Leo: A System for Cost Effective 3D Shaded Graphics

Michael F Deering, Scott R Nelson
Sun Microsystems Computer Corporation[†]

ABSTRACT

A physically compact, low cost, high performance 3D graphics accelerator is presented. It supports shaded rendering of triangles and antialiased lines into a double-buffered 24-bit true color frame buffer with a 24-bit Z-buffer. Nearly the only chips used besides standard memory parts are 11 ASICs (of four types). Special geometry data reformatting hardware on one ASIC greatly speeds and simplifies the data input pipeline. Floating-point performance is enhanced by another ASIC: a custom graphics microprocessor, with specialized graphics instructions and features. Screen primitive rasterization is carried out in parallel by five drawing ASICs, employing a new partitioning of the back-end rendering task. For typical rendering cases, the only system performance bottleneck is that intrinsically imposed by VRAM.

CR Categories and Subject Descriptors: C.1.2 [Processor Architectures]: Multiprocessors; I.3.1 [Computer Graphics]: Hardware Architecture; I.3.3 [Computer Graphics]: Picture/Image Generation *Display algorithms*; I.3.7 [Computer Graphics]: Three Dimensional Graphics and Realism.

Additional Keywords and Phrases: 3D graphics hardware, rendering, parallel graphics algorithms, gouraud shading, antialiased lines, floating-point microprocessors.

1 INTRODUCTION

To expand the role of 3D graphics in the mainstream computer industry, cost effective, physically small, usable performance 3D shaded graphics architectures must be developed. For such systems, new features and sheer performance at any price can no longer be the driving force behind the architecture; instead, the focus must be on affordable desktop systems.

The historical approach to achieving low cost in 3D graphics systems has been to compromise both performance and image quality. But now, falling memory component prices are bringing nearly ideal frame buffers into the price range of the volume market: double buffered 24-bit color with a 24-bit Z-buffer. The challenge is to drive these memory chips at their maximum rate with a minimum of supporting rendering chips, keeping the total system cost and physical size to an absolute minimum. To achieve this, graphics architectures must be repartitioned to reduce chip count and internal bus sizes, while still supporting existing 2D and 3D functionality.

This paper describes a new 3D graphics system, Leo, designed to these philosophies. For typical cases, Leo's only performance limit is that intrinsically imposed by VRAM. This was achieved by a combination of new architectural techniques and advances in VLSI technology. The result is a system without performance or image quality compromises, at an affordable cost and small physical size. The Leo board set is about the size of one and a half paperback novels; the complete workstation is slightly larger than two copies of Foley and Van Dam [7]. Leo supports both the traditional requirements of the 2D X window system and the needs of 3D rendering: shaded triangles, antialiased vectors, etc.

2 ARCHITECTURAL ALTERNATIVES

A generic pipeline for 3D shaded graphics is shown in Figure 1. ([7] Chapter 18 is a good overview of 3D graphics hardware pipeline issues.) This pipeline is truly generic, as at the top level nearly every commercial 3D graphics accelerator fits this abstraction. Where individual systems differ is in the partitioning of this rendering pipeline, especially in how they employ parallelism. Two major areas have been subject to separate optimization: the floating-point intensive initial stages of processing up to, and many times including, primitive set-up; and the drawing-intensive operation of generating pixels within a primitive and Z-buffering them into the frame buffer.

For low end accelerators, only portions of the pixel drawing stages of the pipeline are in hardware; the floating-point intensive parts of the pipe are processed by the host in software. As general purpose processors increase in floating-point power, such systems are starting to support interesting rendering rates, while minimizing cost [8]. But, beyond some limit, support of higher performance requires dedicated hardware for the entire pipeline.

There are several choices available for partitioning the floating-point intensive stages. Historically, older systems performed these tasks in a serial fashion [2]. In time though, breaking the pipe into more pieces for more parallelism (and thus performance) meant that each section was devoting more and more of its time to I/O overhead rather than to real work. Also, computational variance meant that many portions of the pipe would commonly be idle while others were overloaded. This led to the data parallel designs of most recent 3D graphics architectures [12].

[†]2550 Garcia Avenue, MTV18-212
Mountain View, CA 94043-1100
michael.deering@Eng.Sun.COM (415)336-3017
scott.nelson@Eng.Sun.COM (415)336-3106

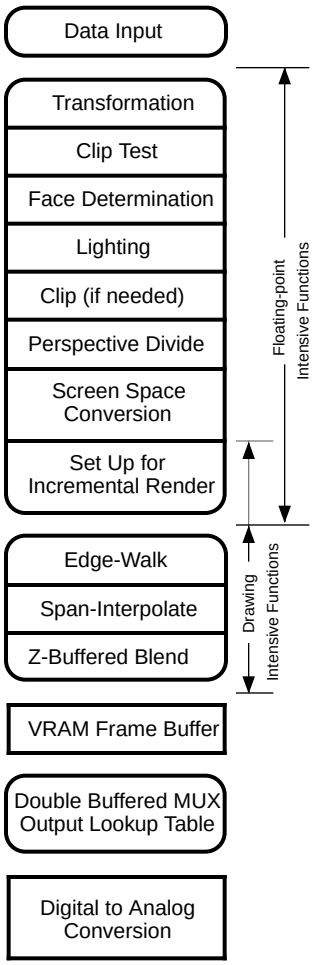

Figure 1: Generic 3D Graphics Pipeline

Here the concept is that multiple parallel computation units can each process the entire floating-point intensive task, working in parallel on different parts of the scene to be rendered. This allows each pipe to be given a large task to chew on, minimizing handshake overhead. But now there is a different load balancing problem. If one pipe has an extra large task, the other parallel pipes may go idle waiting for their slowest peer, if the common requirement of in-order execution of tasks is to be maintained. Minor load imbalances can be averaged out by adding FIFO buffers to the inputs and outputs of the parallel pipes. Limiting the maximum size of task given to any one pipe also limits the maximum imbalance, at the expense of further fragmenting the tasks and inducing additional overhead.

But the most severe performance bottleneck lies in the pixel drawing back-end. The most fundamental constraint on 3D computer graphics architecture over the last ten years has been the memory chips that comprise the frame buffer. Several research systems have attempted to avoid this bottleneck by various techniques [10][4][8], but all commercial workstation systems use conventional Z-buffer rendering algorithms into standard VRAMs or DRAMs. How this RAM is organized is an important defining feature of any high performance rendering system.

3 LEO OVERVIEW

Figure 2 is a diagram of the Leo system. This figure is *not* just a block diagram; it is also a *chip level* diagram, as every chip in the system is shown in this diagram. All input data and window system interactions enter through the LeoCommand chip. Geometry data is reformatted in this chip before being distributed to the array of LeoFloat chips below. The LeoFloat chips are microcoded specialized DSP-like processors that tackle the floating-point intensive stages of the rendering pipeline. The LeoDraw chips handle all screen space pixel rendering and are directly connected to the frame buffer RAM chips. LeoCross handles the back-end color look-up tables, double buffering, and video timing, passing the final digital pixel values to the RAMDAC.

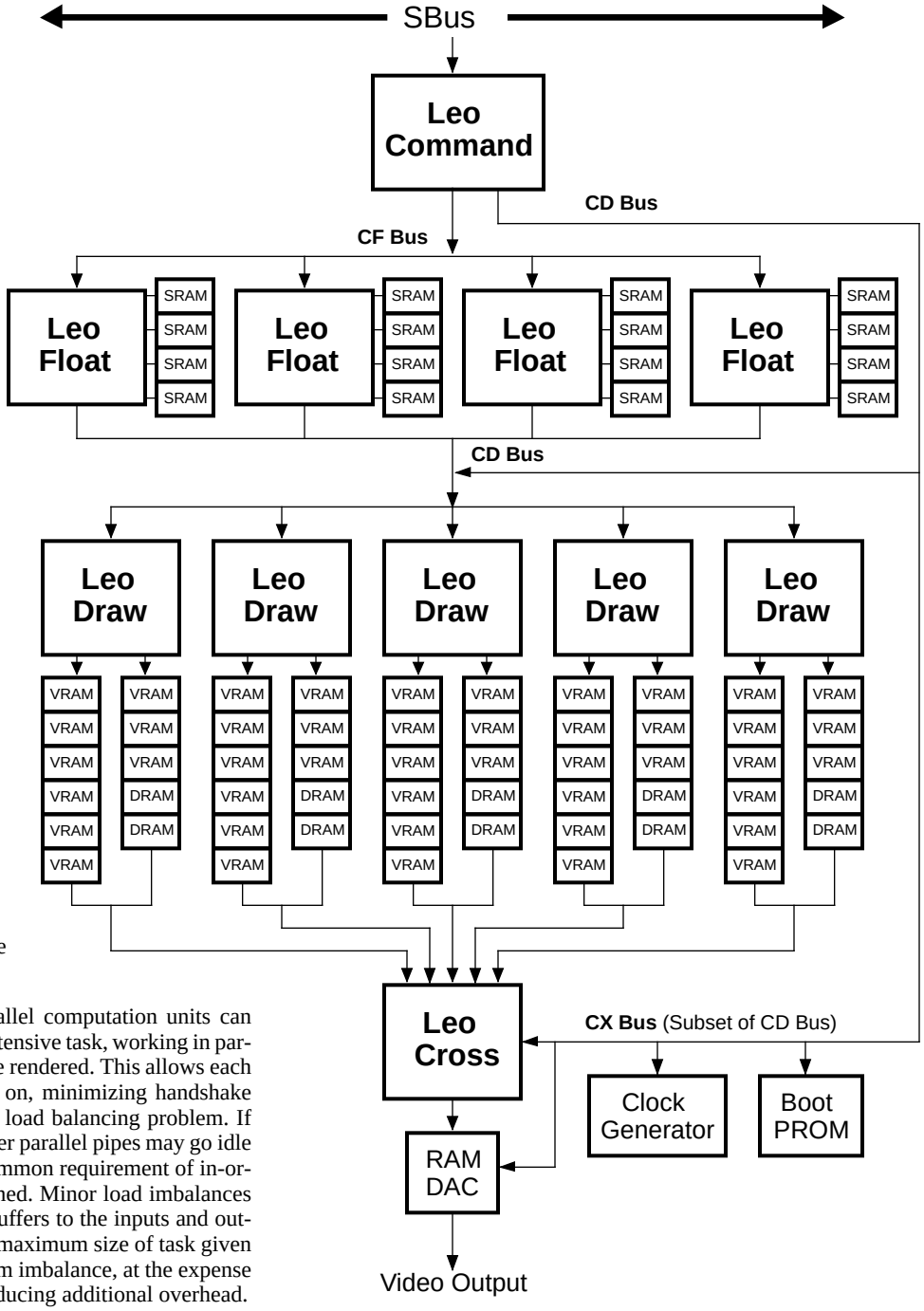

Figure 2: The Leo Block Diagram. Every chip in the system is represented in this diagram.

The development of the Leo architecture started with the constraints imposed by contemporary VRAM technology. As will be derived in the LeoDraw section below, these constraints led to the partitioning of the VRAM controlling LeoDraw chips, and set a maximum back-end rendering rate. This rate in turn set the performance goal for LeoFloat, as well as the data input bandwidth and processing rate for LeoCommand. After the initial partitioning of the rendering pipeline into these chips, each chip was subjected to additional optimization. Throughput bottlenecks in input geometry format conversion, floating-point processing, and pixel rendering were identified and overcome by adding reinforcing hardware to the appropriate chips.

Leo's floating-point intensive section uses data parallel partitioning. LeoCommand helps minimize load balancing problems by breaking down rendering tasks to the smallest isolated primitives: individual triangles, vectors, dots, portions of pixel rasters, rendering attributes, etc., at the cost of precluding optimizations for shared data in triangle strips and polylines. This was considered acceptable due to the very low average strip length empirically observed in real applications. The overhead of splitting geometric data into isolated primitives is minimized by the use of dedicated hardware for this task. Another benefit of converting all rendering operations to isolated primitives is that down-stream processing of primitives is considerably simplified by only needing to focus on the isolated case.

4 INPUT PROCESSING: LEOCOMMAND

Feeding the pipe

Leo supports input of geometry data both as programmed I/O and through DMA. The host CPU can directly store up to 32 data words in an internal LeoCommand buffer without expensive read back testing of input status every few words. This is useful on hosts that do not support DMA, or when the host must perform format conversions beyond those supported in hardware. In DMA mode, LeoCommand employs efficient block transfer protocols on the system bus to transfer data from system memory to its input buffer, allowing much higher bandwidth than simple programmed I/O. Virtual memory pointers to application's geometry arrays are passed directly to LeoCommand, which converts them to physical memory addresses without operating system intervention (except when a page is marked as currently non-resident). This frees the host CPU to perform other computations during the data transfer. Thus the DMA can be efficient even for pure immediate-mode applications, where the geometry is being created on the fly.

Problem: Tower of Babel of input formats

One of the problems modern display systems face is the explosion of different input formats for similar drawing functions that need to be supported. Providing optimized microcode for each format rapidly becomes unwieldy. The host CPU could be used to pretranslate the primitive formats, but at high speeds this conversion operation can itself become a system bottleneck. Because DMA completely bypasses the host CPU, LeoCommand includes a programmable format conversion unit in the geometry data pipeline. This reformatter is considerably less complex than a general purpose CPU, but can handle the most commonly used input formats, and at very high speeds.

The geometry reformatting subsystem allows several orthogonal operations to be applied to input data. This geometric input data is abstracted as a stream of vertex packets. Each vertex packet may contain any combination of vertex position, vertex normal, vertex color, facet normal, facet color, texture map coordinates, pick IDs, headers, and other information. One conversion supports arbitrary re-ordering of data within a vertex, allowing a standardized element order after reformatting. Another operation supports the conversion of multiple numeric formats to 32-bit IEEE floating-point. The source data can be 8-bit or 16-bit fixed-point, or 32-bit or 64-bit IEEE floating-point. Additional miscellaneous reformatting allows the stripping of headers and other fields, the addition of an internally generated sequential pick ID, and insertion of constants. The final reformatting stage re-packages vertex packets into complete isolated geometry primitives (points, lines, triangles). Chaining bits in vertex headers delineate which vertices form primitives.

Like some other systems, Leo supports a generalized form of triangle strip (see Figure 3), where vertex header bits within a strip specify how the incoming vertex should be combined with previous vertices to form the next triangle. A stack of the last three vertices used to form a triangle is kept. The three vertices are labeled oldest, middle, and newest. An incoming vertex of type *replace_oldest* causes the oldest vertex to be replaced by the middle, the middle to be replaced by the newest, and the incoming vertex becomes the newest. This corresponds to a PHIGS PLUS triangle strip (sometimes called a "zig-zag" strip). The replacement type *replace_middle* leaves the oldest vertex unchanged, replaces the middle vertex by the newest, and the incoming vertex becomes the newest. This corresponds to a triangle star. The replacement type *restart* marks the oldest and middle vertices as invalid, and the incoming vertex becomes the newest. Generalized triangle strips must always start with this code. A triangle will be output only when a replacement operation results in three valid vertices. *Restart* corresponds to a "move" operation in polylines, and allows multiple unconnected variable-length triangle strips to be described by a single data structure passed in by the user,

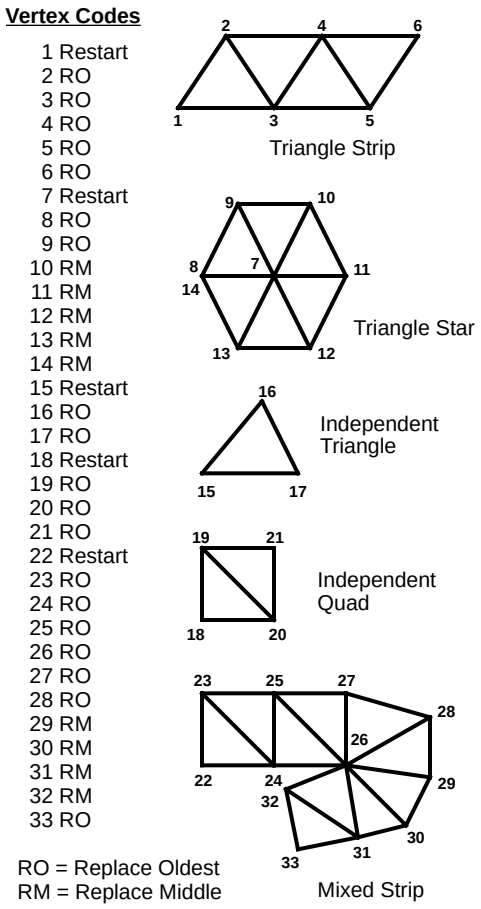

Figure 3: A Generalized Triangle Strip

reducing the overhead. The generalized triangle strip's ability to effectively change from "strip" to "star" mode in the middle of a strip allows more complex geometry to be represented compactly, and requires less input data bandwidth. The restart capability allows several pieces of disconnected geometry to be passed in one DMA operation. Figure 3 shows a *single* generalized triangle strip, and the associated replacement codes. LeoCommand also supports headerless strips of triangle vertices either as pure strips, pure stars, or pure independent triangles.

LeoCommand hardware automatically converts generalized triangle strips into isolated triangles. Triangles are normalized such that the front face is always defined by a clockwise vertex order after transformation. To support this, a header bit in each *restart* defines the initial face order of each sub-strip, and the vertex order is reversed after every *replace_oldest*. LeoCommand passes each completed triangle to the next available LeoFloat chip, as indicated by the input FIFO status that each LeoFloat sends back to LeoCommand. The order in which triangles have been sent to each LeoFloat is scoreboarded by LeoCommand, so that processed triangles are let out of the LeoFloat array in the same order as they entered. Non-sequential rendering order is also supported, but the automatic rendering task distribution hardware works so well that the performance difference is less than 3%. A similar, but less complex vertex repackaging is supported for polylines and multi-polylines via a move/draw bit in the vertex packet header.

To save IC pins and PC board complexity, the internal Leo data busses connecting LeoCommand, LeoFloat, and LeoDraw are 16 bits in size. When colors, normals, and texture map coefficients are being transmitted on the CF-bus between LeoCommand and the LeoFloats, these components are (optionally) compressed from 32-bit IEEE floating-point into 16-bit fixed point fractions by LeoCommand, and then automatically reconverted back to 32-bit IEEE floating-point values by LeoFloat. This quantization does not effect quality. Color components will eventually end up as 8-bit values in the frame buffer. For normals, 16-bit (signed) accuracy represents a resolution of approximately plus or minus an inch at one mile. This optimization reduces the required data transfer bandwidth by 25%.

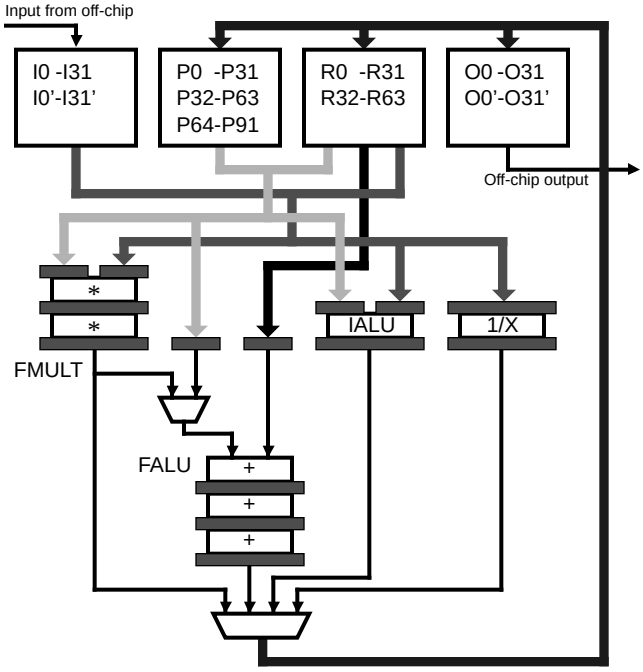

Figure 4: LeoFloat arithmetic function units, registers and data paths.

5 FLOATING-POINT PROCESSING: LEOFLOAT

After canonical format conversion, the next stages of processing triangles in a display pipeline are: transformation, clip test, face determination, lighting, clipping (if required), screen space conversion, and set-up. These operations are complex enough to require the use of a general purpose processor.

Use of commercially available DSP (Digital Signal Processing) chips for this work has two major drawbacks. First, most such processors require a considerable number of surrounding glue chips, especially when they are deployed as multi-processors. These glue chips can easily quadruple the board area dedicated to the DSP chip, as well as adversely affecting power, heat, cost, and reliability. Second, few of these chips have been optimized for 3D graphics.

A better solution might be to augment the DSP with a special ASIC that would replace all of these glue chips. Given the expense of developing an ASIC, we decided to merge that ASIC with a custom DSP core optimized for graphics.

The resulting chip was LeoFloat. LeoFloat combines a 32-bit microcodable floating-point core with concurrent input and output packet communication subsystems (see Figure 4.), similar to the approach of [3]. The only support chips required are four SRAM chips for external microcode store. A number of specialized graphics instructions and features make LeoFloat different from existing DSP processors. Each individual feature only makes a modest incremental contribution to performance, and indeed many have appeared in other designs. What is novel about LeoFloat is the combination of features, whose cumulative effect leads to impressive overall system performance. The following sections describe some of the more important special graphics instructions and features.

Double buffered asynchronous I/O register files. All input and output commands are packaged up by separate I/O packet hardware. Variable length packets of up to 32 32-bit words are automatically written into (or out of) on-chip double-buffered register files (the I and O registers). These are mapped directly into microcode register space. Special instructions allow complete packets to be requested, relinquished, or queued for transmission in one instruction cycle.

Enough internal registers. Most commercial DSP chips support a very small number of internal fast registers, certainly much smaller than the data needed by the inner loops of most 3D pipeline algorithms. They attempt to make up for this with on-chip SRAM or data caches, but typically SRAMs are not multi-ported and the caches not user-schedulable. We cheated with LeoFloat. We first wrote the code for the largest important inner loop (triangles), counted how many registers were needed (288), and built that many into the chip.

Parallel internal function units. The floating-point core functions (32-bit IEEE format) include multiply, ALU, reciprocal, and integer operations, all of which can often be executed in parallel. It is particularly important that the floating-point reciprocal operation not tie up the multiply and add units, so that perspective or slope calculations can proceed in parallel with the rest of geometric processing. Less frequently used reciprocal square root hardware is shared with the integer function unit.

Put all non-critical algorithms on the host. We avoided the necessity of building a high level language compiler (and support instructions) for LeoFloat by moving any code not worth hand coding in microcode to the host processor. The result is a small, clean kernel of graphics routines in microcode. (A fairly powerful macro-assembler with a 'C'-like syntax was built to support the hand coding.)

Software pipeline scheduling. One of the most complex parts of modern CPUs to design and debug is their scoreboard section, which schedules the execution of instructions across multiple steps in time and function units, presenting the programmer with the

illusion that individual instructions are executed in one shot. LeoFloat avoided all this hardware by using more direct control fields, like horizontal microprogrammable machines, and leaving it to the assembler (and occasionally the programmer) to skew one logical instruction across several physical instructions.

Special clip condition codes & clip branch. For clip testing we employ a modified Sutherland-Hodgman algorithm, which first computes a vector of clip condition bits. LeoFloat has a clip test instruction that computes these bits two at a time, shifting them into a special clip-bits register. After the bits have been computed, special branch instructions decode these bits into the appropriate case: clip rejected, clip accepted, single edge clip (six cases), or needs general clipping. There are separate branch instructions for triangles and vectors. (A similar approach was taken in [9].) The branch instructions allow multiple other conditions to be checked at the same time, including backfacing and model clipping.

Register Y sort instruction. The first step of the algorithm we used for setting up triangles for scan conversion sorts the three triangle vertices in ascending Y order. On a conventional processor this requires either moving a lot of data, always referring to vertex data through indirect pointers, or replicating the set-up code for all six possible permutations of triangle vertex order. LeoFloat has a special instruction that takes the results of the last three comparisons and reorders part of the R register file to place vertices in sorted order.

Miscellaneous. LeoFloat contains many performance features traditionally found on DSP chips, including an internal subroutine stack, block load/store SRAM, and integer functions. Also there is a "kitchen sink" instruction that initiates multiple housekeeping functions in one instruction, such as "transmit current output packet (if not clip pending), request new input packet, extract op-code and dispatch to next task."

Code results: equivalent to 150 megaflop DSP. Each 25 MHz LeoFloat processes the benchmark isolated triangle (including cliptest and set-up) in 379 clocks. (With a few exceptions, microcode instructions issue at a rate of one per clock tick.) The same graphics algorithm was tightly coded on several RISC processors and DSP chips (SPARC, i860, C30, etc.), and typically took on the order of 1100 clocks. Thus the 379 LeoFloat instruction at 25 MHz do the equivalent work of a traditional DSP chip running at 75 MHz (even though there are only 54 megaflops of hardware). Of course these numbers only hold for triangles and vectors, but that's most of what LeoFloat does. Four LeoFloats assure that floating-point processing is not the bottleneck for 100-pixel isolated, lighted triangles.

6 SCREEN SPACE RENDERING: LEODRAW

VRAM limits

Commercial VRAM chips represent a fundamental constraint on the possible pixel rendering performance of Leo's class of graphics accelerator. The goal of the Leo architecture was to ensure to the greatest extent possible that this was the *only* performance limit for typical rendering operations.

The fundamental memory transaction for Z-buffered rendering algorithms is a conditional read-modify-write cycle. Given an XY address and a computed RGBZ value, the old Z value at the XY address is first read, and then if the computed Z is in front of the old Z, the computed RGBZ value is written into the memory. Such transactions can be mapped to allowable VRAM control signals in many different ways: reads and writes may be batched, Z may be read out through the video port, etc.

VRAM chips constrain system rendering performance in two ways. First, they impose a minimum cycle time per RAM bank for the Z-buffered read-modify-write cycle. Figure 5 is a plot of this cycle

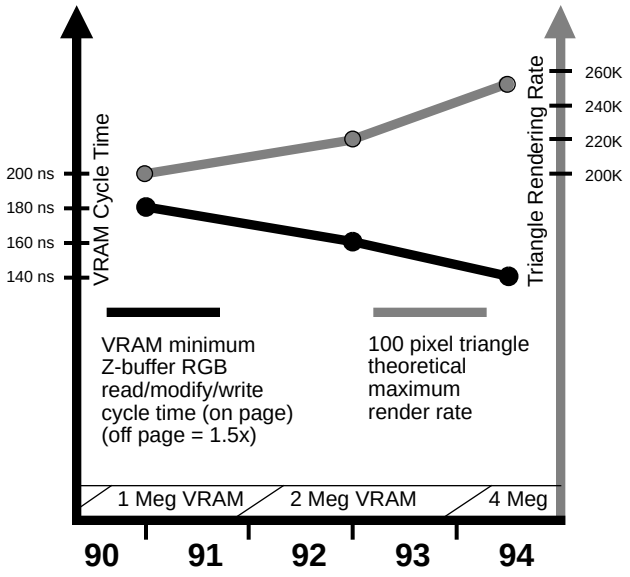

Figure 5: VRAM cycle time and theoretical maximum triangle rendering rate (for five-way interleaved frame buffers).

time (when in "page" mode) and its changes over a half-decade period. VRAMs also constrain the ways in which a frame buffer can be partitioned into independently addressable banks. Throughout the five year period in Figure 5, three generations of VRAM technology have been organized as 256K by 4, 8, and 16-bit memories. For contemporary display resolutions of 1280×1024, the chips comprising a minimum frame buffer can be organized into no more than five separately-addressed interleave banks. Combining this information, a theoretical maximum rendering speed for a primitive can be computed. The second line in Figure 5 is the corresponding performance for rendering 100-pixel Z-buffered triangles, including the overhead for entering page mode, content refresh, and video shift register transfers (video refresh). Higher rendering rates are only possible if additional redundant memory chips are added, allowing for higher interleaving factors, at the price of increased system cost.

Even supporting five parallel interleaves has a cost: at least 305 memory interface pins (five banks of (24 RGB + 24 Z + 13 address/control)) are required, more pins than it is currently possible to dedicate to a memory interface on one chip. Some systems have used external buffer chips, but on a minimum cost and board area system, this costs almost as much as additional custom chips. Thus, on the Leo system we opted for five separate VRAM control chips (LeoDraws).

Triangle scan conversion

Traditional shaded triangle scan conversion has typically been via a linear pipeline of edge-walking followed by scan interpolation [12]. There have been several approaches to achieving higher throughput in rasterization. [2] employed a single edge-walker, but parallel scan interpolation. [4][10] employed massively parallel rasterizers. [6] and other recent machines use moderately parallel rasterizers, with additional logic to merge the pixel rasterization streams back together.

In the Leo design we chose to broadcast the identical triangle specification to five parallel rendering chips, each tasked with rendering only those pixels visible in the local interleave. Each chip performs its own complete edge-walk and span interpolation of the triangle, biased by the chip's local interleave. By paying careful attention to proper mathematical sampling theory for rasterized pixels, the five

chips can act in concert to produce the correct combined rasterized image. Mathematically, each chip thinks it is rasterizing the triangle into an image memory with valid pixel centers only every five original pixels horizontally, with each chip starting off biased one more pixel to the right.

To obtain the speed benefits of parallel chips, most high performance graphics systems have split the edge-walk and span-interpolate functions into separate chips. But an examination of the relative amounts of data flow between rendering pipeline stages shows that the overall peak data transfer bandwidth demand occurs between the edge-walk and span-interpolate sections, induced by long thin triangles, which commonly occur in tessellated geometry. To minimize pin counts and PC board bus complexity, Leo decided to replicate the edge-walking function into each of the five span-interpolation chips.

One potential drawback of this approach is that the edge-walking section of each LeoDraw chip will have to advance to the next scan line up to five times more often than a single rasterization chip would. Thus LeoDraw's edge-walking circuit was designed to operate in one single pixel cycle time (160 ns. read-modify-write VRAM cycle), so it would never hold back scan conversion. Other usual pipelining techniques were used, such as loading in and buffering the next triangle to be drawn in parallel with rasterizing the current triangle. Window clipping, blending, and other pixel post processing are handled in later pipelined stages.

Line scan conversion

As with triangles, the mathematics of the line rasterization algorithms were set up to allow distributed rendering of aliased and antialiased lines and dots, with each LeoDraw chip handling the 1/5 of the frame buffer pixels that it owns. While the Leo system uses the X11 semantics of Bresenham lines for window system operations, these produce unacceptable motion artifacts in 3D wireframe rendering. Therefore, when rendering 3D lines, Leo employs a high-accuracy DDA algorithm, using 32 bits internally for sufficient subpixel precision.

At present there is no agreement in the industry on the definition of a high quality antialiased line. We choose to use the image quality of vector strokers of years ago as our quality standard, and we tested different algorithms with end users, many of whom were still using calligraphic displays. We found users desired algorithms that displayed no roping, angle sensitivities, short vector artifacts, or end-point artifacts. We submitted the resulting antialiased line quality test patterns as a GPC [11] test image. In achieving the desired image quality level, we determined several properties that a successful line antialiasing algorithm must have. First, the lines must have at least three pixels of width across the minor axis. Two-pixel wide antialiased lines exhibit serious roping artifacts. Four-pixel wide lines offer no visible improvement except for lines near 45 degrees. Second, proper endpoint ramps spread over at least two pixels are necessary both for seamless line segment joins as well as for isolated line-ends. Third, proper care must be taken when sampling lines of subpixel length to maintain proper final intensity. Fourth, intensity or filter adjustments based on the slope are necessary to avoid artifacts when rotating wireframe images. To implement all this, we found that we needed at least four bits of subpixel positional accuracy *after* cumulative interpolation error is factored in. That is why we used 32 bits for XY coordinate accuracy: 12 for pixel location, 4 for subpixel location, and 16 for DDA interpolation error. (The actual error limit is imposed by the original, user-supplied 32-bit IEEE floating-point data.)

Because of the horizontal interleaving and preferred scan direction, the X-major and Y-major aliased and antialiased line rasterization algorithms are not symmetric, so separate optimized algorithms were employed for each.

Antialiased dots

Empirical testing showed that only three bits of subpixel precision are necessary for accurate rendering of antialiased dots. For ASIC implementation, this was most easily accomplished using a brute-force table lookup of one of 64 precomputed 3×3 pixel dot images. These images are stored in on-chip ROM, and were generated using a circular symmetric Gaussian filter.

Triangle, line, and dot hardware

Implementation of the triangle and antialiased vector rasterization algorithms require substantial hardware resources. Triangles need single pixel cycle edge-walking hardware in parallel with RGBZ span interpolation hardware. To obtain the desired quality of antialiased vectors, our algorithms require hardware to apply multiple waveform shaping functions to every generated pixel. As a result, the total VLSI area needed for antialiased vectors is nearly as large as for triangles. To keep the chip die size reasonable, we reformulated both the triangle and antialiased vector algorithms to combine and reuse the same function units. The only difference is how the separate sequencers set up the rasterization pipeline.

Per-pixel depth cue

Depth cueing has long been a heavily-used staple of wireframe applications, but in most modern rendering systems it is an extra time expense feature, performed on endpoints back in the floating-point section. We felt that we were architecting Leo not for benchmarks, but for users, and many wireframe users want to have depth cueing on all the time. Therefore, we built a parallel hardware depth cue function unit into each LeoDraw. Each triangle, vector, or dot rendered by Leo can be optionally depth cued at absolutely no cost in performance. Another benefit of per-pixel depth cueing is full compliance with the PHIGS PLUS depth cueing specification. For Leo, per-pixel depth cueing hardware also simplifies the LeoFloat microcode, by freeing the LeoFloats from ever having to deal with it.

Picking support

Interactive graphics requires not only the rapid display of geometric data, but also interaction with that data: the ability to pick a particular part or primitive within a part. Any pixels drawn within the bounds of a 3D pick aperture result in a pick hit, causing the current pick IDs to be automatically DMAed back to host memory.

Window system support

Many otherwise sophisticated 3D display systems become somewhat befuddled when having to deal simultaneously with 3D rendering applications and a 2D window system. Modern window systems on interactive workstations require frequent context switching of the rendering pipeline state. Some 3D architectures have tried to minimize the overhead associated with context switching by supporting multiple 3D contexts in hardware. Leo goes one step further, maintaining two completely separate pipelines in hardware: one for traditional 2D window operations; the other for full 3D rendering. Because the majority of context switch requests are for 2D window system operations, the need for more complex 3D pipeline context switching is significantly reduced. The 2D context is much lighter weight and correspondingly easier to context switch. The two separate graphics pipelines operate completely in parallel, allowing simultaneous access by two independent CPUs on a multiprocessor host.

2D functionality abstracts the frame buffer as a 1-bit, 8-bit, or 24-bit pixel array. Operations include random pixel access, optimized character cell writes, block clear, block copy, and the usual menagerie of

boolean operations, write masks, etc. Vertical block moves are special cased, as they are typically used in vertical scrolling of text windows, and can be processed faster than the general block move because the pixel data does not have to move across LeoDraw chip interleaves. Rendering into non-rectangular shaped windows is supported by special clip hardware, resulting in no loss in performance. A special block clear function allows designated windows (and their Z-buffers) to be initialized to any given constant in under 200 microseconds. Without this last feature, 30 Hz or faster animation of non-trivial objects would have been impossible.

7 VIDEO OUTPUT: LEOCROSS

Leo's standard video output format is 1280×1024 at 76 Hz refresh rate, but it also supports other resolutions, including 1152×900, interlaced 640×480 RS-170 (NTSC), interlaced 768×576 PAL timing, and 960×680 113 Hz field sequential stereo. LeoCross contains several color look-up tables, supporting multiple pseudo color maps without color map flashing. The look-up table also supports two different true color abstractions: 24-bit linear color (needed by rendering applications), and REC-709 non-linear color (required by many imaging applications).

Virtual reality support

Stereo output is becoming increasingly important for use in Virtual Reality applications. Leo's design goals included support for the Virtual Holographic Workstation system configuration described in [5]. Leo's stereo resolution was chosen to support square pixels, so that lines and antialiased lines are displayed properly in stereo, and standard window system applications can co-exist with stereo. Stereo can be enabled on a per-window basis (when in stereo mode windows are effectively quad-buffered). Hooks were included in LeoCross to support display technologies other than CRT's, that may be needed for head-mounted virtual reality displays.

8 NURBS AND TEXTURE MAP SUPPORT

One of the advantages to using programmable elements within a graphics accelerator is that additional complex functionality, such as NURBS and texture mapping, can be accelerated. Texture mapping is supported through special LeoFloat microcode and features of LeoCommand. LeoFloat microcode also includes algorithms to accelerate dynamic tessellation of trimmed NURBS surfaces. The dynamic tessellation technique involves reducing trimmed NURBS surfaces into properly sized triangles according to a display/pixel space approximation criteria [1]; i.e. the fineness of tessellation is view dependent. In the past, dynamic tessellation tended to be mainly useful as a compression technique, to avoid storing all the flattened triangles from a NURBS surface in memory. Dynamic tessellation was not viewed as a performance enhancer, for while it might generate only a third as many triangles as a static tessellation, the triangles were generated at least an order of magnitude or more slower than brute force triangle rendering. In addition it had other problems, such as not handling general trimming. For many cases, Leo's dynamic tesselator can generate and render triangles only a small integer multiple slower than prestored triangle rendering, which for some views, can result in *faster* overall object rendering.

9 RESULTS

Leo is physically a-two board sandwich, measuring $5.7 \times 6.7 \times 0.6$ inches, that fits in a standard 2S SBus slot. Figure 6 is a photo of the two boards, separated, showing all the custom ASICs. Figure 7 is a photo of the complete Leo workstation, next to two of our units of scale and the board set.

Leo can render 210K 100-pixel isolated, lighted, Gouraud shaded, Z-buffered, depth cued triangles per second, with one infinite diffuse and one ambient light source enabled. At 100 pixels, Leo is still VRAM rendering speed limited; smaller triangles render faster. Isolated 10-pixel antialiased, constant color, Z-buffered, depth cued lines (which are actually 12 pixels long due to endpoint ramps, and three pixels wide) render at a 422K per second rate. Corresponding aliased lines render at 730K. Aliased and antialiased constant color, Z-buffered, depth cued dots are clocked at 1100K. 24-bit image rasters can be loaded onto the screen at a 10M pixel per second rate. Screen scrolls, block moves, and raster character draws all also have competitive performance. Figure 8 is a sample of shaded triangle rendering.

10 SIMULATION

A system as complex as Leo cannot be debugged after the fact. All the new rendering mathematics were extensively simulated before being committed to hardware design. As each chip was defined, high, medium, and low level simulators of its function were written and continuously used to verify functionality and performance. Complete images of simulated rendering were generated throughout the course of the project, from within weeks of its start. As a result, the window system and complex 3D rendering were up and running on a complete board set within a week of receiving the first set of chips.

11 CONCLUSIONS

By paying careful attention to the forces that drive both performance and cost, a physically compact complete 3D shaded graphics accelerator was created. The focus was not on new rendering features, but on cost reduction and performance enhancement of the most useful core of 3D graphics primitives. New parallel algorithms were developed to allow accurate screen space rendering of primitives. Judicious use of hardware to perform some key traditional software functions (such as format conversion and primitive vertex reassembly) greatly simplified the microcode task. A specialized floating-point core optimized for the primary task of processing lines and triangles also supports more general graphics processing, such as rasters and NURBS. The final system performance is limited by the *only* chips not custom designed for Leo: the standard RAM chips.

ACKNOWLEDGEMENTS

The authors would like to thank the entire Leo team for their efforts in producing the system, and Mike Lavelle for help with the paper.

REFERENCES

1. **Abi-Ezzi, Salim, and L. Shirman.** Tessellation of Curved Surfaces under Highly Varying Transformations. Proc. Eurographics '91 (Vienna, Austria, September 1991), 385-397.
2. **Akeley, Kurt and T. Jermoluk.** High-Performance Polygon Rendering, Proceedings of SIGGRAPH '88 (Atlanta, GA, Aug 1-5, 1988). In *Computer Graphics* 22, 4 (July 1988), 239-246.
3. **Anido, M., D. Allerton and E. Zaluska.** MIGS - A Multiprocessor Image Generation System using RISC-like Microprocessors. Proceedings of CGI '89 (Leeds, UK, June 1989), Springer Verlag 1990.
4. **Deering, Michael, S. Winner, B. Schediwy, C. Duffy and N. Hunt.** The Triangle Processor and Normal Vector Shader: A VLSI system for High Performance Graphics. Proceedings of SIGGRAPH '88 (Atlanta, GA, Aug 1-5, 1988). In *Computer Graphics* 22, 4 (July 1988), 21-30.

5. **Deering, Michael.** High Resolution Virtual Reality. Proceedings of SIGGRAPH '92 (Chicago, IL, July 26-31, 1992). In *Computer Graphics* 26, 2 (July 1992), 195-202.
6. **Dunnett, Graham, M. White, P. Lister and R. Grimsdale.** The Image Chip for High Performance 3D Rendering. *IEEE Computer Graphics and Applications* 12, 6 (November 1992), 41-52.
7. **Foley, James, A. van Dam, S. Feiner and J Hughes.** Computer Graphics: Principles and Practice, 2nd ed., Addison-Wesley, 1990.
8. **Kelley, Michael, S. Winner, K. Gould.** A Scalable Hardware Render Accelerator using a Modified Scanline Algorithm. Proceedings of SIGGRAPH '92 (Chicago, IL, July 26-31, 1992). In *Computer Graphics* 26, 2 (July 1992), 241-248.
9. **Kirk, David, and D. Voorhies.** The Rendering Architecture of the DN10000VS. Proceedings of SIGGRAPH '90 (Dallas, TX, August 6-10, 1990). In *Computer Graphics* 24, 4 (August 1990), 299-307.
10. **Molnar, Steven, J. Eyles, J. Poulton.** PixelFlow: High-Speed Rendering Using Image Composition. Proceedings of SIGGRAPH '92 (Chicago, IL, July 26-31, 1992). In *Computer Graphics* 26, 2 (July 1992), 231-240.
11. **Nelson, Scott.** GPC Line Quality Benchmark Test. GPC Test Suite, NCGA GPC committee 1991.
12. **Torborg, John.** A Parallel Processor Architecture for Graphics Arithmetic Operations. Proceedings of SIGGRAPH '87 (Anaheim, CA, July 27-31, 1987). In *Computer Graphics* 21, 4 (July 1987), 197-204.

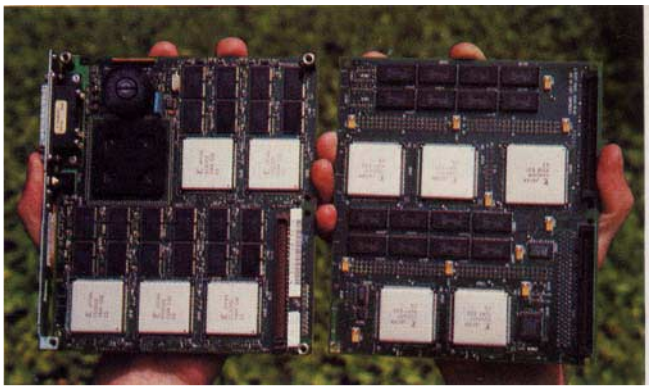

Figure 6: The two boards, unfolded.

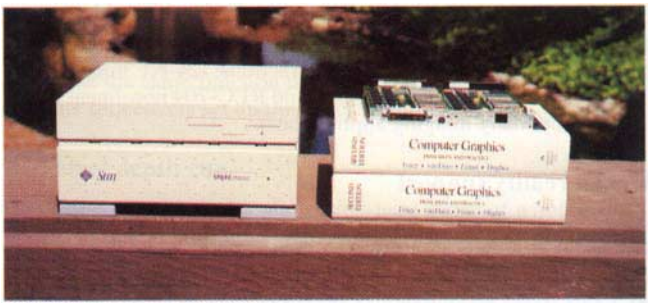

Figure 7: The complete SPARCstation ZX workstation, next to two of our units of scale and the Leo board set.

Figure 8: **Traffic Jam to Point Reyes**. A scene containing 2,322,000 triangles, rendered by Leo Hardware. Stochastically super-sampled 8 times. Models courtesy of Viewpoint Animation Engineering.