A Heterogeneous Multiprocessor Graphics System Using Processor-Enhanced Memories[1]

Henry Fuchs, John Poulton, John Eyles, Trey Greer, Jack Goldfeather[2],
David Ellsworth, Steve Molnar, Greg Turk, Brice Tebbs, Laura Israel

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

Abstract

This paper introduces the architecture and initial algorithms for Pixel-planes 5, a heterogeneous multi-computer designed both for high-speed polygon and sphere rendering (1M Phong-shaded triangles/second) and for supporting algorithm and application research in interactive 3D graphics. Techniques are described for volume rendering at multiple frames per second, font generation directly from conic spline descriptions, and rapid calculation of radiosity form factors. The hardware consists of up to 32 math-oriented processors, up to 16 rendering units, and a conventional 1280x1024-pixel frame buffer, interconnected by a 5 gigabit ring network. Each rendering unit consists of a 128x128-pixel array of processors-with-memory with parallel quadratic expression evaluation. Implemented on fast custom CMOS chips, this array has 208 bits/pixel on-chip and is connected to a video RAM memory system that provides 4,096 bits of off-chip memory. Rendering units can be independently reassigned to any part of the screen or to non-screen-oriented computation. A message-passing operating system encourages algorithms to mix and match capabilities of the massively parallel rendering units with those of the math-oriented processors. As of January 1989, both hardware and software are still under construction, with initial system operation scheduled for summer 1989.

1. Introduction

Many computer applications seek to create an illusion of interaction with a virtual world. Vehicle simulation, geometric modeling and scientific visualization, for example, all require rapid display of computer-generated imagery that changes dynamically according to the user's wishes. Much progress has been made in developing high-speed rendering hardware over the past several years, but even the current generation of graphics systems can render only modest scenes at interactive rates.

For many years our research goal has been the pursuit of truly interactive graphics systems. To achieve the necessary rendering speeds and to provide a platform for real-time algorithm research, we have developed the parallel image generation architecture called *Pixel-planes* [Fuchs 81, Fuchs 82, Poulton 85]. We briefly describe the basic ideas in the architecture:

---

[1] This work supported by the Defense Advanced Research Projects Agency, DARPA ISTO Order No. 6090, the National Science Foundation, Grant No. DCI-8601152, and the Office of Naval Research, Contract No. N0014-86-K-0680.

[2] Department of Mathematics, Carleton College, Northfield, MN.

Each pixel is provided with a minimal, though general, processor, together with local memory to store pixel color, z-depth, and other pixel information. Each processor receives a distinct value of a linear expression in screen-space, Ax + By + C, where A,B,C are data inputs and x,y is the pixel address in screen-space. These expressions are generated in a parallel linear expression evaluator, composed of a binary tree of tiny multiply-accumulator nodes. A custom VLSI chip contains pixel memory, together with the relatively compact pixel processors and the linear expression evaluator, both implemented in bit-serial circuitry. An array of these chips forms a "smart" frame buffer, a 2D computing surface that receives descriptions of graphics primitives in the form of coefficients (A,B,C) with instructions and locally performs all pixel-level rendering computations. Since instructions, memory addresses, and A,B,C coefficients are broadcast to all processors, the smart frame buffer forms a Single-Instruction-Multiple-Datastream computer, and has a very simple connection topology. Instructions (including memory addresses and A,B,C's) are generated in a conventional graphics transformation engine, with the relatively minor additional task of converting screen-space polygon vertices and colors into the form of linear expressions and instructions.

In 1986 we completed a full-scale prototype Pixel-planes system, Pixel-planes 4 (*Pxpl4*) [Poulton 87, Eyles 88], which renders 39,000 Gouraud-shaded, z-buffered polygons per second (13,000 smooth-shaded interpenetrating spheres/second, 11,000 shadowed polygons/second) on a 512x512 pixel full-color display. While this system was a successful research vehicle and is extremely useful in our department's computer graphics laboratory, it is too large and expensive to be practical outside of a research setting. Its main limitations are:

- large amount of hardware, often utilized poorly (particularly when rendering small primitives)
- hard limit on the memory available at each pixel (72 bits)
- no access to pixel data by the transformation unit or host computer
- insufficient front-end computation power

This paper describes its successor, Pixel-planes 5 (*Pxpl5*), which we expect to have running by mid-1989. *Pxpl5* uses screen subdivision and multiple small rendering units in a modular, expandable architecture to address the problem of processor utilization. A full-size system can render in excess of one million Phong-shaded triangles per second. Sufficient "front end" power for this level of performance is provided by a MIMD array of general-purpose math-oriented processors. The machine's multiple processors communicate over a high-speed network. Its organization is sufficiently general that it can efficiently render curved surfaces, volume-defined data and CSG-defined objects. In addition it can rapidly perform various image-processing algorithms. *Pxpl5*'s rendering units each are 5x faster than *Pxpl4* and contain more memory per pixel, distributed in a memory hierarchy: 208 bits of fast local storage on its processor-enhanced memory chips, 4K bits of memory per pixel processor in a VRAM "backing store", and a separate frame buffer that refreshes normal and stereo images on a 1280x1024 72Hz display.

2. Perspective

Raster graphics systems generally contain two distinct parts: a graphics transformation engine that transforms and lights the geometric description of a scene in accordance with the user's viewpoint and a renderer that paints the transformed scene onto a screen.

Designs for fast transformation units have often cast the series of discrete steps in the transformation process onto a pipeline of processing elements, each of which does one of the steps [Clark 82]. As performance requirements increase, however, simple pipelines begin to experience communication bottlenecks, so designers have turned to multiple pipelines [Runyon 87] or have spread the work at some stages of the pipe across multiple processors [Akeley 88]. Vector organizations offer a simple and effective way to harness the power of multiple processors, and have been used in the fastest current graphics workstations [Apgar 88, Diede 88]. Wide vector organizations may have difficulty with data structures of arbitrary size, such as those that implement the PHIGS+ standard, so at least one commercial offering divides the work across multiple processors operating in MIMD fashion [Torberg 87].

The rendering problem has generally been much more difficult to solve because it requires, in principal, computations for every pixel of every primitive in a scene. To achieve interactive speeds on workstation-class machines, parallel rendering engines have become the rule. These designs must all deal with the memory bandwidth bottleneck at a raster system's frame buffer. Three basic strategies for solving this problem are:

Rendering Pipelines. The rendering problem can also be pipelined over multiple processors. The Hewlett-Packard SRX graphics system [Swanson 86], for example, uses a pipeline of processors implemented in custom VLSI that simultaneously perform 6-axis interpolations for visibility and shading, operating on data in a pixel cache.

The frame buffer bandwidth bottleneck can be ameliorated by writing to the frame buffer only the final colors of the *visible* pixels. This can only be achieved if all the primitives that may affect a pixel are known and considered before that pixel is written. Sorting primitives by screen position minimizes the number that have to be considered for any one pixel. Sorting first by Y, then by X achieves a scan-line order that has been popular since the late 1960's and is the basis for several types of real-time systems [Watkins 70]. The basic strategy has been updated by several groups recently. The SuperBuffer design [Gharachorloo 85] contained a processor for every pixel on a scan-line. Data for primitives active on a scan-line pass by this array, and visible pixel colors are emitted at video rates; no separate frame buffer is required. This work continues at IBM/TJW on a system called SAGE [Gharachorloo 88]. Researchers at Schlumberger [Deering 88] recently proposed a system in which visibility and Phong-shading processors in a pipeline are assigned to the *objects* to be rendered on the current scan line. The latter two projects promise future commercial offerings that can render of order 1M triangles per second with remarkably little hardware, though designs for the front ends of these systems have yet to be published. These machines have each cast one particular rendering algorithm into hardware, enabling a lower-cost solution but one not intended for internal programming by users. New algorithms cannot easily be mapped onto hardware for scan-line ordered pipelines. Finally, a difficulty with these designs is ensuring graceful performance degradation for scenes with exceptional numbers of primitives crossing a given scan-line.

Interlaced Processors. As first suggested a decade ago [Fuchs 77, Fuchs 79, Clark 80], the frame buffer memory can be divided into groups of memory chips, each with its own rendering processor, in an interlaced fashion (each processor-with-memory handles every nth pixel on every mth row). The rendering task is distributed evenly across the multiple processors, so the effective bandwidth into the frame buffer increases by a factor of $m \cdot n$. This idea is the basis of several of the most effective current raster graphics systems [Akeley 88, Apgar 88]. Some of these systems, however, are again becoming limited by the bandwidth of commercial DRAMs [Whitton 84]. With increasing numbers of processors operating in SIMD fashion, processor utilization begins to suffer because fewer

processors are able to operate on visible pixels, the "write efficiency" problem discussed in [Deering 88]. Raising the performance of interlaced processors by an order of magnitude will probably require more complex organizations or new memory devices.

Processor-Enhanced Memories. Much higher memory bandwidth can be obtained by combining some processing circuitry on the same chip with dense memory circuits. The most widely used example of a "smart" memory is the Video RAM (VRAM), introduced by Texas Instruments. Its only enhancement is a second, serial-access port into the frame buffer memory; nevertheless these parts have had a great impact on graphics system design. The SLAM system, described some years ago in [Demetrescu 85], combines a 2-D frame buffer memory with an on-chip parallel 1-D span computation unit; it appears to offer excellent performance for some 2D applications but requires external processing to divide incoming primitives into scan-line slices. Recently NEC announced a commercial version of an enhanced VRAM that performs many common functions needed in 2-D windowing systems. This approach has been the focus of our work since 1980; in the Pixel-planes architecture we have attempted to remove the memory bottleneck by performing essentially all pixel-oriented rendering tasks within the frame buffer memory system itself.

The architecture we will describe below employs a MIMD array of processors in its transformation unit and seeks to make more effective use of the processor-enhanced memory approach.

3. Project Goals

We wanted Pixel-planes 5 to be a platform for research in graphics algorithms, applications and architectures, and a testbed for refinements that would enhance the cost effectiveness of the approach. To this end, we adopted the following goals:

- **Fast Polygon Rendering.** Despite all the interest in higher-order primitives and rendering techniques, faster polygon rendering is the most often expressed need for many applications: 3D medical imaging, scientific visualization, 'virtual worlds' research. We therefore set a goal of rendering 1 million Z-buffered Phong-shaded triangles per second, assuming the average triangle's area is 100 pixels and that it is embedded in a triangle strip. We wanted to achieve this rate without using any special structures for rendering just triangles — we wanted a system for much more than triangles.

- **Generality.** For the system to be an effective base for algorithm development, it needed to have a simple, general structure whose power was readily accessible to the algorithm developer programming in a high-level language. We wanted it to have sufficient generality for rendering curved surfaces, volume data, objects described with Constructive Solid Geometry, for rendering scenes using the radiosity lighting model, and (we hoped) for a variety of other 3D graphics tasks that we have not yet considered. It was essential that the system support a PHIGS+ -like environment for application programmers not interested the system's low-level details. Further, the hardware platform should be flexible to allow experiments in hardware architectures.

- **Packaging.** A high-performance configuration that met our primary performance goals should fit within a workstation cabinet with no unusual power requirements. We also wanted a system that could be modularly

built and flexibly configured to trade cost for performance. The system should drive a 1280x1024 display at >60Hz, and be able to update full scene images at >20 frames/second.

4. Parallel Rendering by Screen-space Subdivision

We now describe the scheme we use in *Pxpl5* to attain high levels of performance in a compact, modular, expandable machine. Our previous work has depended on a single, large computing surface of SIMD parallel processors operating on the entire screen space. In the new architecture, we instead have one or more small SIMD engines, called Renderers, that operate on small, separate 128x128-pixel patches in a virtual pixel space. Virtual patches can be assigned on the fly to any actual patch of the display screen. The system achieves considerable speedup by simultaneously processing graphics primitives that fall entirely within different patches on the screen.

The principal cost of this screen-space subdivision scheme is that the primitives handled in the transformation engine must be sorted into "bins" corresponding to each patch-sized region of the screen space. Primitives that fall into more than one region are placed into the bins for all such regions. The simplest (though expensive) way to support these bins is to provide additional storage in the transformation engine for the entire, sorted list of output primitives. Once transformed, sorted, and stored, a new scene is rendered by assigning all available Renderers to patches on the screen and dispatching to these Renderers primitives from their corresponding bins. When a Renderer completes a patch, it can discard its Z-buffer and all other pixel values besides colors; pixel color values are transferred from on-chip pixel memory to the secondary storage system, or "backing store", described below. The Renderer is then assigned to the next patch to be processed. This process is illustrated in Figure 1 for a system configured with four Renderers.

The general idea of multiple independent groups of pixel processors operating on disjoint parts of the display screen was described in several of our earlier publications as "buffered" Pixel-planes. What is new about this implementation is the idea of flexibly mapping small virtual pixel spaces onto the screen space. It allows useful systems to be built with any number of small rendering units, permits cost/performance to be traded nearly linearly, and can render into a window of arbitrary size with only linear time penalty.

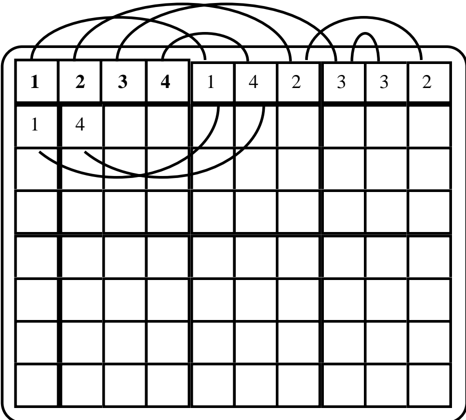

Figure 1. Rendering process for a *Pxpl5* system with 4 Renderers. 1280x1024 screen is divided into 80 128x128 patches. Patches are processed in raster order. Renderers 1-4 are assigned initially to the first four patches. Renderer #1 completes first, and is assigned to the next available patch. Next Renderer 4 completes its first patch and is assigned to the next available patch, and so forth.

The virtual pixel approach is supported in the *Pxpl5* implementation by a memory hierarchy, whose elements are: (1) some 200 bits of fast SRAM associated on-chip with each pixel processor; (2) a "backing store" built from VRAMs, tightly linked to the custom logic-enhanced memory chips; (3) a conventional VRAM frame buffer. The backing store is consists of an array of VRAMs, each connected via its video port to one of our custom memory chips; 1MB VRAMs provide 4Kbits of storage per pixel. The backing store memory is available through the VRAM random I/O port to the rest of the system, which can read and write pixel values in the conventional way. A Renderer uses this memory to save and retrieve pixel values, effectively allowing "context switches" when the Renderer ceases operations on one patch and moves to another. A typical context switch takes about 0.4 msec, the time to render a hundred or so primitives, and can be fully overlapped with pixel processing.

In the simple multi-Renderer scheme described above, the backing store is used to store pixel color values for patches of the screen as the Renderer completes them. When the entire image has been rendered, each of these regions is transferred in a block to the (double-buffered) display memory in the Frame Buffer, from which the display is refreshed.

5. Architectural Overview

We now describe the architectural features of the *Pxpl5* system as well as the motivation for various design decisions. The major elements of the design are:

- **Graphics Processors** (GPs), floating point engines, each with considerable local code and data storage.
- **Renderers**, each a small SIMD array of pixel processors with its own controller.
- **Frame Buffer**, double-buffered, built from conventional Video RAMs, from which the display is refreshed.
- **Host Interface**, which supports communications to/from a UNIX workstation.
- **Ring Network** to interconnect the various processors in a flexible way.

We discuss these elements in more detail below.

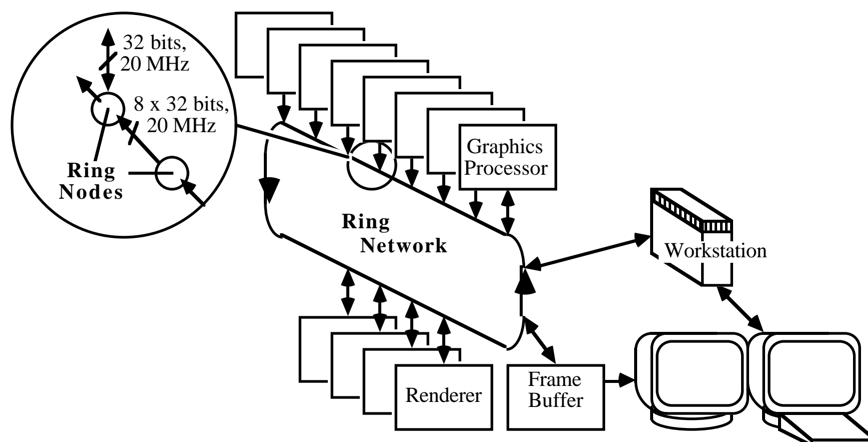

Figure 2. *Pxpl5* block diagram.

5.1 Ring Network

Pxpl5's multi-processor architecture, motivated by the desire to support a variety of graphics tasks, requires a capable communications network. Rather than build several specialized communications busses to support different types of traffic between system elements, we instead provide a single, flexible, very high performance network connecting all parts of the system.

At rendering rates of 1M primitives per second, moving object descriptions from the GPs to the Renderers requires up to 40 million 32-bit words/second (40 MW/sec), even for relatively simple rendering algorithms. Simultaneously, pixel values must be moved from the Renderers to the Frame Buffer at rates up to 40 MW/sec, for real-time interactive applications. At the suggestion of J. William Poduska of Stellar Computer, Inc., we explored technology and protocols for fast ring networks, and eventually settled on a multi-channel token ring. Ring networks have many advantages over busses in high-speed digital systems. They require only point-to-point communication, thus reducing signal propagation and power consumption problems, while allowing a relatively simple communication protocol.

Our network can support eight simultaneous messages, each at 20 MW/sec for a total bandwidth of 160 MW/sec. To avoid deadlock, each transmitting device gains exclusive access first to its intended receiver, then to a data channel, before it can transmit its data packet. Each Ring Node is a circuit composed of commercial MSI bus-oriented data parts and field-programmable controllers. (At the expense of an expensive development cycle, the Ring Network could be reduced to one or a few ASICs). The controllers operate at 20MHz, while data is moved at 40MHz (to save wires). Each client processor in the system has one or more of these Nodes, and each Node provides to the client a 20 MW/sec bi-direction port onto the Ring network.

We have developed a low-level message-passing operating system for the ring devices called the Ring Operating System (ROS). It provides device control routines as well as hardware independent communication. In addition, ROS controls the loading and initialization of programs and data.

5.2 Graphics Processors

The performance goals we have set require sustained computation rates in the "front end" of several hundred MFlops, feasible today only in parallel or vector architectures. We elected to build a MIMD transformation unit; this organization handles PHIGS+-like variable data structures better than would a vector unit, and supports the "bins" needed for our screen subdivision multi-Renderer. Weitek "XL" processors were used primarily because of the existence of mature compilers and assemblers.

Much of the system's complexity is hidden by ROS; the programming model is therefore relatively simple. Load sharing is accomplished by dividing a databases across the GPs, generally with each GP running the same code. Since the GPs are programmable in the C language, users have access to the machine's full capability without needing to write microcode.

5.3 Renderer

Section 4 describes the essentials of the Renderer design, whose block diagram is shown in Figure 3. It is based on a logic-enhanced memory chip built using 1.6μ CMOS technology and operating at 40MHz bit-serial instruction rates. In addition to 256 pixel processing elements, each with 208 bits of static memory, the chip contains a quadratic expression evaluator (QEE) that produces the function $Ax+By+C+Dx^2+Exy+Fy^2$ simultaneously at each pixel [Goldfeather 86b]. Quadratic expressions, while not essential for polygon rendering, are very useful for rendering curved surfaces and for computing a spherical radiosity lighting model (see Section 7.6).

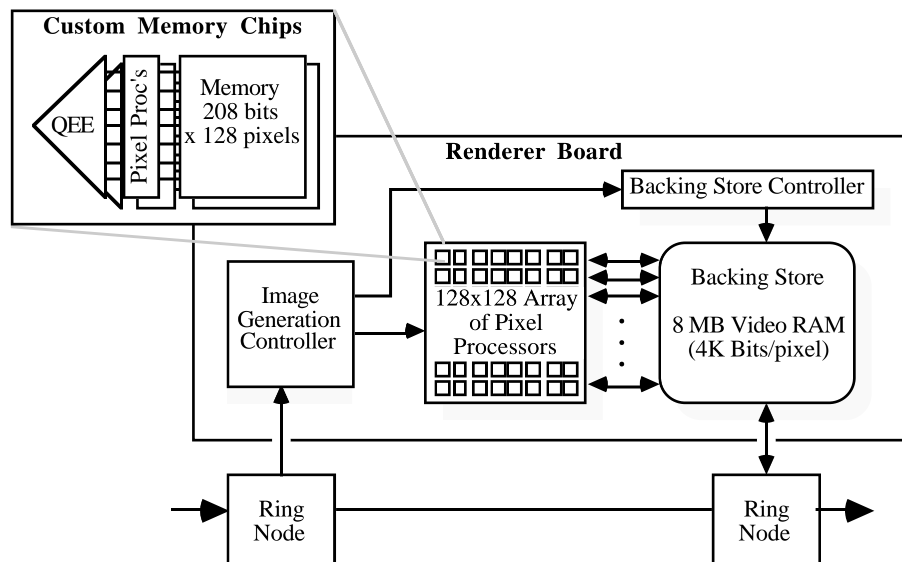

Figure 3. Block diagram of a *Pxpl5* Renderer. Pixel processor array implemented in 64 custom chips, each with 2 columns of 128 pixel processors-with-memory and a quadratic expression evaluator.

A major design issue for the Renderer was choosing the size of the processor array. The effectiveness of the screen-space subdivision scheme for parallel rendering is determined in part by the frequency with which primitives must be processed in more than one region, and this in turn depends on the size of the Renderer's patch. On one hand,

economy of use of the fairly expensive custom chips of the processor array and the need to leverage performance by dividing the rendering work across as many processors as possible argue for smaller Renderer patches. A large Renderer patch, on the other hand, reduces the likelihood that primitives will need to be processed more than once. We elected a 128x128 Renderer size; it is fairly efficient for small primitives, and its hardware conveniently fits on a reasonable size printed circuit board.

5.4 Frame Buffer and Host Interface

The Frame Buffer is built in a fairly conventional way using Video RAMs. It supports a 1280x1024-pixel, 72Hz refresh-rate display, 24-bit true color and a color lookup table. Display modes include stereo (alternating frames) and a hardware 2x zoom. The Frame Buffer is accessed through two Ring Nodes, to provide an aggregate bandwidth of 40 MW/sec into the buffer, allowing up to 24 Hz updates for full-size images. *Pxpl5* is hosted by a Sun 4 workstation. Host communications is via programmed I/O, providing up to about 4 MBytes/sec of bandwidth.

5.5 Performance

Since the transformation engine in *Pxpl5* is based on the same processor used in *Pxpl4*, we estimate, based on the earlier machine's performance, that a GP can process of order 30,000 Phong-shaded triangles per second; 32 GPs are therefore required to meet our performance goal. A single Renderer has a raw performance of about 150,000 Phong-shaded triangles per second; actual performance is reduced somewhat by inefficiencies resulting from primitives that must be processed in more than one patch. Simulations predict an actual performance of around 100,000 triangles/sec, so a configuration to meet the performance goals will require 8-10 Renderers.

6. PPHIGS Graphics Library

Pxpl5 may be programmed at various levels. We anticipate users ranging from application programmers, who simply desire a fast rendering platform with a PHIGS+ -style interface [van Dam 88], to algorithm prototypers, who need access to the renderer's low-level pixel operations and may depart from the PHIGS+ paradigm. To meet these disparate needs, several layers of support software are required. Program initialization and message passing between processors are handled by the Ring Operating System (ROS). A local version of PHIGS+ (Pixel-planes PHIGS or *PPHIGS*) provides a high-level interface for users desiring portable code. This section describes PPHIGS.

PPHIGS makes the hardware appear to the "high-level" graphics programmer very much like any other graphics system: the programmer's code (running on the host) makes calls to the graphics system to build and modify a hierarchical data structure. This structure is traversed by the PPHIGS system to create the image on the screen.

6.1 Database Distribution

Since the applications programming library is based on PHIGS, it allows the programmer to create a display list that is a directed acyclic graph of structures. These structures contain elements that are either graphics primitives, state-changing commands, or calls to execute other structures. To take advantage of the multiple graphics processors in *Pxpl5*, we must distribute the database structure graph across the graphics processors in a way that balances the

computational load, even in the presence of editing and changes in view. To achieve this we distribute each structure's primitives across the GPs instead of placing an entire structure on one GP.

Because PHIGS is a stateful system where child structures inherit information from their parents, the state-changing primitives as well as structure executions must be replicated on each GP. This replication should not be a problem, since we expect the majority of structure elements to be graphics primitives and not state-changing ones. We have devised other distribution schemes for applications that violate this assumption. The display list traversal and rendering routines are not affected by the distribution scheme.

6.2 The Rendering Process

The rendering process is controlled by a designated graphics processor, the master GP or *MGP*. By exchanging messages with other GPs and sending commands to other modules when necessary, the MGP synchronizes operations throughout the system.

Before discussing the steps in the rendering process, we first want to emphasize the distinction between pixel operations that take place on a per primitive basis, such as Z comparison and storage, and those that can be deferred until the end of all primitive processing or *end-of-frame*. Shading calculations from intermediate values stored at the pixels, for instance, need only be performed once per pixel, rather than once per primitive (assuming there is sufficient pixel storage to hold the intermediate values until end-of-frame). At that time, the final pixel colors for every pixel in the 128 x 128 pixel renderer can be calculated from the stored values. This savings results in about a 80x speedup for Phong shading over doing the Phong final calculations after every primitive is processed. For more expensive lighting and shading models (such as texture) this speedup is critical for making the algorithm practical.

The major steps in the rendering process are:

1. The application program running on the host edits the database using PPHIGS library routines and transmits these changes to the GPs.
2. Application requests a new frame. Host sends this request the MGP, which relays it to the other GPs.
3. The GPs interpret the database, generating renderer commands for each graphics primitive. These commands are placed into the local bins corresponding to the screen regions where the primitive lies. Each GP has a bin for every 128x128 pixel region in the window being rendered.
4. The GPs send bins of containing commands to renderers. The renderers execute commands and compute intermediate results.
5. The GP sending the final bin to a renderer also sends end-of-frame commands for the region. The renderers execute these commands and compute final pixel values from the intermediate results.
6. The renderers send computed pixels to the frame buffer.
7. When all regions have been received, the frame buffer swaps banks and displays the newly-computed frame.

The MGP assigns renderers to screen regions while the frame is being rendered. It communicates a renderer assignment to the GPs by sending a message to one GP, which sends its associated bin, and then forwards the message to the next GP, which does the same. At the end, the message is sent back to the MGP, indicating that all the bins have been processed. This method ensures that at most one GP attempts to transmit to a renderer at a given time. This prevents blocked transmissions, which would slow throughput.

The steps of the rendering process can be overlapped in several ways; at maximum throughput, several frames may be in progress at once. The MGP handles synchronization to keep the frames properly separated [Ellsworth 89].

7. Rendering Algorithms

We now discuss various rendering algorithms in turn. Some of these have been published before, in which case, we review their applicability to *Pxpl5* and give performance estimates. We also report new techniques for efficiently displaying procedural textures and conic spline-defined fonts, for calculating radiosity form-factors, and for displaying volume-defined images at interactive rates.

7.1 Phong Shading

Since *Pxpl5* can evaluate quadratic expressions rapidly, we could implement Phong shading using Bishop and Wiemer's Fast Phong Shading technique [Bishop 86]. Unfortunately, this requires large amounts of computation by the front-end processors to determine the quadratic coefficients. Since we estimate that the renderers will usually have more idle capacity than the GPs, we have decided to use a scheme which pushes most of the computation to the renderers and is closer to the original Phong formulation [Phong 73].

As polygons and other primitives are processed, the x, y, and z components of the surface normal are stored in all the pixels where the primitive is visible. For polygons this is done by simple linear interpolation of each component. When all the primitives for a region have been processed, the pixel-parallel end-of-frame operations are performed. The normal vector is normalized by dividing by the square root of its length, which is computed using a Newton iteration. Once this is done the color for each pixel is computed using the standard Phong lighting model.

Simulation indicates that the end-of-frame computation for the Phong lighting model with a single light source consumes around 23,000 renderer cycles or .57 milliseconds. With full screen resolution of 1024 by 1280 and a 16 renderer system, the total end-of-frame time is .57msec • (80/16) or 2.85msec per frame. At 24 frames per second this is 6.8 percent of the rendering time.

7.2 Spheres

Pxpl5 can render spheres using the same algorithm as on *Pxpl4* [Fuchs 85], but is both faster (taking advantage of the QEE), and can generate higher-quality images (Phong shading with 24-bit color). Phong shading is achieved as follows. The expressions for the coordinates of the surface normal for a sphere are:

$$nx = (1/r) \bullet (x - a)$$

$$ny = (1/r) \bullet (y - b)$$

$$nz = \left(\frac{1}{r^2}\right) \bullet \sqrt{r^2 - (x - a)^2 - (y - b)^2}$$

The expression for nz approximated by a parabola:

$$nz = \left(\frac{1}{r^2}\right) \bullet \left(r^2 - (x - a)^2 - (y - b)^2\right)$$

Then the normals are computed at each pixel by broadcasting two linear expressions and one quadratic expression. Results from simulation indicate that this approximation produces satisfactory shading including the specular highlights. Assuming one light source and 24 frames per second, we estimate the system performance to be 1.8M spheres per second for 100 pixel area spheres and 900K spheres per second for 1600 pixel area spheres.

7.3 Shadows

Pxpl4 generates images with shadows very rapidly — nearly half as fast as images without shadows [Fuchs 85]. Unfortunately, this figure does not scale up by the usual 20x for *Pxpl5*, since the performance increase from the screen space subdivision does not extend to shadow volumes, which frequently cross many screen regions. At worst, every shadow volume edge could be processed in every region, increasing the display list size by as much as 80/1.4 = 57x for a 1280 x 1024 image. A nominal *Pxpl5* configuration has 16 renderers, each running at 40MHz as opposed to *Pxpl4*'s 8MHz. The shadow algorithm might be expected to run about the same speed on *Pxpl5* as on *Pxpl4*. Various optimizations do exist; for example, shadow boundary edges need not be processed in regions lying between a polygon and the light source. We have not yet explored these options in depth. Because of the problems mentioned above, we anticipate increasing use of the fast radiosity technique described in Section 7.6.

7.4 Texture Mapping

We have previously reported a technique to compute the *u,v* texture coordinates for polygons in perspective [Fuchs 85]. The speed of this technique is limited by the time to broadcast the individual texture values to the pixels. While 64 x 64 -image textures run at interactive rates on *Pxpl4* (see Figure 4), a more efficient method for *Pxpl5* is to calculate the texture values directly in each pixel. Broadcasting the texture values will be significantly faster on *Pxpl5* than on *Pxpl4*, since texture values can be stored in bins and only broadcast when needed for one or more pixels of a region.

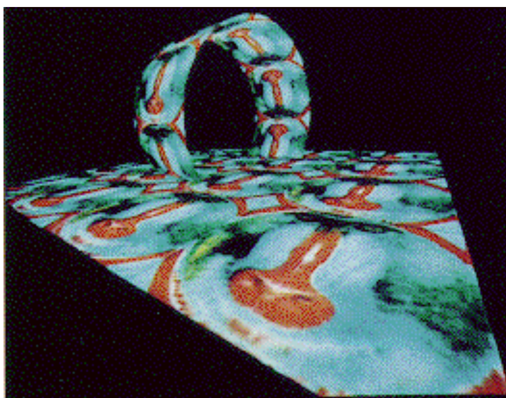

Figure 4. Mandrill mapped onto a plane and hoop on *Pxpl4*. Estimated rendering time on *Pxpl5* is 31 msec.

Procedural Textures. We have begun to explore procedural textures, as shown by Perlin [Perlin 85] and Gardner [Gardner 88], for use in *Pxpl5*. We have written a program for *Pxpl4* that allows one to explore in real-time the space of textures possible using Gardner's technique. This program and software written by Douglass Turner were used to create the textures shown in Figure 5.

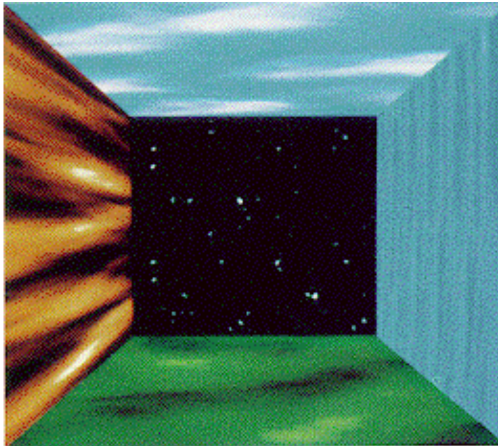

Figure 5. Procedural earth, water, sky, and fire textures with brick image texture (simulated). Estimated rendering time on *Pxpl5* is 5.5 milliseconds.

The two-dimensional Gardner spectral functions are calculated using quadratic approximations for the cosine functions. This requires nine multiplies per term plus one multiply to combine the x and y directions. Different textures for different pixels can be computed simultaneously. The images shown in the figure contains five terms. On *Pxpl5* they would require about 15,000 cycles or 360 microseconds using 10 bits of resolution. These procedural methods can be anti-aliased by eliminating high frequency portions of the texture; terms whose wavelength spans less than one pixel are simply not computed [Norton 82].

Image-based Textures. We have explored both summed area tables [Crow 84] and mip-maps [Williams 83] for anti-aliasing image textures. We feel that mip-maps will work best on *Pxpl5*. During rendering the mip-map interpolation value can be linearly interpolated across the polygon. At end of frame, the mip-map is broadcast to each pixel-processor, and each processor loads the pixel at its u,v coordinate along with neighboring values for interpolation.

7.5 Fonts

Herve Tardif has been developing methods for rapidly rendering fonts. Conic splines, as advocated by several researchers [Pavlidis 83, Pratt 85], are particularly well suited for rendering by *Pxpl5*; with the QEE in the processor-enhanced memories, *Pxpl5* can directly scan convert conic section, from which characters are defined. Initially, a character is represented by a sequence of straight line segments and arcs of conics joined together in the plane. As suggested by Pratt, each arc of a conic is in turn represented by three points M, N, P and a scalar S which measures the departure of the conic from a parabola (Figure 6.a). Hence, a letter can be represented either by a simple closed polygon or, for letters with holes, two or more polygons. The character is initially converted into the

difference between its unique convex hull and the discrepancy with that hull. (Holes are treated the same as other discrepancies.) The process is repeated if the discrepancy region(s) are concave. This process amounts to building a tree whose leaves are convex regions and nodes are set operators [Tor 84] (see Figure 6.b). A character is rendered by traversing its corresponding tree, scan converting each convex region in turn.

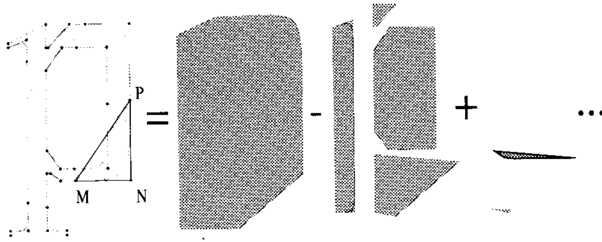

Figure 6. Conic font constructed by regions bounded by lines and conic sections.

Consider a convex region obtained through this process. For edges corresponding to straight line segments the coefficients for that line are sent to the QEE. For two consecutive edges MN and NP representing an arc of conic, the coefficients of the straight line (MP) are first sent to the QEE. Then the quadratic coefficients for the conic section are derived and sent to the QEE in order to scan convert the region enclosed between the line MP and the conic section. This process is repeated until all convex regions have been processed. Figure 6.c shows the regions that are successively scan converted for the letter P. It is possible for the two segments MN and NP that describe an arc to be split into two different convex regions during the decomposition process. In that case, the edges MN and NP are considered as simple segments of the character definition until all convex regions have been processed. Then, the region enclosed between the conic and the segments MN and NP is either added or subtracted from the current construction (see Figure 6.d). Since conic sections are invariant under projective maps, this technique can also be applied to the rendering of planar characters embedded in a 3D environment.

Performance estimates have been obtained from a conic representation of a Times Roman font given to us courtesy of Michael Shantz of Sun Microsystems. The average number of convex polygons per character in this set is 8.12, the average number of straight edges per polygon is 4.13, and the average number of conics per character is 8.4. This indicates that the average character can be scan-converted with 36 linear coefficients and 8.4 quadratic coefficients. This suggests that each renderer can scan-convert over 20,000 letters per second. Assuming each character falls into an average of 1.4 rendering regions, 16 renderers can draw over 225,000 letters per second. Graphics Processors will have difficulty keeping up with this rendering rate. GPs can cache renderer commands for 2D applications.

7.6 Fast Radiosity

The radiosity lighting model offers dramatically improved realism for certain types of images [Immel 86, Wallace 87]. The progressive radiosity approach [Cohen 88] would be well-suited for interactive applications if images could be computed more rapidly. *Pxpl5*'s renderers allow us to greatly accelerate the progressive radiosity method. We have developed an algorithm for computing projections of 3D polygons onto a hemisphere, which speeds the projection, scan conversion and visibility calculations necessary to distribute light from a light source to the patches

in the scene. Instead of storing color values at the pixels, we store the patch id of the nearest visible patch. Once all the patches in a scene have been processed, the visible patch matrix is sent over the ring network to a GP, which scans through the matrix, updating the radiosities of patches indicated by the patch ids.

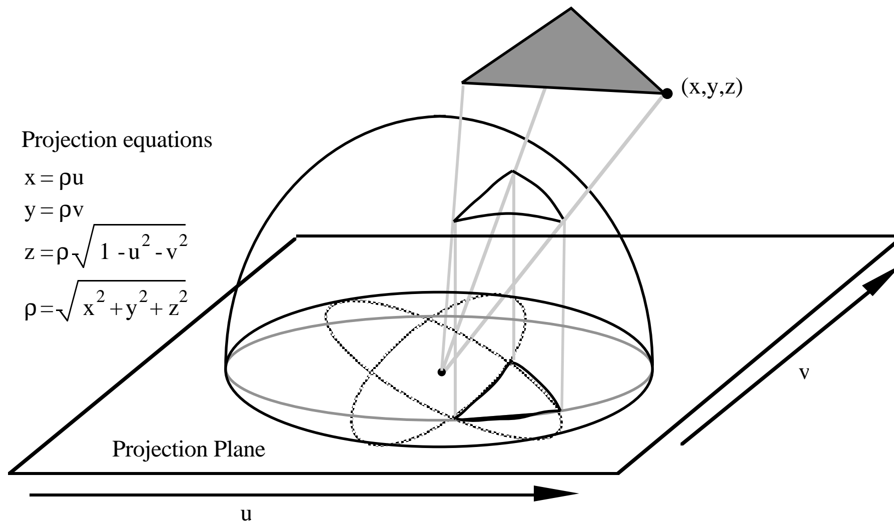

Figure 7. Hemispherical projection of a triangle.

In the unusual scan conversion process, the edges of a polygon map to ellipses in screen-space. The *Pxpl5* QEE computes these ellipses directly to scan convert the polygons' projections into pixel space. Figure 7 illustrates the scan conversion process. Z-buffering can be performed using approximations or by storing a special constant term for each pixel [Goldfeather 89]. Figure 8 shows the result of the hemispherical projection and Z-buffered rendering of a room environment.

This technique can compute these radiosity form-factors in one pass instead of the five passes that would be necessary in a hemi-cube implementation — and even this one pass could be done at *Pxpl5* polygon rendering rates. Since the resolution within a single renderer appears to be more than adequate for this calculation, multiple renderers can be used independently. Each renderer should be able to process about 100,000 quadrilaterals per second. If the GPs cannot keep up, we may be calculate the form factors at reduced resolution, reducing the number of patch id's the GPs need to tally.

Displaying the radiosity image is performed in the conventional manner: vertex colors are computed from patch radiosities, and linear-interpolation is used to blend colors smoothly across each patch.

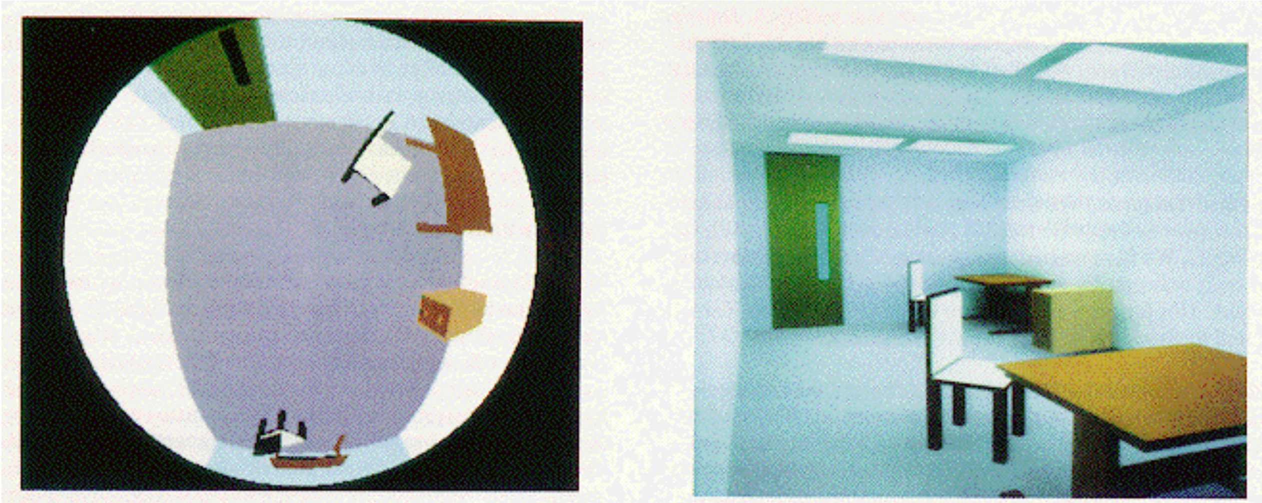

Figure 8. (a) Hemispherical projection of Tebbs and Turk's office, generated on the *Pspl5* simulator. Estimated rendering time on *Pxpl5* is 2.8 milliseconds. (b) Standard view of the same room as in (a), generated on *Pxpl4*. The viewpoint in (a) is from the illuminated light fixture.

7.7 Volume rendering

One example of *Pxpl5's* generality is its ability to perform volume rendering. Marc Levoy plans to implement a version of the algorithm described in [Levoy 88a, 88b, 89]. To briefly summarize the algorithm: We begin with a 3D array of scalar-valued voxels. We first classify and shade the array based on the function value and its gradient to yield a color and an opacity for each voxel. Parallel viewing rays are then traced into the array from an observer position. Each ray is divided into equally spaced sample intervals, and a color and opacity is computed at the center of each interval by tri-linearly interpolating from the colors and opacities of the nearest eight voxels. The resampled colors and opacities are then composited in front-to-back order to yield a color for the ray.

For *Pxpl5*, we propose to store the function value and gradient for several voxels in the backing store of each pixel processor. The processor then performs classification and shading calculations for all voxels in its backing store. The time to apply a monochrome Phong shading model at a single voxel using a pixel processor is about 1 msec. For a 256 x 256 x 256 voxel dataset, each pixel processor would be assigned 64 voxels, so the time required to classify and shade the entire dataset would be about 64 msec.

GPs perform the ray-tracing to generate the image. They are each assigned a set of rays and request sets of voxels from the pixel processors as necessary. The GPs perform the tri-linear interpolation and compositing operations, then transmit the resulting pixel colors to the frame buffer for display.

The success of this approach depends on reducing the number of voxels flowing from the pixel processors to the GPs. Three strategies are planned: First, a hierarchical enumeration of the volumetric dataset [Levoy 88b] will be installed in each graphics processor. This data structure encodes the coherence present in the dataset, telling the graphics processor which voxels are interesting (non-transparent) and hence worth requesting from the pixel

processors. Second, the adaptive sampling scheme described in [Levoy 89] will be used to reduce the number of rays required to generate an initial image. Last, all voxels received by a graphics processor will be retained in a local cache. If the observer does not move during generation of the initial image, the cached voxels will be used to drive successive refinement of the image. If the observer moves, many of the voxels required to generate the next frame may already reside in the cache, depending on how far the observer moves between frames.

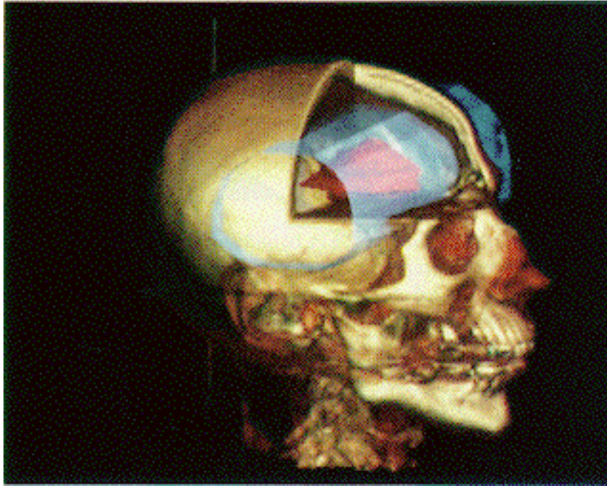

Figure 9. Volume-rendered head from CT data, generated on a Sun 4. Estimated rendering time on *Pxpl5* is 1 second. Photo courtesy of Marc Levoy.

The frame rate we expect from this system depends on which parameters change from frame to frame. Preliminary estimates suggest that for changes in observer position alone, we will be able to generate a sequence of slightly coarse images at 10 frames per second and a sequence of images of the quality of Figure 9 at 1 frame per second. For changes in shading or changes in classification that do not invalidate the hierarchical enumeration, we expect to obtain about 20 coarse or 2 high-quality images per second. This includes highlighting and interactively moving a region of interest, which we plan to implement by heightening the opacity of voxels inside the region and attenuating the opacities of voxels outside the region. If the user changes the classification mapping so that the set of interesting voxels is altered, the hierarchical enumeration must be recomputed. We expect this operation to take several seconds.

7.8 Rendering CSG-defined Objects

We and others have developed algorithms to directly render Constructive Solid Geometry (CSG) defined objects on graphics systems with deep frame buffers [Goldfeather 86a, Jansen 86, Rossignac 86]. On *Pxpl4* we developed a CSG modeler that displays small datasets at interactive rates [Goldfeather 88].

Pxpl5 provides several opportunities to increase rendering speed: the QEE on *Pxpl5* renders curved-surfaced primitives without breaking them into polygonal facets; having more bits per pixel allows surfaces that are used multiple times to be stored and re-used, rather than being re-rendered, greatly increasing performance; finally, the screen-subdivision technique advocated in [Jansen 87] provides a way to take advantage of *Pxpl5*'s multiple renderers. *Pxpl4* interactively renders CSG objects with dozens of primitives (Figure 10). We expect *Pxpl5* to interactively render objects with hundreds of primitives.

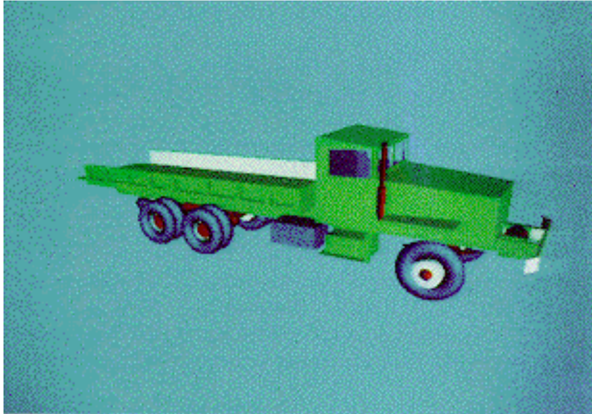

Figure 10. CSG-modeled truck generated on *Pxpl4*. Estimated rendering time on *Pxpl5* is 40 milliseconds.

7.9 Transparency

A number of methods for rendering transparent surfaces are possible, given the generality and power of *Pxpl5*. The most promising is to enhance the bin sorting in each GP to generate twice as many bins, one for transparent and another for opaque primitives for each region. The transparent primitives are rendered after all the opaque ones. Since we expect relatively few transparent polygons, each of the "transparent" bins can be sorted from back to front and rendered by simple composition. For difficult cases, in which a cluster of transparent polygons cannot be sorted in Z (as in a basket-weave of transparent strips), multiple Z values can be stored at each pixel to control the compositing step. With this approach, difficult primitives may need to be sent to renderers several times to ensure correct blending.

A second method, stochastically sampling the pixels of transparent primitives, is currently being used on *Pxpl4*. It is very simple and efficient, but requires several anti-aliasing passes to produce an acceptable image (without antialiasing, primitives appear splotchy). With *Pxpl5*'s increased speed this method may perform so well that more complicated algorithms are not needed.

8. Conclusions

It is too early to conclude much about the potential usefulness of Pixel-planes 5. We hope that with its generality and simple conceptual structure, it will prove useful for our local colleagues' activities in highly interactive 3D

graphics. We are convinced that even experimental machines like this one should be built for a community of users who can dispassionately evaluate their utility. The heavy local use of its predecessor, *Pxpl4*, has contributed substantially to the ideas in *Pxpl5*.

With the rapid development of high-performance graphics engines in the past few years, it is difficult to determine which of the many approaches will continue to be useful in the future. Among the safest predictions is that scan-line ordered pipeline processing will continue to be a cost-effective solution to rendering specific sets of primitives and that parallel screen subdivision will continue to be useful for general purpose image generation. Within this latter approach, we speculate that there may be a convergence between current parallel solutions (4x4-pixel footprint of the Stellar GS-1000, the 4x5 footprint of SGI, the 8x8 footprint of Pixel Machine) and the 128x128-pixel footprint of *Pxpl5*. Once the size of the footprint becomes large enough so most primitives fall into only a single region, the rendering can be done independently for each region with little penalty for duplication of primitives among the multiple regions. With VLSI and ULSI technology, it will be increasingly practical to have such footprints that are sufficiently large to simplify the processing in this way.

Current status of Pxpl5. As of January 10, hardware and software are being built. Of the three custom CMOS VLSI chips being designed, the processor-enhanced memory and the backing memory interface are both in final simulations, projected to be sent to fabrication in the next few weeks. The third chip, the renderer controller, is in middle of layout. Detailed simulation of the board-level logic design is well along, and PCBs are being designed. A small version of the Ring Network with a pair of Graphics Processors is expected to become operational in March, with a small complete system running in July. On the software front, a high-level language porting base is running simple code. Renderer simulator is yielding useful images.

9. Acknowledgements

We wish to thank: our colleagues on the Pixel-planes team, Michael Bajura, Andrew Bell, Howard Good, Chip Hill, Victoria Interrante, Jonathan Leech, Marc Levoy, Ulrich Neumann, John Rhoades, Herve Tardif, and Russ Tuck for many months of dedicated creative work; Vernon Chi for the design of a novel clock distribution scheme for the system; J. William Poduska and Andries van Dam for valuable criticism and advice on architecture design; Douglass Turner for the texture rendering program; John Rohlf for implementing textures on *Pxpl4* and computing the radiosity office image; Randy Brown, Penny Rheingans, and Dana Smith for the office model; UNC Department of Radiation Oncology for volumetric data set; US Army Ballistic Research Laboratory for CSG truck data; John Thomas and Brad Bennett for laboratory support; Sharon Walters for engineering assistance.

10. References

[Akeley 88] Akeley, K. and T. Jermoluk, "High-Performance Polygon Rendering," *Computer Graphics*, 22(4), (Proceedings of SIGGRAPH '88), pp 239-246.

[Apgar 88] Apgar, B., B. Bersack, A. Mammen, "A Display System for the Stellar Graphics Supercomputer Model GS1000," *Computer Graphics*, 22(4), (Proceedings of SIGGRAPH '88), pp 255-262.

[Bishop 86] *Computer Graphics,* 20(4) (Proceedings of SIGGRAPH '86), pp. 103-106.

[Clark 80] Clark, J. and M. Hannah, "Distributed Processing in a High-Performance Smart Image Memory," *LAMBDA (VLSI Design)*, Q4, 1980, pp 40-45.

[Clark 82] Clark, J. July, 1982. "The Geometry Engine: A VLSI Geometry System for Graphics," *Computer Graphics*, 16(3), (Proceedings of SIGGRAPH '82), pp 127-133.

[Cohen 88] Cohen, Michael F., Shenchang Eric Chen, John R. Wallace, and Donald P. Greenberg, "A Progressive Refinement Approach to Fast Radiostiy Image Generation," *Computer Graphics* 22(4) (Proceedings of SIGGRAPH '88), pp. 75-84.

[Crow 84] Crow, F., "Summed-Area Tables for Texture Mapping," *Computer Graphics* 18(4) (Proceedings of SIGGRAPH '84), pp. 207-212.

[Deering 88] Deering, M., S. Winner, B. Schediwy, C. Duffy, N. Hunt, "The Triangle Processor and Normal Vector Shader: A VLSI System for High Performance Graphics," *Computer Graphics*, 22(4), (Proceedings of SIGGRAPH '88), pp 21-30.

[Demetrescu 85] Demetrescu, S., "High Speed Image Rasterization Using Scan Line Access Memories," *Proceedings of the 1985 Chapel Hill Conference on VLSI*, Rockville, MD, Computer Science Press, pp 221-243.

[Diede 88] Diede, T., C. Hagenmaier, G. Miranker, J. Rubenstein, W. Worley, "The Titan Graphics Supercomputer Architecture," *Computer*, 21(9), pp 13-30.

[Ellsworth 89] Ellsworth, D., "Pixel-planes 5 Rendering Control," University of North Carolina Department of Computer Science Technical Report TR89-003.

[Eyles 88] Eyles, J., J. Austin, H. Fuchs, T. Greer, J. Poulton, "Pixel-planes 4: A Summary," *Advances in Computer Graphics Hardware II*, Eurographics Seminars, 1988, pp 183-208.

[Fuchs 77] Fuchs, H., "Distributing a Visible Surface Algorithm over Multiple Processors," *Proceedings of the ACM Annual Conference*, 449-451.

[Fuchs 79] Fuchs, H., B. Johnson, "An Expandable Multiprocessor Architecture for Video Graphics," *Proceedings of the 6th ACM-IEEE Symposium on Computer Architecture*, April, 1979, pp 58-67.

[Fuchs 81] Fuchs, H. and J. Poulton. 3rd Quarter, 1981. "Pixel-planes: A VLSI-Oriented Design for a Raster Graphics Engine," *VLSI Design*, 2(3), pp 20-28.

[Fuchs 82] Fuchs, H., J. Poulton, A. Paeth, and A. Bell. January, 1982. "Developing Pixel Planes, A Smart Memory-Based Raster Graphics System," *Proceedings of the 1982 MIT Conference on Advanced Research in VLSI*, Dedham, MA, Artech House, pp 137-146.

[Fuchs 85] Fuchs, H., J. GoldFeather, J.P. Hultquist, S. Spach, J. Austin, F.P. Brooks, Jr., J. Eyles, and J. Poulton, "Fast Spheres, Textures, Transparencies, and Image Enhancements in Pixel-Planes," *Computer Graphics*, 19(3), (Proceedings of SIGGRAPH '85), pp. 111-120.

[Gardner 88] Gardner, G., "Functional Modeling of Natural Scenes, Functional Based Modeling," *SIGGRAPH Course Notes*, vol. 28, 1988, pp. 44-76.

[Gharachorloo 85] Gharachorloo, N., and C. Pottle, "SUPER BUFFER: A Systolic VLSI Graphics Engine for Real Time Raster Image Generation," *Proceedings of the 1985 Chapel Hill Conference on VLSI*, Rockville, MD, Computer Science Press, pp 285-305.

[Gharachorloo 88] Gharachorloo, N., S. Gupta, E. Hokenek, P. Balasubramanian, B. Bogholtz, C. Mathieu, C. Zoulas, "Subnanosecond Pixel Rendering with Million Transistor Chips, " *Computer Graphics*, 22(4), (Proceedings of SIGGRAPH '88), pp 41-49.

[Goldfeather 86a] Goldfeather, J., Jeff P.M. Hultquist, and Henry Fuchs, "Fast Constructive Solid Geometry Display in the Pixel-Powers Graphics System", *Computer Graphics*, 20(4), (Proceedings of SIGGRAPH '86), pp. 107-116.

[Goldfeather 86b] Goldfeather, J. and H. Fuchs, "Quadratic Surface Rendering on a Logic-Enhanced Frame-Buffer Memory System," *IEEE Computer Graphics and Applications*, 6(1), pp 48-59.

[Goldfeather 88] Goldfeather, J., S. Molnar, G. Turk, and H. Fuchs, "Near Real-Time CSG Rendering using Tree Normalization and Geometric Pruning," University of North Carolina Department of Computer Science Technical Report TR88-006. To appear in CG&A, 1989.

[Goldfeather 89] Goldfeather, J., "Progressive Radiosity Using Hemispheres," University of North Carolina Department of Computer Science Technical Report TR89-002.

[Immel 86] Immel, D., M. Cohen, and D. Greenberg, "A Radiosity Method for Non-Diffuse Environments," *Computer Graphics*, 20(4), (Proceedings of SIGGRAPH '86), pp. 133-142.

[Jansen 86] Jansen, F., "A Pixel-Parallel Hidden Surface Algorithm for Constructive Solid Geometry," *Proceedings of Eurographics '86*, Elsevier Science Publishers B.V. (North-Holland): Amsterdam, New York, pp 29-40.

[Jansen 87] Jansen, F. and R. Sutherland, "Display of Solid Models with a Multi-processor System," *Proceedings of Eurographics '87*, Elseviers Science Publications, 1987, pp 377-387.

[Levoy 88a] Levoy, M., "Display of Surfaces from Volume Data," *IEEE Computer Graphics and Applications*, 8(3), May 1988, pp 29-37.

[Levoy 88b] Levoy, M., "Efficient Ray Tracing of Volume Data," University of North Carolina Department of Computer Science Technical Report TR88-029. (In review, *ACM Transactions on Graphics*).

[Levoy 89] Levoy, M., "Volume Rendering by Adaptive Refinement," *The Visual Computer*, 5(3), June, 1989 (to appear).

[Norton 82] Norton, A., "Clamping: A Method of Antialiasing Textured Surfaces by Bandwidth Limiting in Object Space," *Computer Graphics*, 16(3), (Proceedings of SIGGRAPH '82), pp 1-8.

[Pavlidis 83] Pavlidis, T., "Curve Fitting with Conic Splines," *ACM Transactions on Graphics*, 2(1), January 1983.

[Perlin 85] Perlin, K., "An Image Synthesizer," *Computer Graphics*, 19(3), (Proceedings of SIGGRAPH '85), pp. 151-159.

[Phong 73] Phong, B.T., "Illumination for Computer-Generated Picturres," Ph.D. Dissertation, University of Utah, Salt Lake City, 1973.

[Poulton 85] Poulton, J., H. Fuchs, J.D. Austin, J.G. Eyles, J. Heinecke, C-H Hsieh, J. Goldfeather, J.P. Hultquist, and S. Spach. 1985. "PIXEL-PLANES: Building a VLSI-Based Graphic System," *Proceedings of the 1985 Chapel Hill Conference on VLS*I, Rockville, MD, Computer Science Press, pp 35-60.

[Poulton 87] Poulton, J., H. Fuchs, J. Austin, J. Eyles, T. Greer. "Building a 512x512 Pixel-planes System," *Proceedings of the 1987 Stanford Conference on Advanced Research in VLSI*, MIT Press, pp 57-71.

[Pratt 85] Pratt, V., "Techniques for Conic Splines," *Computer Graphics*, 19(3), (Proceedings of SIGGRAPH '85), pp. 151-159.

[Rossignac 86] Rossignac, J., A. Requicha, "Depth Buffering Display Techniques for Constructive Solid Geometry," *IEEE Computer Graphics and Applications*, 6(9), pp 29-39.

[Runyon 87] Runyon, S., "AT&T Goes to 'Warp Speed' with its Graphics Engine," *Electronics Magazine*, July 23, 1987, pp 54-56.

[Swanson 86] Swanson, R., L. Thayer, "A Fast Shaded-Polygon Renderer," *Computer Graphics*, 20(4), (Proceedings of SIGGRAPH '86), pp 95-101.

[Tor 84] Tor, S. and A. Middleditch, "Convex Decomposition of Simple Polygons," *ACM Transactions on Graphics*, 3(4), October 1984, pp 244-265.

[Torberg 87] Torberg, J., "A Parallel Processor Architecture for Graphics Arithmetic Operations," *Computer Graphics*, 21(4), (Proceedings of SIGGRAPH '87), pp 197-204.

[van Dam 88] van Dam, A., Chairman, PHIGS+ Committee, "PHIGS+ Functional Description, Revision 3.0," *Computer Graphics,* 22(3), July, 1988, pp 125-218.

[Wallace 87] Wallace, J., M. Cohen, and D. Greenberg, "A Two-Pass Solution to the Rendering Equations: A Synthesis of Ray-Tracing and Radisoity Methods," *Computer Graphcis*, 21(4) (Proceedings of SIGGRAPH '87), pp. 311-320.

[Watkins 70] Watkins, G., "A Real-Time Visible Surface Algorithm, " University of Utah Computer Science Department, UTEC-CSc-70-101, June 1970, NTIS AD-762 004.

[Whitton 84] Whitton, M., "Memory Design for Raster Graphics Displays," *IEEE Computer Graphics and Applications*, 4(3), March 1984, pp 48-65.

[Williams 83] Williams L., "Pyramidal Parametrics," *Computer Graphics* 17(3) (Proceedings of SIGGRAPH '83), pp. 1-11.