The PixelFlow Texture and Image Subsystem

Steven Molnar

Department of Computer Science
University of North Carolina
Chapel Hill, NC 27599-3175

Abstract

Texturing and imaging have become essential tasks for high-speed, high-quality rendering systems. They make possible effects such as photo-textures, environment maps, decals, modulated transparency, shadows, environment maps, and bump maps, to name just a few.

These operations all require high-speed access to a large "image" memory closely connected to the rasterizer hardware. The design of such memory systems is challenging because there are many competing constraints: memory bandwidth, memory size, flexibility, and, of course, cost.

PixelFlow is an experimental hardware architecture designed to support new levels of geometric complexity and to incorporate realistic rendering effects such as programmable shading. This required an extremely flexible and high-performance texture/image subsystem. This paper describes the PixelFlow texture/image subsystem, the design decisions behind it and its advantages and limitations. Future directions are also described.

1 INTRODUCTION

The PixelFlow architecture was conceived with two principal aims in mind: 1) to increase the complexity of scenes that can be rendered in real time (*i.e.* numbers of primitives per second), and 2) to increase rendering quality through programmable shading, antialiasing, and a rich set of texture/image operations.

PixelFlow meets the first goal with a scalable rendering architecture based on image-composition [MOLN92]. PixelFlow consists of a set of independent rendering nodes, each of which renders a subset of the primitives to be displayed. The partial images from each rendering node are then merged together over a high-speed compositing network to produce a final image containing all of the primitives. The basic rendering speed of a PixelFlow node is approximately 0.5 million triangles/second with 8-sample antialiasing. An *N*-node system is approximately *N* times as fast.

To increase image quality, PixelFlow supports programmable shading, as advocated by [WHIT82, COOK87, HANR90] and popularized by the RenderMan shading language [UPST90]. Complex interactions of light with surfaces are supported by allowing users to write arbitrary programs to compute the color at each surface. These *shaders* can take into account effects such as anisotropy, textures, local and global lighting, procedural noise functions, perturbed surface-normal vectors, shadows, etc.

This paper describes the the global shared memory system that supports texture and imaging operations and makes many of these effects possible. It is hoped that the ideas, experience (and mistakes) in developing this texture subsystem will aid the designers of future graphics systems targeted for high-quality rendering.

This paper is organized as follows: Section 2 gives an overview of the PixelFlow hardware. Section 3 describes the design considerations that led to the texture subsystem design. Sections 4 and 5 describe the texture subsystem itself. Sections 6 and 7 give a discussion and conclusion.

2 PIXELFLOW OVERVIEW

To make real-time shading possible, PixelFlow uses a technique called *deferred shading* [DEER88; TEBB92]. The system boards are divided into three classes, *renderers* and *shaders*, and *frame buffers*, as shown in Figure 1. Renderers transform and rasterize their portions of the display dataset and produce partially rendered pixels in a form suitable for compositing. These pixels contain depth values, surface-normal vectors, texture coordinates, and other parameters needed for shading (these are sometimes called *appearance parameters*).

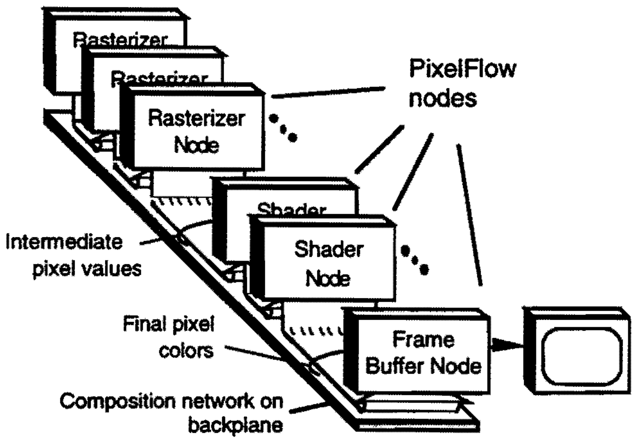

Figure 1. Overview of PixelFlow system.

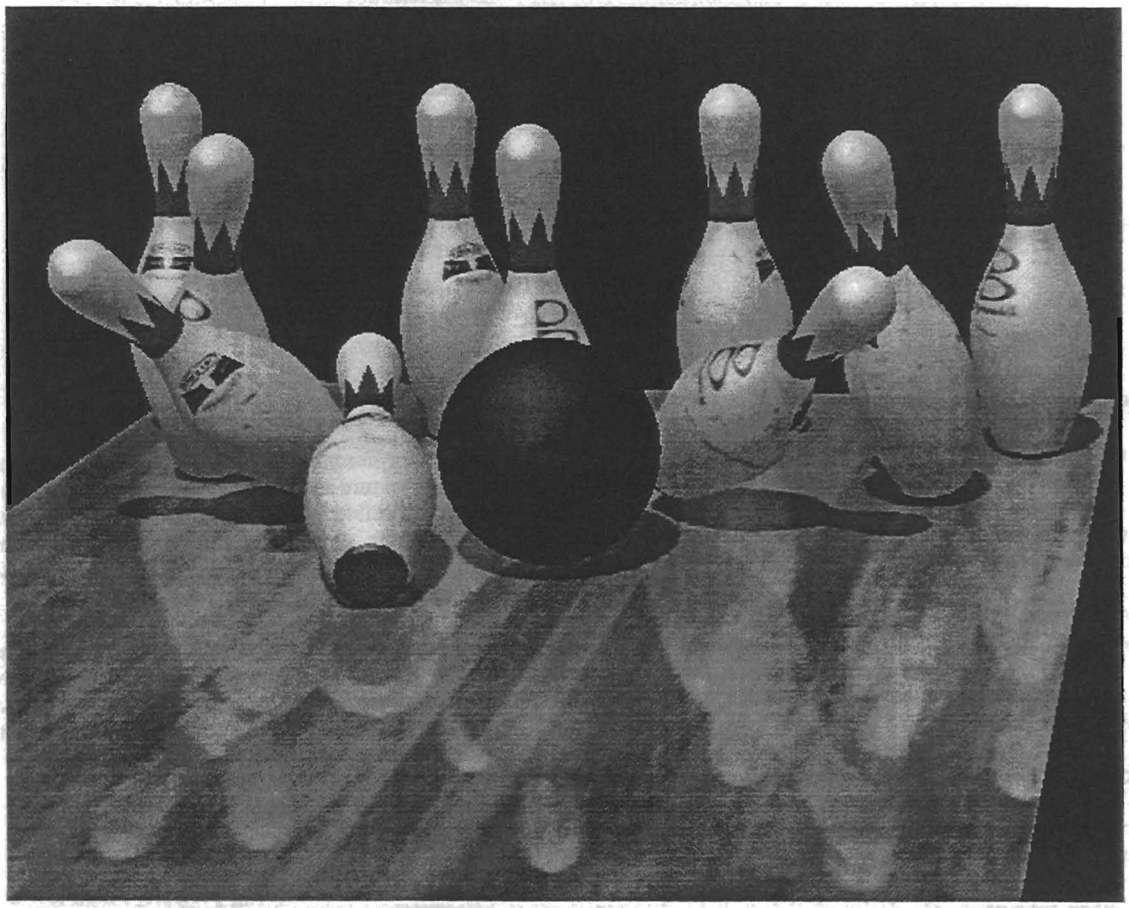

Figure 2: Scene from bowling video rendered on PixelFlow simulator. Texture/shading effects include: shadow-mapped lighting (ball, alley, pins), mip-map textures (wood texture, decals and scuff-marks on pins), bump mapping (dents on pins), reflection mapping (alley), and Phong shading. Simulated performance is 30 msec on a 3-rasterizer/12-shader PixelFlow system.

These appearance parameters are fed into the image composition network. Corresponding pixels from each of the renderers are composited together, and the resulting visible pixels are loaded onto a designated shader board. The shaders then evaluate a shading model for the pixels, including texturing, lighting, Phong or any other type of shading. The shading model may be different for each pixel and depends on the property of the visible surface. Shaded pixels are simply RGB values, which then are sent to the frame buffer for display. Any number of boards can be assigned to rasterization or shading. PixelFlow will be able to generate images such as that shown in Figure 2 at rates of 30 Hz or more [LAST95].

By organizing the system in this way, shading is deferred until after visible surfaces have been determined. This means that only visible pixels are shaded, not pixels that later are obscured by other primitives. It also concentrates the shading operations onto particular system boards. In the remainder of this paper, we will focus on the texturing operations that are performed on shader boards (though texturing sometimes must be performed on renderer boards , as in texture-modulated transparency).

Each type of PixelFlow board has the same basic hardware. Figure 3 shows a high-level view of the components on a single PixelFlow circuit board. There are two main parts: 1) a *geometry processor* (*GP*), a general-purpose floating-point processor, and 2) a SIMD *rasterizer*, comprising 32 enhanced-memory chips (EMCs) and associated texture-lookup hardware.

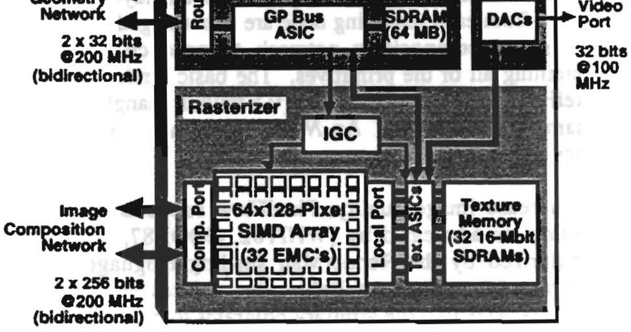
GP
RISC
RISC
Video Card (Optional)
Geometry Network
Router
GP Bus ASIC
SDRAM (64 MB)
DACs
Video Port
2 x 32 bits @200 MHz (bidirectional)
32 bits @100 MHz
Rasterizer
IGC
Image Composition Network
Comp. Port
64x128-Pixel SIMD Array (32 EMC's)
Local Port
Tex. ASICs
Texture Memory (32 16-Mbit SDRAMs)
2 x 256 bits @200 MHz (bidirectional)

Figure 3: Components on a PixelFlow board.

The geometry processor is a high-performance RISC processor comprising two Hewlett-Packard PA-7200 CPUs and 64 to 128 MBytes of SDRAM memory. On renderer boards, the geometry processor transforms primitives into screen coordinates and feeds them to the rasterizer. On all nodes, it controls the rasterizer.

The rasterizer performs all pixel operations. Each of the 32 EMCs contains 256 8-bit processing elements (PEs) and 384 bytes of memory per PE. Together the PEs on all 32 EMCs form a SIMD processor that can operate on arrays of 128x64 pixels in parallel. The rasterizer is used to scan-convert and shade pixels. It also contains an 8-bit linear expression evaluator, which can evaluate expressions of the form $F(x, y) = Ax + By + C$ (where x and y are the pixel address) for each of the 128x64 pixels in parallel.

For texturing operations, the 32 EMCs are divided into four groups of eight, called *modules*. Each module is provided with its own (nearly) independent texture subsystem. The texture subsystem consists of a *Texture ASIC* (*TASIC*) and eight 16 Mbit 100-MHz *synchronous DRAM* (*SDRAM*) memories. The texture subsystem receives addresses from the EMCs and looks up the corresponding texture values while the SIMD array performs texture setup and shading operations.

On video boards (frame-buffer or frame-grabber boards), SDRAM memory is used to store pixels, making an external frame buffer unnecessary. Access to SDRAM memory is time-shared on video boards between video reads/writes and normal texture accesses.

3 DESIGN CONSIDERATIONS

We now review the major design considerations that led to the PixelFlow texture subsystem design. The overriding consideration was memory bandwidth, the rate at which data may be retrieved from memory. Other secondary, but important, considerations were the organization of the memory itself, and the coupling between the memory system and the rasterizer.

3.1 Memory Bandwidth

The major design consideration for a texturing system is the texture memory bandwidth—the rate at which data must be retrieved from memory. The target for PixelFlow was to be able to mip-map texture and shade high-resolution (1280x1024), 5-sample antialiased images at 30 Hz on four or fewer shader boards.

Mip-map textures of 2D images [WILL83] were chosen as the canonical texture operation because they are the most bandwidth-intensive, common texture operation. A single mip-map lookup requires eight accesses to texture memory, one for each corner of the two active resolution levels. If texels are 4 bytes, this means that the shaders together must be able look up $1280 \cdot 1024 \cdot 5 \cdot 30 \cdot 8 \cdot 4 = 6.3$ GBytes/second. A single shader, therefore, must look up 1.6 GBytes/second—a formidable number!

To support this amount of bandwidth requires a parallel memory system. To minimize cost, we wanted to use memories whose I/O interface was as fast as possible. A wide variety of DRAM parts are now available with substantially different interfaces. We chose synchronous DRAMs (SDRAMs) because of their high speed, high density, relatively low cost, and presumed availability. A single, byte-wide 100 MHz 16 Mbit SDRAM (the fastest speed currently available) has a peak read/write bandwidth of 100 MBytes/second. With a burst size of 4 bytes (for 4-byte texels), the maximum attainable bandwidth is 8 bytes every 11 clock cycles, or 72.7 MBytes/second.

To meet our performance target requires 1.6 GBytes/sec / 72.7 MBytes/sec/SDRAM = 22 SDRAM chips. Rounding up to the nearest power of 2 gives a target of 32 SDRAMs per board.

3.2 Interleaved Memory Organization

A second important consideration was maximizing the amount of useful storage in the memory chips. Given that any pixel can look up any texel, a naive organization would require that every SDRAM store identical copies of the same texture data. This would reduce the $32 \cdot 2 = 64$ MBytes of raw memory to 2 MBytes of usable storage—a gross waste of memory resources. We wanted lookups to occur at the maximum rate and to minimize the amount of redundancy.

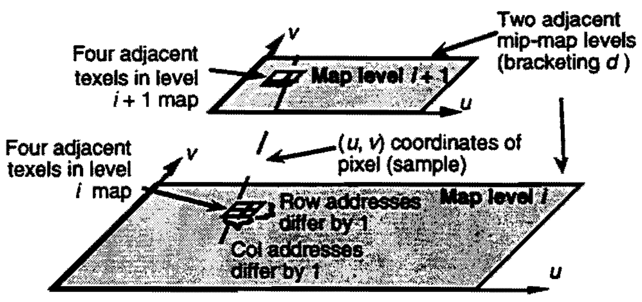
Four adjacent texels in level i+1 map
v
Map level i+1
u
Two adjacent mip-map levels (bracketing d)
(u, v) coordinates of pixel (sample)
Four adjacent texels in level i map
v
Row addresses differ by 1
Map level i
Col addresses differ by 1
u

Figure 4: Memory accesses to mip-map one pixel.

We can reduce the redundancy by a factor of eight using memory *interleaving*. Consider the memory accesses required to mip-map a pixel, shown pictorially in Figure 4.

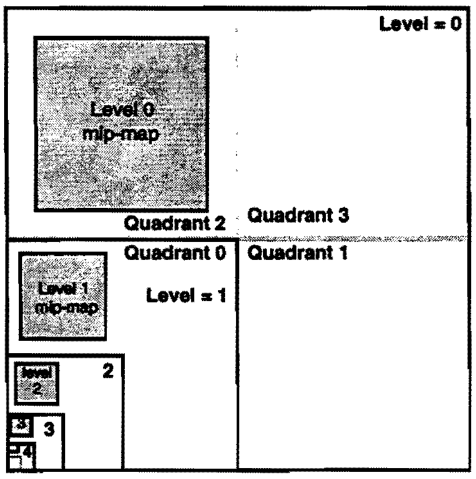

Figure 5: Conventional layout of mip-map in memory. Level 0 occupies quadrants 1–3; higher levels occupy quadrant 0, which is recursively subdivided for succeeding levels.

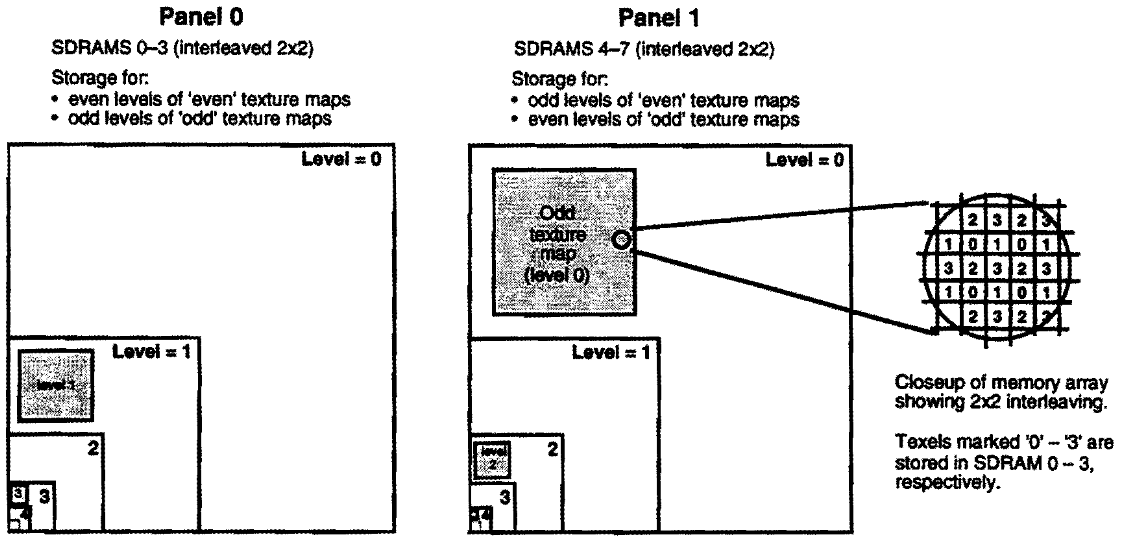
Panel 0
SDRAMS 0–3 (interleaved 2x2)
Storage for:
• even levels of 'even' texture maps
• odd levels of 'odd' texture maps
Level = 0
Level = 1
level 1
2
3
Panel 1
SDRAMS 4–7 (interleaved 2x2)
Storage for:
• odd levels of 'even' texture maps
• even levels of 'odd' texture maps
Level = 0
Odd texture map (level 0)
Level = 1
level 2
2
3
Closeup of memory array showing 2x2 interleaving.
Texels marked '0' – '3' are stored in SDRAM 0 – 3, respectively.

Figure 6: PixelFlow memory organization: 2 panels of 2x2-interleaved memories.

Four values are retrieved from each of the two adjacent mip-map levels bracketing the desired resolution level *d*. Within each resolution level, the texels retrieved are neighbors in *u* and *v*, so one will have an even row address and an even column address, one will have an even row address and an an odd column address, and so forth.

If we divide texture memory into eight partitions or *banks*, corresponding to each of these eight possibilities (even/odd resolution level; even/odd row address; even/odd column address), we are guaranteed that any mip-map lookup will require precisely one access to each bank.

Mip-map textures normally are stored in the arrangement shown in Figure 5. The highest resolution map (level = 0) is stored in any of the three outer quadrants labeled 1, 2, or 3. The level 1 texture map is stored in quadrant 0. Lower-resolution maps are also stored in quadrant 0, which is quadrasected recursively for each succeeding level. Using this organization, mip-map textures can be packed into square memory arrays with no wasted storage. The storage required for a mip-map is 4/3 the storage required for its highest-resolution level [WILL83]. Address calculations are simple as well, since the address of a lower-level texture can be obtained by shifts and adds.

We can retain these advantages and achieve eight-way interleaving by providing two 2x2 interleaved 'panels' of memory as shown in Figure 6. Now, level 0 maps can be assigned to regions marked '0' in either panel. Whichever panel is chosen, the next resolution level is stored in the opposite panel. The third resolution level can be stored in the same panel, and so forth, as shown in the figure.

With this approach, there are two types of maps: *even* maps and *odd* maps. Even maps have their highest-resolution level (and other even levels) in Panel 0 and their odd levels in Panel 1. Odd maps have their highest-resolution (and other even levels) in Panel 1 and their odd levels in Panel 0.

Interleaving texture memory in this manner reduces the amount of redundancy by a factor of eight. Unfortunately for PixelFlow, this is not enough. Further redundancy is needed to meet the bandwidth requirements. As a result, the 32 SDRAMs are divided into four groups or modules of eight chips, with textures stored redundantly in the four modules. The eight chips within a module correspond to the eight banks described here. We will return to the redundancy issue in Section 6.

3.3 Ping-Pong SDRAM Accesses

The SDRAMs contain a feature that accelerates random memory accesses but complicates a designer's life: each SDRAM memory contains two internal memory banks. Each bank can be accessed in a method analogous to fast-page mode of a conventional DRAM: a row address is given followed by potentially multiple column addresses. The SDRAMs contain internal column address counters to generate multiple sequential column addresses for "burst" operations, a sequence of writes or reads to/from consecutive memory locations.

The two-bank design of the SDRAMs allows the row address command to be overlapped with burst reads/writes to/from the other bank, hiding the row access time. If banks are accessed alternately, this can nearly double the memory bandwidth of each chip. The problem is ensuring that banks will be hit alternately. An obvious way to do this is to store data redundantly in the two banks on each chip and have successive lookups access the data in whichever bank is next. A more complex method, but one that makes texture writing more efficient, is to store identical data in bank 0 of one panel and bank 1 of the other panel[1]. We have adopted this latter approach.

[1]When writing texture memory, it is desirable to keep both panels busy. If textures are replicated in both banks of the same chip, writing a given texture will affect only one of the two panels. To keep the other panel busy, two texture maps would have to be

Replicated textures are slower to write than non-replicated textures, since the data must be stored twice. They also reduce the effective amount of texture storage by a factor of two (from 16 MBytes to 8 MBytes), so they are only used when the fastest random lookups are needed.

3.4 SIMD Control

Another important issue is how to control texture operations. SDRAMs, like conventional DRAMs, have the property that successive accesses to the same row in memory are faster than operations that cross rows. Texture lookups generally exhibit a good deal of coherence. That is, texture lookups from nearby pixels are likely to access nearby texels. However, there is no guarantee that this will occur.

It is possible to organize a memory system that takes advantage of coherence when it occurs, performing row accesses only where necessary. This can be done by storing textures so that nearby texels lie on the same row as far as possible, generally by assigning square arrays of texels to SDRAM rows. Of course, successive accesses will not always hit the same row, and row crossings on different memories and different modules will occur at different times. To take advantage of this requires controlling the memories independently, akin to MIMD control in a multiprocessor.

The alternative is to be conservative and to perform a full row/column access for every texture lookup. This has the advantage of not restricting the way textures are stored in memory; the organization need not preserve coherence. Also, control of the memory subsystem is simpler since all texture lookups occur in lock step. This is akin to SIMD control in a multiprocessor.

We have chosen to use SIMD control because of its simplicity and because it fits naturally with PixelFlow's SIMD rasterizer engine.

3.5 Address and Filtering Calculations on EMCs

As anyone who has implemented texturing knows, performing the actual lookups is only one piece of the puzzle. The mip-map resolution level must be determined. Texture coordinates must be transformed, taking perspective warping into account. Texture addresses must be computed from texture coordinates. Finally, texture values must be filtered.

All of these steps are compute-intensive. Computing the resolution level for mip-maps requires a logarithm and several multiplies. Perspective-correction requires a high-precision reciprocal z calculation and several multiplies and adds. Filtering for mip-maps requires trilinear interpolation, another series of multiplies and adds.

What is worse is that other texture algorithms require other calculations. Environment maps, depth-map shadows, and bump maps require different address calculations [GREE86; REEV87; SEGA92]. Non-mip-map textures require different types of filtering.

It is possible to devise hardware that converts pixel values into addresses and does the required filtering, but it would require many modes and would have to be heavily pipelined or highly parallel to be fast enough. This, indeed, is the approach that most texture subsystem designers have taken [RICH89; LARS90; AKEL93]. For PixelFlow, we were not content to support only a few prescribed forms of texture access. We desired texturing to be as programmable as the rest of the rasterizer to allow flexibility in devising shading and other pixel/texture manipulation algorithms.

Furthermore, PixelFlow has a tremendous computation resource available—the SIMD processing elements. The PEs on a single board provide an aggregate of nearly a trillion 8-bit integer operations per second. In addition, they are completely programmable, allowing users to change algorithms at will. We decided to use the pixel processors for all address and filtering calculations.

4 SYSTEM COMPONENTS

We now describe the system components in more detail, emphasizing how they implement the functions above.

4.1 Image Generation Controllers

The texture subsystem (in fact the entire rasterizer) is controlled by a pair of microcoded sequencers called *Image Generation Controllers* (IGCs). The IGCs execute high-level instructions for the EMCs and TASICs, controlling their cycle-by-cycle operation. The *EIGC* controls the EMCs and synchronizes operations over the image-composition network. The *TIGC* controls the TASICs, the SDRAMs, and handshakes with the optional video port. A set of semaphores interlocks operation of the two IGCs.

4.2 Enhanced Memory Chips

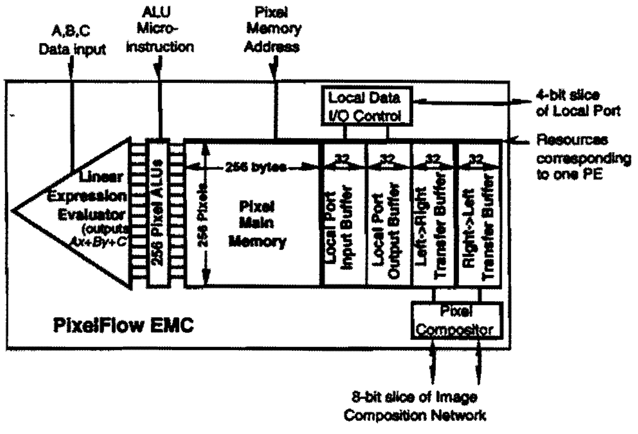

Figure 7: Logical diagram of a PixelFlow EMC (figure courtesy of John Eyles).

paired together and both written at the same time. This has ugly software implications. With the method we have adopted, the data is replicated across panels, so a single texture write affects both panels; the redundant data can be written in a single pass without wasting one panel's memory cycles.

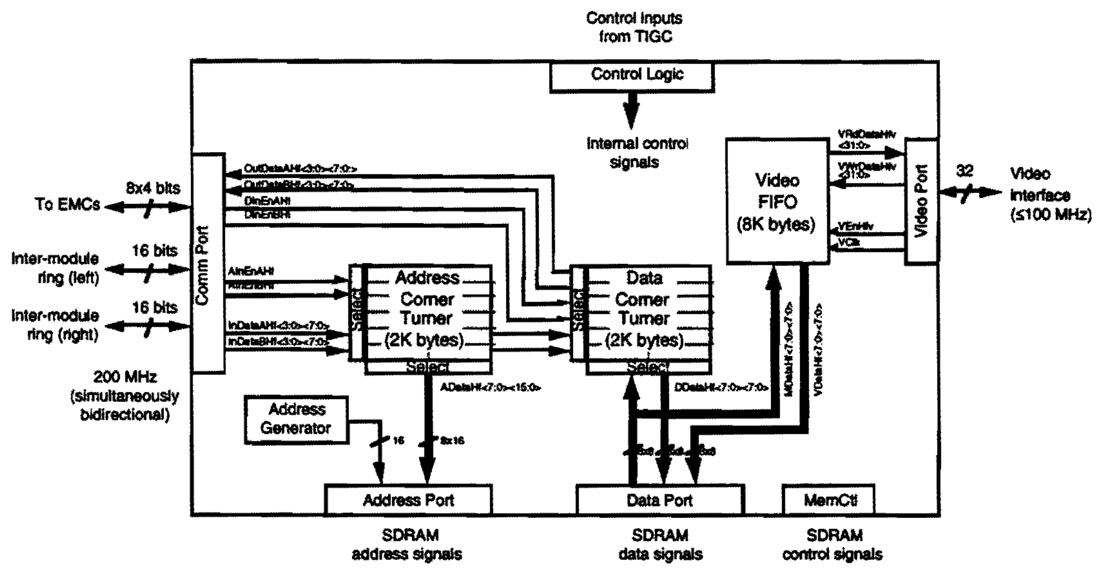

Figure 8: Block diagram of logical Texture ASIC (two physical TASIC chips).

The EMCs comprise the main computational resource for the texture subsystem. Each EMC contains 256 *processing elements* (PEs), each with its own 8-bit ALU, an output of a *linear-expression evaluator* (LEE), 256 bytes of local memory, two 32-byte *transfer buffers*, and two 32-byte *local-port buffers*. Figure 7 shows a logical diagram of an EMC.

Each PE's ALU is a general-purpose 8-bit processor, including an enable register which allows operations to be performed on a subset of the PEs. The PE can use LEE results (described below) or local memory as operands and can write results back to local memory. It can also transfer data between memory, the carry register, and the I/O buffers.

The LEE evaluates bilinear expressions $Ax + By + C$ for each PE of the array in parallel. A, B, and C are coefficients loaded from the IGC and (x, y) represent the PE's (x, y) address. The LEE is used primarily to interpolate parameters during rasterization, but can also be used during shading.

Each PE is provided with 256+4•32 bytes of local memory. The memory is divided into five partitions: a 256-byte main partition, which is used for most computation, and four 32-byte partitions used for external communication. Two of these, the local port buffers, are connected to the local port. The local port is connected to the TASICs, so that data can be exchanged between the local buffer and attached external memory. The other two, the image-composition buffers, are connected to the image-composition port.

The image-composition port and local port allow pixel data to be transferred serially to/from the EMCs to other EMCs (for compositing) or to/from the TASICs (to perform texture lookups or pixel-data writes to texture or video memory). Data from each PE is presented serially at each port. The number of bytes transferred to/from each PE and their location in the communication buffer are configured by software. The image-composition port is an 8-bit port; the local port is a 4-bit port. Both run at 200 MHz. with simultaneous bi-directional communication.

The local-port has a special feature that is useful for texture operations. Each PE has a *mark* register which can be set under software control to determine whether it will participate in the next texture operation. Only marked PEs transmit or receive data over their local port.

4.3 Texture ASICs

The array of eight *Texture ASICS* (*TASICS*) implements a data-parallel communication interface between pixel memory in the EMCs, texture/video memory, the GP, and optional video circuitry. The TASICs perform the buffering and data conversion required to read and write SDRAM memories and contain internal counters to refresh a video display or read video data from a frame grabber.

For packaging reasons, the TASICs are bit-sliced by two; the two *physical TASICs* function together as a single *logical TASIC*. Throughout the remainder of this paper, we will refer to logical TASICs (both bit slices), rather than physical TASICs. Figure 8 shows a block diagram of a logical TASIC.

The *Address Corner Turner* (*ACT*) is a 2K byte dual-ported memory that "corner-turns" serial addresses arriving from the EMCs or GNI and buffers them up into the parallel format required at the SDRAM address pins. Address data from the eight EMCs in the module stream into this memory from the left. The eight addresses from a single EMC are read out from below.

The *Data Corner Turner* (*DCT*) is similar to the ACT, except it buffers data rather than addresses and can transfer data in both directions, from EMCs to SDRAMs, or vice versa. Like the ACT, it is a 2K byte dual-ported memory. When writing textures to SDRAM memory, it receives data from eight EMCs from the left. The eight data values from a single EMC are read out from below. The DCT can be configured to send data in either direction and to transfer either 1, 2, or 4 byte data types.

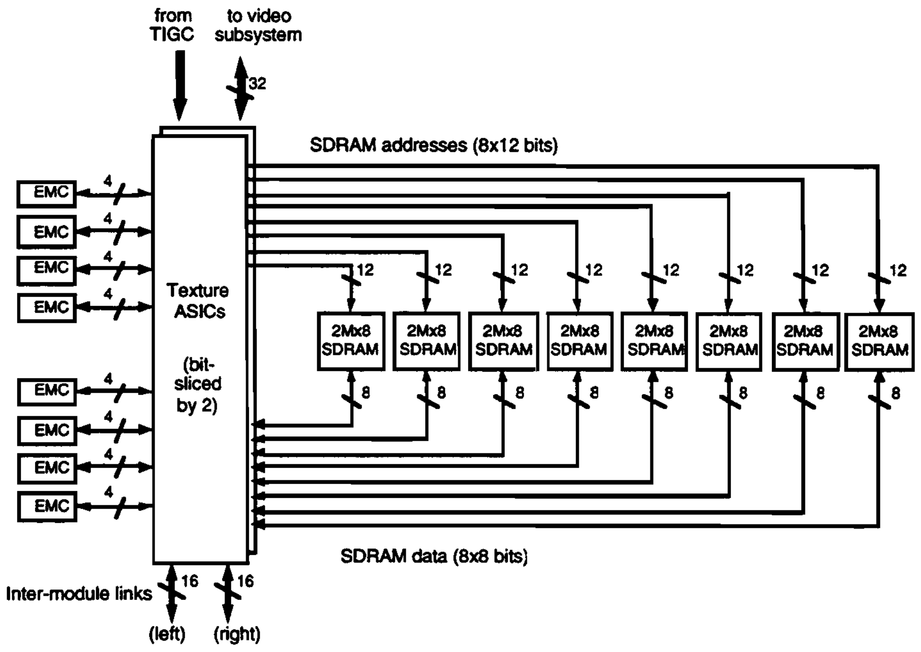

Figure 9: Interconnection between components in a module (one of four modules).

The TASIC has connections to the EMCs and the SDRAMs of the module. It also has connections to the TASICs of the other modules via a 16-bit ring network called the *inter-module ring*. This ring network provides communication between modules and the GP, allowing pixel data to be shared between modules and allowing the GP to participate in pixel calculations. Like the EMC-to-TASIC connections, the inter-module ring operates at 200 MHz, bidirectionally.

The datapath between the EMC port, the inter-module ring, and the ACT/DCT is configurable to allow data to be transferred in a variety of directions between the modules of a rasterizer and the GP.

Under certain circumstances, such as block texture writes (explained in Section 5) and transfers of texel data to and from the GP, it is impractical to provide memory addresses from the EMCs. To support these kinds of operations, the TASIC contains an internal address generator, consisting of programmable registers and counters.

Multiplexing circuitry at the memory address outputs selects among the different possible address sources. Some address sources can be XORed together as well, providing more flexibility when generating addresses.

The TASICs' video interface consists of an 8K byte *video FIFO* (VFIFO) and a set of registers and counters to keep track of pixel addresses within scanlines and video fields. The VFIFO buffers pixels into batches and handles the asynchronous boundary between the rasterizer and video subsystems. The video address registers store the row/column addresses of the starting scan line of up to eight independent video fields. Additional counters store the address of the current scan line and pixel. A set of registers also stores the order in which up to eight fields are to be displayed, so interleaved and/or stereo displays can be refreshed continuously without intervention by the GP. Special TIGC instructions update these address and field registers. They are used to swap buffers when double-buffering and to synchronize GP operation with video scanout when desired.

4.4 SDRAM Memory

The module is the "functional unit" of the texture subsystem. The eight SDRAM memories of each module are connected to the module's TASIC, as shown in Figure 9. The chips are controlled globally by the TIGC, so that all the SDRAMs on a board do a memory operation at the same time, but the addresses for each SDRAM are independent. The total storage per module is 16 MBytes. Each SDRAM can read or write data to/from a random location in memory at a peak rate of 100 Mbytes/sec, so the raw memory bandwidth per module is 8 • 100 MBytes/sec = 800 MByte/sec. The attainable memory bandwidth per module for four-byte reads/writes is approximately 580 MBytes/sec.

Addresses for texture reads/writes can come from one of three sources: 1) computed on EMCs and sent over the local-port to TASICs, 2) sent to TASICs over the geometry network, 3) generated on the TASICs themselves using their internal address generators.

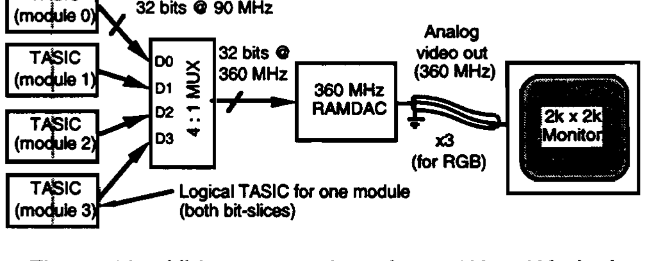

Figure 10: Video connections for a 2K x 2K-pixel frame-buffer board.

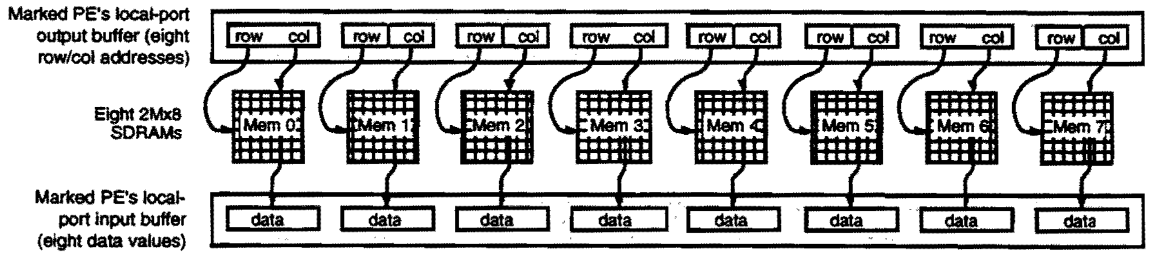
Marked PE's local-port output buffer (eight row/col addresses)
row
col
Eight 2Mx8 SDRAMs
Mem 0
Mem 1
Mem 2
Mem 3
Mem 4
Mem 5
Mem 6
Mem 7
Marked PE's local-port input buffer (eight data values)
data

Figure 11: Basic filtered texture read. Each PE specifies eight independent row/column addresses and receives a data value from each of the eight SDRAMs in the module.

4.5 Video Interface

On frame-buffer and frame-grabber boards (boards with an optional video interface), the SDRAMs store video data as well as textures. The SDRAMs provide storage for 4Kx4K 32-bit pixels, sufficient to double-buffer images for any display up to 4Kx2K pixels.

The TASICs of each module provide a 32-bit bidirectional video interface that can be clocked by an external video clock at up to 100 MHz. The pixels in a scan line are interleaved over the modules, so that every 4 (contiguous) pixels on a horizontal scan line come from different modules. By multiplexing the outputs of the four modules together, as shown in Figure 10, the overall pixel rate can be as high as 4 • 100 Mpixels/sec = 400 Mpixels/sec, sufficient to update a 2K x 2K monitor at 60 Hz.

When driving less aggressive video devices, video scan-out/scan-in requires only a fraction of the SDRAM bandwidth. For example, when refreshing a 1280x1024 at 60 Hz, video scanout requires less than 10% of SDRAM memory bandwidth. In such cases, portions of memory not needed for storing images can store texels for image-based texturing or other data as in any other system board.

5 ALGORITHMS AND PERFORMANCE

As discussed before, the most common texture operation is the 2D mip-map lookup, in which each PE looks up a value from a random location in each of the eight SDRAMs in its module. We will describe this basic operation first and variations later.

First, each PE calculates eight independent row and column addresses for the eight SDRAMs in its module. It then copies these into its local port buffer. Next, the texture subsystem reads these addresses from the local-port, applies them to the eight SDRAM memories, and reads the eight corresponding data values from the SDRAMs. It then loads these eight data values into the PEs' local-port input buffer. These operations are depicted schematically in Figure 11.

The texture subsystem performs this type of lookup at the maximum rate supported by the SDRAMs. Other classes of memory operation exhibit more coherence and can run faster, or require communication across module boundaries. We describe these variations in the following sections.

5.1 Coherent vs. Non-Coherent Reads/Writes

Mip-map and other texture lookups require that each texel be addressed independently. However, when writing texture data or frame-buffer data, the data generally is in array form already. There is no need to specify a row address for each data element. Rather, we can use fast-page mode to transfer data more rapidly.

For these reasons, PixelFlow provides two types of texture commands: *scatter commands*, which perform a full row/column access for each data item transferred, and *block commands*, which use fast-page mode to store coherent data. Scatter commands use addresses computed by the EMCs and transferred to the TASICs along with the data. Block commands use the TASICs' internal address generator to generate addresses.

5.2 Local vs. Global Writes

Data can be written from the EMCs of all modules to the modules' SDRAMs in a manner similar to texture reads. Alternatively, it can be broadcast from the EMCs of one module to the SDRAMs of all modules. This is useful for converting rendered images into textures. Other data transfer modes are available for sending data from one module to another and to transfer data to/from the GP.

5.3 Immediate Texturing

So far, we have assumed that texturing is done during deferred shading—after visible surfaces have been determined. An important class of texture operations, texture operations that affect visibility, do not fit this model. Texture-modulated transparency is an important algorithm of this type. The solution is to perform immediate texturing—texturing during rasterization. This is what virtually all other texture engines do. PixelFlow can do it also, but performance issues arise.

First, immediate texturing means texturing all rasterized pixels, whether they are visible in the final image or not. Second, partially transparent objects have to be blended properly with objects in front of and behind them. To do this properly requires front-to-back traversal, negating the benefits of the z-buffer algorithm. A common solution is to use screen-door transparency, enabling a fraction of the sub-pixel samples when antialiasing. This permits only a few levels of transparency, however, (as many as there are sub-pixel samples) and can produce artifacts when multiple transparent objects cover each other.

Operation	Data size (bytes)	Texture time (μsec)		
		Formula	50-pixel triangle	128x64 pixel region
Scatter read	8 x 2	0.92 + 0.32*n*	3.5	82.8
(mip-map)	8 x 4	0.92 + 0.48*n*	4.8	123.8
Local block read/write	2	0.65 + 0.025*n*	N/A	7.1
(backing-store or video transfer)	4	0.65 + 0.045*n*	N/A	12.3
Global block write	2	2.6 + 0.100*n*	N/A	28.2
(texture write to all modules)	4	2.6 + 0.182*n*	N/A	49.1

Figure 12: Times for various texture operations.

Operation	EMC/ TASIC	Compute time (μsec)	
		50-pixel triangle	128x64 pixel region
Invert 4-byte Z	EMC	9.9	9.9
Multiply u, v by invertex Z	EMC	4.1	4.1
Compute mip level	EMC	3.8	3.8
Compute texture addresses	EMC	12.2	12.2
Texture lookup	TASIC	5.3	141.7
Trilinear interpolate	EMC	4.5	4.5
Total EMC time	EMC	34.5	34.5
Total TASIC time	TASIC	3.5	123.8
Max of EMC and TASIC times		34.5 μsec	123.8 μsec

Figure 13: Comparison of immediate-texturing and deferred shading performance.

There are additional performance issues particular to PixelFlow. When doing immediate texturing, only a fraction of the PEs are involved at any given time. The mark register in the EMCs can be used to identify pixels that need texturing. However, this is one case where using PEs to calculate addresses and filter texture values really hurts: the SIMD array is utilized very poorly. Also, the pipelined nature of the texture lookup engine (particularly the serial transfer of address and data to/from the ACT/DCT) adds latency. This latency of approximately 80 cycles is insignificant when texturing a whole 128x64 array of pixels, but becomes important when only a few PEs are enabled.

As a result, the rasterizer performance of 0.5 million polygons/sec falls to about 25,000 polygons/sec when immediate texturing. A useful optimization is available if partially transparent object are tiled with primitives that do not overlap. The primitives can be rasterized sequentially and textured as a single batch, obtaining many of the benefits of deferred texturing. A major goal of any future texture subsystem for PixelFlow will be to improve immediate texturing performance.

5.4 Performance

Figure 12 gives the raw speed of the basic texture instructions on one rasterizer. For scatter reads, eight values (sufficient for a mip-map lookup) are assumed to be read or written per PE. For block reads/writes, one value is assumed to be read/written per PE. 'Local' block read/writes transfer data between the EMCs and SDRAMs of all four modules in parallel. This is typical of video and pixel-memory backing-store transfers. 'Global' block writes write data redundantly from the EMCs of one module to the SDRAMS of all four modules. This is typically done when a rendered image is converted into a texture. The times here are four times longer because texels from only one module can be written at a time.

The 'data size' column gives the texel size in bytes. In all cases, the amount of time required for the texture operation depends on the maximum number of marked PEs *n* in each EMC in the participating module (or modules). The 'formula' column gives the texture time as a function of *n*. The '50-pixel triangle' column assumes *n* = 8, typical when immediate-texturing 50-pixel triangles. The '128x64-pixel-region' column assumes *n* = 256 (full-sized region).

Figure 13 gives overall texturing times (including time for EMCs to compute texture addresses and to filter looked-up data) for two types of texture operations: immediate-texturing 50-pixel triangles and deferred texturing 128x64 regions. These times were obtained by running code fragments on the PixelFlow hardware simulator.

There are several things to notice from this table. First, note that the time required for EMCs to compute addresses and filter texture data is the same no matter how many PEs are marked. This follows from the SIMD nature of the rasterizer. Also, note that the EMC time is insignificant compared to the texture lookup time when deferred shading. When deferred shading, approximately 8,000 regions can be textured per second, which meets our goal of mip-map texturing 1280x1024 images with 5 samples per pixel at 30 Hz on four shaders.

However, when immediate texturing, the EMC time dominates, limiting the number of polygons that can be immediate-textured per second. From these results, we predict a maximum immediate texturing rate of about 28,000 triangles per second, a factor of almost 20 below the raw rasterization rate. We will discuss ways to improve this performance in the following section.

6 DISCUSSION

The PixelFlow texture subsystem met our primary design goals: to perform near the theoretical maximum for mip-map textures when deferred shading, and to be sufficiently programmable and flexible that it can be used for a wide variety of realistic-rendering effects. Members of our software team have written code to exercise it and have generated numerous images that demonstrate its features, such as the image shown in Figure 2.

6.1 Comparison with Previous Approaches

Our texture subsystem departs from the mainstream in two important ways: it uses deferred shading to minimize the number of texture/shading calculations that need to be performed and it uses a SIMD rasterizer engine to minimize the cost per processing element.

These characteristics give it a very different performance envelope than the texture systems on most commercial machines. On commercial machines that use immediate texturing, effects such as texture-modulated transparency are no more expensive than normal texture lookups. Immediate texturing on PixelFlow is costly, but PixelFlow's deferred texturing performance is much higher, particularly when multiple shaders are used in parallel.

Antialiasing has different ramifications for the different approaches. Immediate texturing systems generally texture and shade a single sample in each rasterized pixel and share this information among all the samples within the pixel. On PixelFlow, every sample can correspond to a different surface, and in the general case, must be textured/shaded independently. This can erase the benefit of deferred shading. For example, if pixels are sampled eight times, this could require the same work as immediate shading with a depth complexity of eight. So immediate texturing favors scenes with large primitives and low depth complexity and deferred shading favors scenes with small primitives and high depth complexity. We are exploring optimizations in which only the *k* dominant (most visible) surfaces in a pixel are shaded. Preliminary results indicate that shading three surfaces per pixel gets the vast majority of pixels right and there are ways to approximate the pixels that are wrong.

The programmable nature of the EMCs and the fact that they do all address and filter calculations makes the PixelFlow texture subsystem extremely flexible. We plan to experiment, and hope others will experiment, with a wide variety of algorithms that span the domains of rendering and imaging.

6.2 Implementation Lessons

The design presented here is the third in a series of evolving designs. The earlier designs had a simpler, cleaner software interface, but were slower. In the previous designs, virtually all of the details of the ACT/DCT, address generation, etc. were hidden from the user, but texture writes utilized less than half of the SDRAMs' write bandwidth, since addresses had to be provided by the EMCs. Henry Rich, of IVEX Corporation, proposed adding address generation logic to the TASICs to enable the use of block transfer commands. These indeed improved the texture subsystem write performance by nearly the theoretical factor of two for block texture writes, but made the system more difficult to program. Fortunately, much of this difficulty is hidden within a library that implements higher-level texture commands.

The inclusion of the video FIFO and registers and counters to generate addresses for video scan-in/scan-out proved to be a great win. These allow a wide variety of frame-buffer/frame-grabbers to be built by adding only a small amount of external hardware on an attached I/O card. For most frame-buffers/frame-grabbers, the only external logic that is needed is a timing generator and the analog video components. Since the amount of bandwidth required for video is a small fraction of the overall texture bandwidth for most video devices, the impact on the system is low. This approach has been used successfully by others [LARS90]; we recommend it.

Implementing the video subsystem was not without problems, however. Details such as supporting genlock, the desire for video scanout to continue even if the rest of the system is hung, and the asynchronous interface between the rasterizer and video subsystem added troublesome constraints to the design and took us several design iterations to get right.

The idiosyncratic nature of SDRAMs proved challenging. We committed to SDRAMs early and were thrashed about by ever-changing specifications. The dual-bank nature of SDRAMs, while improving peak bandwidth, complicates the memory system design and the user's programming model. It will be interesting to see whether other memory families currently being introduced, such as RAMBUS, extended output DRAM, etc. can provide the advantages of SDRAMs without the headaches.

6.3 Limitations and Future Work

The texture subsystem described here has three main shortcomings: slow immediate texturing, redundant calculations when supersampling, and painful replication of textures. We are reasonably confident that all of these can be solved in the future.

To make immediate texturing faster, we must add dedicated circuitry for address calculations and filtering. As mentioned before, other rendering systems do this already [RICH89; LARS90; AKEL93]. We hope to learn from their experience as they publish details on their work. At the same time, we hope to learn from our users. We have endeavored to give them a very general platform for texturing with capabilities not found on other machines. It will be interesting to see what features and modes they exercise and find useful. We will take these into account when we design future hardware for immediate texturing.

We believe that dominant-surface shading can eliminate many of the redundant calculations performed when supersampling.

We are pursuing two methods to reduce the resplication of textures: the use of a memory hierarchy and increased use of memory interleaving.

It may be possible to have a single, large texture repository with multiple, fast local caches of modest size. The large repository would store the bulk of the texture data, while only relatively small working sets would be stored in the caches. The success of this approach depends on the working set being manageably small and not changing rapidly. The replacement criteria would be different than for standard caches.

Interleaving by eight, as done in the PixelFlow design, works because of the nature of the mip-map algorithm. It may be possible to interleave further by hashing addresses, statistically distributing memory accesses over number of interleaved memory banks. This would require a MIMD control model, as opposed to the SIMD model we currently use, but has the potential to significantly diminish the amount of texture memory replication. This approach may be particularly attractive for systems that already use MIMD control.

7 CONCLUSION

The PixelFlow texture subsystem was designed to provide flexible, high-speed access to a large, shared texture/image memory. It allows PixelFlow to compute images such as the bowling image (Figure 2) in real time. With sufficiently clever programming, it can be used for a variety of other image-based algorithms, including: shadows, general function lookups, inter-PE communication, backing store for EMC memory, image resampling, optical predistortion, remapping for low latency, bump mapping, environment mapping, lightsource mapping, and hopefully others that are yet to be invented.

The system's SIMD control made the design simple and matched well to the existing SIMD rasterizer, but sacrificed some performance available in coherence between neighboring pixels. The 8-way interleaved memory design we use appears to reduce memory replication as much as possible while retaining SIMD control. The system's main limitations are poor performance when immediate texturing, the need to texture each sample when supersampling, and an eight-fold replication of texture data. We don't believe these limitations are intrinsic and hope to solve them in the future. In the meantime, we hope that the experience gained here will be useful to designers of texture subsystems on other high-performance rendering systems.

ACKNOWLEDGEMENTS

PixelFlow is a large team effort. Many people have contributed to the final form of the PixelFlow texture subsystem: John Eyles and John Poulton were instrumental in developing the initial design. Yulan Wang, Greg Welch, Brian Jones, Bruce Ledet, Lawrence Kesteloot, and Paul Keller wrote software to exercise the design (and exorcise bugs). Henry Rich argued for maximizing texture bandwidth at almost any cost and spurred a major set of revisions that allow us to capture the full bandwidth of the SDRAMs when writing as well as reading. Trey Greer contributed ideas on improving immediate-rendering performance and avoiding texture replication.

This research is supported by the Advanced Research Projects Agency, ISTO Order No. A410, the National Science Foundation, Grant No. MIP-9306208, and by generous grants of equipment from Hewlett-Packard and Digital Equipment Corporation.

REFERENCES

AKEL93 Akeley, K., "RealityEngine Graphics, "SIGGRAPH 93, pp. 109-116.

COOK87 Cook, R.L., Carpenter L., and Catmull E., "The Reyes Image Rendering Architecture", *SIGGRAPH 87*, Vol. 21, No. 4, pp. 95-102.

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.

GREE86 Greene, N., "Environment Mapping and Other Applications of World Projections", *IEEE CG&A*, Vol. 6, No. 11, Nov, 1986, pp. 21 - 29.

HANR90 Hanrahan, P. and Jim L., "A Language for Shading and Lighting Calculations", *SIGGRAPH 90*, Vol. 24, No. 4, pp. 289-298.

LARS90 Larson, R., B. Morrison, and A. Goris, "Hardware Texture Mapping," Hewlett-Packard Internal Report, Hewlett-Packard Company, Ft. Collins, CO, 1990.

LAST95 Lastra, A., S. Molnar, Y. Wang, and M. Olano, "Real-Time Programmable Shading," *Proceedings of the 1995 Symposium on Interactive Computer Graphics*, April 1995.

MOLN92 Molnar, S., Eyles J., and Poulton J., "PixelFlow: High-Speed Rendering Using Image Composition," *SIGGRAPH 92*, Vol. 26, No. 3, pp. 231-240.

REEV87 Reeves W.T., Salesin D. H., and Cook R. L., "Rendering Antialiased Shadows With Depth Maps," *SIGGRAPH 87*, Vol. 21, No. 4, pp. 283-291.

RICH89 Rich, H., "Tradeoffs in Creating a Low-Cost Visual Simulator," *Proceedings of the 11th Interservice/Industry Training Systems Conference*, 1989, pp. 214-223.

SEGA92 Segal, M., C. Korobkin, R. van Widenfelt, J. Foran, and P. Haeberli, "Fast Shadows and Lighting Effects using Texture Mapping," *SIGGRAPH 92*, Vol. 26, No. 3, pp. 249-252.

TEBB92 Tebbs, B., U. Neumann, J. Eyles, G. Turk, and D. Ellsworth, "Parallel Architectures and Algorithms for Real-Time Synthesis of High Quality Images using Deferred Shading," UNC CS Technical Report TR92-034.

UPST90 Upstill S., *The RenderMan Companion*, Addison-Wesley, 1990.

WHIT82 Whitted, T., and Weimer D. M., " A software Testbed for the development of 3D Raster Graphics Systems", *ACM Transaction on Graphics*, Vol. 1, No. 1, Jan. 1982, pp. 43-58.

WILL83 Williams, L., "Pyramidal Parametrics,", *SIGGRAPH 83*, Vol. 17, No. 3, pp. 1-11.