REVISED JULY 1999

WRL Research Report 98/1

Neon: A (Big) (Fast) Single-Chip 3D Workstation Graphics Accelerator

Joel McCormack
Robert McNamara
Christopher Gianos
Larry Seiler
Norman P. Jouppi
Ken Correll
Todd Dutton
John Zurawski

COMPAQ **Western Research Laboratory** 250 University Avenue Palo Alto, California 94301 USA

The Western Research Laboratory (WRL), located in Palo Alto, California, is part of Compaq's Corporate Research group. Our focus is research on information technology that is relevant to the technical strategy of the Corporation and has the potential to open new business opportunities. Research at WRL ranges from Web search engines to tools to optimize binary codes, from hardware and software mechanisms to support scalable shared memory paradigms to graphics VLSI ICs. As part of WRL tradition, we test our ideas by extensive software or hardware prototyping.

We publish the results of our work in a variety of journals, conferences, research reports and technical notes. This document is a research report. Research reports are normally accounts of completed research and may include material from earlier technical notes, conference papers, or magazine articles. We use technical notes for rapid distribution of technical material; usually this represents research in progress.

You can retrieve research reports and technical notes via the World Wide Web at:

http://www.research.digital.com/wrl/home

You can request research reports and technical notes from us by mailing your order to:

Technical Report Distribution
Compaq Western Research Laboratory
250 University Avenue
Palo Alto, CA 94301 U.S.A.

You can also request reports and notes via e-mail. For detailed instructions, put the word "Help" in the subject line of your message, and mail it to:

wrl-techreports@pa.dec.com

Neon: A (Big) (Fast) Single-Chip 3D Workstation Graphics Accelerator

Joel McCormack[1], Robert McNamara[2], Christopher Gianos[3], Larry Seiler[4], Norman P. Jouppi[1], Ken Correl[4], Todd Dutton[3], John Zurawski[3]

Revised July 1999

Abstract

High-performance 3D graphics accelerators traditionally require multiple chips on multiple boards. Specialized chips perform geometry transformations and lighting computations, rasterizing, pixel processing, and texture mapping. Multiple chip designs are often scalable: they can increase performance by using more chips. Scalability has obvious costs: a minimal configuration needs several chips, and some configurations must replicate texture maps. A less obvious cost is the almost irresistible temptation to replicate chips to increase performance, rather than to design individual chips for higher performance in the first place.

In contrast, Neon is a single chip that performs like a multichip design. Neon accelerates OpenGL 3D rendering, as well as X11 and Windows/NT 2D rendering. Since our pin budget limited peak memory bandwidth, we designed Neon from the memory system upward in order to reduce bandwidth requirements. Neon has no special-purpose memories; its eight independent 32-bit memory controllers can access color buffers, Z depth buffers, stencil buffers, and texture data. To fit our gate budget, we shared logic among different operations with similar implementation requirements, and left floating point calculations to Digital's Alpha CPUs. Neon's performance is between HP's Visualize fx[4] and fx[6], and is well above SGI's MXE for most operations. Neon-based boards cost much less than these competitors, due to a small part count and use of commodity SDRAMs.

[1] Compaq Computer Corporation Western Research Laboratory, 250 University Avenue, Palo Alto, CA 94301. [Joel.McCormack, Norm.Jouppi]@compaq.com

[2] Compaq Computer Corporation Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301. Bob.McNamara@Compaq.com

[3] Compaq Computer Corporation Alpha Development Group, 334 South Street, Shrewsbury, MA 01545-4172. [Chris.Gianos, Todd.Dutton, John.Zurawski] @Compaq.com

[4] At Digital Equipment Corporation (later purchased by Compaq) for the development of Neon, now at Real Time Visualization, 300 Baker Avenue, Suite #301, Concord, MA 01742. [seiler,correll]@rtviz.com

This report is a superset of *Neon: A Single-Chip 3D Workstation Graphics Accelerator*, published in the *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, August 1998, and *The Implementation of Neon: A 256-bit Graphics Accelerator*, published in the April/May issue of *IEEE Micro.*

1. Introduction

Neon borrows much of its design philosophy from Digital's Smart Frame Buffer [21] family of chips, in that it extracts a large proportion of the peak memory bandwidth from a unified frame buffer, accelerates only rendering operations, and efficiently uses a general-purpose I/O bus.

Neon makes efficient use of memory bandwidth by reducing page crossings, by prefetching pages, and by processing batches of pixels to amortize read latency and high-impedance bus turnaround cycles. A small texture cache reduces bandwidth requirements during texture mapping. Neon supports 32, 64, or 128 megabytes of 100 MHz synchronous DRAM (SDRAM). The 128 megabyte configuration has over 100 megabytes available for textures, and can store a 512 x 512 x 256 3D 8-bit intensity texture.

Unlike most fast workstation accelerators, Neon doesn't accelerate floating-point operations. Digital's 500 MHz 21164A Alpha CPU [7] transforms and lights 1.5 to 4 million vertices per second. The 600 MHz 21264 Alpha [12][16] should process 2.5 to 6 million vertices/second, and faster Alpha CPUs are coming.

Since Neon accepts vertex data after lighting computations, it requires as little as 12 bytes/vertex for (x, y) coordinate, color, and Z depth information. A well-designed 32-bit, 33 MHz Peripheral Component Interconnect (PCI) supports over 8 million such vertices/second; a 64-bit PCI supports nearly twice that rate. The 64-bit PCI transfers textures at 200 megabytes/second, and the 64 and 128 megabyte Neon configurations allow many textures to stay in the frame buffer across several frames. We thus saw no need for a special-purpose bus between the CPU and graphics accelerator.

Neon accelerates rendering of Z-buffered Gouraud shaded, trilinear perspective-correct texture-mapped triangles and lines. Neon supports antialiased lines, Microsoft Windows lines, and X11 [27] wide lines.

Performance goals were 4 million 25-pixel, shaded, Z-buffered triangles/second, 2.5 million 50-pixel triangles/second, and 600,000 to 800,000 50-pixel textured triangles/second. Early in the design, we traded increased gate count for reduced design time, which had the side-effect of increasing the triangle setup rate to over 7 million Gouraud shaded, Z-buffered triangles per second. This decision proved fortunate—applications are using ever smaller triangles, and the software team doubled their original estimates of vertex processing rates.

This paper, a superset of previous papers about Neon, discusses how our focus on efficiently using limited resources helped us overcome the constraints imposed by a single chip. We include much that is not novel, but many recent specifications and papers describe designs that perform incorrect arithmetic or use excessive amounts of logic. We therefore describe most of the techniques we used in Neon to address these issues.

2. Why a Single Chip?

A single chip's pin count constrains peak memory bandwidth, while its die size constrains gate count. But there are compensating implementation, cost, and performance advantages over a multichip accelerator.

A single-chip accelerator is easier to design. Partitioning the frame buffer across multiple chips forces copy operations to move data between chips, increasing complexity, logic duplication, and pin count. In contrast, internal wires switch faster than pins and allow wider interfaces (our Fragment Generator ships nearly 600 bits downstream). And changing physical pin interfaces is harder than changing internal wires.

A single-chip accelerator uses fewer gates, as operations with similar functionality can share generalized logic. For example, copying pixel data requires computing source addresses, reading data, converting it to the correct format, shifting, and writing to a group of destination addresses. Texture mapping requires computing source addresses, reading data, converting it, filtering, and writing to a destination address. In Neon, pixel copying and texture mapping share source address computation, a small cache for texel and pixel reads, read request queues, format conversion, and destination steering. In addition, pixel copies, texture mapping, and pixel fill operations use the same destination queues and source/destination blending logic. And unlike some PC accelerators, 2D and 3D operations share the same paths through the chip.

This sharing amplifies the results of design optimization efforts. For example, the chunking fragment generation described below in Section 5.2.5 decreases SDRAM page crossings. By making the chunk size programmable, we also increased the hit rate of the texture cache. The texture cache, in turn, was added to decrease texture bandwidth requirements—but also improves the performance of 2D tiling and copying overlay pixels.

A single-chip accelerator can provide more memory for texture maps at lower cost. For example, a fully configured RealityEngine replicates the texture map 20 times for the 20 rasterizing chips; you pay for 320 megabytes of texture memory, but applications see only 16 megabytes. A fully configured InfiniteReality [24] replicates the texture "only" four times—but each rasterizing board uses a redistribution network to fully connect 32 texture RAMs to 80 memory controllers. In contrast, Neon doesn't replicate texture maps, and uses a simple 8 x 8 crossbar to redistribute texture data internally. The 64 megabyte configuration has over 40 megabytes available for textures after allocating 20 megabytes to a 1280 x 1024 display.

3. Why a Unified Memory System?

Neon differs from many workstation accelerators in that it has a single general-purpose graphics memory system to store colors, Z depths, textures, and off-screen buffers.

The biggest advantage of a single graphics memory system is the dynamic reallocation of memory bandwidth. Dedicated memories imply a dedicated partitioning of memory bandwidth—and wasting of bandwidth dedicated to functionality currently not in use. If Z buffering or texture mapping is not enabled, Neon has more bandwidth for the operations that are enabled. Further, partitioning of bandwidth changes instantaneously at a fine grain. If texel fetches overlap substantially in a portion of a scene, so that the texture cache's hit rate is high, more bandwidth becomes available for color and Z accesses. If many Z buffer tests fail, and so color and Z data writes occur infrequently, more bandwidth becomes available for Z reads. This automatic allocation of memory bandwidth enables us to design closer to average memory bandwidth requirements than to the worst case.

A unified memory system offers flexibility in memory allocation. For example, using 16-bit colors rather than 32-bit colors gains 7.5 megabytes for textures when using a 1280 x 1024 screen.

A unified memory system offers greater potential for sharing logic. For example, the sharing of copy and texture map logic described above in Section 2 is possible only if textures and pixels are stored in the same memory.

A unified memory system has one major drawback—texture mapping may cause page thrashing as memory accesses alternate between texture data and color/Z data. Neon reduces such thrashing in several ways. Neon's deep memory request and reply queues fetch large batches of texels and pixels, so that switching between texel accesses and pixel accesses occurs infrequently. The texel cache and fragment generation chunking ensure that the texel request queues contain few duplicate requests, so that they fill up slowly and can be serviced infrequently. The memory controllers prefetch texel and pixel pages when possible to minimize switching overhead. Finally, the four SDRAM banks available on the 64 and 128 megabyte configurations usually eliminate thrashing, as texture data is stored in different banks from color/Z data. These techniques are discussed further in Section 4 below.

SGI's O2 [20] carries unification one step further, by using the CPU's system memory for graphics data. But roughly speaking, CPU performance is usually limited by memory latency, while graphics performance is usually limited by memory bandwidth, and different techniques must be used to address these limits. We believe that the substantial degradation in both graphics and CPU performance caused by a completely unified memory isn't worth the minor cost savings. This is especially true after the major memory price crash of 1998, and the minor crash of 1999, which have dropped SDRAM prices to under $1.00/megabyte.

4. Is Neon Just Another PC Accelerator?

A single chip connected to a single frame buffer memory with no floating point acceleration may lead some readers to conclude "Neon is like a PC accelerator." The dearth of hard data on PC accelerators makes it hard to compare Neon to these architectures, but we feel a few points are important to make.

Neon is in a different performance class from PC accelerators. Without floating point acceleration, PC accelerators are limited by the slow vertex transformation rates of Intel and x86-compatible CPUs. Many PC accelerators also burden the CPU with computing and sending slope and gradient information for each triangle; Neon uses an efficient packet format that supports strips, and computes triangle setup information directly from vertex data. Neon does not require the CPU to sort objects into different chunks like Talisman [3][28] nor does it suffer the overhead of constantly reloading texture map state for the different objects in each chunk.

Neon directly supports much of the OpenGL rendering pipeline, and this support is general and orthogonal. Enabling one feature does not disable other features, and does not affect performance unless the feature requires more memory bandwidth. For example, Neon can render OpenGL lines that are simultaneously wide and dashed. Neon supports all OpenGL 1.2 source/destination blending modes, and both exponential and exponential squared fog modes. All pixel and texel data are accurately computed, and do not use gross approximations such as a single fog or mip-map level per object, or a mip-map level interpolated across the object. Finally, all three 3D texture coordinates are perspective correct.

5. Architecture

Neon's performance isn't the result of any one great idea, but rather many good ideas—some old, some new—working synergistically. Some key components to Neon's performance are:

- a unified memory to reduce idle memory cycles,
- a large peak memory bandwidth (3.2 gigabytes/second with 100 MHz SDRAM),
- the partitioning of memory among 8 memory controllers, with fine-grained load balancing,
- the batching of fragments to amortize read latencies and bus turnaround cycles, and to allow prefetching of pages to hide precharge and row activate overhead,
- chunked mappings of screen coordinates to physical addresses, and chunked fragment generation, which reduce page crossings and increase page prefetching,
- a screen refresh policy that increases page prefetching,
- a small texel cache and chunked fragment generation to increase the cache's hit rate,
- deeply pipelined triangle setup logic and a high-level interface with minimal software overhead,
- multiple formats for vertex data, which allow software to trade CPU cycles for I/O bus cycles,
- the ability for applications to map OpenGL calls to Neon commands, without the inefficiencies usually associated with such direct rendering.

Section 5.1 below briefly describes Neon's major functional blocks in the order that it processes commands, from the bus interface on down. Sections 5.2 to 5.6, however, provide more detail in roughly the order we designed Neon, from the memory system on up. This order better conveys how we first made the memory system efficient, then constantly strove to increase that efficiency as we moved up the rendering pipeline.

5.1. Architectural Overview

Figure 1 shows a block diagram of the major functional units of Neon.

The PCI logic supports 64-bit transfers at 33 MHz. Neon can initiate DMA requests to read or write main memory.

The PCI logic forwards command packets and DMA data to the Command Parser. The CPU can write commands directly to Neon via Programmed I/O (PIO), or Neon can read commands from main memory using DMA. The parser accepts nearly all OpenGL [26] object types, including line, triangle, and quad strips, so that CPU cycles and I/O bus bandwidth aren't wasted by duplicated vertex data. Finally, the parser oversees DMA operations from the frame buffer to main memory via Texel Central.

The Fragment Generator performs object setup and traversal. The Fragment Generator uses half-plane edge functions [10][16][25] to determine object boundaries, and generates each object's fragments with a fragment "stamp" in an order that enhances the efficiency of the memory system. (A fragment contains the information required to paint one pixel.) Each cycle, the stamp generates a single textured fragment, a 2 x 2 square of 64-bit *RGBAZ* (red, green, blue, alpha transparency, *Z* depth) fragments, or up to 8 32-bit color or 32 8-bit color indexed fragments along a scanline. When generating a 2 x 2 block of fragments, the stamp interpolates six channels for each fragment: red, green, blue, alpha transparency, Z depth, and fog intensity. When generating a single texture-mapped fragment, the stamp interpolates eight additional channels: three texture

coordinates, the perspective correction term, and the four derivatives needed to compute the mip-mapping level of detail. Setup time depends upon the number of channels and the precision required by those channels, ranging from over 7 million triangles/second that are lit and Z-buffered, down to just over 2 million triangles/second that are trilinear textured, lit, fogged, and Z-buffered. The Fragment Generator tests fragments against four clipping rectangles (which may be inclusive or exclusive), and sends visible fragments to Texel Central.

Texel Central was named after Grand Central Station, as it provides a crossbar between memory controllers. Any data that is read from the frame buffer in order to derive data that is written to a different location goes through Texel Central. This includes texture mapping, copies within the frame buffer, and DMA transfers to main memory. Texel Central also expands a row of an internal 32 x 32 bitmap or an externally supplied 32 bit word into 256 bits of color information for 2D stippled fill operations, expanding 800 million 32-bit *RGBA* fragments/second or 3.2 billion 8-bit color indexed fragments/second.

Texture mapping is performed at a peak rate of one fragment per cycle *before* a Pixel Processor tests the Z value. This wastes bandwidth by fetching texture data that are obscured, but pre-textured fragments are about 350 bits and post-textured fragments are about 100 bits. We couldn't afford more and wider fragment queues to texture map after the Z depth test. Further, OpenGL semantics don't allow updating the Z buffer until after texture mapping, as a textured fragment may be completely transparent. Such a wide separation between reading and writing Z values would significantly complicate maintaining frame buffer consistency, as described in Section 5.2.2 below. Finally, distributing pretextured fragments to the Memory Controllers, then later texturing only the visible fragments would complicate maintaining spatial locality of texture accesses, as described in Section 5.3.4 below.

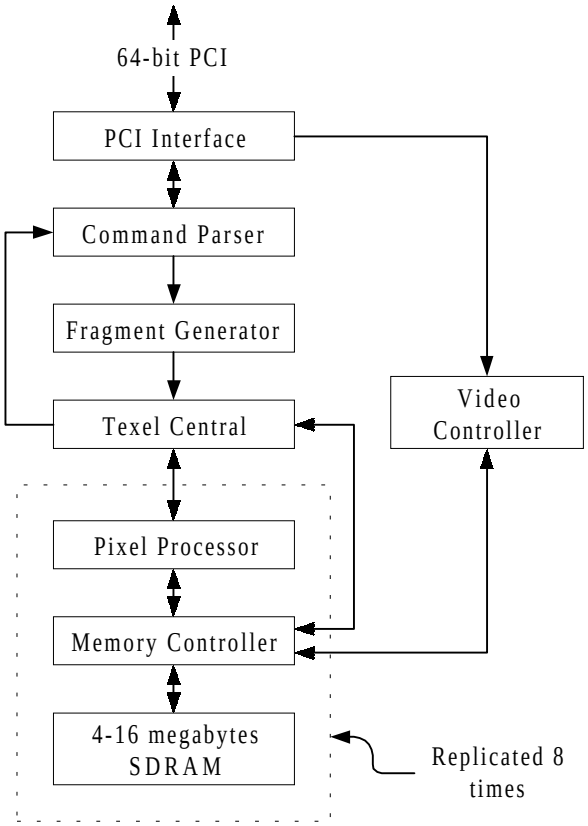

Figure 1: Neon block diagram

Texel Central feeds fragments to the eight Pixel Processors, each of which has a corresponding Memory Controller. The Pixel Processors handle the back end of the OpenGL rendering pipeline: alpha, stencil, and Z depth tests; fog; source and destination blending (including raster ops and OpenGL 1.2 operations like minimum and maximum); and dithering.

The Video Controller refreshes the screen, which can be up to 1600 x 1200 pixels at 76 Hz, by requesting pixel data from each Memory Controller. Each controller autonomously reads and interprets overlay and display format bytes. If a pixel's overlay isn't transparent, the Memory Controller immediately returns the overlay data; otherwise it reads and returns data from the front, back, left, or right color buffer. The Video Controller sends low color depth pixels (5/5/5 and 4/4/4) through "inverse dithering" logic [5], which uses an adaptive digital filter to restore much of the original color information. Finally, the controller sends the filtered pixels to an external RAMDAC for conversion to an analog video signal.

Neon equally partitions frame buffer memory among the eight Memory Controllers. Each controller has five request queues: Source Read Request from Texel Central, Pixel Read and Pixel Write Request from its Pixel Processor, and two Refresh Read Requests (one for each SDRAM bank) from the Video Controller. Each cycle, a Memory Controller services a request queue using heuristics that reduce wasted memory cycles.

A Memory Controller owns all data associated with a pixel, so that it can process rendering and screen refresh requests independently of the other controllers. Neon stores the front/back/left/right buffers, Z, and stencil buffers for a pixel in a group of 64 bits or 128 bits, depending upon the number of buffers and the color depth. To improve 8-bit 2D rendering speeds and to decrease screen refresh overhead, a controller stores a pixel's overlay and display format bytes in a packed format on a different page.

5.2. Pixel Processors and Memory Controllers

Neon's design began with the Pixel Processors and Memory Controllers. We wanted to effectively use the SDRAM's large peak bandwidth by maximizing the number of controllers, and by reducing read/write turnaround overhead, pipeline stalls due to unbalanced loading of the controllers, and page crossing overhead.

5.2.1. Memory Technology

We evaluated several memory technologies. We quickly rejected extended data out (EDO) DRAM and RAMBUS RDRAM due to inadequate performance (the pre-Intel RAMBUS protocol is inefficient for the short transfers we expected), EDO VRAM due to high cost, and synchronous graphic RAM (SGRAM) due to high cost and limited availability. This left synchronous DRAM (SDRAM) and 3D-RAM.

3D-RAM [6], developed by Sun and Mitsubishi, turns read/modify/write operations into write-only operations by performing Z tests and color blending inside the memory chips. The authors claim this feature gives it a "3-4x performance advantage" over conventional DRAM technology at the same clock rate, and that its internal caches further increase performance to "several times faster" than conventional DRAM.

We disagree. A good SDRAM design is quite competitive with 3D-RAM's performance. Batching eight fragments reduces read latency and high-impedance bus turnaround overhead to ½ cycle per fragment. While 3D-RAM requires color data when the Z test fails, obscured fragment writes never occur to SDRAM. In a scene with a depth complexity of three (each pixel is covered on average by three objects), about 7/18 of fragments fail the Z test. Factoring in batching and Z failures, we estimated 3D-RAM's rendering advantage to be a modest 30 to 35%. 3D-RAM's support for screen refresh via a serial read port gives it a total performance advantage of about 1.8-2x SDRAM. 3D-RAM's caches didn't seem superior to intelligently organizing SDRAM pages and prefetching pages into SDRAM's multiple banks; subsequent measurement of a 3D-RAM-based design confirmed this conclusion.

3D-RAM has several weaknesses when compared to SDRAM. It does not use 3-input multipliers like those described below in Section 5.3.7, so many source and destination blends require two cycles. (Some of these blends can be reduced to one cycle if the graphics chip does one of the two multiplies per channel.) Blending is limited to adding the source and destination factors: subtraction, min, and max aren't supported. 3D-RAM's blending logic incorrectly processes 8-bit data using base 256 arithmetic, rather than OpenGL's base 255 arithmetic (see Section 5.2.6 below). 3D-RAM computes the product $FF_{16} \times FF_{16}$ as FE_{16}, and so thinks that $1 \times 1 < 1$! 4/4/4/4 color pixels (four bits each of red, green, blue, and alpha transparency) suffer more severe arithmetic errors; worse, 3D-RAM cannot dither high-precision color data down to 4/4/4/4, leading to banding artifacts when blending. Support for 5/6/5 or 5/5/5/1 color is almost nonexistent. Working around such deficiencies wastes space and time, as the graphics accelerator must duplicate logic, and 3D-RAM sports a slow 20 nsec read cycle time.

3DRAM does not take a Z/color pair in sequential order; the pair is presented to separate 3DRAM chips, and a Z buffer chip communicates the result of the Z test to a corresponding color data chip. As a result, half the data pins sit idle when not Z buffering.

3D-RAM parts are 10 megabits—the RAM is 5/8 populated to make room for caches and for Z compare and blending logic. This makes it hard to support anything other than 1280 x 1024 screens. 3D-RAM is 6 to 10 times more expensive per megabyte than SDRAM. Finally, we'd need a different memory system for texture data. The performance advantage during Z buffering didn't outweigh these problems.

5.2.2. Fragment Batching and Overlaps

Processing fragments one at a time is inefficient, as each fragment incurs the full read latency and high impedance bus turnaround cycle overhead. Batch processing several fragments reduces this overhead to a reasonable level. Neon reads all Z values for a batch of fragments, compares each to the corresponding fragment's Z value, then writes each visible fragment's Z and color values back to the frame buffer.

Batching introduces a read/write consistency problem. If two fragments have the same pixel address, the second fragment must not use stale Z data. Either the first Z write must complete before the second Z read occurs, or the second Z "read" must use an internal bypass. Since it is rare for overlaps to occur closely in time, we found it acceptable to stop reading pixel data until the first fragment's write completes. (This simplifying assumption does not hold for anti-aliasing graphics accelerators, which generate two or more fragments at the same location along adjoining object edges.)

We evaluated several schemes to create batches with no overlapping fragments, such as limiting a batch to a single object; all these resulted in average batch lengths that were unacceptably short. We finally designed a fully associative eight-entry overlap detector per Memory Controller, which normally creates batches of eight fragments. (The size of the batch detector is matched to the total buffering capacity for writing fragments.) The overlap detector terminates a batch and starts a new batch if an incoming fragment has the same screen address as an existing fragment in the batch, or if the overlap detector is full. In both cases, it marks the first fragment in the new batch, and "forgets" about the old batch by clearing the associative memory. When a memory controller sees a fragment with a "new batch" mark, it writes all data associated with the current batch before reading data for the new batch. Thus, the overlap detector need not keep track of all unretired fragments further down the pixel processing pipeline.

To reduce chip real estate for tags, we match against only the two bank bits and the column address bits of a physical address. This aliases all pairs of A and B banks, as shown in Figure 2. Note how the red triangle spans four physical pages, and how its fragments are aliased into two pages. If two fragments are in the same position on different pages in the same SDRAM bank, the detector falsely flags an overlap. For example, the blue triangle appears to

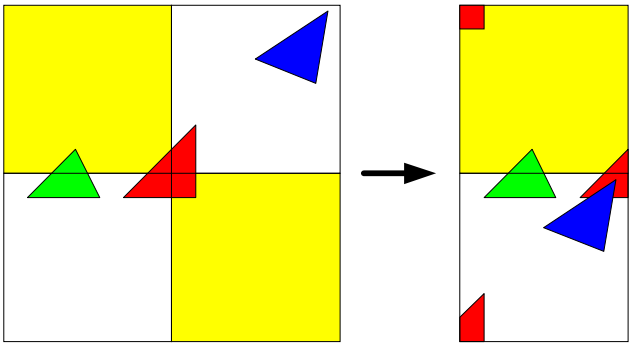

Figure 2: The partial tag compare aliases all pairs of A and B bank pages, sometimes creating false overlaps

overlaps the red triangle in the aliased tag space. This "mistake" can actually *increase* performance. In such cases, it is usually faster to terminate the batch, and so turn the bus around twice to complete all work on the first page and then complete all work on the second page, than it is to bounce twice between two pages in the same bank (see Section 5.2.4 below).

5.2.3. Memory Controller Interleaving

Most graphics accelerators load balance memory controllers by interleaving them in one or two dimensions, favoring either screen refresh or rendering operations. An accelerator may cycle through all controllers across a scanline, so that screen refresh reads are load balanced. This one-dimensional interleaving pattern creates vertical strips of ownership, as shown in Figure 3. Each square represents a pixel on the screen; the number inside indicates which memory controller owns the pixel.

The SGI RealityEngine [1] has as many as 320 memory controllers. To improve load balancing during rendering, the RealityEngine horizontally and vertically tiles a 2D interleave pattern, as shown in Figure 4. Even a two-dimensional pattern may have problems load balancing the controllers. For example, if a scene has been tessellated into vertical triangle strips, and the 3D viewpoint maintains this orientation (as in an architectural walk-through), a subset of the controllers get overworked.

Neon load balances controllers for both rendering and screen refresh operations by rotating a one-dimensional interleaving pattern by two pixels from one scanline to the next, as shown in Figure 5. This is also a nice pattern for texture maps, as any 2 x 2 block of texels resides in different memory controllers. (The SGI InfiniteReality [24] uses a rotated pattern like Neon within a single rasterizing board, but does not rotate the 2-pixel wide vertical strips owned by each of the four rasterizing boards, and so has the same load balancing problems as an 8-pixel wide non-rotated interleave.)

In retrospect, Neon nicely balances work among the Memory Controllers, but at such a fine grain that the controllers make too many partially prefetched page crossings. Small objects tend to include only a few locations on a

	0	1	2	3	4	5	6	7	0	1
	0	1	2	3	4	5	6	7	0	1
	0	1	2	3	4	5	6	7	0	1

Figure 3: Typical 1D pixel interleaving

	0	1	2	3	4	5	6	7	0	1
	8	9	10	11	12	13	14	15	8	9
	0	1	2	3	4	5	6	7	0	1

Figure 4: Typical 2D pixel interleaving

	0	1	2	3	4	5	6	7	0	1
	2	3	4	5	6	7	0	1	2	3
	4	5	6	7	0	1	2	3	4	5
	6	7	0	1	2	3	4	5	6	7
	0	1	2	3	4	5	6	7	0	1

Figure 5: Neon's rotated pixel interleaving

given page in each controller. Narrow vertical triangle strips exacerbate the problem, as Neon's pages are usually wide but not very high (see Section 5.2.4 below). Consequently, for such triangles the controllers frequently cannot hide all of the precharge & row activate overhead when switching banks.

Making each square in Figure 5 represent a 2 x 2 or even a 4 x 4 pixel area increases memory efficiency by increasing the number of pixels some controllers access on a page, while hopefully reducing to zero the number of pixels other controllers access on that page. This larger granularity still distributes work evenly among controllers, but requires a much larger screen area to average out the irregularities. This in turn requires increased fragment buffering capacity in the Memory Controllers, in order to prevent starvation caused by one or more controllers emptying their incoming fragment queues. We couldn't afford larger queues in Neon, but newer ASICs should have enough real estate to remedy this inefficiency.

5.2.4. SDRAM Page Organization

SDRAM's have two or four banks, which act as a two or four entry direct mapped page cache. A page of SDRAM data must be loaded into a bank with a row activate command before reading from the page. This load is destructive, so a bank must be written back with a pre-

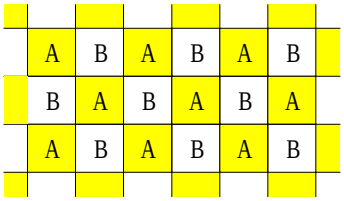

Figure 6: Page interleaving with two banks

charge command before loading another page into the bank. These commands take several cycles, so it is desirable to access as much data as possible on a page before moving to a new page. It is possible to prefetch a page into one bank—that is, precharge the old page and row activate a new page—while reading or writing data to a different bank. Prefetching a page early enough hides the prefetch latency entirely.

Neon reduces the frequency of page crossings by allocating a rectangle of pixels to an SDRAM page. Object rendering favors square pages, while screen refresh favors wider pages. Neon keeps screen refresh overhead low by allocating on-screen pages with at worst an 8 x 1 aspect ratio, and at best a 2 x 1 aspect ratio, depending upon pixel size, number of color buffers, and SDRAM page size. Texture maps and off-screen buffers, with no screen refresh constraints, use pages that are as square as possible. Three-dimensional textures use pages that are as close to a cube of texels as possible.

In the 32 megabyte configuration, each Memory Controller has two banks, called A and B. Neon checkerboards pages between the two banks, as shown in Figure 6. All horizontal and vertical page crossings move from one bank to the other bank, enhancing opportunities for prefetching.

In the 64 and 128 megabyte configurations, each controller has four banks. Checkerboarding all four banks doesn't improve performance sufficiently to warrant the complication of prefetching two or three banks in parallel. Instead, these configurations assign two banks to the bottom half of memory, and the other two banks to the top half. Software preferentially allocates pixel buffers to the bottom two banks, and texture maps to the top two banks, to eliminate page thrashing between drawing buffer and texture map accesses.

5.2.5. Fragment Generation Chunking

Scanline-based algorithms generate fragments in an order that often prohibits or limits page prefetching. Figure 7 shows a typical fragment generation order for a triangle that touches four pages. The shaded pixels belong to bank A. Note how only the four fragments numbered 0 through 3 access the first A page before fragment 4 accesses the B page, which means that the precharge and row activate overhead to open the first B page may not be completely hidden. Note also that fragment 24 is on the first B page, while fragment 25 is on the second B page. In this case the page transition cannot be hidden at all.

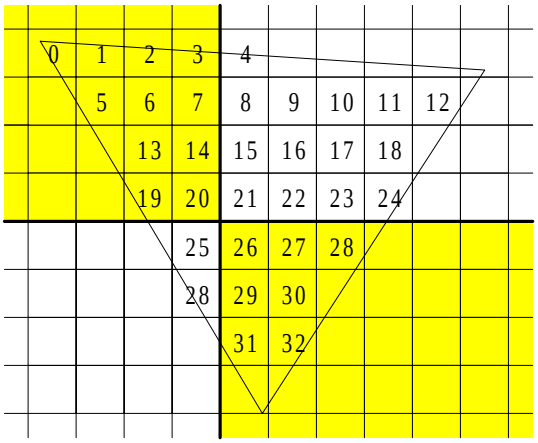

Figure 7: Scanline fragment generation order

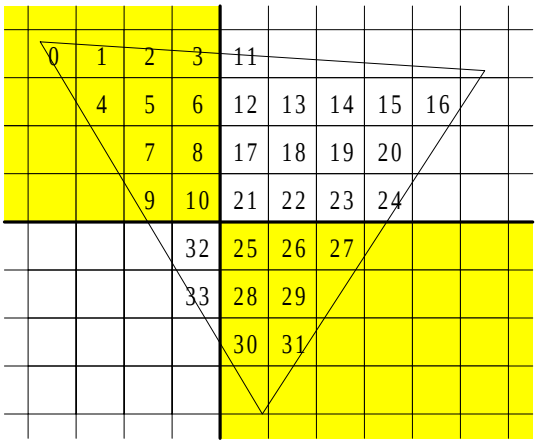

Figure 8: Neon's chunking fragment generation order

To further increase locality of reference, the Fragment Stamp generates an object in rectangular "chunks." When not texture mapping, a chunk corresponds to a page, so that the stamp generates an object's fragments one page at a time. This decreases page crossings, and gives the maximum possible time to prefetch the next page. Figure 8 shows the order in which Neon generates fragments for the same triangle. Note how the "serpentine" order in which chunks are visited further increases the number of page crossings that can exploit prefetching.

5.2.6. Repeated Fraction Arithmetic

We concentrated not only upon the efficiency of pixel processing, but also upon arithmetic accuracy. Since many designs do not blend or dither pixel values correctly, we describe the arithmetic behind these operations in this and the next section.

If the binary point is assumed to be to the left of an n-bit fixed point color value, the value represents a discrete number in the inclusive range $[0, 1 - 2^{-n}]$. However, OpenGL and common sense require that the number 1 be representable. We can accomplish this by dividing an n-bit

value by 2^n-1 rather than by 2^n. This is not as difficult as it sounds: $v_n/(2^n\text{-}1)$ is representable in binary form by infinitely repeating the n-bit number v_n to the right of the binary point. This led us to refer to such numbers as "repeated fractions."

Jim Blinn provides a detailed description of repeated fraction numbers in [4]. Briefly, ordinary binary arithmetic is inadequate for multiplication. The product's implicit divisor is $(2^n\text{-}1)^2$, and so the product must be converted to a bit pattern whose implicit divisor is 2^n-1. Simply rounding the product to n bits is equivalent to dividing by 2^n rather than by 2^n-1, and so biases the result toward 0. This is why 3D-RAM computes $1 \times 1 < 1$. If multiple images or transparent surfaces are composited with this erroneous bias, the resulting color may be significantly darker than desired.

We can use ordinary binary arithmetic to compute the repeated fraction product p of two n-bit repeated fraction numbers a and b:

$q = a*b + 2^{n-1}$;
$p = (q + (q >> n)) >> n$;

This adjustment can be implemented with an extra carry-propagate adder after the multiply, inside the multiplier by shifting two or more partial sums, or as part of the dithering computations described in the following section.

5.2.7. Dithering

Dithering is a technique to spread errors in the reduction of high-precision n-bit numbers to lower-precision m-bit numbers. If we convert an n-bit number v_n to an m-bit number v_m by rounding (adding ½ and truncating) and shifting:

$$v_m = (v_n + 2^{n-m-1}) >> (n\text{-}m)$$

we will probably see color banding if m is less than about 8 to 10 bits, depending upon room lighting conditions. Large areas are a constant color, surrounded by areas that are a visibly different constant color.

Instead of adding the constant rounding bit 2^{n-m-1}, a dithering implementations commonly add a variable rounding value $d(x, y)$ in the half-open range from [0, 1). (Here and below, we assume that d has been shifted to the appropriate bit position in the conversion.) The rounding value is usually computed as a function of the bottom bits of the (x, y) position of the pixel, and must have an average value of 0.5 when evaluated over a neighborhood of nearby (x, y) positions. Dithering converts the banding artifacts to noise, which manifests itself as graininess. If too few bits of x and y are used to compute d, or if the dither function is too regular, dithering also introduces dither matrix artifacts, which manifest themselves at repeated patterns of darker and lighter pixels.

The above conversion is correct for binary numbers, but not for repeated fractions. We can divide the repeated fraction computations into two parts. First, compute the real number in the closed interval [0, 1] that the n-bit number represents:

$$r = v_n / (2^n - 1)$$
$$= 0.\, v_n\, v_n\, v_{n\ldots} \text{ (base 2)}$$

Next, convert this into an m-bit number:

$$v_m = \text{floor}(r * (2^m - 1) + d(x, y))$$
$$= \text{floor}((r << m) - r) + d(x, y))$$

Rather than convert the binary product to a repeated fraction number, then dither that result, Neon combines the repeated fraction adjustment with dithering, so that dithering operates on the 2n-bit product. Neon approximates the above conversions to a high degree of accuracy with:

$q = a*b$;
$v_m = (q + (q >> (n-1)) - (q >> m) - (q >> (m+n-1)) + d(x, y) + 2^{n-e-1}) >> (2*n - m)$

Similar to adding a rounding bit (i.e. 2^{n-1}) below the top n bits as in Section 5.2.6 above, here we add a rounding bit 2^{m-e-1} below the dither bits. The value e represents how far the dither bits extend past the top m bits of the product. Neon computes 5 unique dither bits, and expands these by replication if needed so that they extend 6 bits past the top 8 bits of the product.

Finally, certain frame buffer operations should be idempotent. In particular, if we read a low-precision m-bit repeated fraction number from the frame buffer into a high-precision n-bit repeated fraction register, multiply by 1.0 (that is, 2^n–1), dither, and write the result back, we should not change the m-bit value. If n is a multiple of m, this happens automatically. But if, for example, n is 8 and m is 5, certain m-bit values will change. This is especially true if 5-bit values are converted to 8-bit values by replication [31], rather than to the closest 8-bit value. Our best solution to this problem was to clamp the dither values to lie in the half-open interval $[\varepsilon(m, n), 1 - \varepsilon(m, n))$, where ε is relatively small. For example $\varepsilon(5, 8)$ is 3/32.

5.3. Texel Central

Texel Central is the kitchen sink of Neon. Since it is the only crossbar between memory controllers, it handles texturing and frame buffer copies. Pixel copying and texture mapping extensively share logic, including source address computation, a small cache for texel and pixel reads, read request queues, format conversion, and destination steering. Since it has full connectivity to the Pixel Processors, it expands a row of the internal 32 x 32 bitmap or an externally supplied bitmap to foreground and background colors for transparent or opaque stippling.

The subsections below describe the perspective divide pipeline, a method of computing OpenGL's mip-mapping level of detail with high accuracy, a texture cache that reduces memory bandwidth requirements with fewer gates than a traditional cache, and the trilinear filtering multiplier tree.

5.3.1. Perspective Divide Pipeline

Exploiting Heckbert and Moreton's observations [14], we interpolate the planar (affine) texture coordinate channels $u' = u/q$, $v' = v/q$, and $w' = w/q$. For each textured fragment, we must then divide these by the planar perspective channel $q' = 1/q$ to yield the three-dimensional perspective-correct texture coordinates (u, v, w). Many implementations compute the reciprocal of $1/q$, then perform three multiplies. We found that a 12-stage, 6-cycle divider pipeline was both smaller and faster. This is because we use a small divider stage that avoids propagating carries as it accumulates the quotient, and we decrease the width of each stage of the divider.

The pipeline is built upon a radix-4 non-restoring divider stage that yields two bits of quotient. A radix-4 divider has substantial redundancy (overlap) in the incremental quotient bits we can choose for a given dividend and divisor. A typical radix-4 divider [11] exploits this redundancy to restrict quotients to 0, ±1, and ±2, avoiding quotients of ±3 so that a 2-input adder can compute the new partial remainder. This requires a table indexed by five remainder bits and three divisor bits (excluding the leading 1 bit) to choose two new quotient bits. It also means that when a new negative quotient is added to the previous partial quotient, the carry bit can propagate up the entire sum.

Neon instead exploits the redundancy to avoid an incremental quotient of 0, and uses a 3-input adder to allow an incremental quotient of ±3. This simplifies the table lookup of new quotient bits, requiring just three partial remainder bits and one divisor bit (excluding the leading 1). It also ensures that the bottom two bits of the partial quotient can never be 00, and so when adding new negative quotient bits to the previously computed partial quotient, the carry propagates at most one bit. Here are the three cases where the (unshifted) previous partial quotient ends in 01, 10, and 11, and the new quotient bits are negative.

```
  ab0100      ab1000      ab1100
+ 1111xy    + 1111xy    + 1111xy
  ab00xy      ab01xy      ab10xy
```

Neon does not compute the new partial remainders, nor maintain the divisor, to the same accuracy throughout the divide pipeline. After the third 2-bit divider stage, their sizes are reduced by two bits each stage. This results in an insignificant loss of accuracy, but a significant reduction in gate count.

5.3.2. Accurate Level of Detail Computation

Neon implements a more accurate computation of the mip-mapping [32] level of detail (*LOD*) than most hardware. The *LOD* is used to bound, for a given fragment, the instantaneous ratio of movement in the texture map coordinate space (u, v) to movement in screen coordinate space (x, y). This avoids aliasing problems caused by undersampling the texture data.

Computing OpenGL's desired *LOD* requires determining the distances moved in the texture map in the u and v directions as a function of moving in the x and y directions on the screen. That is, we must compute the four partial derivatives $\partial u/\partial x$, $\partial v/\partial x$, $\partial u/\partial y$, and $\partial v/\partial y$.

If $u'(x, y)$, $v'(x, y)$, and $q'(x, y)$ are the planar functions $u(x, y)/q(x, y)$, $v(x, y)/q(x, y)$, and $1/q(x, y)$, then:

$$\partial u/\partial x = (q'(x, y) * \partial u'/\partial x - u'(x, y) * \partial q'/\partial x) / q'(x, y)^2$$
$$\partial v/\partial x = (q'(x, y) * \partial v'/\partial x - v'(x, y) * \partial q'/\partial x) / q'(x, y)^2$$
$$\partial u/\partial y = (q'(x, y) * \partial u'/\partial y - u'(x, y) * \partial q'/\partial y) / q'(x, y)^2$$
$$\partial v/\partial y = (q'(x, y) * \partial v'/\partial y - v'(x, y) * \partial q'/\partial y) / q'(x, y)^2$$

(We've dropped the dependency on x and y for terms that are constant across an object.) The denominator is the same in all four partial derivatives. We don't compute $q'(x, y)^2$ and divide, as suggested in [8], but instead implement these operations as a doubling and a subtraction of $\log_2(q')$ after the $\log_2$ of the lengths described below.

The numerators are planar functions, and thus it is relatively easy to implement setup and interpolation hardware for them. If an application specifies a mip-mapping texture mode, Neon computes numerators from the vertex texture coordinates, with no additional software input.

Neon uses the above partial derivative equations to compute initial values for the numerators using eight multiplies, in contrast to the 12 multiplies described in [8]. The setup computations for the x and y increments use different equations, which are obtained by substituting the definitions for $u'(x, y)$, $v'(x, y)$, and $q'(x, y)$, then simplifying:

$$\partial u/\partial x = ((\partial q'/\partial y * \partial u'/\partial x - \partial q'/\partial x * \partial u'/\partial y) * y + q'(0, 0) * \partial u'/\partial x - u'(0,0) * \partial q'/\partial x) / q'(x, y)^2$$
$$\partial v/\partial x = ((\partial q'/\partial y * \partial v'/\partial x - \partial q'/\partial x * \partial v'/\partial y) * y + q'(0, 0) * \partial v'/\partial x - v'(0,0) * \partial q'/\partial x) / q'(x, y)^2$$
$$\partial u/\partial y = ((\partial q'/\partial x * \partial u'/\partial y - \partial q'/\partial y * \partial u'/\partial x) * x + q'(0, 0) * \partial u'/\partial y - u'(0,0) * \partial q'/\partial y) / q'(x, y)^2$$
$$\partial v/\partial y = ((\partial q'/\partial x * \partial v'/\partial y - \partial q'/\partial y * \partial v'/\partial x) * x + q'(0, 0) * \partial v'/\partial y - v'(0,0) * \partial q'/\partial y) / q'(x, y)^2$$

First, note that the numerators of $\partial u/\partial x$ and $\partial v/\partial x$ depend only upon y, and that $\partial u/\partial y$ and $\partial v/\partial y$ depend only upon x. Second, note that the $\partial u/\partial y$ and $\partial v/\partial y$ x increments are the negation of the $\partial u/\partial x$ and $\partial v/\partial x$ y increments, respectively. Finally, we don't need the constant offsets—the initial values of the numerators take them into account. We thus use four multiplies to obtain two increments.

OpenGL next determines the length of the two vectors $(\partial u/\partial x, \partial v/\partial x)$ and $(\partial u/\partial y, \partial v/\partial y)$, takes the maximum length, then takes the base 2 logarithm:

$$LOD = \log_2(\max(\mathrm{sqrt}((\partial u/\partial x)^2 + (\partial v/\partial x)^2), \mathrm{sqrt}((\partial u/\partial y)^2 + (\partial v/\partial y)^2)))$$

Software does four multiplies for the squares, and converts the square root to a divide by 2 after the $\log_2$.

Note that this *LOD* computation *requires* the computation of all four derivatives. The maximum can change from one square root to the other within a single object. Accelerators that whittle the *LOD* computation down to a

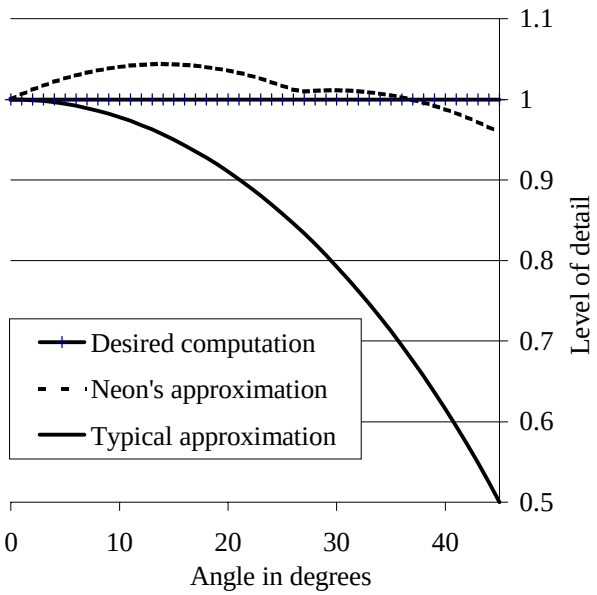

Figure 9: Various level of detail approximations

single interpolated channel may incur substantial errors, and cannot comply with OpenGL's lax requirements.

OpenGL allows implementations to compute the *LOD* using gross approximations to the desired computation. Hardware commonly takes the maximum of the partial derivative magnitudes:

$$LOD = \log_2(\max(\text{abs}(\partial u/\partial x), \text{abs}(\partial v/\partial x), \text{abs}(\partial u/\partial y), \text{abs}(\partial v/\partial y)))$$

This can result in an *LOD* that is too low by half a mipmap level, an error which reintroduces the aliasing artifacts that mip-mapping was designed to avoid.

Neon uses a two-part linear function to approximate the desired distances. Without loss of generality, assume that $a > 0$, $b > 0$, $a > b$. The function:

if ($b < a/2$) return $a + b/4$ else return $7a/8 + b/2$

is within ± 3% of $\text{sqrt}(a^2 + b^2)$. This reduces the maximum error to about ±0.05 mipmap levels—a ten-fold increase in accuracy over typical implementations, for little extra hardware. The graph in Figure 9 shows three methods of computing the level of detail as a texture mapped square on the screen rotates from 0° through 45°. In this example, the texture map is being reduced by 50% in each direction, and so the desired *LOD* is 1.0. Note how closely Neon's implementation tracks the desired *LOD*, and how poorly the typical implementation does.

5.3.3. Texel Cache Overview

Texel Central has eight fully associative texel caches, one per memory controller. These are vital to texture mapping performance, since texel reads steal bandwidth from other memory transactions. Without caching, the 8 texel fetches per cycle for trilinear filtering require the entire peak bandwidth of memory. Fortunately, many texel fetches are redundant; Hakura & Gupta [13] found that each trilinearly filtered texel is used by an average of four fragments. Each cache stores 32 bytes of data, so holds 8 32-bit texels, 16 16-bit texels, or 32 8-bit texels. Neon's total cache size is a mere 256 bytes, compared to the 16 to 128 *kilo*byte texel caches described in [13]. Our small cache size works well because chunking fragment generation improves the hit rate, the caches allow many more outstanding misses than cache lines, the small cache line size of 32 bits avoids fetching of unused data, and we never speculatively fetch cache lines that will not be used.

The texel cache also improves rendering of small X11 and Windows 2D tiles. An 8 x 8 tile completely fits in the caches, so once the caches are loaded, Texel Central generates tiled fragments at the maximum fill rate of 3.2 gigabytes per second. The cache helps larger tiles, too, as long as one scanline of the tile fits into the cache.

5.3.4. Improving the Texel Cache Hit Rate

In order to avoid capacity misses in our small texel cache, fragments that are close in 2D screen space must be generated closely in time. Once again, scanline-based fragment generation is non-optimal. If the texel requirements of one scanline of a wide object exceed the capacity of the cache, texel overlaps across adjacent scanlines are not captured by the cache, and performance degrades to that of a single-line cache. Scanline generators can alleviate this problem, but not eliminate it. For example, fragment generation may proceed in a serpentine order, going left to right on one scanline, then right to left on the next. This always captures *some* overlap between texel fetches on different scanlines at the edges of a triangle, but also halves the width at which cache capacity miss problems appear.

Neon attacks this problem by exploiting the chunking fragment generation described in Section 5.2.5 above. When texturing, Neon matches the chunk size to the texel cache size. Capacity misses still occur, but usually only for fragments along two edges of a chunk. Neon further reduces redundant fetches by making chunks very tall and one pixel wide (or vice versa), so that redundant fetches are mostly limited to the boundaries between chunk rows.

Figure 10 shows fragment generation order for texture mapping, where the chunks are shown as 4 x 1 for illustration purposes. (Chunks are actually 8 x 1 for 32-bit and 16-bit texels, and 16 x 1 for 8-bit texels.) The chunk boundaries are delineated with thick lines. Neon restricts chunks to be aligned to their size, which causes triangles to be split into more chunk rows than needed. Allowing chunks to be aligned to the stamp size (which is 1 x 1 when texturing) would eliminate this inefficiency: the top of the triangle would then start at the top of the first chunk row, rather than some point inside the row.

If each texel is fetched on behalf of four fragments, chunking reduces redundant fetches in large triangles by nearly a factor of 8, and texel read bandwidth by about 35%, when compared to a scanline fragment generator.

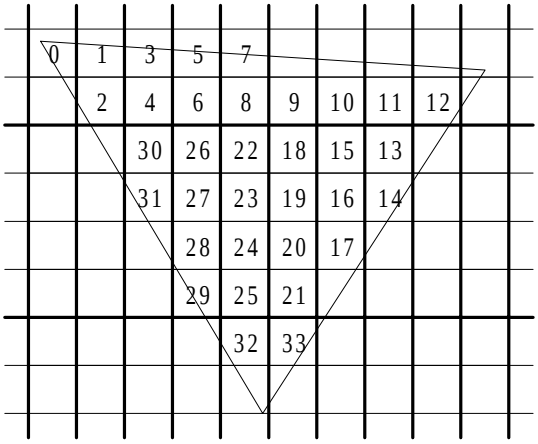

Figure 10: Chunking improves the texel cache hit rate

5.3.5. Texel Cache Operation

A texel cache must not stall requests after a miss, or performance would be worse than not using a cache at all! Further, the cache must track a large number of outstanding misses—since several other request queues are vying for the memory controller's attention, a miss might not be serviced for tens of cycles.

A typical CPU cache requires too much associative logic per outstanding miss. By noting that a texel cache should always return texels in the same order that they were requested, we eliminated most of the associative bookkeeping. Neon instead uses a queue between the address tags and the data portion of the texel cache to maintain hit/miss and cache line information. This approach appears to be similar to the texel cache described in [33].

Figure 11 shows a block diagram of the texel cache. If an incoming request address matches an Address Cache entry, the hardware appends an entry to the Probe Result Queue. This entry records that a hit occurred at the cache line index of the matched address.

If the request doesn't match a cached address, the hardware appends an entry to the Probe Result Queue indicating a miss. This miss entry records the current value of the Least Recently Written Counter (LRWC) as the cache index—this is the location that the new data will eventually be written to in the Data Cache. The cache logic appends the requested address to the Address Queue, writes the address into the Address Cache line at the location specified by the LRWC, and increments the LRWC. The Memory Controller eventually services the entry in the Address Queue, reads the texel data from memory, and deposits the corresponding texel data at the tail of the Data Queue.

To supply texture data that was cached or read from memory to the texel filter tree, the cache hardware examines the head entry of the Probe Result Queue each cycle. A "hit" entry means that the requested data is available in the Data Cache at the location specified by the cache index. When the requested data is consumed, the head entry of the Probe Result Queue is removed.

If the head entry indicates a "miss" and the Data Queue is non-empty, the requested data is in the head entry of the Data Queue. When the data is consumed, it is written into the Data Cache at the location specified by the cache index. The head entries of the Probe Result and Data Queues are then removed.

5.3.6. Unifying Texel Filtering Modes

Neon is designed to trilinear filter texels. All other texel filtering operations are treated as subsets of this case by adjusting the (u_0, v_0, u_1, v_1, *LOD*) coordinates, where (u_0, v_0) are coordinates in the lower mipmap level and (u_1, v_1) are coordinates in the next higher mipmap level. For example, filters that use the nearest mip-map level add 0.5 to the *LOD*, and then zero the fractional bits. Point-sample filters that use the nearest texel in a mip-map do the same to the u_0, v_0, u_1, and v_1 coordinates. Filtering modes that don't use mip-maps zero the entire *LOD*.

Although all filtering modes look like a trilinear filtering after this coordinate adjustment, each mode consumes only as much memory bandwidth as needed. Before probing the address cache, a texel's *u*, *v*, and *LOD* values are examined. If the texel's value is irrelevant, because it will be weighted by a coefficient of zero, then the request is not made to the address or data portions of the cache.

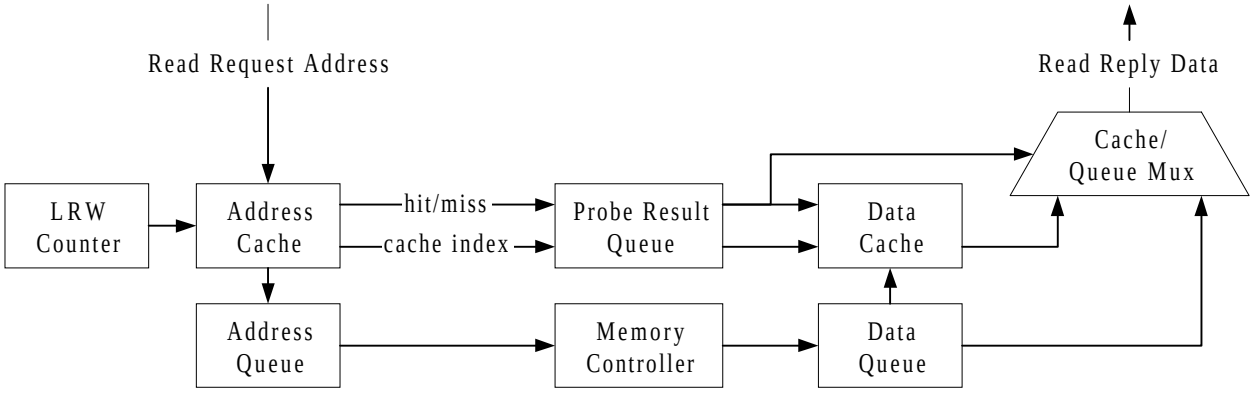

Figure 11: Texel cache block diagram

5.3.7. Filter Tree Structure

Neon's trilinear filter multipliers directly compute the function:

$a*(1.0-c) + b*c$

This requires minor changes to a standard multiplier. The value (1.0-*c*) is represented as ~*c*+1. For each bit of *c*, rather than adding a shifted *b* or 0, the multiplier adds a shifted *b* or *a*. That is, at each bit in the multiplier array, an AND gate is replaced with a multiplexer. An extra row is also needed to unconditionally add in *a*.

Trilinear filtering uses seven of these multipliers, where the *c* input is the fractional bits of *u*, *v*, or *LOD*, as shown in Figure 12. Each 2 x 2 x 2 cube shows which texels have been blended. The front half of the cube is the lower mip-map level, the back half is the higher mip-map level. The first stage combines left and right pairs of texels, by applying the fractional u_0 and u_1 bits to reduce the eight texels to four intermediate values. The second stage combines the top and bottom pairs, using the fractional v_0 and v_1 bits to reduce the four values to the two bilinear filtered results for each mip-map level. The third stage blends the two bilinearly filtered values into a trilinearly filtered result using the fractional *LOD* bits.

It's easy to see that this tree can implement any 2D separable filter in which $f(u) = 1 - f(1 - u)$, by using a simple one-dimensional filter coefficient table. For example, it could be used for a separable cubic filter of radius 1:

$f(u) = 2*\text{abs}(u^3) - 3*u^2 + 1$

Less obviously, we later realized that the filter tree can implement *any* separable filter truncated to 0 beyond the 2 x 2 sampling square. For example, the Gaussian filter:

$f(u, v) = e^{-\alpha (u^2 + v^2)}$ when $u < 1$ and $v < 1$
$f(u, v) = 0$ otherwise

is separable into:

$$f(u, v) = e^{-\alpha u^2} e^{-\alpha v^2}$$

If we remap the fractional bits of *u* as:

$$map[u] = e^{-\alpha u^2} / (e^{-\alpha u^2} + e^{-\alpha (1-u)^2})$$

and do the same for *v*, for both mip-map levels, and then feed the mapped fractional bits into the filter tree, it computes the desired separable function. The first level of the tree computes:

$$t_{bottom} = (t_{00} * e^{-\alpha u^2} + t_{10} * e^{-\alpha (1-u)^2}) / (e^{-\alpha u^2} + e^{-\alpha (1-u)^2})$$
$$t_{top} = (t_{01} * e^{-\alpha u^2} + t_{11} * e^{-\alpha (1-u)^2}) / (e^{-\alpha u^2} + e^{-\alpha (1-u)^2})$$

The second level of the tree computes:

$$\begin{aligned} t &= (t_{\text{bottom}} * e^{-\alpha v^2} + t_{\text{top}} * e^{-\alpha (1-v)^2}) / (e^{-\alpha v^2} + e^{-\alpha (1-v)^2}) \\ &= (t_{00} * e^{-\alpha u^2} * e^{-\alpha v^2} + t_{10} * e^{-\alpha (1-u)^2} * e^{-\alpha v^2} \\ &\quad + t_{01} * e^{-\alpha u^2} * e^{-\alpha (1-v)^2} + t_{11} * e^{-\alpha (1-u)^2} * e^{-\alpha (1-v)^2}) \\ &\quad / (e^{-\alpha u^2} + e^{-\alpha (1-u)^2}) * (e^{-\alpha v^2} + e^{-\alpha (1-v)^2}) \end{aligned}$$

The third level of the tree linearly combines the Gaussian results from the two adjacent mip-maps. Using a Gaussian filter rather than a bilinear filter on each mip-map improves the quality of texture magnification, though it reduces the sharpness of minified images. It also improves the quality of anisotropic texture minification, as discussed further in [22].

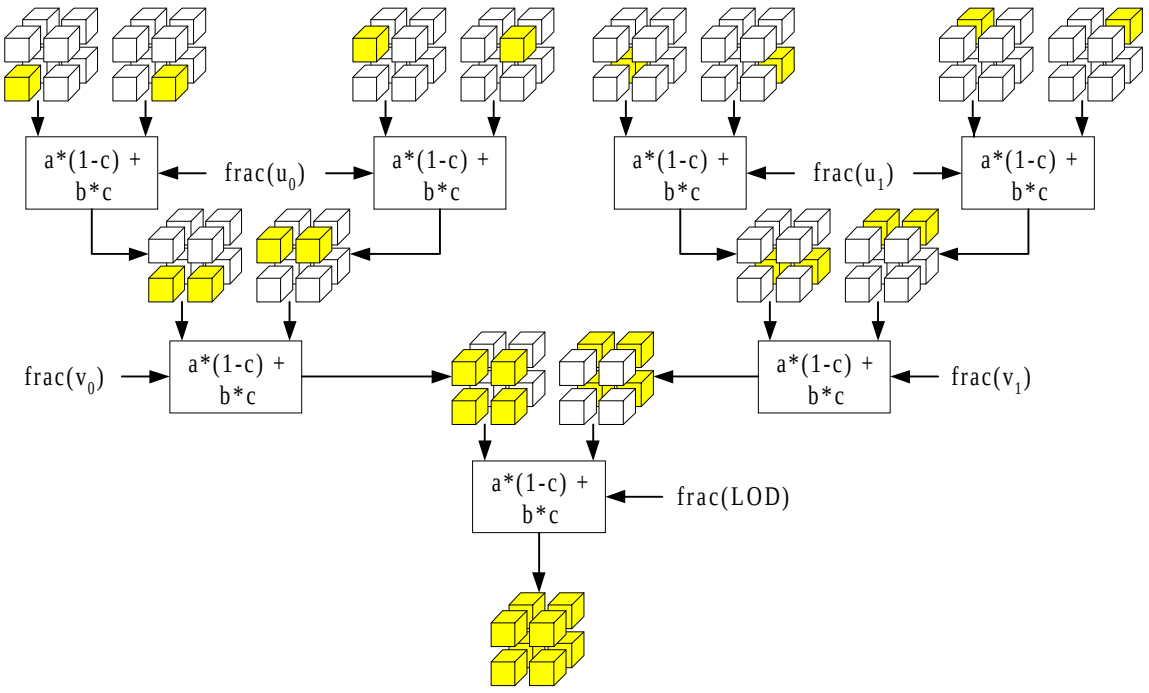

Figure 12: Filter multiplier tree

5.4. Fragment Generator

The Fragment Generator determines which fragments are within an object, generates them in an order that reduces memory bandwidth requirements, and interpolates the channel data provided at vertices.

The fragment generator uses half-plane edge functions [10][16][25] to determine if a fragment is within an object. The three directed edges of a triangle, or the four edges of a line, are represented by planar (affine) functions that are negative to the left of an edge, positive to the right, and zero on an edge. A fragment is inside an object if it is to the right of all edges in a clockwise series, or to the left of all the edges in a counterclockwise series. (Fragments exactly on an edge of the object use special inclusion rules.) Figure 13 shows a triangle described by three clockwise edges, which are shown with bold arrows. The half-plane where each edge function is positive is shown by several thin "shadow" lines with the same slope as the edge. The shaded portion shows the area where all edge functions are positive.

For most 3D operations, a 2 x 2 fragment stamp evaluates the four edge equations at each of the four positions in the stamp. Texture mapped objects use a 1 x 1 stamp, and 2D objects use an 8 x 1 or 32 x 1 stamp. The stamp bristles with several probes that evaluate the edge equations outside the stamp boundaries; each cycle, it combines these results to determine in which direction the stamp should move next. Probes are cheap, as they only compute a sign bit. We use enough probes so that the stamp avoids moves to locations outside the object (where it does not generate any fragments) unless it must in order to visit other positions inside the object. When the stamp is one pixel high or wide, several different probes may evaluate the edge functions at the same point. The stamp movement algorithm handles coincident probes without special code for the myriad stamp sizes. Stamp movement logic cannot be pipelined, so simplifications like this avoid making a critical path even slower.

The stamp may also be constrained to generate all fragments in a 2^m by 2^n rectangular "chunk" before moving to the next chunk. Neon's chunking is not cheap: it uses three additional 600-bit save states and associated multiplexers. But chunking improves the texture cache hit rate and decreases page crossings, especially non-prefetchable crossings. We found the cost well worth the benefits. (Chunking could be a lot cheaper—we recently discovered that we could have used a single additional save state.)

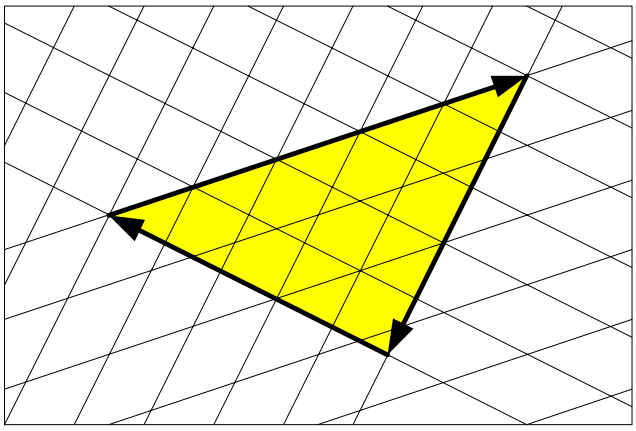

Figure 13: Triangle described by three edge functions

The Fragment Generator contains several capabilities specific to lines. The setup logic can adjust endpoints to render Microsoft Windows "cosmetic" lines. Lines can be dashed with a pattern that is internally generated for OpenGL lines and some X11 lines, or externally supplied by software for the general X11 dashed line case. We paint OpenGL wide dashed lines by sweeping the stamp horizontally across scanlines for *y*-major lines, and vertically across columns for *x*-major lines. Again, to avoid slowing the movement logic, we don't change the movement algorithm. Instead, the stamp always moves across what it thinks are scanlines, and we lie to it by exchanging *x* and y coordinate information on the way in and out of the stamp movement logic.

Software can provide a scaling factor to the edge equations to paint the rectangular portion of X11 wide lines. (This led us to discover a bug in the X11 server's wide line code.) Software can provide a similar scaling factor for antialiased lines. Neon nicely rounds the tips of antialiased lines and provides a programmable filter radius; these features are more fully described in [23]. The OpenGL implementation exploits these features to paint antialiased square points up to six pixels in diameter that look like the desired circular points.

5.5. Command Parser

The Command Parser decodes packets, detects packet errors, converts incoming data to internal fixed-point formats, and decomposes complex objects like polygons, quads, and quad-strips into triangle fans for the fragment generator. Neon's command format is sufficiently compact that we use the PCI bus rather than a high-speed proprietary bus between the CPU and the graphics device. A well-implemented 32-bit, 33 MHz PCI provides over 100 megabytes/second for DMA and sequential PIO (Programmed I/O) writes, while a 64-bit PCI provides over 200 megabytes/second.

We don't initiate activity with out-of-order writes to registers or frame buffer locations, but use low-overhead variable-length sequential commands to exploit streaming transfers on the PCI. The processor can write commands directly to Neon, or can write to a ring buffer in main memory, which Neon reads using DMA.

Neon supports multiple command ring buffers at different levels of the memory hierarchy. The CPU preferentially uses a small ring buffer that fits in the on-chip cache, which allows the CPU to write to it quickly. If Neon falls behind the CPU, which then fills the small ring buffer, the CPU switches to a larger ring buffer in slower memory. Once Neon catches up, the CPU switches back to the smaller, more efficient ring buffer.

5.5.1. Instruction Set

Polygon vertex commands draw independent triangles, triangle strips, and triangle fans, and independent quadrilaterals and quad strips. They consist of a 32-bit command header word, a 32-bit packet length, and a variable amount of *per-vertex* or *per-object* data, such as Z depth information, RGB colors, alpha transparency, eye distance for fog, and texture coordinates. Per-vertex data is provided at each vertex, and is smoothly interpolated across the object. Per-object data is provided only at each vertex that completes an object (for example, each third vertex for independent triangles, each vertex after the first two for triangle strips), and is constant across the object. Thus, the CPU provides only as much data as is actually needed to specify a polygon; there is no need to replicate data when painting flat shaded triangles or when painting strips. We don't provide *per-packet* data, since it would save only one word over changing the default color and Z registers with a register write command.

Line vertex commands draw independent lines and line strips. In addition to per-vertex and per-object data, line commands also allow several types of *per-pixel* data. This lets us implement new functionality in software while taking advantage of Neon's existing capabilities. Painting a triangle using lines and per-pixel data wouldn't offer blinding performance, but it would be faster than having to paint using Neon's Point command.

The Point vertex command draws a list of points, and takes per-vertex data. Points may be wide and/or antialiased. (Antialiased points aren't true circles, but are antialiased squares with a wide filter, so look good only for points up to about five or six pixels wide.)

Rectangle commands paint rectangles to the frame buffer, or DMA rectangular regions of the frame buffer to main memory. Rectangles may be solid filled, or foreground and background stippled using the internal 32 x 32 stipple pattern, or via stipple data in the command packet. Rectangles may also fetch source data from several sources: from inline data in the packet, from main memory locations specified in the packet, from an identically sized rectangle in frame buffer memory, from a 2^m x 2^n tile, or from an arbitrary sized texture map using any of the texture map filters. This last capability means that Neon can rescale an on-screen video image via texture mapping and deposit the result into main memory via DMA with no intermediate buffers.

The Interlock command ensures that a buffer swap doesn't take place until screen refresh is outside of a small critical region (dependent upon the window size and location), in order to avoid tearing artifacts. And a multichip variant of the interlock command guarantees that a buffer swap takes place only when a group of Neon chips are all ready to swap, so that multiple monitors can be animated synchronously.

5.5.2. Vertex Data Formats

Neon supports multiple representations for some data. For example, RGBA color and transparency can be supplied as four 32-bit floating point values, four packed 16-bit integers, or four packed 8 bit integers. The *x* and *y* coordinates can be supplied as two 32-bit floating point values, or as signed 12.4 fixed-point numbers. Using floating point, the six values (*x, y, z, r, g, b*) require 24 bytes per vertex. Using Neon's most compact representation, they require only 12 bytes per vertex. These translate into about 4 million and 8 million vertices/second on a 32-bit PCI.

If the CPU is the bottleneck, as with lit triangles, the CPU uses floating-point values and avoids clamping, conversion, and packing overhead. If the CPU can avoid lighting computations, and the PCI is the bottleneck, as with wireframe drawings, the CPU uses the packed formats. Future Alpha chips may saturate even a 64-bit PCI or an AGP-2 bus with floating point triangle vertex data, but may also be able to hide clamping and packing overhead using new instructions and more integer functional units. Packed formats on a 64-bit PCI allows transferring about 12 to 16 million (*x, y, z, r, g, b*) vertices per second.

5.5.3. Better Than Direct Rendering

Many vendors have implemented some form of *direct rendering*, in which applications get direct control of a graphics device in order to avoid the overhead of encoding, copying, and decoding an OpenGL command stream [18]. (X11 command streams are generally not directly rendered, as X11 semantics are harder to satisfy than OpenGL's.) We were unhappy with some of the consequences of direct rendering. To avoid locking and unlocking overhead, CPU context switches must save and restore both the architectural and internal implementation state of the graphics device on demand, including in the middle of a command. Direct rendering applications make new kernel calls to obtain information about the window hierarchy, or to accomplish tasks that should not or cannot be directly rendered. These synchronous kernel calls may in turn run the X11 server before returning. Applications that don't use direct rendering use more efficient asynchronous requests to the X11 server.

Neon uses a technique we called "Better Than Direct Rendering" (BTDR) to provide the benefits of direct rendering without these disadvantages. Like direct rendering, BTDR allows client applications to create hardware-specific rendering commands. Unlike direct rendering, BTDR leaves dispatching of these commands to the X11 server. In effect, the application creates a sequence of hardware rendering commands, then asks the X11 server to call them as a subroutine. To avoid copying client-generated commands, Neon supports a single-level call to a command stream stored anywhere in main memory. Since only the X11 server communicates directly with the accelerator, the accelerator state is never context switched preemptively, and we don't need state save/restore logic.

Since hardware commands are dispatched in the correct sequence by the server, there is no need for new kernel calls. Since BTDR maintains atomicity and ordering of commands, we believe (without an existence proof) that BTDR could provide direct rendering benefits to X11, with much less work and overhead than Mark Kilgard's D11 proposal [19].

5.6. Video Controller

The Video Controller refreshes the display, but delegates much of the work to the memory controllers in order to increase opportunities for page prefetching. It periodically requests pixels from the memory controllers, "inverse dithers" this data to restore color fidelity lost in the frame buffer, then sends the results to an IBM RGB640 RAMDAC for color table lookup, gamma correction, and conversion to an analog video signal.

5.6.1. Opportunistic Refresh Servicing

Each screen refresh request to a memory controller asks for data from a pair of A and B bank pages. The memory controller can usually finish rendering in the current bank , ping-pong between banks to satisfy the refresh request, and return to rendering in the other bank—using prefetching to hide all page crossing overhead. For example, if the controller is currently accessing an A bank page when the refresh request arrives, it prefetches the refresh B page while it finishes rendering the rest of the fragments on the A bank page. It then prefetches the refresh A page while it fetches pixels from the refresh B page, prefetches a new B page for rendering while it fetches pixels from the refresh A page, and finally returns to rendering in the new B page.

Screen refresh reads cannot be postponed indefinitely in an attempt to increase prefetching. If a memory controller is too slow in satisfying the request, the Video Controller forces it to fetch refresh data immediately. When the controller returns to the page it was rendering, it cannot prefetch it, as this page is in the same bank as the second screen refresh page.

The Video Controller delegates to each Memory Controller the interpretation of overlay and display format bytes, and the reading of pixels from the front, back, left, or right buffers. This allows the memory controller to immediately follow overlay and display format reads with color data reads, further increasing prefetching efficiency. To hide page crossing overhead, the memory controller must read 16 16-bit pixels from each overlay and display format page, but only 8 32-bit pixels from each color data page. The memory controller thus alternates between:

1. Reading overlay and display format from an A and B bank pair of pages (32 16-bit pixels), then
2. Reading color data from a different A and B bank pair (16 32-bit pixels) if the corresponding overlay (that was just read in step 1) is transparent.

and sometime later:

3. Reading color data from an A and B bank pair (16 32-bit pixels) if the corresponding overlay (read awhile ago in step 1) is transparent.

If the overlay isn't transparent, the controller doesn't read the corresponding 32-bit color data. If the root window and 2D windows use the 8-bit overlay, then only 3D windows fetch 32-bit color data, which further increases memory bandwidth available for rendering.

5.6.2. Inverse Dithering

Dithering is commonly used with 16 and 8-bit color pixels. Dithering decreases spatial resolution in order to increase color resolution. In theory, the human eye integrates pixels (if the pixels are small enough or the eye is myopic enough) to approximate the original color. In practice, dithered images are at worst annoyingly patterned with small or recursive tessellation dither matrices, and at best slightly grainy with the large void-and-cluster dither matrices we have used in the past [29].

Hewlett-Packard introduced "Color Recovery™" [2] to perform this integration digitally and thus improve the quality of dithered images. Color Recovery applies a 16 pixel wide by 2 pixel high filter at each 8-bit pixel on the path out to the RAMDAC. In order to avoid blurring, the filter is not applied to pixels that are on the opposite side of an "edge," which is defined as a large change in color.

HP's implementation has two problems. Their dithering is non-mean preserving, and so creates an image that is too dark and too blue. Their reconstruction filter does not compensate for these defects in the dithering process. And the 2 pixel high filter requires storage for the previous scanline's pixels, which would need a lot of real estate for Neon's worst case scanlines of 1920 16-bit pixels. The alternative—fetching pixels twice—requires too much bandwidth.

Neon implements an "inverse dithering" process similar to Color Recovery, but dynamically chooses between several higher quality filters, all of which are only one pixel high. We used both mathematical analysis of dithering functions and filters, as well as empirical measurements of images, to choose a dither matrix, the coefficients for each filter, and the selection criteria to determine which filter to apply to each pixel in an image. We use small asymmetrical filters near high-contrast edges, and up to a 9-pixel wide filter for the interior of objects. Even when used on Neon's lowest color resolution pixels, which have 4 bits for each color channel, inverse dithering results are nearly indistinguishable from the original 8 bits per channel data. More details can be found in [5] and [30].

5.7. Performance Counters

Modern CPUs include performance counters in order to increase the efficiency of the code that compilers generate, to provide measurements that allow programmers to tune their code, and to help the design of the next CPU.

Neon includes the same sort of capability with greater flexibility. Neon includes two 64-bit counters, each fully programmable as to how conditions should be combined before being counted. We can count multiple occurrences of some events per cycle (e.g., events related to the eight memory controllers or pixel processors). This allows us to directly measure, in a single run, statistics that are ratios or differences of different conditions.

5.8. "Sushi Boat" Register Management

Register state management in Neon is decentralized and pipelined. This reduces wiring congestion—rather than an explosion of signals between the Command Parser and the rest of the chip, we use a single existing pathway for both register reads and writes. This also reduces pipeline stalls needed to ensure a consistent view of the register state. (The "sushi boat" name comes from Japanese restaurants that use small boats in a circular stream to deliver sushi and return empty trays.)

Registers are physically located near logic that uses them. Several copies of a register may exist to limit physical distances from a register to dependent logic, or to reduce the number of pipeline stages that are dependent upon a register's value. The different copies of a register may contain different data at a given time.

Register writes are sent down the object setup/fragment generation/fragment processing pipeline. The new value is written into a local register at a point that least impacts the logic that depends upon it. Ideally, a register write occurs as soon as the value reaches a local register. At worst, several pipe stages use the same copy of a register, and thus a handful of cycles must be spent draining that portion of the pipeline before the write commits.

Register reads are also sent down the pipeline. Only one copy of the register loads its current value into the read command; other copies simply let the register read pass unmodified. The end of the pipeline feeds the register back to the Command Parser.

6. Physical Characteristics

Neon is a large chip. Its die is 17.3 x 17.3 mm, using IBM's 0.35 µm CMOS 5S standard cell process with 5 metal layers [15]. (Their 0.25 µm 6S technology would reduce this to about 12.5 x 12.5 mm, and 0.18 µm 7S would further reduce this to about 9 x 9 mm.) The design uses 6.8 million transistors and sample chips run at the 100 MHz design frequency.

The chip has 628 signal pins, packaged in an 824-pin ceramic column grid array. The 8 memory controllers each use 32 data pins and 24 address, control, and clock pins; an additional two pins for SDRAM clock phase adjustment make a total of 450 signal pins to memory. The 64-bit PCI interface uses 88 pins. The video refresh portion of the RAMDAC interface uses 65 pins. Another 15 pins provide a general-purpose port—a small FPGA connects the port to the RAMDAC, VGA, and programmable dot clock registers, as well as to board configuration switches. One pin is for the core logic clock, and the remaining 9 pins are for device testing.

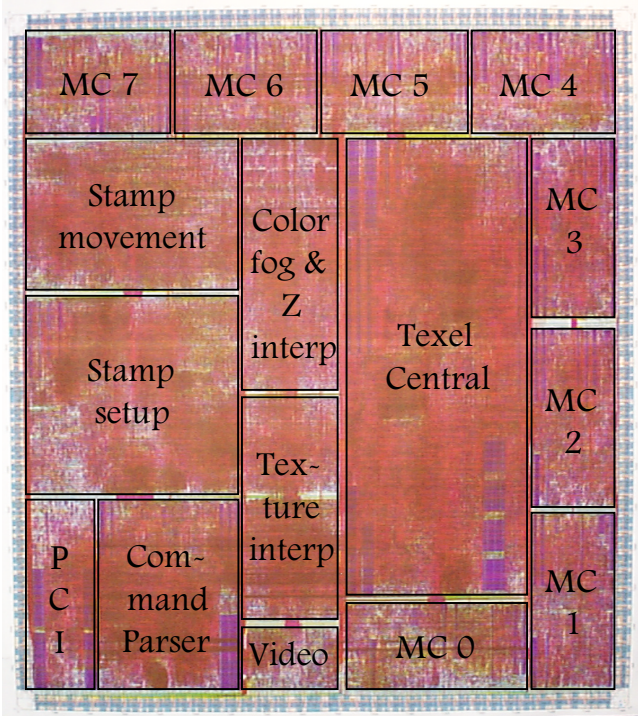

Figure 14: Neon die plot

Figure 14 shows a plot of the metal layers of the die. Data flows in through the PCI interface, right to the Command Parser, up to the Fragment Generator setup logic, up again to the stamp movement logic, right to the interpolation of vertex data, right into Texel Central, and finally out to the eight Pixel Processor/Memory Controllers on the periphery of the. The Video Refresh block is small because it includes logic only for sending requests for pixel data, and for inverse dithering; the line buffers are resident in the memory controller blocks. The congested wiring channels between blocks are a consequence of IBM's suggestion that interblock wiring flow through small areas on the sides of each block.

7. CAD and Verification Environment

We designed Neon using the C programming language, rather than Verilog or VHSIC Hardware Description Language (VHDL). This section discusses the advantages of using C, our simulator, the C to Verilog compiler, special-purpose gate generators, and our custom verification software.

7.1. C vs. Verilog and VHDL

The C language has several advantages over Verilog. In particular, C supports signed numbers and record structures, which we used extensively in Neon. On the other hand, C has no way of specifying bit lengths. We solved

this deficiency by using a C++ compiler, and added two new types with C++ templates. *Bits*[*n*] is an unsigned number of *n* bits; *Signed*[*n*] is a 2's complement number of *n* bits, including the sign bit. In retrospect, we should also have added the type constructor *Range*[*lower*, *upper*], in order to further enhance the range-checking in *c2v*, described below in Section 7.3.

VHDL has all of these language features. We still saw C as a better choice. In addition to the advantages discussed below, coding in C gave us the entire C environment while developing, debugging, and verifying Neon. For example, early, high-level models of the chip used library calls to mathematical functions.

7.2. Native C Simulator

We've used our own 2-state event-driven simulator for years. It directly calls the C procedures used to describe the hardware, so we can use standard C debugging tools. It topologically sorts the procedures, so that the event flag scanning proceeds from top to bottom. (Apparent "loops" caused by data flowing back and forth between two modules with multiple input/output ports are handled specially.) Evaluating just the modules whose input changes invokes on average 40% to 50% of Neon's presynthesis high level behavioral code. (This could have been smaller, as many designers were sloppy about importing large wire structures *in toto*, rather than the few signals that they needed.) The simulator evaluated only 7% to 15% of the synthesized gate-level wirelist each cycle.

The simulator runs about twice as fast as the best commercial simulator we benchmarked. We believe this is due to directly compiling C code, especially the arithmetic operations in high-level behavioral code; and to the low percentage of procedures that must be called each cycle, especially in the gate-level structural code. Even better, we have no per-copy licensing fee. During Neon's development, we simulated over 289 billion cycles using 22 Alpha CPUs. We simulated 2 billion cycles with the final full-chip structural model.

7.3. C to Verilog Translation

We substantially modified *lcc* [9], a portable C compiler, to create *c2v*, a C to Verilog translator. This translator confers a few more advantages to using C. In particular, *c2v* evaluates the numeric ranges of expressions, and expands their widths in the Verilog output to avoid overflow. For example:

Bits[2] *a*, *b*, *c*, *d*;
if (*a*+*b*) < (*c*+*d*) …

is evaluated in Verilog or VHDL using the maximum precision of the variables—two bits—and so can yield the wrong answer. The *c2v* translator forces the expression to be computed with three bits. *c2v* computes the tightest possible bounds for expressions, including those that use Boolean operators, in order to minimize the gates required to evaluate the expression correctly.

In addition, *c2v* checks assignments to ensure that the right-hand side of an expression fits into the left-hand side. This simple check statically caught numerous examples of code trying to assign, for example, three bits of state information into a 2-bit field.

7.4. Synthesis vs. Gate Generators

Initial benchmarks using Synopsys to generate adders and multipliers yielded structures that were larger and slower than we expected. IBM's parameterized libraries weren't any better. These results, coupled with our non-standard arithmetic requirements (base 255 arithmetic, *a**(1-*c*) + *b***c* multiplier/adders, etc.) led us to design a library of gate generators for addition, multiplication, and division. We later added a priority encoder generator, as Synopsys was incapable of efficiently handling chained *if…then…else if…* statements. We also explicitly wired multiplexers for Texel Central's memory controller crossbar and format conversion: Synopsys took a day to synthesize a structure that was twice as large, and much slower, than the one we created by hand.

From our experiences, we view Synopsys as a weak tool for synthesizing data paths. However, the only alternative seems to be wiring data paths by hand.

7.5. Hardware Verification

We have traditionally tested graphics accelerators with gigabytes of traces from the X11 server. Designers generate traces from the high-level behavioral model by running and visually verifying graphics applications. With Neon, we expected the behavioral model to simulate so slowly that it would be impossible to obtain enough data.

A number of projects at Digital, including all Alpha processors, have used a language called Segue for creating test suites. Segue allows the pseudo-random selection of weighed elements within a set. For example, the expression X = {1:5, 2:15, 3:80} randomly selects the value 1, 2 or 3 with a probabilities of 5%, 15% or 80%. Segue is well suited for generating stimulus files that require a straightforward selection of random data, such as corner-case tests or random traffic on a PCI bus. Unfortunately, Segue is a rudimentary language with no support for complex data processing capabilities. It lacks multidimensional arrays, pointers, file input, and floating-point variables, as well as symbolic debugging of source code. C supports the desired programming features, but lacks the test generation features. Because we use C and C++ extensively for the Neon design and the CAD suite, we decided to enhance C++ to support the Segue sets. We call this enhanced language Segue++.

Segue++ is an environment consisting of C++ class definitions with behavior like Segue sets, and a preprocessor to translate the Segue++ code to C++. We could have used the new C++ classes without preprocessing, but the

Triangle size	Random triangles	Random strips	Aligned strips	Peak generation
10-pixel	N/A	N/A	7.8	7.8
25-pixel	2.6	4.2	5.4	7.5
50-pixel	1.6	2.3	2.8	4.5
25-pixel, trilinear textured	N/A	2.0	2.3	4.0
50-pixel, trilinear textured	0.75	1.3	1.4	2.0

Table 1: Shaded, Z-buffered triangles, millions of triangles/second

limitations of C++ operator overloading makes the code difficult to read and write. Furthermore, the preprocessor allows us to define and select set members inside C++ code, and we can imbed C++ expressions in the expressions for set members. The users of Segue++ have all the features of the C++ language as well as the development environment, including symbolic debugging linked back to the original Segue++ code.

Segue++ proved invaluable for our system tests, which tested the complete Neon chip attached to a number of different PCI transactors. The system tests generate a number of subtests, with each subtest using different functionality in the Neon chip to render the same image in the frame buffer. For example, a test may render a solid-colored rectangle in the frame buffer. One subtest may download the rectangle from memory using DMA, another may draw the rectangle using triangles, etc. Each subtest can vary global attributes such as the page layout of the frame buffer, or the background traffic on the PCI bus. We discovered a surprisingly rich set of possible variations for each subtest. The set manipulation features of Segue++ allowed us to generate demanding test kernels, while the general programming features of Segue++ allowed us to manipulate the test data structure to create the subtests. The system tests found many unforeseen interaction-effect bugs in the Neon design.

8. Performance

In this section, we discuss some performance results, based on cycle-accurate simulations of a 100 MHz part. (Power-on has proceeded *very* slowly. Neon was cancelled shortly before tape-out, so any work occurs in people's spare time. The chip does run at speed, and the few real benchmarks we have performed validate the simulations.)

We achieved our goal of using memory efficiently. When painting 50-pixel triangles to a 1280 x 1024 screen refreshed at 76 Hz, screen refresh consumes about 25% of memory bandwidth, rendering consumes another 45%, and overhead cycles that do not transfer data (read latencies, high-impedance cycles, and page precharging and row addressing) consume the remaining 30%. When filling large areas, rendering consumes 60% of bandwidth.

As a worst-case acid test, we painted randomly placed triangles with screen refresh as described above. Each object requires at least one page fetch. Half of these page fetches cannot be prefetched at all, and there is often insufficient work to completely hide the prefetching in the other half. The results are shown in the "Random triangles" column of Table 1. (Texels are 32 bits.)

Type of line	Random strips
10-pixel, constant color, no Z	11.0
10-pixel, shaded, no Z	10.6
10-pixel, shaded, Z-buffered	7.8
10-pixel, shaded, Z-buffered, antialiased	4.7

Table 2: Random line strips, millions of lines/second

We also painted random strips of 10 objects; each list begins in a random location. This test more closely resembles the locality of rendering found in actual applications, though probably suffers more non-prefetchable page transitions than a well-written application. Triangle results are shown in the "Random strips" column of Table 1, line results are shown in Table 2.

The only fill rates we've measured are not Z-tested, in which case Neon achieves 240 million 64-bit fragments/second. However, the "Aligned strip" column in Table 1 shows triangle strips that were aligned to paint mostly on one page or a pair of pages, which should provide a lower bound on Z-tested fill rates. Note that 50-pixel triangles paint 140 million Z-buffered, shaded pixels/second, and 70 million trilinear textured, Z-buffered, shaded pixels/second. In the special case of bilinearly magnifying an image, such as scaling video frames, we believe Neon will run extremely close to the peak texture fill rate of 100 million textured pixels/second.

The "Peak generation" column in Table 1 shows the maximum rate at which fragments can be delivered to the memory controllers. For 10-pixel triangles, the limiting factor is setup. For larger triangles, the limiting factor is object traversal: the 2 x 2 stamp generates on average 1.9 fragments/cycle for 25-pixel triangles, and 2.3 fragments/cycle for 50-pixel triangles. For textured triangles, the stamp generates one fragment/cycle.

Neon's efficient use of memory bandwidth is impressive, especially when compared to other systems for which we have enough data to compute peak and obtained band-

width. For example, we estimate that the SGI Octane MXE, using RAMBUS RDRAM, has over twice the peak bandwidth of Neon—yet paints 50-pixel Z-buffered triangles about as fast as Neon. Even accounting for the MXE's 48-bit colors, Neon extracts about twice the performance per unit of bandwidth. The MXE uses special texture-mapping RAMs, and quotes a "texture fill rate" 38% higher than Neon's peak texture fill rate. Neon uses SDRAM and steals texture mapping bandwidth from other rendering operations. Yet their measured texture mapped performance is equivalent. Tuning of the memory controller heuristics might further improve Neon's efficiency.

We have also achieved our goals of outstanding price/performance. When compared to other workstation accelerators, Neon is either a lot faster, a lot cheaper, or both. For example, HP's fx[6] accelerator is about 20% to 80% faster than Neon—at about eight times our anticipated list price.

Good data on PC accelerators is hard to come by (many PC vendors tend to quote peak numbers without supporting details, others quote performance for small screens using 16-bit pixels and texels, etc.). Nonetheless, when compared to PC accelerators in the same price range, Neon has a clear performance advantage. It appears to be about twice as fast, in general, as Evans & Sutherland's REALimage technology (as embodied in the Mitsubishi 3DPro chip set), and the 3Dlabs GLINT chips.

9. Conclusions

Historically, fast workstation graphics accelerators have used multiple chips and multiple memory systems to deliver high levels of graphics performance. Low-end workstation and PC accelerators use single chips connected to a single memory system to reduce costs, but their performance consequently suffers.

The advent of 0.35 μm technology coupled with ball or column grid arrays means that a single ASIC can contain enough logic and connect to enough memory bandwidth to compete with multichip 3D graphics accelerators. Neon extracts competitive performance from a limited memory bandwidth by using a greater percentage of peak memory bandwidth than competing chip sets, and by reducing bandwidth requirements wherever possible. Neon fits on one die, because we extensively share real estate among similar functions—which had the nice side effect of making performance tuning efforts more effective. Newer 0.25 μm technology would reduce the die size to about 160 mm^2 and increase performance by 20-30%. Emerging 0.18 μm technology would reduce the die to about 80 mm^2 and increase performance another 20-30%. This small die size, coupled with the availability of SDRAM at less that a dollar a megabyte, would make a very low-cost, high-performance accelerator.

10. Acknowledgements

Hardware Design & Implementation: Bart Berkowitz, Shiufun Cheung, Jim Claffey, Ken Correll, Todd Dutton, Dan Eggleston, Chris Gianos, Tracey Gustafson, Tom Hart, Frank Hering, Andy Hoar, Giri Iyengar, Jim Knittel, Norm Jouppi, Joel McCormack, Bob McNamara, Laura Mendyke, Jay Nair, Larry Seiler, Manoo Vohra, Robert Ulichney, Larry Wasko, Jay Wilkinson.

Hardware Verification: Chris Brennan, John Eppling, Tyrone Hallums, Thom Harp, Peter Morrison, Julianne Romero, Ben Sum, George Valaitis, Rajesh Viswanathan, Michael Wright, John Zurawski.

CAD Tools: Paul Janson, Canh Le, Ben Marshall, Rajen Ramchandani.

Software: Monty Brandenberg, Martin Buckley, Dick Coulter, Ben Crocker, Peter Doyle, Al Gallotta, Ed Gregg, Teresa Hughey, Faith Lin, Mary Narbutavicius, Pete Nishimoto, Ron Perry, Mark Quinlan, Jim Rees, Shobana Sampath, Shuhua Shen, Martine Silbermann, Andy Vesper, Bing Xu, Mark Yeager.

Keith Farkas commented extensively on far too many drafts of this paper.

Many of the techniques described in this paper are patent pending.

References

[1] Kurt Akeley. RealityEngine Graphics. *SIGGRAPH 93 Conference Proceedings*, ACM Press, New York, August 1993, pp. 109-116.

[2] Anthony C. Barkans. Color Recovery: True-Color 8-Bit Interactive Graphics. *IEEE Computer Graphics and Applications*, IEEE Computer Society, New York, volume 17, number 1, January/February 1997, pp. 193-198.

[3] Anthony C. Barkans. High Quality Rendering Using the Talisman Architecture. *Proceedings of the 1997 SIGGRAPH/Eurographics Workshop on Graphics Hardware*, ACM Press, NY, August 1997, pp. 79-88.

[4] Jim Blinn. Jim Blinn's Corner: Three Wrongs Make a Right. *IEEE Computer Graphics and Applications*, volume 15, number 6, November 1995, pp. 90-93.

[5] Shiufun Cheung & Robert Ulichney. Window-Extent Tradeoffs in Inverse Dithering. *Proceedings of Society for the Imaging Science and Technology (IS&T) 6th Color Imaging Conference*, IS&T, Springfield, VA, Nov. 1998, available at http://www.crl.research.digital.com/who/people/ulichney/bib.htm.

[6] Michael F. Deering, Stephen A. Schlapp, Michael G. Lavelle. FBRAM: A New Form of Memory Optimized for 3D Graphics. *SIGGRAPH 94 Conference Proceedings*, ACM Press, New York, July 1994, pp. 167-174.

[7] John H Edmondson, et. al. Internal Organization of the Alpha 21164, a 300-MHz 64-bit Quad-issue CMOS RISC Microprocessor. *Digital Technical Journal*, Digital Press, volume 7, number 1, 1995.

[8] Jon P. Ewins, Marcus D. Waller, Martin White & Paul F. Lister. MIP-Map Level Selection for Texture Mapping. *IEEE Transactions on Visualization and Computer Graphics*, IEEE Computer Society, New York, volume 4, number 4, October-December 1998, pp. 317-328.

[9] Christopher Fraser & David Hanson. *A Retargettable C Compiler: Design and Implementation*, Benjamin/Cummings Publishing, Redwood City, CA, 1995.

[10] Henry Fuchs, et. al. Fast Spheres, Shadows, Textures, Transparencies, and Image Enhancements in Pixel-Planes. *SIGGRAPH 85 Conference Proceedings*, ACM Press, New York, July 1985, pp. 111-120.

[11] David Goldberg. Computer Arithmetic. In John L. Hennessy & David A. Patterson's *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers, 1990, pp. A50-A53.

[12] Lynn Gwennap. Digital 21264 Sets New Standard. *Microprocessor Report*, volume 10, issue 14, October 28, 1996.

[13] Siyad S. Hakura & Anoop Gupta. The Design and Analysis of a Cache Architecture for Texture Mapping. *Proceedings of the 24th International Symposium on Computer Architecture (ISCA)*, ACM Press, New York, June 1997, pp. 108-120.

[14] Paul S. Heckbert & Henry P. Morton. Interpolation for Polygon Texture Mapping and Shading. In *State of the Art in Computer Graphics: Visualization and Modeling*, Springer-Verlag, 1991, available at http://www.cs.cmu.edu/~ph/.

[15] *IBM CMOS 5S ASIC Products Databook*, IBM Microelectronics Division, Hopewell Junction, NY, 1995, available at http://www.chips.ibm.com/techlib.products/asics/databooks.html.

[16] Brian Kelleher. *PixelVision Architecture*, Technical Note 1998-013, System Research Center, Compaq Computer Corporation, October 1998, available at http://www.research.digital.com/SRC/publications/src-tn.html

[17] Jim Keller. The 21264: A Superscalar Alpha Processor with Out-of-Order Execution. Presentation at *Microprocessor Forum*, October 22-23 1996, slides available at http://www.digital.com/info/semiconductor/a264up1/index.html.

[18] Mark J. Kilgard, David Blythe & Deanna Hohn. System Support for OpenGL Direct Rendering. *Proceedings of Graphics Interface 1995*, available at http://www.sgi.com/software/opengl/whitepapers.html.

[19] Mark J. Kilgard. D11: A High-Performance, Protocol-Optional, Transport-Optional Window System with X11 Compatibility and Semantics. *The X Resource*, issue 13, Proceedings of the 9th Annual X Technical Conference, 1995, available at http://reality.sgi.com/opengl/d11/d11.html.

[20] Mark J. Kilgard. Realizing OpenGL: Two Implementations of One Architecture. *Proceedings of the 1997 SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pp. 45-55, available at http://reality.sgi.com/mjk/twoimps/twoimps.html.

[21] Joel McCormack & Robert McNamara. *A Smart Frame Buffer*, Research Report 93/1, Western Research Laboratory, Compaq Computer Corporation, January 1993, available at http://www.research.digital.com/wrl/techreports/pubslist.html.

[22] Joel McCormack, Ronald Perry, Keith I. Farkas & Norman P. Jouppi. *Simple and Table Feline: Fast Elliptical Lines for Anisotropic Texture Mapping*, Research Report 99/1, Western Research Laboratory, Compaq Computer Corporation, July 1999, available at http://www.research.digital.com/wrl/techreports/pubslist.html

[23] Robert McNamara, Joel McCormack & Norman Jouppi. *Prefiltered Antialiased Lines Using Distance Functions*, Research Report 98/2, Western Research Laboratory, Compaq Computer Corporation, October 1999, available at http://www.research.digital.com/wrl/techreports/pubslist.html.

[24] John S. Montrym, Daniel R. Baum, David L. Dignam & Christopher J. Migdal. InfiniteReality: A Real-Time Graphics System. *SIGGRAPH 97 Conference Proceedings*, ACM Press, New York, August 1997, pp. 293-302.

[25] Juan Pineda. A Parallel Algorithm for Polygon Rasterization. *SIGGRAPH 88 Conference Proceedings*, ACM Press, New York, August 1988, pp. 17-20.

[26] Mark Segal & Kurt Akeley. *The OpenGL Graphics System: A Specification (Version 1.2)*, 1998, available at http://www.sgi.com/software/ opengl/manual.html.

[27] Robert W. Scheifler & James Gettys. *X Window System, Second Edition*, Digital Press, 1990.

[28] Jay Torborg & James Kajiya. Talisman: Commodity Realtime 3D Graphics for the PC. *SIGGRAPH 96 Conference Proceedings*, ACM Press, New York, August 1996, pp. 353-363.

[29] Robert Ulichney. The Void-and-Cluster Method for Dither Array Generation. *IS&T/SPIE Symposium on Electronic Imaging Science & Technology*, volume

1913, pp. 332-343, 1993, available at http://www.crl.research.digital.com/who/people/ulichney/bib.htm.

[30] Robert Ulichney, One-Dimensional Dithering. *Proceedings of International Symposium on Electronic Image Capture and Publishing (EICP 98)*, SPIE Press, Bellingham, WA, SPIE volume 3409, May, 1998, available at http://www.crl.research.digital.com/who/people/ulichney/bib.htm.

[31] Robert Ulichney and Shiufun Cheung, Pixel Bit-Depth Increase by Bit Replication. *Color Imaging: Device-Independent Color, Color Hardcopy, and Graphic Arts III, Proceedings of SPIE* vol. 3300, Jan. 1998, pp. 232-241, available at http://www.crl.research.digital.com/who/people/ulichney/bib.htm.

[32] Lance Williams. Pyramidal Parametrics. *SIGGRAPH 83 Conference Proceedings*, ACM Press, New York, July 1983, pp 1-11.

[33] Stephanie Winner, Mike Kelley, Brent Pease, Bill Rivard & Alex Yen. Hardware Accelerated Rendering of Antialiasing Using a Modified A-buffer Algorithm. *SIGGRAPH 97 Conference Proceedings*, ACM Press, New York, August 1997, pp. 307-316.