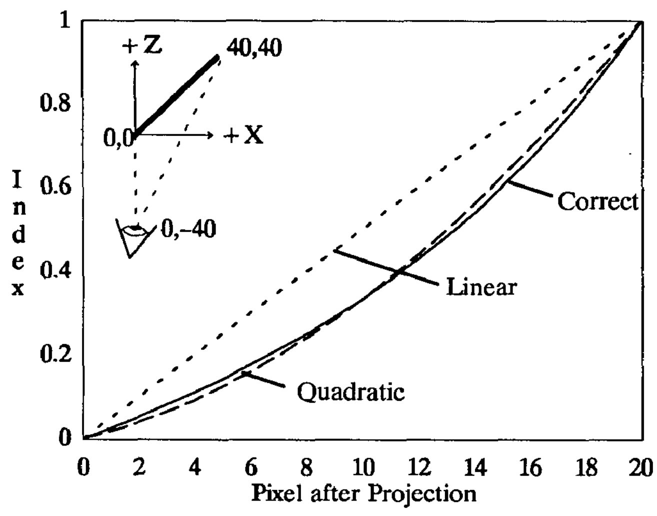
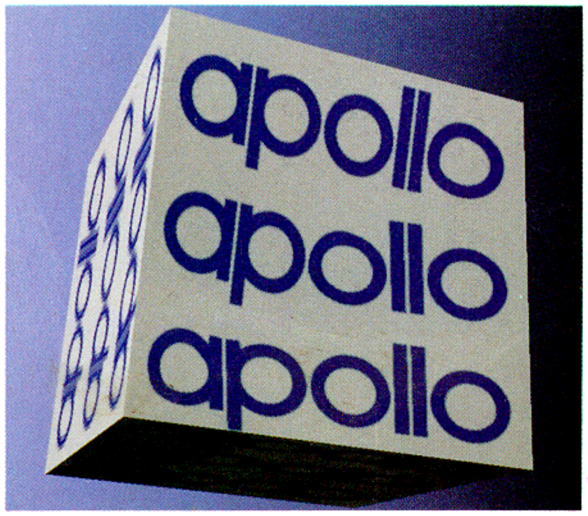
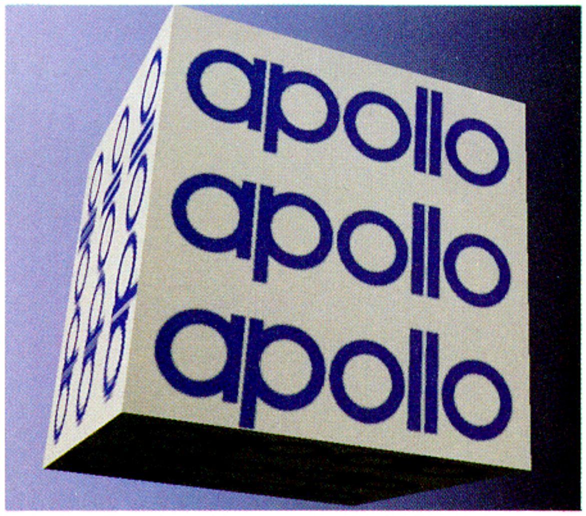


Figure 5.





Quadratic

Figure 6.

Figure 6. shows a cube with a texture map interpolated linearly and interpolated quadratically. The cube is constructed from only two large triangles per side.

Texture Illumination

Illumination can be factored in by reusing the alpha buffering multipliers to multiply the sampled texture color with a interpolated darkening function. Either monochrome or RGB illumination can be modeled. When texture mapping, alpha buffering, and Z buffering are used simultaneously, the visual cues from all three techniques combine well. See Figure 7.

Dithering

Dithering allows a 40-plane frame buffer to do interactive 3-D rendering. By allocating only 12 bits per image (4 bits each of red, green, blue), double-buffered drawing with a 16-bit Z buffer fits into 40 planes. However, 4 bits per color component results in severe color banding. A simple 2x2 ordered dither [Floyd75] can correct this by providing the visual impression of 64 shades of each color component, yet it requires only a few gates and imposes no speed penalty. See the split-screen comparison in Figure 8. The 2x2 pattern is small enough to be invisible on a monitor. Dithering can also be used with other pixel formats. Used with 24-bit color, it provides the appearance of 30 bit color without the added cost of a deeper image memory or higher-precision DACs.

FRAME BUFFER

The frame buffer is the heart of any raster graphics device. Its size determines screen resolution, its depth determines display quality and features, and its organization strongly affects performance. Although we would have preferred to integrate the frame buffer into main memory, the 318 Mbytes/sec. drawing bandwidth and the 437 Mbytes/sec. video requirement were achievable only with a specialized implementation. It is integrated into virtual memory.

Interleaving

We use horizontal 6:1 interleaving to achieve 27.5 ns. sequential pixel writing while individual RAM cycles take 165 ns. to complete. All interleaving requires the ability to control multiple concurrent RAM cycles. We further implemented dynamic cycle selection to maximize RAM throughput. The six RAM banks can perform different cycle types at the same time, such as a full "row/column" cycle in one bank while a page-mode cycle is done in another. While many gates are required to manage six variable cycles concurrently, it greatly improves the bandwidth of the frame buffer and is less expensive than adding more RAMs. We stagger alternate scanlines by 3 pixels, providing a 26% improvement in random vector drawing to achieving 1.1 million 10-pixel vectors per second,

Mixed VRAMs and DRAMs

One way to minimize the frame buffer cost was to use a mix of VRAMs and DRAMs. We used 1 megabit RAMs with similar timing. The DRAMs can hold Z values, alpha values, and off-screen images such as texture maps, pop-up menus, and tile patterns.

By using 1 Megabit VRAMs in the image memory, we are able to fit a competitive 40-plane 3-D graphics subsystem on only one

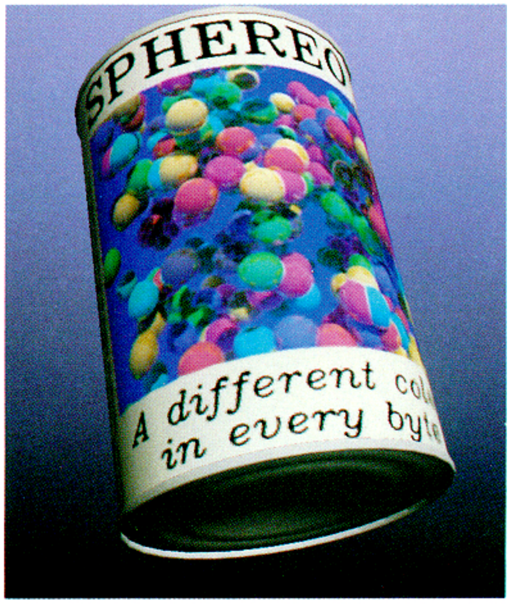


Figure 7.

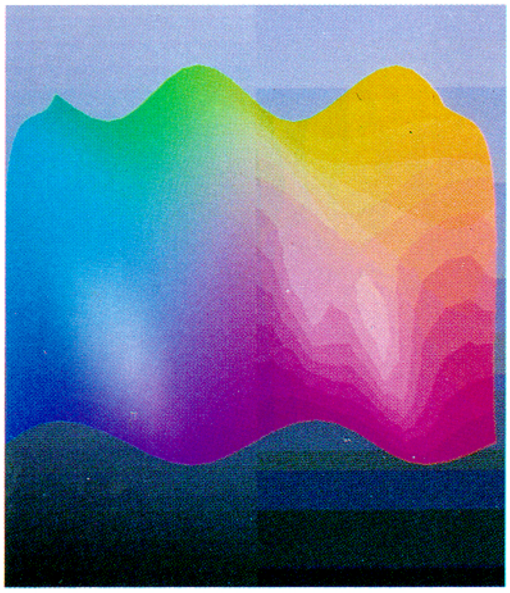


Figure 8.

board. Committing to these VRAMs was a major risk, since they had to be chosen several years before production chips became available. To reduce that risk, we chose to use only the basic features and cycles common to all potential vendors. This means omitting "block write," where 16 pixels are cleared by a single cycle, and not using any on-RAM ALU.

CONCLUSIONS

We built a competitive high-end graphics workstation, the DN10000VS, using a holistic approach which emphasized integration and simplification of the graphics hardware. Integration meant improving the CPU for graphics and then making it shoulder more of the load. This proved to be successful, with a considerable increase in performance for both graphics and customer code. Simplification is successful in the control logic and in address synthesis, avoiding the painful ordeal of microcode, but was not tractable for the frame buffer. Color synthesis is also not simplified; instead, we succumbed to the temptation to add several advanced rendering features, such as alpha buffering and texture mapping.

Multiprocessing proves to be a flexible way of scaling performance, for graphics as well as for applications. It improves a wide breadth of rendering methods and options since it is orthogonal to coding and CPU improvements; as such it contrasts sharply with many hardware accelerators which target a narrow and inextensible set of operations.

In hindsight, the hardware/software boundary was a bit too low. Calculating the color/Z/alpha differentials, adjusting the initial pixel value, and subpixel Bresenham addressing setup in hardware all would have improved the system balance. The 80-plane configuration has the untapped potential of twice the 40-plane bandwidth. Casting features in raw hardware without the possibility of microcode workarounds forced far more extensive simulation and many sleepless nights. Nonetheless, compared with the level of effort microcode entails, and its dangerous invitation to defer facing design complexity until coding time, we believe our approach superior for today's technology.

Finally, the holistic approach unravelled at the frame buffer and color synthesis logic, where raw bandwidth and sophisticated synthesis methods required expensive hardware. This is not a fundamental flaw with the approach but rather the recognition that at the fringes of any system are specialized tasks which defy integration.

ACKNOWLEDGEMENTS

We would like to thank all members of DN10000VS project and in particular the FANG design team, who worked together to make this effort succeed. We were assisted considerably by our anonymous reviewers, and by Victor Odryna and Casey Dowdell, who offered much constructive criticism. We truly appreciate Dr. William Ackerman's tireless efforts to achieve multiprocessing success. And we especially thank Dr. Terence Lindgren, who helped mightily in the presentation of these ideas, offering many insights into subpixel address synthesis and texture perspective approximation.

REFERENCES

Akeley, Kurt, T. Jermoluk, "High-Performance Polygon Rendering," *Computer Graphics*, Vol. 22, No. 4, August 1988, pp. 239-246.

Apgar, Brian, B. Bersack, A. Mammen, "A Display System for the Stellar Graphics Supercomputer Model GS1000," *Computer Graphics*, Vol. 22, No. 4, August 1988, pp. 255-262.

Apollo Computer, "Series 10000 Technical Reference Library: Volume 1 - Processors and Instruction Set," Order No. 0011720-A00, Apollo Computer Inc. 1988.

Bishop, Gary, D. Weimer, "Fast Phong Shading," *Computer Graphics*, Vol. 20, No. 4, July 1986, pp. 103-106.

Bresenham, Jack, "Algorithm for Computer Control of a Digital Plotter," *IBM Systems Journal* 4,1, 1965, pp. 25-30.

Deering, Michael, S. Winner, B. Schediwy, C. Duffy, N. Hunt, "The Triangle Processor and Normal Vector Shader: A

VLSI System for High Performance Graphics," *Computer Graphics*, Vol. 22, No. 4, July 1988, pp. 21-30.

Floyd, R. W., L. Steinberg, "An Adaptive Algorithm for Spatial Gray Scale," SID 75, International Symposium of Digital Technical Papers, 1975, 36.

Fuchs, Henry, J. Poulton, J. Eyles, T. Greer, J. Goldfeather, D. Ellsworth, S. Molnar, G. Turk, B. Tebbs, L. Israel, "Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor-Enhanced Memories," *Computer Graphics*, Vol. 23, No. 3, July 1989, pp. 79-88.

Gharachorloo, Nader, S. Gupta, E. Hokenek, P. Balasubramanian, B. Bogholtz, C. Mathieu, C. Zoulas, "Subnanosecond Pixel Rendering with Million Transistor Chips," *Computer Graphics*, Vol. 22, No. 4, July 1988, pp. 41-49.

Heckbert, Paul, "Survey of Texture Mapping," *IEEE Computer Graphics and Applications*, Vol. 6, No. 11, November, 1986, pp. 56-67.

Kirk, David, O. Lathrop, D. Voorhies, U. S. Patent Application for, "Quadratic Interpolation for Shaded Image Generation," U. S. Serial No. 077,202.

Lathrop, Olin, "Accurate Rendering by Subpixel Addressing," *IEEE Computer Graphics and Applications*, (to appear).

Lindgren, Terence, "Principles Guiding Line Generation and their Application to Start and End Rule Selection for Vector Drawing", *Private Communications*, 1988

Mach, Ernst, "The Analysis of Sensations and the Relation of the Physical to the Psychical," Dover Publications, New York, 1959.

Myer, T., I. Sutherland, "On the Design of Display Processors," *CACM*, Vol. 11, No. 6, June 1968, pp. 410-414.

Porter, Thomas, T. Duff, "Compositing Digital Images," *Computer Graphics*, Vol. 18, No. 3, July, 1984, pp. 253-259.

Potmesil, Michael, E. Hoffert, "The Pixel Machine: A Parallel Image Computer," *Computer Graphics*, Vol. 23, No. 3, July 1989, pp. 69-78.

Ratliff, Floyd, "Mach Bands: Quantitative Studies on Neural Networks in the Retina," Holden-Day, Inc., San Francisco, 1965.

Rhoden, Desi, C. Wilcox, "Hardware Acceleration for Window Systems," *Computer Graphics*, Vol. 23, No. 3, July 1989, pp. 61-67.

Swanson, Roger, L. Thayer, "A Fast Shaded-Polygon Renderer," *Computer Graphics*, Vol. 20, No. 4, August, 1986, pp. 95-101.

Torborg, John, "A Parallel Processor Architecture for Graphics Arithmetic Operations," *Computer Graphics*, Vol. 21, No. 4, July, 1987, pp. 197-204.

Williams, Lance, "Pyramidal Parametrics," *Computer Graphics*, Vol. 17, No. 3, July, 1983, pp. 1-11.

Voorhies, Douglas, "Reduced-Complexity Graphics," *IEEE Computer Graphics and Applications*, Vol. 9, No. 4, July, 1989, pp. 63-70.

Voorhies, Douglas, D. Kirk, O. Lathrop, "Virtual Graphics," *Computer Graphics*, Vol. 22, No. 4, August, 1988, pp. 247-253.