PixelFlow: The Realization

John Eyles[*], *Steven Molnar*[*], *John Poulton*[†], *Trey Greer*[*], *Anselmo Lastra*[†], *Nick England*[†], *Lee Westover*[*]

[*] Hewlett-Packard Company
Chapel Hill Graphics Lab

[†] Department of Computer Science
University of North Carolina

ABSTRACT

PixelFlow is an architecture for high-speed, highly realistic image generation, based on the techniques of object-parallelism and image composition. Its initial architecture was described in [MOLN92]. After development by the original team of researchers at the University of North Carolina, and co-development with industry partners, Division Ltd. and Hewlett-Packard, PixelFlow now is a much more capable system than initially conceived and its hardware and software systems have evolved considerably. This paper describes the final realization of PixelFlow, along with hardware and software enhancements heretofore unpublished.

CR Categories and Subject Descriptors: C.5.4 [Computer System Implementation]: VLSI Systems; I.3.1 [Computer Graphics]: Hardware Architecture; I.3.3 [Computer Graphics]: Picture/Image Generation; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism.

Additional Key Words and Phrases: object-parallel, rendering, compositing, deferred shading, scalable.

1 INTRODUCTION

PixelFlow is an architecture for high-speed image generation that was designed to be linearly scaleable to unprecedented levels of performance and to implement realistic rendering techniques such as user-programmable shading, texturing, antialiasing, and shadows. Achieving these goals required a new architecture that was substantially different than the interleaved screen-subdivision approach that is nearly universal in today's commercial graphics architectures (*e.g.* [E&S96;SGI97]).

Author addresses:

[*] 1512 East Franklin St., Suite 200, Chapel Hill, NC 27514
{jge,molnar,greer,lee}@chapelhill.hp.com

[†] Sitterson Hall C.B #3175, Chapel Hill, NC 27599-3175
{jp,lastra,nick}@cs.unc.edu

PixelFlow uses an object-parallel approach called image-composition to achieve its high speed. Display primitives are distributed over an array of identical renderers, each of which computes a full-screen image of its fraction of the primitives. A dedicated, high-speed communication network called the *Image-Composition Network* merges these images in real time, based on visibility information, to produce an image of the entire scene [MOLN92].

The PixelFlow architecture is extremely flexible, allowing configurations from deskside systems, drawing tens of millions of triangles per second, to multiple-rack systems, drawing hundreds of millions of triangles per second. Near-linear performance increases are obtained by adding renderers.

1.1 System Overview

A PixelFlow system consists of one or more chassis, each containing up to 9 *Flow Units* (the PixelFlow name for renderer). Each Flow Unit consists of: a *Geometry Processor Board* (GP), a conventional floating-point microprocessor with DRAM memory; and a *Rasterizer Board* (RB), a SIMD array of 8,192 byte-serial processing elements, each with 384 bytes of local memory.

A Flow Unit is a powerful graphics engine in itself, capable of rendering up to 3 million antialiased polygons per second and performing complex shading calculations (such as bump mapping, shadows, and user-programmable shading) in real time. Geometry Processor Boards provide the front-end floating-point computation needed for transforming primitives and generating rendering instructions for the Rasterizer Boards. Rasterizer Boards turn screen-space descriptions of primitives into pixel values and perform sophisticated shading calculations.

The Image-Composition Network is implemented as a daisy-chained connection between Rasterizer Boards of neighboring Flow Units. A second communication network, the *Geometry Network*, is a packet-routing network which connects Geometry Processor Boards.

Any subset of Flow Units can be provided with I/O or video adapter daughter-cards that provide host-interface or video (frame buffer or frame grabber) capabilities. It is possible to build very large systems with multiple host interfaces attached to a parallel host, and multiple displays to support multiple-user applications. Such a system also can be re-configured by software into several smaller systems. Figure 1 shows a typical two-chassis PixelFlow system configuration.

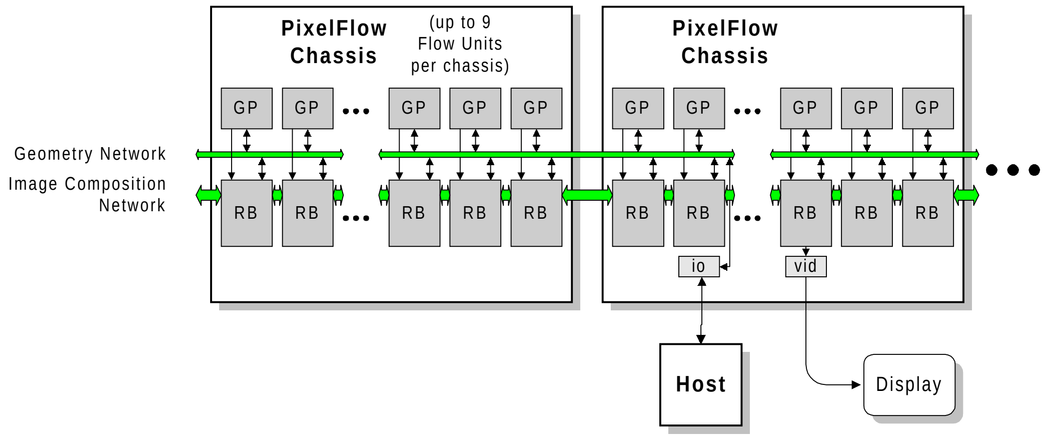


Figure 1: Typical PixelFlow System.

1.2 System Operation

Individual Flow Units can be designated, by software, as one of three types:

- **Renderers** (not to be confused with the general term *renderer* above) process a portion of the database to generate regions of pixel data ready for shading. The Geometry Processor Board transforms primitives to screen-space and sorts them into bins according to screen region. The Rasterizer Board rasterizes primitives one region at a time. After all renderers have processed a given region, the region is composited across the Image-Composition Network and the composited pixel-values are deposited onto one or more shaders.
- **Shaders** apply texture and lighting models to regions of raw pixel data, producing RGB color values that are forwarded to the frame buffer.
- **Frame buffers** send or receive video data via an attached video adapter card.

To compute a frame, the GPs on each renderer first transform their fraction of the primitives into screen coordinates and sort them into bins corresponding to regions of the screen. The renderers then process the regions one at a time, rasterizing all of the primitives that affect the current region before moving on to the next.

Once a given region has been rasterized on all of the renderers, the composition network merges the pixel data together and loads the region of composited pixel data onto a shader. Regions are assigned to shaders in round-robin fashion, with each shader processing every *n*th region. Shaders operate on entire regions in parallel, to convert raw pixel attributes into final RGB values, blend multiple samples together for antialiasing, and forward final color values to the frame buffer for display.

1.3 Design Evolution

The PixelFlow architecture has evolved considerably since its initial conception, described in [MOLN92]. PixelFlow was initially developed at the University of North Carolina at Chapel Hill as an NSF- and DARPA-sponsored research project. In 1994, Division Ltd. of Bristol, UK acquired commercial rights to the technology and established a laboratory in Chapel Hill to complete development of the project. In mid-1996, this laboratory and rights to the PixelFlow technology were acquired by Hewlett-Packard. The final design is significantly faster, more complex, and technically more aggressive than originally conceived.

The following sections describe the final PixelFlow architecture in more detail. Special attention will be given to aspects of the architecture that have not been described before.

2 ARCHITECTURAL FEATURES

PixelFlow was designed to demonstrate the advantages of image-composition architectures, to provide a research platform for real-time 3D graphics algorithms and applications, and to provide workstation graphics capability with unprecedented levels of performance and realism. In this section we describe its major architectural features and the rationale under which they were chosen.

2.1 Image Composition Architecture

PixelFlow's most characteristic feature is that it is an image-composition architecture. Image-composition is an object-parallel rendering approach in which the primitives in the scene are distributed over a parallel array of renderers, each of which is responsible for generating a full-screen image of its fraction of the primitives (Figure 2). To compute a frame, each renderer computes a full-screen image of its fraction of the primitives. It then feeds color and visibility information for its pixels into a local compositor (C), which also receives a stream of pixels from the compositors (and renderers) upstream. The compositor selects the visible pixel from its two input ports and forwards it to the compositors downstream. The compositors together form an *Image-Composition Network*, the output of which contains the pixels of the final image. The Image-Composition Network can be built as a pipeline, as shown in Figure 2, or as a binary tree; PixelFlow uses a pipeline, since it is easier to implement and the additional latency is negligible.

The bandwidth in every link of the Image-Composition Network is identical; it is determined by frame rate, screen size, and the amount of data per pixel or sample, and is independent of the number of polygons in the scene. Thus the network (and system) can be extended to incorporate an arbitrary number of renderers

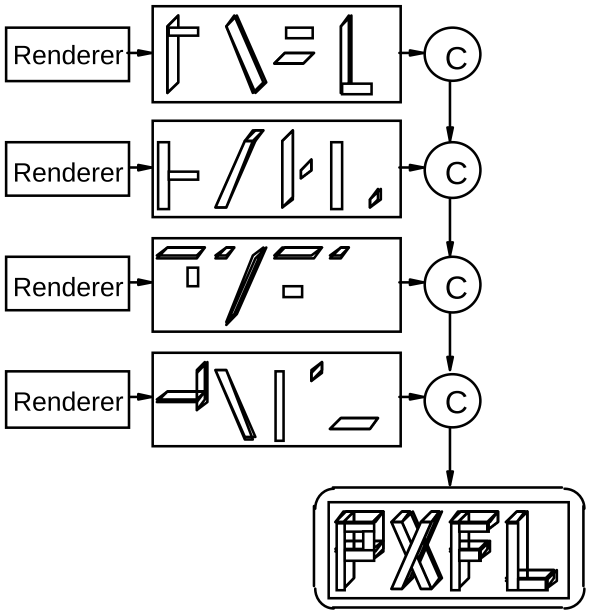


Figure 2: Object-parallel rendering by image composition.

into the system. This gives the architecture its unique and most important property: linear scalability to arbitrarily high levels of performance.

2.2 Supersampling Antialiasing with *Z*-buffer Visibility

PixelFlow performs antialiasing by supersampling. Each renderer computes its full-screen image with 4, 8, or more samples per pixel. Samples are computed in parallel at jittered, subpixel locations. Up to eight samples per pixel are computed simultaneously, each with independent colors (or shading attributes) and *z* values. The compositors perform a simple *z* comparison for each sample, forwarding the appropriate sample downstream. After composition, the samples are blended together to form the final image.

Supersampling was chosen because it is general, the compositor hardware is simple (therefore fast), and the number of samples can be varied to trade speed for image quality. It has two disadvantages: first, the composition network must support the worst-case bandwidth—that of every sample within a pixel hitting a different surface—and this bandwidth is large; second, rendering transparent surfaces requires screen-door or multipass algorithms [MAMM89].

2.3 Logic-Enhanced Memory Rasterizer

The PixelFlow Rasterizer Board uses the logic-enhanced memory approach used in our earlier Pixel-Planes designs [EYLE88, FUCH89], in which a large SIMD array of pixel processors is implemented using custom VLSI chips that integrate simple processing elements with high-speed, low-latency image memory. This approach eliminates the traditional bandwidth bottleneck between rasterizer and image memory, permitting more sophisticated rasterization algorithms and higher polygon rates [POUL92, TORB96]. PixelFlow's enhanced memory chips have byte-wide ALUs and operate at 100 MHz, enabling a 3 million triangle-per-second rasterizer to be built on a single circuit board. Building custom rasterizer chips also allowed a low-cost implementation of the Image-Composition Network (see below).

2.4 Region-Based Rendering

The logic-enhanced memory approach has one big disadvantage: it is not feasible (today) to implement enough image memory on custom chips to provide a full-screen image. This means that a full-screen image must be generated in multiple steps.

PixelFlow renderers operate by sequentially processing small regions of the screen. The region size is determined by the number of samples per pixel and ranges from 32x32 to 64x128 pixels. After each renderer rasterizes a given region, the renderers scan out that region's rasterized pixels over the Image-Composition Network in synchrony with the other renderers. This compositing of regions is the "heartbeat" of the system.

Before rasterization can begin, the GP must sort primitives into bins corresponding to the screen regions. This extra step requires memory on the GP and adds latency. Also, some primitives, particularly those of large screen extent, fall into more than one region, increasing the effective polygon count by a factor equal to the average number of regions per primitive. This number can range from 1.3 to 1.7 for typical datasets [MOLN94]. Region-based rendering algorithms also may suffer from load imbalances when primitives clump into regions; a particular danger is that primitives may clump into different regions on different renderers, potentially starving the compositing network. PixelFlow mitigates these problems by providing buffering in the logic-enhanced memory rasterizer for several regions of pixel data.

2.5 Deferred Shading

PixelFlow uses *deferred shading*, an approach that reduces the calculations required for complex shading models by factoring them out of the rasterization step [DEER88; ELLS91]. PixelFlow rasterizers do not compute pixel colors directly; instead, they compute geometric and intrinsic pixel attributes, such as surface-normal vectors and surface color; these attributes, not pixel colors, are composited. The composited pixels (or samples), containing these shading attributes, are deposited onto designated renderer boards called shaders. The shaders look up texture values for the pixels and compute final pixel color values, based on surface normal, light sources, etc. Shading information is shared among subpixel samples that hit the same surface, up to a maximum of three surfaces per pixel. For ultimate quality rendering, every subpixel sample can be shaded independently. After shading, regions of shaded pixels are forwarded to a frame buffer for display.

The advantage of this approach is that a bounded number of shading calculations are performed per pixel, no matter what the depth complexity of the scene is or how many renderers are in the system. Thus, shading performance is decoupled from rasterization performance: the number of shaders required is determined only by the resolution of the image, the number of surfaces shaded per pixel, the complexity of the shading model, and the frame rate.

PixelFlow's SIMD rasterizer is an ideal processor for deferred shading. Shading calculations can be performed for many pixels simultaneously; if all pixels are shaded with the same algorithm, the SIMD rasterizer achieves near 100% processor utilization. This allows up to 800 billion byte-operations per second of shading performance on a single board. The rasterizer's texture subsystem, using commodity SDRAM texture memory, supports texturing, environment mapping, shadows, and so forth [MOLN95].

Deferred shading requires higher bandwidth in the Image-Composition Network, since pixel shading attributes require more data than the three to four bytes required for RGB color values. Transparent surfaces, including texture-modulated transparency, are handled by sending transparent polygons to shaders and accumulating transparent layers using Mammen's algorithm [MAMM89]. The performance impact is determined by the number of transparent primitives and the number of transparency layers.

2.6 Ultra-Fast Image-Composition Network

To support high-resolution displays, fast frame rates, and deferred shading, the Image-Composition Network must provide enormous bandwidth—tens of Gbytes/second. We accomplish this in a cost-effective way by integrating the compositors onto the logic-enhanced memory chips that implement the SIMD rasterizer array. The network is formed by daisy-chaining connections between the logic-enhanced memories on neighboring boards. Hence the network consists entirely of point-to-point communication between identical custom chips on neighboring boards, so state-of-the-art techniques for high-speed, low-power interconnect can be employed to provide the necessary bandwidth.

The Image-Composition Network on PixelFlow is 256 wires wide and runs at 200 MHz, with data traveling in both directions on the same wire at the same time. The total bandwidth, therefore, is 2 • 200 MHz • 256 wires = 100 Gbit/second. This is sufficient to render 1280x1024-pixel images with sophisticated shading and 4 samples per pixel at greater than 60 frames per second.

2.7 System Configurability

Because renderers, shaders, and frame buffer boards all share the same underlying hardware (with the exception of I/O and video daughter-cards), applications can tune the number of renderers and shaders to achieve optimum speed, based on the number of primitives and the complexity of the shading. Also, a large machine can be partitioned into smaller machines to support multiple users, or as a single, large machine when ultimate performance is desired.

3 HARDWARE COMPONENTS

PixelFlow is a modular graphics system, composed of one or more chassis, each containing up to 9 Flow Units (the maximum configuration is 256 Flow Units or more than 28 chassis!). Figure 3 shows a one-chassis PixelFlow configuration.

The system is built around a horizontal *midplane*. Geometry Processor Boards plug into the underside of the midplane. Rasterizer Boards plug into the top of the midplane. The midplane contains the daisy-chain wiring for the Geometry and Image-Composition Networks, as well as clock and power distribution. Figure 4 shows the components and interconnections of a Flow Unit.

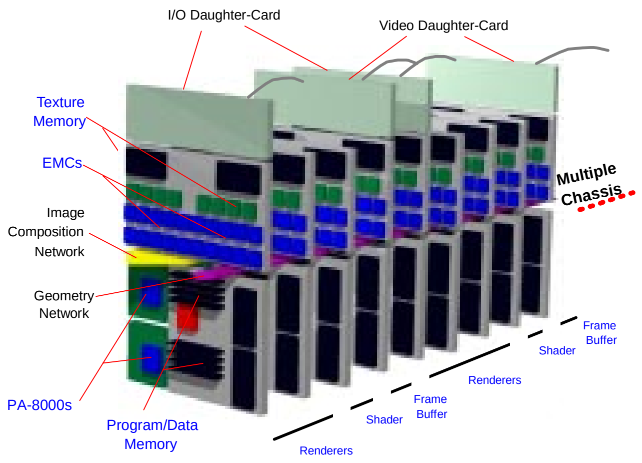


Figure 3: One-chassis PixelFlow system.

3.1 Geometry Processor

The Geometry Processor (GP) is a fast floating-point processor that may be configured with one or two CPUs and up to 256 Mbytes of memory.

CPUs. The CPUs are Hewlett-Packard PA-RISC PA-8000 modules. In a dual processor GP, the two processors are cache coherent. The PA-8000 runs at 180 MHz, issuing a peak of two floating point multiply-accumulates and two integer ops per cycle. The processor modules include large instruction and data caches.

Memory. GP memory consists of 64 to 512 Mbytes of SDRAM memory, serving both as main memory for the GP and as a large FIFO queue for buffering commands for the rasterizer.

RHInO. A custom ASIC, the *RHInO* (Runway Host and I/O) connects the processors with memory, the Geometry Network, and the Rasterizer Board. Its primary function is to service memory requests from the two processors and its various I/O ports. It also contains two DMA engines, one for transmitting rendering commands from SDRAM memory to the Rasterizer Board, and one for sending and receiving data from the Geometry Network.

Geometry Network. The Geometry Network is a high-speed packet-routing network that connects the GPs to each other. This is particularly useful for connecting the host to Flow Units that do include an I/O daughter-card. It is implemented using a bit-slice pair of *Geometry Network Interface* (GeNIe) ASICs; they physically reside on the Rasterizer Board.

The GeNIe provides three ports onto the Geometry Network for each Flow Unit. One port goes to the GP itself (via the RHInO); one port goes to the optional I/O adapter; a third goes to the Inter-TASIC Ring on the Rasterizer Board for loading textures and reading frame-buffer contents. Each port supports I/O traffic of up to 240 Mbytes/second. The Geometry Network supports broadcasts to groups of receivers. The overall Geometry Network bandwidth is 800 Mbytes/sec in each direction. Non-overlapping transfers may occur simultaneously.

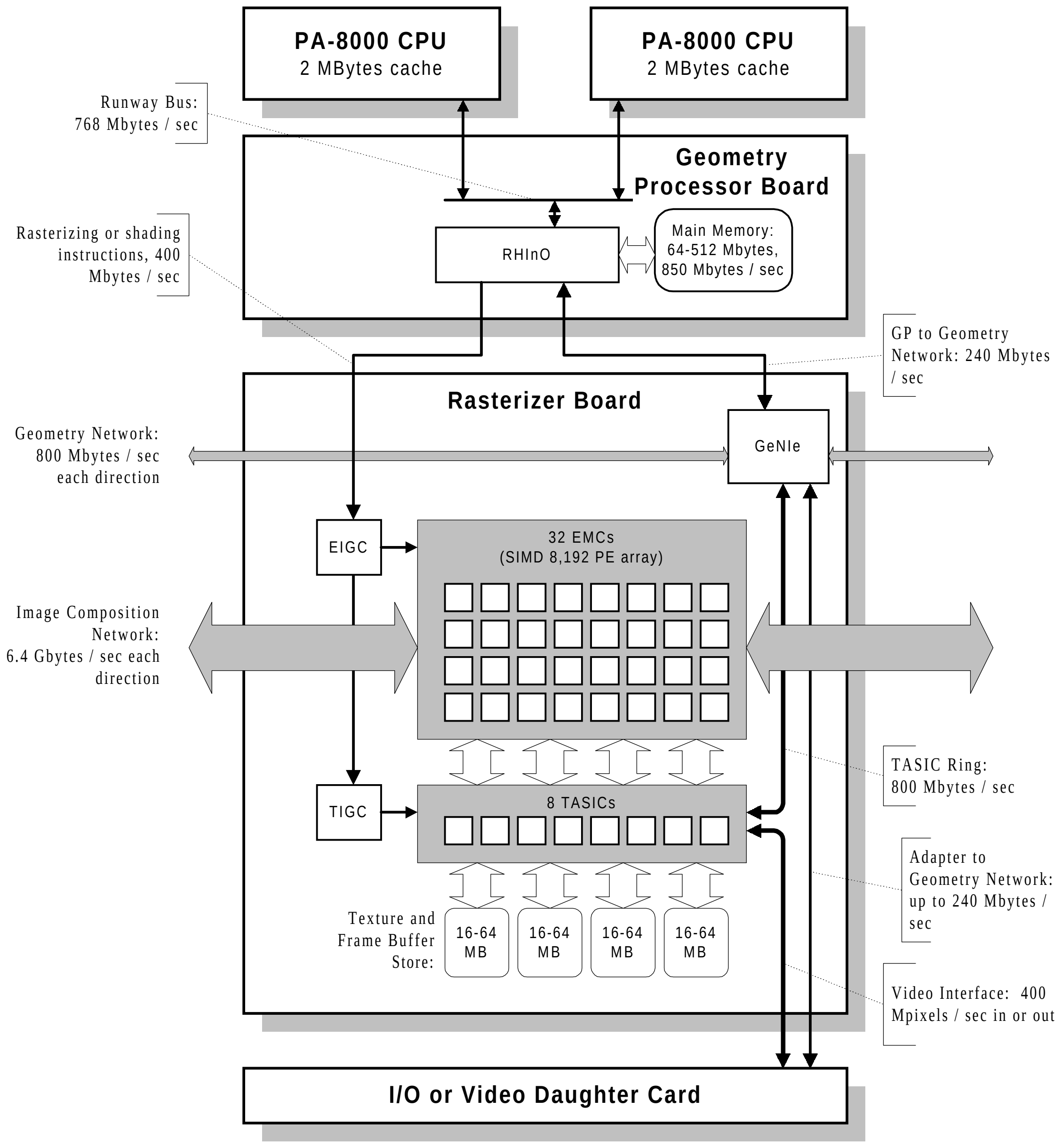

PA-8000 CPU
2 MBytes cache
PA-8000 CPU
2 MBytes cache
Runway Bus: 768 Mbytes / sec
Geometry Processor Board
Rasterizing or shading instructions, 400 Mbytes / sec
RHInO
Main Memory: 64-512 Mbytes, 850 Mbytes / sec
GP to Geometry Network: 240 Mbytes / sec
Rasterizer Board
Geometry Network: 800 Mbytes / sec each direction
GeNIe
EIGC
32 EMCs
(SIMD 8,192 PE array)
Image Composition Network: 6.4 Gbytes / sec each direction
TASIC Ring: 800 Mbytes / sec
TIGC
8 TASICs
Adapter to Geometry Network: up to 240 Mbytes / sec
Texture and Frame Buffer Store:
16-64 MB
16-64 MB
16-64 MB
16-64 MB
Video Interface: 400 Mpixels / sec in or out
I/O or Video Daughter Card

Figure 4: PixelFlow Flow Unit with CPUs and IO/Video adapter.

3.2 SIMD Pixel Processor Array

The heart of the Rasterizer Board is a SIMD array of 8,192 processing elements (PEs). This array is mapped to screen regions of different sizes, depending on the number of samples per pixel, as follows:

Samples per Pixel	Region Size (pixels)
1	128 x 64
4	32 x 64
8	32 x 32

The PE array is divided into four modules, each tightly coupled to a texture/video subsystem.

The SIMD array and texture/video subsystem operate under the control of a pair of Image-Generation Controller chips (IGCs), which perform cycle-by-cycle sequencing of the SIMD array and provide data for the EMCs' linear expression evaluator.

The PE array is implemented on 32 logic-enhanced memory chips (EMCs), each containing 256 PEs. Figure 5 shows a block diagram of an EMC.

Each PE consists of an arithmetic/logical unit (ALU) and 384 bytes of local memory. This includes 256 bytes of main memory, and four 32-byte partitions associated with two I/O ports, the Local Port and the Image Composition port. A *linear expression evaluator* computes values of the bilinear expression *Ax+By+C* in parallel for every PE; the pair *(x,y)* is the address of each PE on a subpixel resolution grid, and *A*, *B*, and *C* are user-specified as part of the SIMD instruction stream. The ALU performs arithmetic

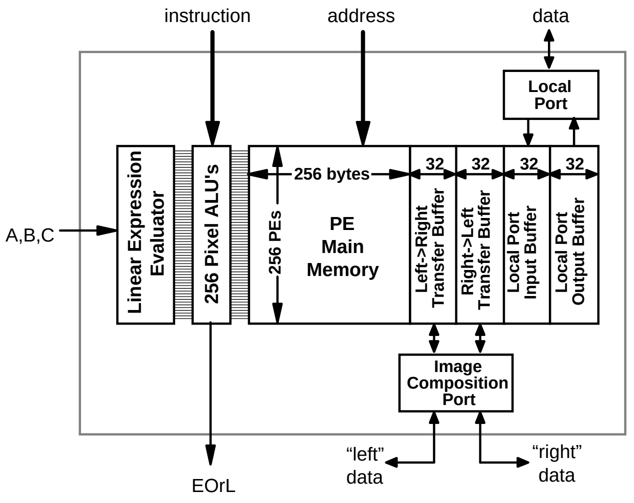


Figure 5: Block diagram of Enhanced Memory Chip.

and logical operations on local memory and on the local value of the bilinear expression.

Figure 6 shows a functional diagram of one PE. The major components are described in the following sections.

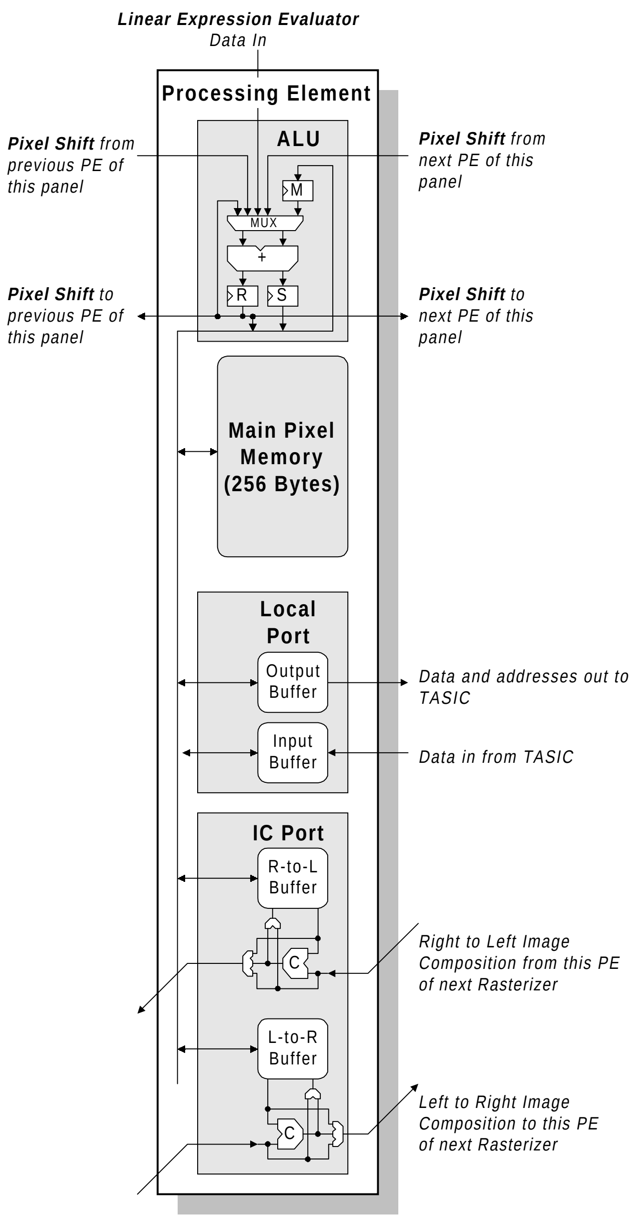
ALU. The ALU implements an 8-bit add and a full range of bitwise logical functions. There are three 8-bit registers: the R, S, and M registers. The R and S registers can be loaded with the core result. The R register can be fed back to the core, and either register can be written to memory. The M register is loaded with a byte read from memory; it also can be loaded with the R or S register value. The R and S registers can be combined into a single 16-bit accumulator, to accelerate multiplies. A carry register is provided for multi-byte computations. Each PE includes an *enable* register. PEs may be *disabled*, by clearing this register, on the basis of computation results; memory writes do not occur at PEs that are disabled.

Linear Expression Evaluator. The linear expression evaluator operates byte-serially to provide each processor with one byte of the bilinear expression on every clock cycle; this can be thought of as an immediate operand. The result for each set of coefficients generally must be preceded by two guard bytes, since A and B are multiplied by 14-bit numbers.

The PEs are assigned *x*,*y* addresses on a subpixel grid with resolution of 1/8th pixel. The PEs are grouped into sets of 1, 4 or 8, each group corresponding to a pixel. The PEs in each group are assigned *x*,*y* subpixel addresses in a 2-pixel-wide box about the pixel center; the pattern of subpixel addresses is the same for each group, and this pattern defines the antialiasing kernel.

Inter-PE Communication. The PEs on each EMC are connected by a shift path that allows each ALU to use the R register of either of its neighbors as an operand. When antialiasing, the 4 or 8 samples for a single pixel are mapped to contiguous PEs; the shift path is used to combine these samples into a single PE, where they can be filtered into an aggregate display value. This is usually done on a shader, after composition.

Local Memory. An 8-bit wide memory data bus connects the M, R, and S registers to the 384 bytes of local memory; a byte of data



Linear Expression Evaluator
Data In
Processing Element
ALU
Pixel Shift from previous PE of this panel
Pixel Shift from next PE of this panel
M
MUX
+
R
S
Pixel Shift to previous PE of this panel
Pixel Shift to next PE of this panel
Main Pixel Memory (256 Bytes)
Local Port
Output Buffer
Input Buffer
Data and addresses out to TASIC
Data in from TASIC
IC Port
R-to-L Buffer
C
Right to Left Image Composition from this PE of next Rasterizer
L-to-R Buffer
C
Left to Right Image Composition to this PE of next Rasterizer

Figure 6: Functional diagram of Processing Element.

may be read from or written to memory on each clock cycle. The 384 bytes of local memory for each PE are arranged as:

- 256 bytes of "main" memory
- 32 bytes of Local Port input buffer
- 32 bytes of Local Port output buffer
- 32 bytes of Image-Composition Network left-to-right transfer buffer
- 32 bytes of Image-Composition Network right-to-left transfer buffer

The four 32-byte partitions of memory are used for I/O operations, using the two communication ports described below. These partitions are part of the same address space as the 256 bytes of main memory, and all 384 bytes can be accessed by the ALU. While communication port operations are in progress, the

ALU cannot access these addresses; this lockout is accomplished using semaphores in the control processors. The ALU may continue to access the main memory and any of the 32-byte buffers not involved in I/O operations; this allows I/O to occur simultaneously with normal pixel computations.

Image-Composition Port. The Image-Composition Port consists of 8 *left* pins and 8 *right* pins per EMC. The *left* pins are connected to the *right* pins of the corresponding EMC on the adjacent board, forming a 256-bit wide daisy-chained point-to-point connection along the midplane. These pins operate at 200 MHz (double the system clock rate), with simultaneous bi-directional data flow (each pin has an input data stream, and a simultaneous output data stream). The Image-Composition Network consists of two pathways superimposed onto this bi-directional interconnect: on the *left-to-right* pathway, each PE synchronously receives pixel data from the board to the left, combines this data with the data in the 32-byte *left-to-right transfer buffer*, and forwards the result to the board to the right; similarly, the *right-to-left* pathway combines data from right to left, using the right-to-left transfer buffer. The two pathways can be formed into a loop on a set of adjacent boards; in this way, large systems can be configured as multiple small systems, each with its own independent Image-Composition Network.

The Image-Composition Network operates on one screen region of pixel data at a time. Its primary function is the real-time compositing operation required to combine the partial images from the multiple renderers. The basic composite operation is a *z*-compare (up to 8 bytes) between the incoming pixel data and the pixel data in the local transfer buffer; the composited pixel (or sample), with the smaller *z* value, is forwarded. More generally, the network is used for rapidly moving pixel data, including writing data back *into* the transfer buffer. For each region transfer, a *compositor mode* is specified for each direction; the forwarded pixel is (1) the composited pixel, (2) the incoming pixel, or (3) the local pixel, and the pixel written back into the transfer buffer is (1) nothing, (2) the incoming pixel, or (3) the composited pixel. Thus, there are 9 modes; the four used in the basic rendering algorithm are shown in Figure 7.

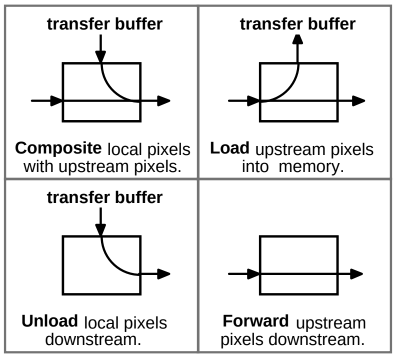

transfer buffer
transfer buffer
Composite local pixels with upstream pixels.
Load upstream pixels into memory.
transfer buffer
Unload local pixels downstream.
Forward upstream pixels downstream.

Figure 7: Compositor modes.

Composite mode is used by renderer boards as regions are composited together to from a final pre-shading image. *Load* mode is used to deposit this composited image into a shader. *Unload* mode is used, to dump final shaded pixels out of the shader (to be received by the frame buffer using *load* mode). *Forward* mode allows data to pass through any boards not participating in a given transfer operation.

Local Port. The Local Port consist of 4 bi-directional pins per EMC. Data in the 32-byte Local Port output buffer is output nibble-serially on these pins. The input data stream from the pins is written into the Local Port input buffer. The Local Port is connected to the texture/video subsystem. Typically, the output buffer is loaded with texture-memory addresses; these are output to the texture/video subsystem, which looks up the texels in texture maps, and returns texel data to the input buffer.

The local input port and local output port operate independently, although they share the same communications substrate. Each port can access all PEs, or a subset of the PEs defined by loading a memory-mapped *mark* register. A *content-dependent decoder* gives the local port access to only the marked PEs. This substantially reduces texture-lookup time when only a subset of pixels in a region needs texturing.

3.3 Texture / Video Subsystem

The texture/video subsystem consists of 8 texture-datapath ASICs (TASICs) and 64 to 256 Mbytes of SDRAM memory. The TASIC chips provide the interface between the Local Ports of the EMCs and texture/image memory; they transfer addresses computed in the PE array to the SDRAMs and transfer texture-lookup data back to the PE array. The SDRAM memory is used as a texture store on shader boards and as a frame store on frame-buffer boards. To provide sufficient bandwidth for Mip-map texture lookups, texture memory is replicated on 4 separate *modules*. Each module consists of 8 EMCs, one copy of the texture memory (16 to 64 Mbytes), and 2 TASIC chips.

Each copy of the texture memory is divided into eight *banks*. The texture memory is designed to simultaneously read eight texels when each of the eight texels comes from a different bank. Prefiltered (Mip-map) texture maps can be interleaved across the banks so that the eight texels required for one pixel are stored in the eight separate banks.

To read from texture memory, the participating PEs each write 8 addresses into their Local Port output buffers. The texture read operation takes this set of eight addresses from each PE in turn, applying the addresses to the eight banks of memory and returning eight 4-byte results to the PE's Local Port input buffer.

The time required for the texture read operation is 0.9 µsec + 0.64 µsec • the number of PEs participating in the worst-case EMC (that is, the EMC with the most PEs marked). For full-screen texture operations, all 32 EMCs will have all 256 PEs marked, so the time is 165 µsec. Pixels are interleaved across the EMCs so that the pixels of a small screen area will be evenly distributed across the EMCs. A 30% speedup is available by replicating textures *within* each module, halving the effective texture store.

Texture memory writes proceed similarly to reads, except that the texture memory addresses can either come from the PEs or can be generated locally on the TASICs.

Generalized table-lookup operations are supported, allowing functions such as bump mapping, environment mapping, and image warping. The shader can be loaded with an image, from which it computes a Mip-map that can then be loaded into texture memory.

Inter-TASIC Ring. For texture reads, each module needs independent access to its local copy of texture data; for texture writes, each module needs write access to all four copies of the texture data. The Inter-TASIC Ring provides each module's PEs with read and write access to the texture memory on all four modules. This enables a 1-to-4 write mode for efficiently writing texture data to all four modules at once.

The Inter-TASIC Ring also connects to the GeNIe chips, allowing the rasterizer to send and receive texel or pixel data over the Geometry Network. This is useful for dynamically loading textures. More details on the texture/video subsystem can be found in [MOLN95].

3.4 Rasterizer Control

The rasterizer is controlled by two Image Generation Controller ASICs (IGCs). The IGCs parse the instruction stream from the Geometry Processor board and issue micro-instructions to the SIMD PE array and texture/video subsystem.

EIGC. One IGC, denoted the *EIGC*, controls the array of PEs on the EMCs. It provides the SIMD micro-instruction to the PEs. This micro-instruction includes control fields for each component of the ALU, the address for local PE memory, and the ABC coefficient set for the linear expression evaluator. The EIGC accepts the coefficients as 32-bit or 64-bit integers, or single- or double-precision floats; it converts these coefficients into byte-serial fixed-point form required by the linear expression evaluator.

The EIGC also controls image composition operations. A *ready/go token chain* attaches to the EIGC on each Rasterizer Board. For each of the two image-composition pathways, the *ready* signal propagates upstream (*e.g.* from right to left, for the left-to-right pathway); once this *ready* signal reaches the first board in the chain, that board initiates the *go* signal, which propagates downstream. As each EIGC in the chain receives the *go* signal, it initiates the transfer operation on the EMCs on that rasterizer.

TIGC. The other IGC, denoted the *TIGC*, provides micro-instructions to the array of TASICs. These instructions control texture memory reads and writes and operations over the Inter-TASIC Ring.

Instruction Stream. The EIGC and TIGC receive a single instruction stream from the GP, via a DMA controller in the RHInO chip. The IGCs independently parse this stream; they contain a set of semaphores which arbitrate between EIGC and TIGC instructions. Each IGC also contains two input FIFOs; another set of semaphores arbitrates between instruction streams in the two FIFOs. This allows the EIGC to have one instruction stream for rasterizing primitives on the PE array and a separate instruction stream for image-composition operations; these two instruction streams execute independently, which is essential for load-balancing.

3.5 Daughter Cards

The basic GP/RB Flow Unit described above can function either as a renderer or as a shader. A Flow Unit can be customized to perform additional system functions by adding one of several types of daughter cards.

Frame Buffer. To use a Flow Unit as a frame buffer, a video daughter card is added. The eight TASICs on a rasterizer board can provide a stream of up to 400 million 32-bit pixels per second from the texture/image memory store. This pixel rate supports very-high resolution displays. Alternatively, the TASIC video port can support up to 8 independent video channels, so multi-channel (stereo, etc.) frame buffers are possible as well.

Frame Grabber. A frame-grabber daughter card allows a stream of video to be acquired by a Rasterizer. It can be stored in video memory, transferred to the PEs for pixel-level processing, transferred to the GP over the Inter-TASIC Ring, or transferred to other boards over the Image-Composition Network or the Geometry Network.

Host Interface. A Flow Unit can be provided with a link to a host computer by adding a n I/O daughter card. Multiple host interfaces are permitted in a system, connecting either to independent hosts, or to a parallel host computer.

3.6 Technology Issues

The PixelFlow system incorporates five custom ASIC designs: IGC, EMC, TASIC, GeNIe, and RHInO. All chips were designed using scaleable design rules, with lambda = 0.35µm, tuned to provide maximum density on Hewlett-Packard's process, a 3-metal n-well CMOS process with 0.6µm channel length. Critical circuit designs, such as the memories and high-speed inter-connect pads, were done using full-custom, mask-level layout; ordinary logic design was done using full-custom and standard-cell approaches. Figure 8 summarizes the five custom chips.

	Transistors	Die Size (mm)	Package
RHInO	1.10 million	15.5 x 14.2	504-pin CPGA
GeNIe	1.36 million	11.3 x 11.3	352-pin BGA
EMC	3.10 million	14.6 x 11.0	208-pin MQUAD
IGC	1.36 million	11.0 x 11.0	352-pin BGA
TASIC	0.63 million	11.0 x 11.0	352-pin BGA

Figure 8: PixelFlow custom chips.

Implementation of these five ASIC designs, and system support for the devices, required us to employ a number of innovative technological solutions. We briefly describe these here.

1-Transistor Dynamic Memory. A major challenge in the EMC design was to build a dense DRAM memory on a high-speed logic IC process. DRAM processes use special features such as trench capacitors to increase memory density and reliability. Don Speck designed a fast, low-power 1-transistor DRAM memory, which can be implemented on a logic process, for Caltech's MOSAIC processor [SPEC91]. We adapted this design for use on the EMC.

In our approach, the 384 bytes of PE memory are divided into 8 bit-slices, and each bit-slice is divided into twelve *chunks*. Each chunk contains 32 storage cells, each consisting of an access transistor and a MOS capacitor. Bit-lines only span a single chunk. A simple interface circuit connects bit-lines within a chunk to a global memory data bus. This allows the bit-lines to run at very high speed, since the bit-lines are loaded with only a small fraction of the storage cells. It also saves power, since only one chunk is active on any given cycle. The design of the data-bus proved critical, however, requiring differential signaling with low voltage-swings to achieve the necessary speed without consuming excessive power.

High-Speed Simultaneous Bi-Directional Signaling. In order to provide the enormous bandwidth required of the Image-Composition Network with reasonable cost in chip pinout, backplane connections, and power, a high-speed/low-power interconnect was needed. The solution took the form of a high-speed, simultaneous bi-directional signaling pad design. These pads transmit data from board to board across the midplane both directions on the same wire at once—and at 200 MHz (double the rasterizer clock speed of 100 MHz). This design was used not only in the Image-Composition Network, but also in the connections from the EMCs to the TASICs, in the Inter-Ring, and in the Geometry Network.

The design of these bi-directional pads was based on the work of Dennison et al. [DENN93], which uses current-mode signaling and on-chip termination. In our approach each pad includes a transmitter consisting of a single-ended current source; each of the two chips, at each end of the signal line, injects a fixed current into the line to send a '1' data bit, or zero current to send a '0' data bit. The voltage across each chip's on-chip termination resistor takes on one of three values, according to whether neither, one, or both of the chips are injecting current. Each chip compares this voltage to a reference voltage level, determined by whether it is sending a '0' or a '1'. The comparator result is the data bit being sent by the chip at the opposite end of the signal line.

Clock Distribution. The simultaneous bi-directional signaling pads only have 5 nsec to transmit signals between boards. This necessitates a low-skew clocking scheme, both within a single board, and between neighboring boards.

Low-skew clocking within a board was accomplished by extending Chi's *salphasic* clocking methodology [CHI94] (used in one dimension on the backplane of Pixel-Planes 5) to two dimensions. The central idea of salphasic clocking is to establish a standing wave across transmission lines (or clock planes) that span the system. Each device in the system needs only to detect the zero crossing of this standing wave, rather than receiving a traveling clock edge, as in conventional clocking. Standing waves have the property that every point along the transmission medium has the same phase, so very low-skew clock distribution is possible—without having to match clock trace lengths and clock buffer delays.

The PixelFlow Rasterizer Board has a differential pair of salphasic clock planes. Each of the custom ASICs has two clock input pins, which receive the differential signal from the clock planes. An on-chip delay-locked loop circuit generates an on-chip clock with edges aligned to the zero-crossings of the differential clock; a single-phase clocking scheme is used inside the custom chips.

A PixelFlow system can include hundreds of Flow Units, presenting a daunting task for backplane clock distribution. Since the Image-Composition Network and Geometry Network connect only neighboring boards, clock skew need be controlled only locally; this leads naturally to our approach, in which each board centers its clock phase between that of its two neighbors [PRAT95]. Each board's clock planes are driven by a voltage-controlled crystal oscillator, which is phase-locked midway between its two neighbors, and with a weak pull towards the center of its operating range. This scheme requires no system-wide clock reference and no active parts on the backplane.

Packaging and Power. PixelFlow's unconventional mid-plane design was chosen because this arrangement allows very short signal paths between chips on one Rasterizer Board and those on the neighboring Rasterizer Board, while simultaneously allowing short signal paths between speed-critical chips on the GP and Rasterizer. Minimizing trace lengths for the high-speed interconnect also required a compact board layout. We used aggressive high-density package designs, such as ball-grid array (BGA) and metal quad flat-pak (MQUAD) packages; parts were mounted on both sides of the circuit board.

Power is distributed throughout the system mid-plane at 48 volts DC. "Point-of-load" DC-to-DC converters on each board convert this to the levels needed on the board. This power distribution scheme ensures that all the custom ASICs receive supply voltages within 100 mV of their nominal 3.3V level.

4 SOFTWARE COMPONENTS

The application programmer's interface to PixelFlow is an OpenGL [OPEN92] library with extensions to support the architecture of the machine and an enhanced lighting and shading capability.

As a research project, we've also developed a shading language, PfMan, modeled on the RenderMan shading language [HANR90], for coding the user-programmable portions of the pipeline (shading, lighting and even new primitives). In this section, we describe the programmer's view of PixelFlow as well as some of the internal software.

4.1 PixelFlow OpenGL

Three major motivations and constraints drove the design of our OpenGL implementation. First was the desire at UNC to demonstrate programmability in the hardware pipeline [LAST95]. Although OpenGL provides a fairly rich model for shading and lighting, that model is fixed. We needed to add library calls to interface to the new programmable shading, lighting, and primitives. Second, the region-based rendering and deferred shading characteristics of the PixelFlow architecture constrained us to a more frame oriented design than most OpenGL implementations. Finally, we wished to provide support for the future research use of a parallel host machine with multiple processors feeding the graphics pipeline simultaneously.

Support for programmability includes library routines for loading user-coded shading and lighting functions. As described in the next section, a programmer can write high-level language procedures for shading and lighting, compile them using the PfMan compiler, and store the result on disk. He can then, through OpenGL extensions, instruct the system to load and link the code on the shading node (at the moment we're statically linking the code, but expect to dynamically load by the end of the year). We provide other OpenGL extensions to set the values of user-defined shading parameters that can change on a primitive by primitive basis. Examples include not only standard parameters, such as surface normal and texture coordinates, but also arbitrary others, such as noise frequency, etc.

The constraints of region-based rendering and deferred shading mandated rendering based on frames (we use the term "frame" in a fairly general sense). Before primitives are transformed on renderers, all global parameters (such as viewing transformations and the state necessary for shading) must be specified. We delimit the end of this setup stage with the call *glStartGeometry()*. Since, in PixelFlow, all of the primitives must be sorted based on their screen-space location before rasterization, we also need a way for the user to inform the system that no more primitives are on the way. We use the function *glEndFrame()* as a delimiter. The OpenGL Architectural Review Board may set a standard to support tiled architectures, therefore the names of the delimiting functions may change (currently proposed names are *glBeginScene* and *glEndScene*).

PixelFlow OpenGL is implemented on the client/server model, as are most OpenGL systems. Small subroutines on the host workstation package tokens representing OpenGL calls and send them, via the Geometry Network, to the appropriate PixelFlow nodes. Some commands, such as those specifying the frustum and projection, are broadcast to all renderers, while others, such as primitives bracketed by *glBegin()* and *glEnd()* calls are distributed round-robin to single renderers. Shader-specific commands, such as those that load a new shading function, are only sent to shaders.

In order to enable full performance rendering, regardless of the workstation to PixelFlow bandwidth, display lists are stored on the GPs. The two GPs on each renderer function as independent OpenGL servers for most of the frame, but the two streams of rasterization commands are merged as they are sent to the IGCs.

4.3 Rendering Control

Rendering control coordinates and synchronizes the actions of the multiple Flow Units. Initially, rendering control designates each Flow Unit as either a renderer, shader, or frame buffer.

- On renderers, rendering control transforms and clips primitives and then generates *bins* of IGC instructions corresponding to each screen region. The instructions for primitives whose bounding box crosses a region boundary are copied to each affected bin. This process is repeated for every primitive assigned to that renderer.
- On shaders, rendering control generates IGC instructions to texture, light, and shade regions of composited pixels.
- On frame buffers, rendering control generates instructions to store pixel values in texture/image memory and to scan pixel values out via the video daughter-card. Frame buffers may also perform post-processing on image data.

Once instructions are generated, *rendering recipes* coordinate and synchronize the actions of each Flow Unit. They control when each Flow Unit perform such tasks as shading, antialiasing, and blending of samples. They also orchestrate transfers over the Image-Composition Network. Since all Flow Units must act together to set the compositors properly and to send or receive the correct amount of data, these transfers form the “heartbeat” of the system.

4.3 Programmability

Currently, in our experimental shading language, we can write PfMan procedures to program three stages of the pipeline: shading, lighting and the rendering of primitives. We expect eventually to experiment with user programmability at the atmospheric stage (to allow user-specified fog, for example) and at the frame buffer (for image operations, such as warping).

User programmable surface shaders and lights are similar to those supported by the RenderMan shading language. We’ve added support for fixed-point arithmetic to increase performance and have worked to optimize the execution by eliminating as much redundant computation as we can. A major performance improvement comes from running all of the lighting code at once, for all of the pixels, regardless of the surface shader at a particular pixel. This technique makes the best use of the SIMD nature of the PE array.

A feature that is rarely supported, even in software renderers, is the ability for the user to add geometric primitives to the system. Using PfMan, a user can add primitives of two different types: those that call rendering code for other primitives, and those that directly issue instructions to the IGC. The former type of primitive is useful when the user wants to package standard primitives in an economical way, for example height fields. Primitives that directly render can add basic functionality, such as true spheres [FUCH85].

4.4 Multipass algorithms

Some shading effects, such as shadows and reflections, may need multiple passes to complete. We treat these as multiple frames and will provide host libraries to insulate the programmer from the complexity. A typical case is that of a shadow-casting light. We must first render a shadow (depth) map from the viewpoint of the light [WILL78]. This is the first frame of the process and is particularly simple since we only need the *z* values at each pixel of the shadow map. Using the Image Composition Network, we load the depth map into the texture memory of all of the shaders and execute the second frame, which is the rendering of the actual scene.

5 PERFORMANCE

5.1 Performance Model

System performance is determined by: (1) the time required by the renderers to transform and rasterize the polygons in the screen regions, (2) the time required by the shaders to shade and texture the screen regions, and (3) the time required by the Image-Composition Network to composite the screen regions. System performance may be limited by any of these effects, depending upon various factors described below.

- **Rasterization.** The time required to transform and rasterize each screen region is determined by the number of polygons in the region and the complexity of the rendering algorithm. Rasterizers are capable of processing 2.8 million Gouraud-shaded triangles per second or 1.4 million textured, Phong-shaded triangles per second. By adding additional renderers, the number of primitives that must be transformed and rasterized by each renderer is reduced; this gives the architecture its appealing characteristic of linear scalability.
- **Shading.** The time required to shade each region is determined by the complexity of the shading and texturing algorithm. A given number of shaders imposes a maximum frame rate. Higher frame rates can be achieved by increasing the number of shaders, thereby reducing the number of regions that must be processed by each shader. Shading time generally is limited by the time required to perform texture

lookups. Each shader performs 50 million full (8-texel) Mip-Map texture lookups per second.

- **Compositing.** The time required to composite each region is determined solely by the number of bytes of pixel attributes to be composited. For a given screen size, number of antialiasing samples, and number of bytes of pixel attributes, image-composition bandwidth places a hard limit on the frame rate.[1] The network requires 1.28 * *N* µsec to composite a screen region (*N* is the number of bytes of pixel data per PE).

The interaction of these factors is shown in Figure 9.

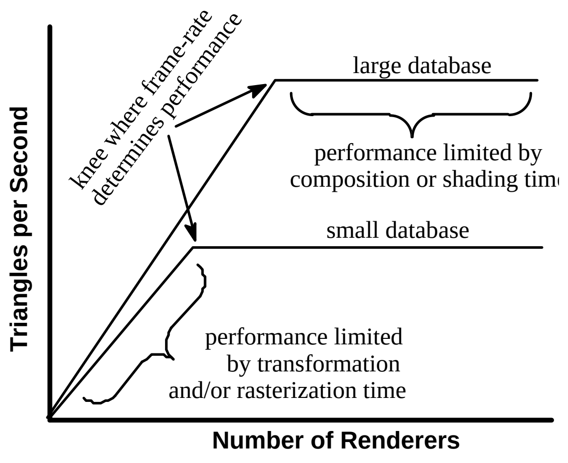


Figure 9: Theoretical performance curves.

This figure shows that, for a given size data base and rendering algorithm, system performance scales linearly with the number of renderers until a knee is reached, where compositing and/or shading time dominates rasterization. For more complex data bases, this knee is reached at a higher number of renderers. Below the knee, adding renderers increases polygon rate; this is the property of linear scalability characteristic of image composition architectures. Above the knee, frame rate is limited by shader and/or Image-Composition Network performance.

Load imbalances. When performance is determined by composition and shading time, performance is quite predictable. However, when performance is determined by rasterization time, two effects can reduce system performance. First, primitives may fall into more than one screen region, so the effective number of primitives is increased (by approximately 30% to 70% for typical scenes [MOLN94]). Second, primitives may not be evenly distributed across the regions, so some regions can be rasterized in very little time, while other regions require significantly more time to rasterize than is taken to composite and shade. This effect is particularly egregious when the screen distribution of primitives varies from renderer to renderer, since renderers will then take turns being the bottleneck.

[1] This limit can be circumvented by configuring the machine as a number of smaller machines, each responsible for a subset of the regions of the screen. However, each smaller machine must transform the entire database or use a hierarchical culling scheme.

We mitigate these load-imbalance effects by providing several regions of buffering between rasterization and compositing. Simulations on test scenes indicate that this is adequate if the primitives in the scenes are distributed round-robin across the renderers.

5.2 Current Status

At the time this of this writing, small systems containing three to five Flow Units are running in our laboratory. All hardware has been demonstrated to run at full design speed.

The major components of system software are running and we have generated a number of test images. The software is still being optimized, so we do not yet have empirical data for GP or overall performance. Preliminary performance data should be available when this paper is presented.

6 SUMMARY

PixelFlow is one of the first graphics architectures to use real-time image composition with multi-primitive renderers. Its combination of multi-million-triangle-per-second renderers, deferred shaders, and high-performance compositing network give it linearly scaleable performance up to 100 million polygons per second or more.

Virtually all of the components of PixelFlow are programmable: its Geometry Processors are conventional microprocessors; its Rasterizers are programmable SIMD computing surfaces; its Image-Composition Network is a general datapath for moving and compositing pixel data. In addition to standard rendering algorithms, such as Gouraud- and Phong-shading of polygonal primitives, PixelFlow can render primitives such as spheres, quadrics, and volume data with high-quality shading methods, such as local light sources, procedural and image-based texturing, and shadow and environment mapping. Figure 10 shows a number of sample images rendered on PixelFlow.

A PixelFlow system can be configured in a variety of ways. Hosted by a single workstation, it can render retained-mode datasets. Coupled to a parallel supercomputer, it can serve as a visualization subsystem for immediate-mode rendering. Using PixelFlow silicon, a 2-3 million-triangle-per-second rasterizer can be built on a single board.

ACKNOWLEDGMENTS

The PixelFlow team, both at The University of North Carolina and at Hewlett-Packard, includes many individuals not mentioned in the author list, who have contributed significant ideas and much hard work to this project. At UNC these include faculty/staff members: Vern Chi, Henry Fuchs, Kurtis Keller, and Steve Tell; and research assistants: Dan Aliaga, Jon Cohen, David Ellsworth, Arthur Gregory, Lawrence Kesteloot, Jonathan Leech, Jonathan McAllister, Peter McMurry, Carl Mueller, Bryon Nordquist, Marc Olano, Krish Ponamgi, Voicu Popescu, Greg Pruett, Jiang Qian, Jason Smith, Yulan Wang, Greg Welch, and Rob Wheeler. At HP they include: Greg Allen, Peter Arnold, John Austin, Randy Bailey, Tony Bennett, Larry Harrold, Fred Heaton, Justin Heinecke, Rich Holloway, Paul Keller, Mike Keshk, Paul Layne, Mike Myers, Chris Practico, Brad Ritter, Durward Rogers, Doug Schiff, Mel Snyder, Herman Towles, and

Jim Weaver. Others include: Phil Atkin, Charles Grimsdale, Henry Rich, Ray McConnell, and Paul Winser.

This research is supported in part by the Defense Advanced Research Projects Agency, DARPA ISTO Order No. 7510, and the National Science Foundation, Grant No. MIP-9000894.

REFERENCES

CHI94 Chi, V.L., "Salphasic Distribution of Clock Signals for Synchronous Systems," *IEEE Transactions on Computers*, Vol. 43, No. 5, May 1995, pp597-602.

DEER88 Deering, M., S. Winner, B. Schediwy, C. Duffy, and N. Hunt, "The Triangle Processor and Normal Vector Shader: A VLSI System for High Performance Graphics," *SIGGRAPH '88*, Vol. 22, No. 4, pp. 21–30.

DENN93 Dennison, L., W. Lee, and W. Dally, "High-Performance Bi-Directional Signaling in VLSI Systems," *Proc. Of 1993 Symposium on Research on Integrated Systems*, MIT Press, pp. 300-319.

ELLS91 Ellsworth, D.E. "Parallel Architectures and Algorithms for Real-Time Synthesis of High-Quality Images Using Deferred Shading," *Workshop on Algorithms and Parallel VLSI Architectures*, Pont-à-Mousson, France, June 12, 1990.

E&S96 Evans & Sutherland Computer Corporation, "Freedom Series™ Whitepaper," http:/www.es.com/ Products/Workstation/WhitePaper.html.

EYLE88 Eyles, J., J. Austin, H. Fuchs, T. Greer, and J. Poulton, "Pixel-Planes 4: A Summary," *Adv. in Computer Graphics Hardware II* (1987 Eurographics Workshop on Graphics Hardware), Eurographics Seminars, 1988, pp. 183-208.

FUCH85 Fuchs, H., J. Goldfeather, J. Hultquist, S. Spach, J. Austin, F. Brooks, J. Eyles, and J. Poulton, "Fast Spheres, Shadows, Textures, Transparencies, and Image Enhancements in Pixel-planes," *SIGGRAPH '85*, Vol. 19, No. 3, pp. 111-120.

FUCH89 Fuchs, H., J. Poulton, J. Eyles, T. Greer, J. Goldfeather, D. Ellsworth, S. Molnar, G. Turk, B. Tebbs, and L. Israel, "Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor-Enhanced Memories," *SIGGRAPH '89*, Vol. 23, No. 3, pp. 79–88.

HANR90 Hanrahan, P., J. Lawson, "A Language for Shading and Lighting Calculations," *Siggraph '90*, Vol. 24, No. 4, pp. 289-298.

LAST95 Lastra, A., S. Molnar, M. Olano, and Y. Wang, "Real-Time Programmable Shading", *Proceedings 1995 Symposium on Interactive 3D Graphics*, (Monterey, CA), April 1995, pp. 59-66.

MAMM89 Mammen, A., "Transparency and Antialiasing Algorithms Implemented with the Virtual Pixel Maps Technique," *IEEE CG&A*, Vol. 9, No. 4, July 1989, pp. 43-55.

MOLN92 Molnar, S., J. Eyles, and J. Poulton, "PixelFlow: High-Speed Rendering Using Image Composition," *SIGGRAPH 92*, Vol. 26, No. 2, pp. 231-240.

MOLN94 Molnar, S., M. Cox, D. Ellsworth, and H. Fuchs, "A Sorting Classification of Parallel Rendering," *IEEE CG&A*, Vol. 14, No. 4, July 1994, pp. 23-32.

MOLN95 Molnar, S., "The PixelFlow Texture and Image Subsystem," *Proc. Of the 10th Eurographics Workshop on Graphics Hardware*, Maastricht, The Netherlands, Aug. 28-29, 1995., pp. 3-13.

OPEN92 OpenGL Architectural Review Board, "OpenGL Reference Manual", Addison Wesley, 1992.

POUL92 Poulton, J., J. Eyles, and S. Molnar, "Breaking the Frame-Buffer Bottlenack: The Case for Logic-Enhanced Memories," *IEEE CG&A*, Vol. 12. No. 6, Sept., 1992, pp. 65-74.

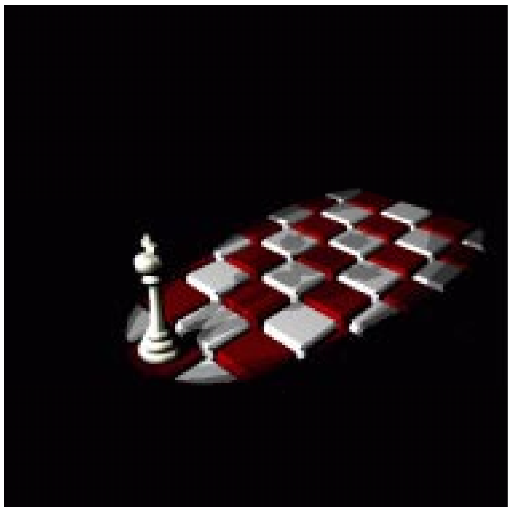
PRAT95 Pratt, G. and J. Nguyen, "Distributed Synchronous Clocking," *Proc. Advanced Research in VLSI*, 27-29 Mar 1995, Chapel Hill, NC, IEEE Comp. Society Press, pp. 316-330.

SGI97 Silicon Graphics Computer Systems, "Onyx2: Graphics Performance," http://www.sgi.com/ Products/hardware/graphics/technology/ graphics.html.

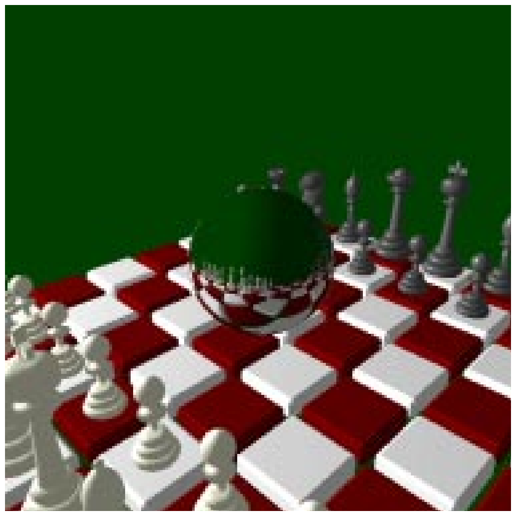
SPEC91 Speck, D., "The Mosaic Fast 512K Scalable CMOS DRAM," *Proceedings of the 1991 University of California at Santa Cruz Conference on Advanced Research in VLSI*, 1991, pp. 229–244.

TORB96 Torborg, J. and J. Kajiya, "Talisman: Commodity Realtime 3D Graphics for the PC," *SIGGRAPH '96*, Vol. 30, No. 3, pp. 353-364.

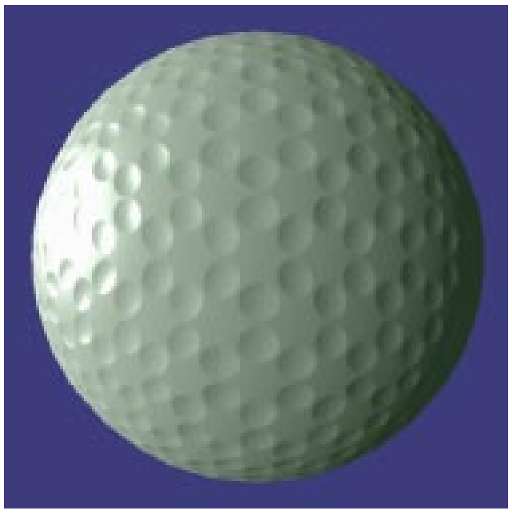
WILL78 Williams, L., " Casting Curved Shadows on Curved Surfaces," *SIGGRAPH '78*, pp. 270-274.



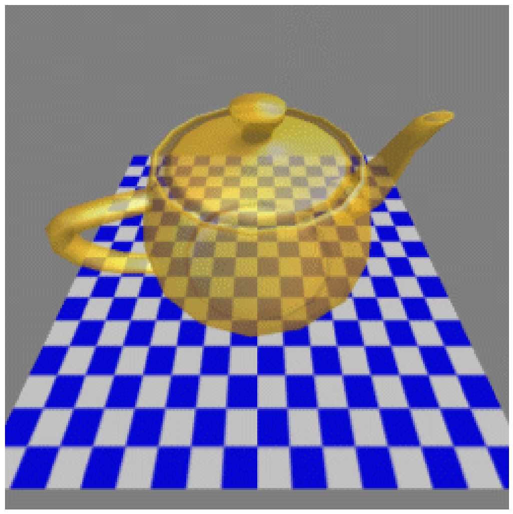
a) Depth-map shadows with three lightsources.



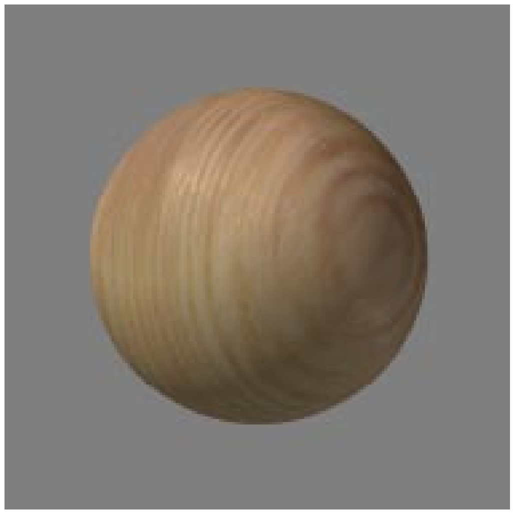
b) Reflection-mapped sphere over chess board.



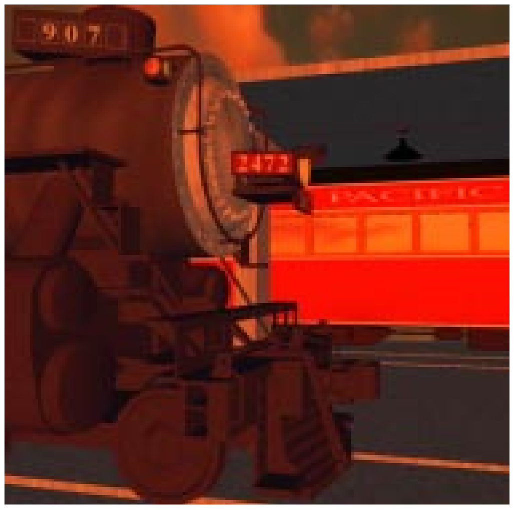
c) Bump-mapped sphere with Phong shading.



d) Scene with texturing and transparent layers.



e) Solid-textured sphere.



f) “Evening Station” (model by John Fujii).

Eyles et. al., Figure 10:   Sample images rendered on PixelFlow.