# Harnessing Parallelism for High-Performance Interactive Computer Graphics

Anselmo Lastra
Henry Fuchs
John Poulton
University of North Carolina
Chapel Hill, NC 27599
*{lastra|fuchs|poulton@cs.unc.edu}*

**Abstract**

This paper provides a summary of the Pixel-Planes project from its inception in 1980 to the present (1996). The goals of the project have been to advance the state of the art in interactive computer graphics and visualization. During those sixteen years, the Pixel-Planes group at the University of North Carolina have built several prototype systems, including two full-scale machines, and are in the process of building a third. These machines have exhibited record-breaking performance and have enabled experiments in many areas of computer graphics. Not only have the team's research publications influenced many of the commercial machines currently on the market but some of the Pixel-Planes integrated circuits have been directly used on three commercial image generators.

## Introduction

This paper provides a summary of the Pixel-Planes project from its inception in 1980 to the present (1996). The project began in the Fall of 1980 during a VLSI design class taught by Henry Fuchs at the University of North Carolina (UNC). Fuchs had been thinking about ways to implement a distributed frame buffer for high-performance interactive graphics and the project-oriented class, based on the text by Mead and Conway [Mead80], presented a wonderful opportunity to try out some of his ideas. One of the students, John Poulton, took on the task of implementing the first Pixel-Planes chip.

Since then, team members have designed six generations of pixel-processing chips and two full-scale prototype machines with a third in construction. These machines have exhibited record-breaking performance and have enabled experiments in many areas of computer graphics. The technology has been licensed to three companies that have produced several products using the chips designed at UNC. Several generations of students have trained in the design of systems for high-performance graphics while working on the Pixel-Planes team.

## Interactive Graphics Pipeline

To introduce the problem of building high-performance graphics engines, we first need to examine the type and quantity of computation required. Figure 1 illustrates the traditional graphics pipeline. Primitives (typically triangles or other polygons) enter the pipeline in a user-centered, object coordinate system [Foley90]. The pipeline is conceptually (and often physically) divided into two stages, one for the geometry processing, the other for rasterization.

A reason for this partitioning is that the work performed in the two stages is fundamentally different. In the geometry processing stage, the vertices of the primitives are transformed to a screen coordinate system, vertices are lit and transformed to projection coordinates, primitives are clipped to fit within the viewing frustum, perspectively transformed, and set up for the rasterization stage. This part of the work is floating-point intensive and produces polygon vertices in screen coordinates (integers) along with shading information.

The rasterization stage is responsible for scan conversion and for determining visibility, typically a pixel by pixel process. Note that computing visibility is essentially a sorting problem, sorting by depth to find the pixel or polygon closest to the viewpoint. After visibility has been determined, the color is computed. While early graphics engines just assigned a constant color to each polygon, the common method today is Gouraud shading, a linear

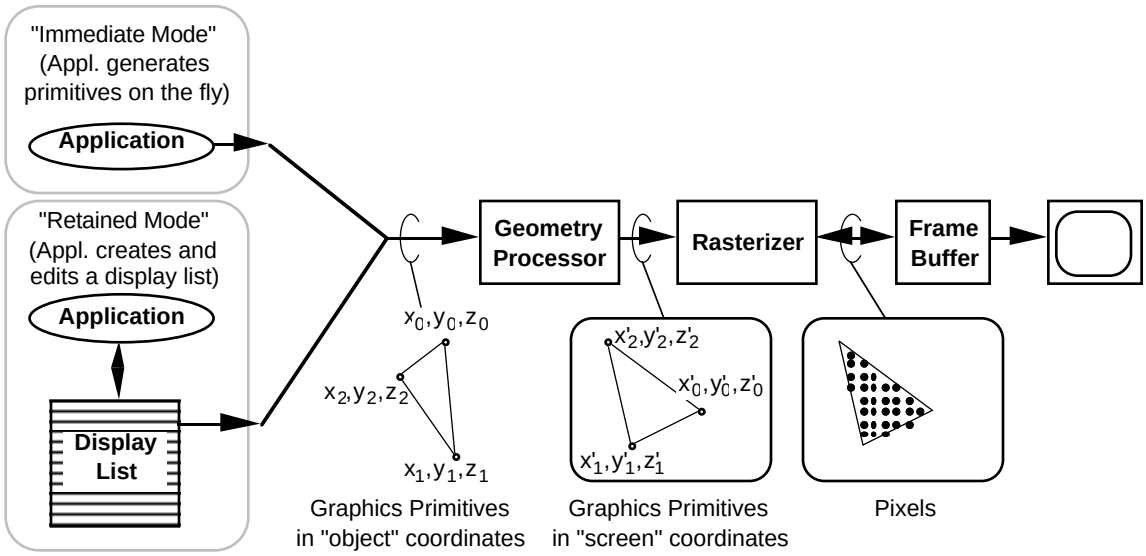



Figure 1. The Graphics Pipeline.

interpolation between the color at the vertices. More complex schemes, such as Phong shading [Foley90], used on Pixel-Planes 5, require more computation. The majority of the work done in the rasterization stage is fixed-point integer computation, but the most difficult problem is the memory bandwidth required, since access to a different location in the frame buffer is necessary for practically every operation.

### Why Is Interactive Graphics Difficult?

Molnar and Fuchs [Molnar90] estimate that a total of approximately 340 million floating point operations are required to process one million vertices (roughly equivalent to one million triangles) within the geometric stage of the graphics pipeline. Our experience on Pixel-Planes 5 is similar, although the work performed was not exactly equivalent (we moved lighting to the rasterization stage but our setup calculations are more involved). We could process 50,000 full triangles per second on a processor (40MHz Intel i860) with peak floating point performance of 80 megaflops, equaling roughly 533 megaflops for a million vertices. A third data point is the new Infinite Reality from Silicon Graphics. They report performance of 480 megaflops from each geometry processor and a peak rendering rate of ten million triangles for a machine with eight geometry processors for a figure of 384 megaflops for a million vertices.

As we said earlier, the bandwidth to the frame buffer is the most difficult problem for rasterization. Molnar and Fuchs estimate a total requirement of approximately 250 million accesses to memory in order to rasterize a million 100 pixel polygons. A basic problem with increasing memory bandwidth performance is that it takes more memory parts. While memory capacities have increased dramatically by several orders of magnitude over the past fifteen years, the number of pins per memory chip and the bandwidths per chip have not kept pace. This requires us to somehow replicate memory.

Initial proposals and prototypes to provide the necessary bandwidth focused on distributing frame buffers in one of two ways, *interleaved* [Fuchs77, Fuchs79] where adjacent pixels are distributed to different processors in a checkerboard pattern, and *partitioned* [Parke80] where the display is divided into large contiguous blocks and distributed to multiple processors. Memory (and processor) interleaving depend on polygons large enough to distribute the work fairly evenly among processors. Partitioning schemes typically have to work on multiple primitives at a time. The Pixel-Planes project grew out of a radical approach to the problem – couple pixel-processors with memory and provide a processor for every pixel.

### Pixel-Planes

One can think of a "processor per pixel" as adding local memory to all of these pixel processors, but perhaps it's more proper to think of it as adding processing power to the frame buffer (creating processor-enhanced or "smart" memories). However, Pixel-Planes is not simply a processor-per-pixel system which blindly replicates much of the scan conversion work at each processor.

A major contribution of Pixel-Planes was a way to perform as much global computation as possible for all of the processors at once. The key concept was the realization that a simple plane equation of the form $F(x, y) = Ax + By + C$ (where x and y are the screen coordinates of each pixel) is an excellent formulation for much of the work performed during rasterization. The plane equation is first used to enable pixel processors on the correct side of each polygon edge. It is then used to interpolate depth z at each pixel in order to determine visibility, and finally it is used to linearly interpolate color.

Besides a one-bit ALU, the Pixel-Planes logic-enhanced memory chips include a global computational tree [Fuchs81] to compute the result of the plane equations for each pixel. The Pixel-Planes 4 version of the enhanced-memory chip (EMC) contained 72 bits of memory for each of 128 pixels on a chip. Subsequent versions have contained more memory. The Pixel-Planes 5 [Fuchs89] EMC provides 208 bits for each of 256 pixels. The PixelFlow [Molnar92] EMC has an eight-bit ALU, 2048 bits of general-purpose memory, 1024 bits of communications registers, and a built-in compositor for each of 256 pixels per chip [Poulton92]. Figure 2 shows a block diagram of a PixelFlow EMC.

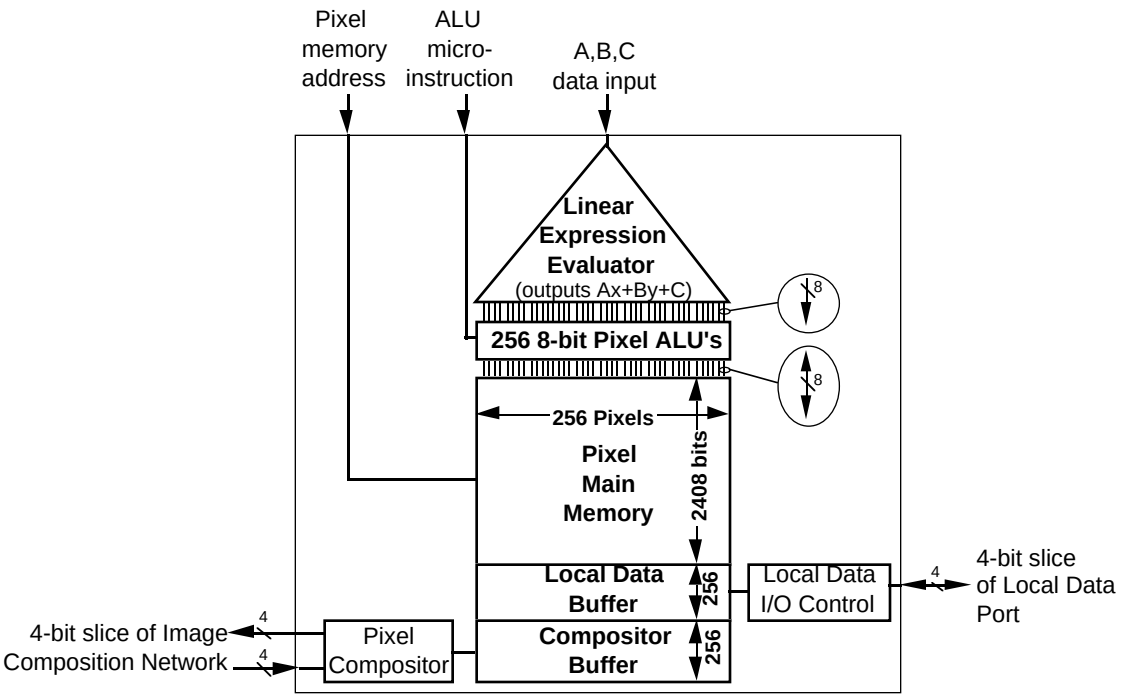



Figure 2 - PixelFlow Enhanced-Memory Chip.

**Pixel-Planes 1, 2, and 3**

Pixel-Planes 1 was the original chip, an outgrowth of the Fall 1980 VLSI design course. It contained a 2 by 2 grid of pixel memory and a branch of the computational tree. This work is documented in [Fuchs81]. In collaboration with Alan Paeth and Alan Bell of Xerox PARC, a new generation of chips, Pixel-Planes 2, was developed [Fuchs82a]. Each of these chips contained 64 pixels with 16 bits per pixel of memory. These were used at UNC to build a 4 x 64 pixel prototype. This prototype was used to verify that a basic set of rendering operations could be executed on this architecture.

In collaboration with Paeth, a third set of chips, Pixel-Planes 3, was built. These chips included a more complex ALU, and 32 bits per pixel. There were still 64 pixels per chip but the video scan-out was integrated into the chips. These were assembled in 1983 to make a 64 x 64 pixel prototype.

**Pixel-Planes 4**

In 1984, new pixel-memory chips were designed, along with a chip that functioned as a program controller. These were assembled into a small system, Pixel-Planes 4.1 [Poulton85], which was first demonstrated at the 1985 VLSI Conference in Chapel Hill. The group's research results were presented at SIGGRAPH '85, but there was general skepticism among the attendees that the full-size performance predicted during the presentation could be achieved on an actual implementation. This is the typical problem with a so-called *paper design*. It's very difficult to tell whether a fatal flaw exists in a design that appears plausible.

The skepticism of the audience at SIGGRAPH prompted the project members to speed up the design and construction of a full-sized, working, and generally usable system. This effort required much more rigor than the construction of small prototypes. The complete machine had to include a host computer, a base of system software, and applications. This machine, Pixel-Planes 4.2 but usually referred to simply as Pixel-Planes 4, was completed just in time for SIGGRAPH '86, where it was demonstrated. It served as the workhorse for research in the University of North Carolina graphics laboratory until it was retired in 1992. The full Pixel-Planes 4 system is described by Eyles [Eyles87].

Pixel-Planes 4 implemented a full-size 512 by 512 pixel frame buffer using 2048 enhanced-

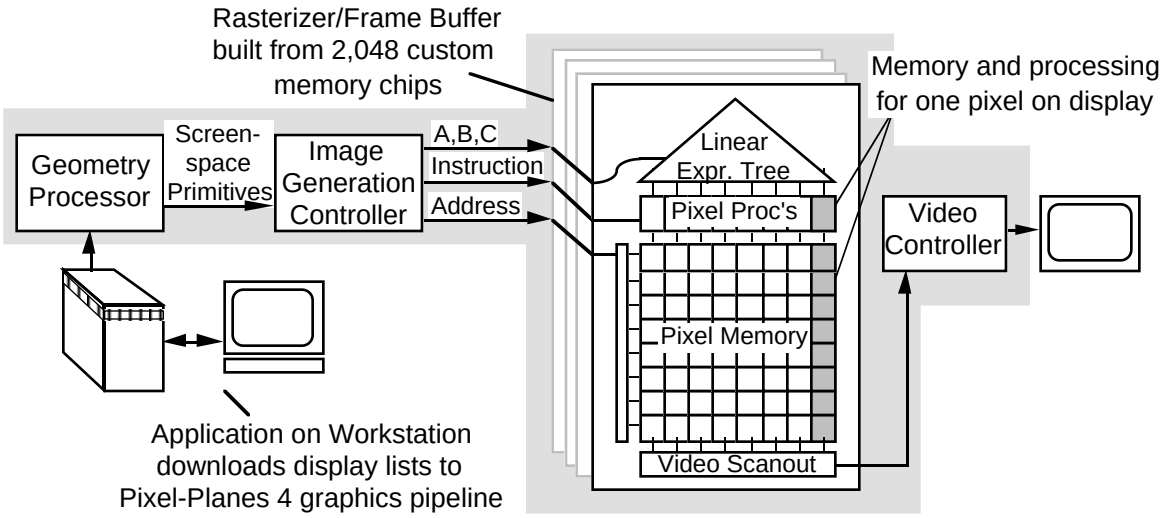



Figure 3. Block diagram of the Pixel-Planes 4 system.

memory integrated circuits, running at 10 MHz. Each chip implemented a 128-pixel column of the display with 72 bits per pixel and included video scan-out circuitry. The frame buffer was housed on thirty two boards on a common backplane. Geometry processing was provided by a processor based on a Weitek 8032 chip set. The system was hosted by DEC MicroVAX workstation. A PHIGS-like graphics library was developed for programming applications on the host workstation.

System performance was trailblazing for its time at 35,000 polygons per second. Furthermore, many novel algorithms were demonstrated on Pixel-Planes 4, including shadow casting, fast rendering of spheres [Fuchs 85], antialiasing by progressive refinement [Bergman86], rendering of quadric surfaces [Goldfeather86b] and direct rendering of constructive solid geometry (CSG) objects [Goldfeather86a, Goldfeather89a].

**Pixel-Planes 5**

Many valuable lessons were learned from the design and implementation of a full-sized Pixel-Planes 4 prototype. A major lesson was that since the size of individual polygons is rarely close to the size of the frame buffer, many pixel processors were idle during the rendering of most polygons. This fact in itself is not a particular problem, but the realization that a set of smaller frame buffers could be used presented an opportunity to increase parallelism to a new level, parallelism by object. Performance could be increased dramatically if multiple polygons were being processed at one time.

Clearly, in order to achieve maximum performance, we want to have multiple geometry processors and rasterization processors. But how can we arrange them to work together? To examine the space of solutions, first let's consider that there's inherently a sorting step in a system with fully-parallel components. Recall that polygons are presented to the graphics pipeline represented in object coordinates but must eventually be placed at the proper location on the screen. That is, polygons enter the geometry processing step in some order that's independent of the final placement on the display. Thus there must be a *sorting* step somewhere in the pipeline to get the information to whichever processors are working on the correct pixels. One can think of the parallel-graphics architecture space as one of ways to perform that inherent sort. This way of establishing a taxonomy is one of the contributions of the Pixel-Planes project [Molnar94]. Pixel-Planes 5 uses a strategy that we now refer to as *sort-middle*. A conceptual diagram is shown in figure 4.

The actual implementation is quite a bit more flexible than that shown on the conceptual diagram. As illustrated on figure 5, the various component parts of Pixel-Planes 5 are connected

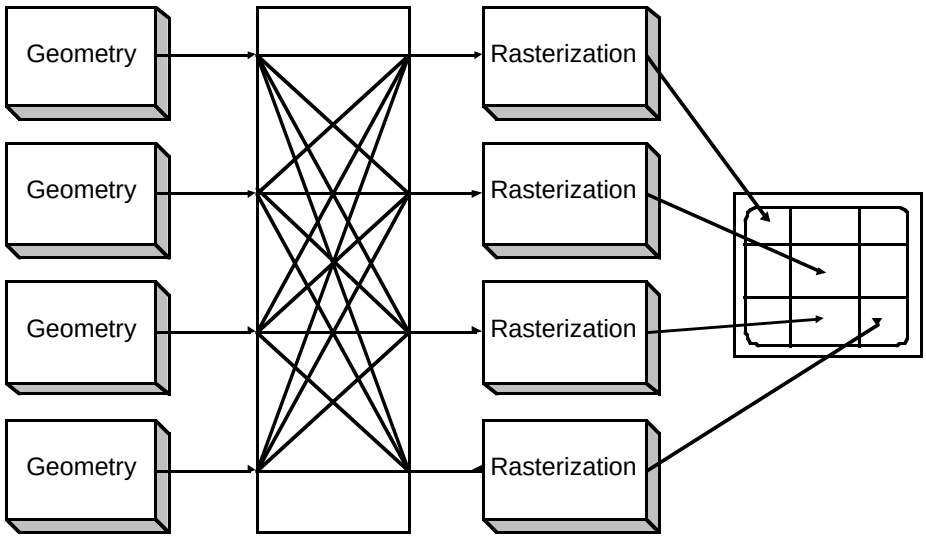



Figure 4 - Sorting on Pixel-Planes 5. The geometry processors bucket sort the primitives which are then sent over the network to the renderer that corresponds to the appropriate region(s) of the display screen. Note that a renderer may be assigned more than one region during a frame, but works solely on one region until all of the primitives for that region have been rendered.

by a ring network. The components consist of geometry processors, rasterization processors (called renderers), various types of frame buffers, frame grabbers for video input, a high-speed HiPPI interface providing a data connection to a Cray supercomputer, and an interface to connect to the Sun 4 workstation which serves as the host processor (providing disks, networking, program loading, etc.).

The ring network is a general-purpose message passing system. It is 32 bits wide and runs at 160 MHz, with eight time slots allowing a total of eight simultaneous messages. All data flow over this network, including polygon data, commands to the rasterization processors, final pixel color destined for the frame buffers, and control messages. A layer of software interfaces to the network, allowing programmers to send messages easily and efficiently.

The rasterization on Pixel-Planes 5 is performed by a set of small Pixel-Planes-based engines called renderers. Each renderer is housed on one board and contains a 128 by 128 enhanced-memory array executing at 40MHz. Each pixel of the array has 208 bits of local, on-chip memory and a backing store of 4096 bits of memory per pixel implemented using commercial random-access memory chips. A renderer has an instruction sequencer to run the pixel array and communications ports to interface to the ring. There are two sets of communications ports, one for instructions to the pixel processors and the other, connected to the backing store, for the pixel data. Renderers are not assigned to a fixed location on the screen, but are re-mapped dynamically as described below. A system can have any number of renderers from a minimum of one to a maximum of as many as will fit within the addressing space of the ring network. On a large system we usually use twenty renderers.

Geometry processing is performed on nodes containing an Intel i860XR processor running at 40 MHz. Each geometry processor has 8 Mbytes of local memory and communications ports to the ring network. There are two geometry processors per board. A Pixel-Planes 5 system needs to contain at least two geometry processors (one serves not as a geometry processor, but as a master controller – see below) but may have as

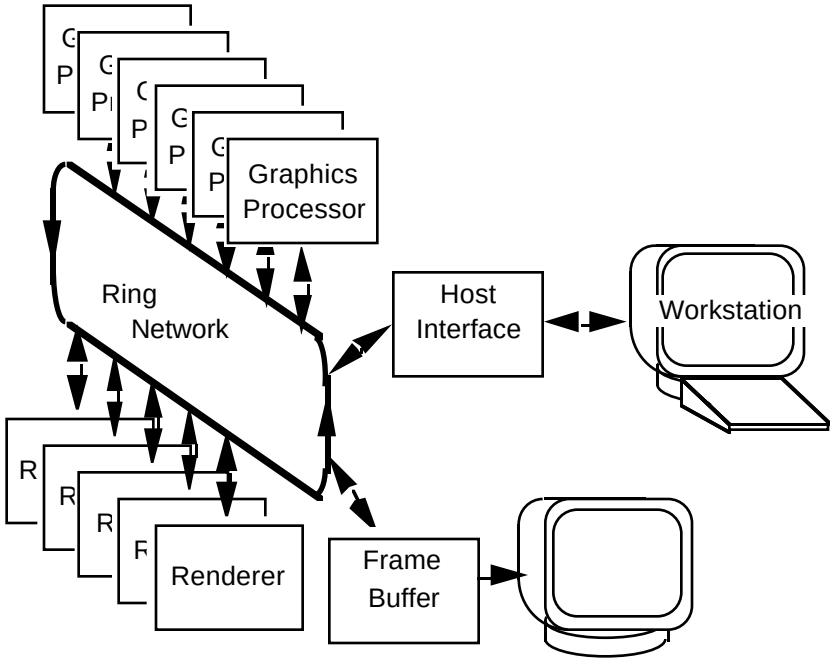



Figure 5 - Block diagram of Pixel-Planes 5.

many as will fit. We usually configure systems with twice as many geometry processors as renderers (it's been our experience that this roughly balances the geometry/rasterization load for most data sets). The largest systems we've used contain up to 50 geometry processors.

Polygons are initially distributed in either a round-robin or a random order among the geometry processors. The geometry processors not only perform the normal transformation, clipping, etc., but also bucket sort the primitives by region of the screen to determine which renderers need to receive rasterization instructions for each primitive. If a primitive overlaps multiple regions, the rendering instructions are replicated into the appropriate buckets.

Renderers may be though of as virtual frame buffers [Gharachorloo 89] (although the actual scan-out on Pixel-Planes 5 is performed on auxiliary frame buffer boards) which are mapped to a region of the screen until all of the polygons falling on that region of the screen are processed, at which time the final color for the whole region is sent to the frame buffer (via backing store and the ring network). The renderer is then assigned to another screen region. Region assignment is performed by first collecting, from all of the geometry processors, information about polygon distribution. This is done by a master processor which creates a work queue and assigns regions to renderers by a heuristic which begins with the most complex screen regions and proceeds to the simplest (an optimal assignment is a NP-complete problem).

This heuristic assignment algorithm avoids processing complex screen regions at the end, which may tend to imbalance the renderer loads and delay processing of a frame. Since some renderers may be idle at the end of a frame, rasterization for the next frame is begun before the current frame is complete (this works well for stereo pairs, for example). Actual control is performed by sending tokens (messages) to the geometry processors instructing them to send rendering data for the appropriate region to the appropriate processor. It was a surprise to us that this token distribution consumed a large quantity of the processing time and had to be streamlined to obtain the expected performance. The region assignment and load balancing is discussed in more detail by Ellsworth [Ellsworth96].

Pixel-Planes 5 uses a retained-mode application programmer's interface very similar to PHIGS+ (it would have been practically identical except that the PHIGS+ specification was not finalized until after the Pixel-Planes 5 implementation), so the primitives are retained in the memory of the graphics processors. This divorces maximum performance from the bandwidth limits imposed by the host to graphics engine interface, but necessitated novel algorithms for display-list distribution and editing [Ellsworth90].

One of the more novel concepts used on Pixel-Planes 5 is known as *deferred shading* [Deering88, Tebbs92]. Instead of determining pixel colors during the initial scan conversion (this process is known as shading), all of the parameters necessary to determine the color (the *appearance parameters*) are computed and stored instead. Only when all primitives have been scan converted (all of the visible-surface determination is complete) is the shading computed. The advantage is that no shading is performed on pixels that are not visible in the final image. Deferred shading is not only useful when complex shading is being performed, but is also an excellent match for a SIMD architecture such as that of the Pixel-Planes enhanced memories. When computing deferred shading, essentially all of the pixel processors are working, achieving excellent utilization. Since shading is very efficient, we normally compute higher-quality Phong-shading instead of Gouraud shading commonly used in today's commercial systems.

This efficiency, coupled with the increased pixel memory (208 bits), also allowed us to experiment with procedural shading, computing pixel color using a programmable instead of a fixed algorithm [Gardner85, Perlin85, Gardner 88] . Although procedural shading is common in high-quality off-line rendering [Upstill90], it has, to our knowledge, never been done elsewhere at interactive rates. The color plate shows some of the textures generated procedurally, including bricks, ceiling panels, and a procedural painting. A simple, interpreted, assembler-level language was designed and implemented to support procedural shading. The main limitation to the complexity of the textures was the amount of pixel-level memory available. This work is described in detail by Rhoades, et. al. [Rhoades92].

The performance of Pixel-Planes 5 was unequaled in its day. In 1991, we demonstrated a rendering rate of over 2.3 million Phong-shaded triangles per second completely rendered to the display. In 1992, the machine ran the largest benchmark (from the Graphics Performance Characterization Committee of the National Computer Graphics Association, an industry group) over three times faster than the best results for commercial systems [Cramblitt 92]. In fact, only this year (1996) has a commercial system posted a better score on that benchmark.

The display of curved primitives, such as NURBS (non-uniform rational B-splines) is even more computationally demanding than that of polygonal primitives. Kumar [Kumar95] used the parallelism provided by the Pixel-Planes 5 geometry processors to tessellate the NURBS primitives for complex models at interactive rates. He took advantage of the natural frame-to-frame coherence of interactive rendering by cacheing tessellations over multiple frames since, for many of the primitives, the same tessellation was correct for many frames. Re-tessellation is necessary only when a primitive gets substantially closer or farther from the viewpoint.

The general-purpose nature of the architecture has proven to be very useful. We have built devices for special purposes, including dual (for stereo) NTSC frame buffers to drive head-mounted displays, dual frame-sequential color frame buffers for driving a custom head-mounted display built at UNC (using monochrome CRTs and a color shutter), frame grabbers for introducing live video (for experiments in augmented reality [Bajura92]), and a HiPPI interface for communication with a Cray supercomputer.

Since Pixel-Planes 5 functions quite well as a general-purpose parallel machine and, unlike most parallel machines, also contains frame buffers, we've been able to use it very successfully for volume rendering. This work is described in [Levoy89, 90a, 90b, Yoo91, Neumann92]. Andrew Bell also implemented a software system for the interactive computation of radiosity initially conceived by Goldfeather [Goldfeather 89b].

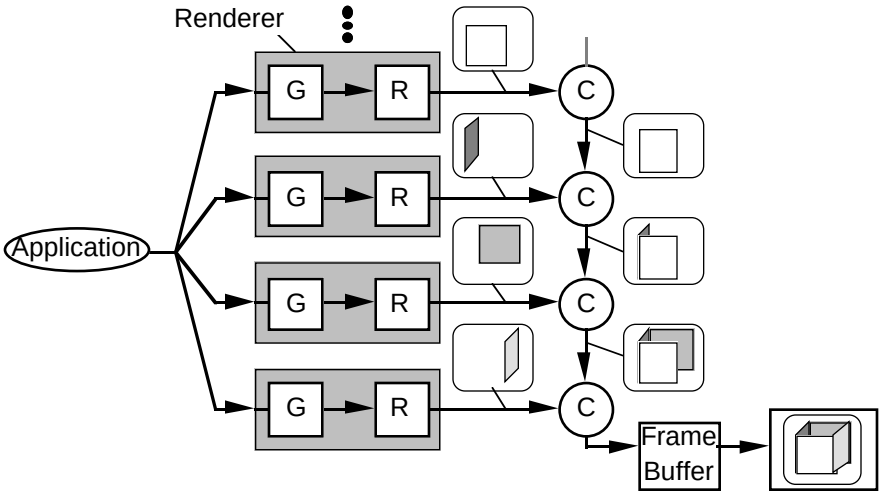



Figure 6 - An image composition architecture.

**Technology Transfer**

Two companies, IVEX Incorporated of Atlanta, Georgia, and Division Group PLC of Bristol, UK, have licensed Pixel-Planes 5 technology. Both companies are selling machines using the custom ICs designed for Pixel-Planes 5.

IVEX produces image generators for the visual simulation market. They have incorporated Pixel-Planes 5 rendering chips into their VDS-2000 scene generator [IVEX94]. They developed their software on a UNC Pixel-Planes 5 prototype. Their image generator features antialiased, textured polygonal imagery, enhanced key-point-based terrain generation, and special light-point hardware.

Division has produced two lines of Pixel-Planes based products. One, the VPX series, is based on small EISA boards containing Pixel-Planes 5 enhanced memories. They have sold the ProVision 100VPX, hosted on 486-based UNIX platform, and the ProVision 10VPX, hosted on Hewlett-Packard 9000 series workstations. Rendering performance of the VPX is 300,000 polygons per second. Division's other product is Pixel-Planes 6, an enhanced version of the Pixel-Planes 5 system. This system used faster, more densely packaged geometry processors, and included texture memory on the renderers. Rendering performance is claimed to be up to 4 million polygons per second. The first installation was at Matsushita Electric Works (Japan).

**Image Composition and PixelFlow**

While the sort-middle rendering architecture of Pixel-Planes 5 is efficient, there is a major problem if we try to increase performance. While we may add processors for geometry processing and renderers for rasterization, all of the primitives must still flow over the network connecting the two sets. Furthermore, that network must function as a full a crossbar – every geometry processor must be able to route primitives to every renderer. To increase performance, the capacity of the network must be proportionately increased, a daunting task.

Ideally, we would like a system to which we can add geometry and rasterization processors as necessary, and see a commensurate increase in performance. There is a space of architectures that promises to provide that flexibility: *image composition* architectures, described by Molnar [Molnar91]. Geometry processors and renderers in an image composition architecture (figure 6) function as complete, independent graphics machines that render complete, full-screen-resolution images. However, after the images are rendered, the pixels are routed through an image composition network. By sending the depth value for the pixel, as well as the color, we can compare the pixels from adjacent processors and send on the one that is visible (closest to the

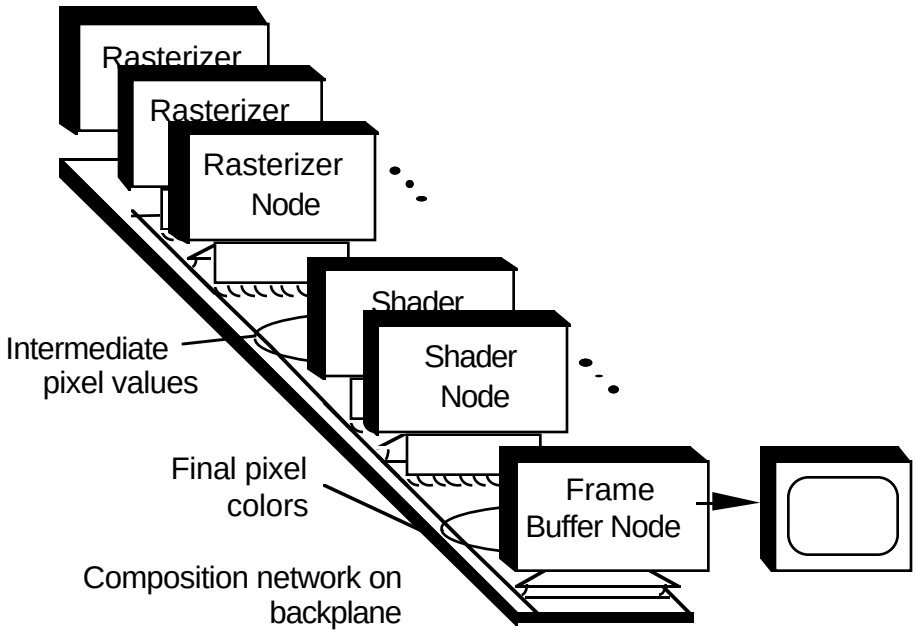



Figure 7 - PixelFlow block diagram.

viewpoint). Note that network capacity need not increase as we add processors. We're still sending only the number of pixels it takes to scan-out the display. The only cost is added latency for each processor we add, but only one pixel's worth of added latency, not a problem in practice.

The real cost of image composition machines is that the composition network bandwidth must be high enough to accommodate the maximum pixel traffic – the highest-resolution display to be used times the maximum frame rate. It turns out that it also needs to have enough bandwidth to allow for supersampling (rendering an image at a sub-pixel resolution which is later filtered to reduce aliasing), usually a factor of five to eight. However, once we've supplied a fast enough network, we can scale performance as desired.

To test this concept, we've been designing and building PixelFlow [Molnar92], a prototype image composition machine. PixelFlow is based on a passive backplane that functions as transport for the image composition network. The network uses bi-directional signaling [Lam90] to increase bandwidth. The actual image compositors are built into a new generation of enhanced memories. The network runs bi-directionally on 256 signal wires at 200 MHz, for a peak network bandwidth of over 100 gigabits per second, theoretically supporting a display size of over 4K by 4K at 72 Hz. However, the reason that the network was designed for this performance is to support deferred shading.

As shown in figure 7, three functional types of nodes are connected to the image composition network: rasterization nodes, shading nodes, and a frame buffer node. As on Pixel-Planes 5, primitives are first scan converted and the appearance parameters are generated. The depth value at each of the rasterization nodes is used to composite the appearance parameters. The parameters for the visible pixel are delivered to one of the shading nodes, which is responsible for shading the pixels.

The three types of nodes are actually identical hardware, except for the frame buffer which will usually contain an attached video card. The only other differences are in the software running on a particular node. A block diagram of a PixelFlow node is shown in figure 8. Geometry processing is performed on two Hewlett-Packard PA-RISC processors sharing 64 MB of memory. Rasterization is on a 128 by 64 pixel enhanced memory array. A controller (labeled EIGC) sequences the array. A second (physically identical) controller runs the texture application-specific integrated circuits (TASICs) which interface to texture memory [Molnar95]. Dual controllers are used to allow the pixel processors

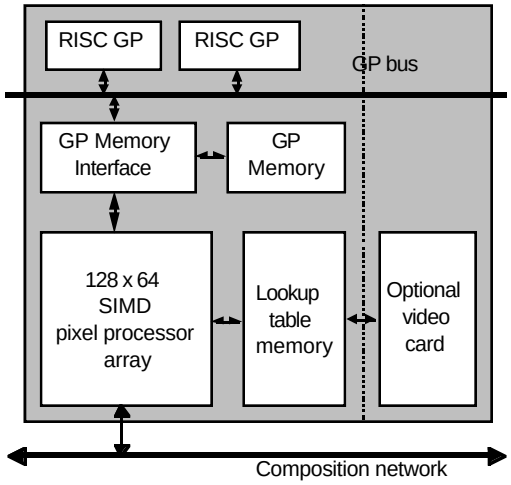



Figure 8 - Block diagram of a PixelFlow node.

to work while a texture lookup is occurring. We expect at least 64MB of texture memory.

The enhanced memories for PixelFlow contain 8-bit ALUs instead of the previous one-bit units, and have 2048 bits of general-purpose memory and 1024 bits of communications registers per pixel.

Each PixelFlow node also has an interface to a message-passing network to be used for general-purpose communications and control. There is also a connection for an I/O card. The card may contain simple video circuitry (the texture memory also serves as the frame buffer), mostly a register and a digital to analog converter. It may also contain an analog to digital converter for video input, or a general-purpose digital interface. The most common interface will be to a workstation, but interfaces to local-area networks, disks, and other peripherals may also be designed.

The application programmer's interface for PixelFlow will be a modified version of OpenGL [OpenGL92]. To enhance the programmable shading that proved so successful on Pixel-Planes 5, we are developing a compiler for a high-level shading language [Lastra95] based on the RenderMan shading language [Hanrahan90, Upstill90]. The color plate shows an image rendered on a PixelFlow simulator that illustrates several effects possible with programmable shading: shadows, reflections, bumps, labels, and image-based textures. We expect to be able to render images such as that at high frame rates.

At the time of this writing, three of the five custom integrated circuits for PixelFlow have been fabricated, one is in fabrication, and the final one is in the last stages of design. Of the three completed chips, two have been tested and one is awaiting a test fixture. Three companies have licensed the technology. The Hewlett-Packard Corporation has joined the Pixel-Planes 5 licensees, Division and IVEX.

**The Future**

What happens when the average size of a polygon falls to below the size of a pixel? Is it still worth using polygons as a representation? We believe that, at that point, it may be better to look for an alternative representation. A natural representation is a set of reference images. This type of rendering was pioneered by Chen and Williams [Chen93] and a simple version is being used in Apple's Quicktime VR [Chen95].

In the Plenoptic modeling [McMillan95] work at the University of North Carolina, McMillan and Bishop warp images, with disparity (a representation of depth), that have been acquired by digitally capturing a series of unstructured outdoor images, to render a new image from an arbitrary viewpoint. As long as the new viewpoint is close to that of an original image, the rendered image is good.

There are several advantages to image-based rendering. The biggest potential gain is to be able to render natural scenes with the richness of the real world. However, another major advantage is that the time to render is largely independent of the image quality. Even if the reference images have been rendered in traditional ways from synthetic data, the interactive rendering has been decoupled from the original rendering. That means that as much time as necessary may be spent to render the reference images to high quality, but the time to render a frame when viewing interactively is fixed. This feature may also be used to increase the apparent frame rate of traditional polygonal rendering systems [Regan 94].

These features of image-based rendering, the ability to use natural images as input models and the decoupling of generation from viewing, hold great promise for the future.

## Observations on Building Experimental Systems

The foremost return from the building of experimental systems in universities may well be the educational value of the enterprise. System building is largely a participatory exercise. While some of the theory may be learned in class, the most valuable skills are acquired through the master/apprentice system. A sign of the success of this system is that many former Pixel-Planes students and staff are now building systems in industry and academia.

In the research arena, the long-time focus of our department is on *experimental* computer science, with the conviction that experiments on computer systems need to be performed on actual machines, not on paper designs. In fact, for many of the projects the focus is *inherently and necessarily on high performance platforms*, not solely on architectural demonstrations. This is because the systems are used for such applications as interactive computer graphics where an experiment is only valid if the frame rate is adequate. Furthermore, architectural prototypes alone do not provide confirmation that the ideas truly work. The machines must be full-scale and full-speed to really prove out the designs. It is unlikely that we would have had the success that we've experienced in transferring our ideas to industry if they had not first been proven on actual systems.

## Acknowledgments

The credit for the research accomplishments of the Pixel-Planes and PixelFlow projects go to the people who have worked on the project over the years: Daniel Aliaga, John Austin, Mike Bajura, Andrew Bell, Brad Bennett, Fred Brooks, Jeff Butterworth, Vern Chi, Jon Cohen, Grant Cooper, Brad Crittenden, Claire Durand, David Ellsworth, Niall Emmart, Nick England, John Eyles, Henry Fuchs, Jack Goldfeather, Howard Good, Trey Greer, Edward "Chip" Hill, Justin Heinecke, Paul Hilts, Sonya Holder, Linda Houseman, Vincent Illiano, Victoria Interrante, Kurtis Keller, Paul Keller, Heather Kerns, Lawrence Kesteloot, Pat Ko, Roman Kuchkuda, Subodh Kumar, Anselmo Lastra, Jonathan Leech, Jeff Mauldin, Jon McAllister, Peter McMurry, Steven Molnar, Mark Monger, Carl Mueller, Ulrich Neumann, Bryon Nordquist, Marc Olano, Sherry Palmer, Mark Parris, Krish Ponamgi, Voicu Popescu, John Poulton, Greg Pruett, Jiang Qian, John Rhoades, Eddie Saxe, Jason Smith, Susan Spach, Andrei State, Brice Tebbs, Steve Tell, John Thomas, Russ Tuck, Greg Turk, Sharon Walters, Yulan Wang, Laura Weaver, Greg Welch, and Rob Wheeler.

The Pixel-Planes and PixelFlow projects have been supported by the National Science Foundation under grant numbers ECS-8300970, MIP-8601552, MIP-9000894, and MIP-9306208, and by DARPA under contract DAAG 29-83-K-1048, and order numbers 6090, 7510, and A410. We have also received significant equipment contributions from the Hewlett-Packard Corporation.

## References

[Bajura 92] Bajura, M., H. Fuchs, R. Ohbuchi, "Merging Virtual Objects with the Real World: Seeing Ultrasound Imagery within the Patient", *Computer Graphics* 26(2) (Proceedings of SIGGRAPH 92), July 1992, pp. 203-210.

[Bergman 86] Bergman, L., H. Fuchs, E. Grant, S. Spach, "Image Rendering by Adaptive Refinement," *Computer Graphics*, 20(3), (Proceedings of SIGGRAPH '86), pp. 29-37.

[Chen 93] Chen, S. E. and L. Williams, "View Interpolation for Image Synthesis," *Proceedings of SIGGRAPH 93*, (Anaheim, CA) July 1993, pp. 279-288.

[Chen 95] Chen, S. E., "Quicktime VR: An Image-Based Approach to Virtual Environment Navigation," *Proceedings of SIGGRAPH 95*, (Los Angeles, CA) August 1995, pp. 29-38.

[Cramblitt92] Cramblitt, B., "UNC Uses PLB to Chart Performance of World's Fastest Graphics Computer", *Graphics Performance Committee Quarterly Report*, **2**(4), 4th Quarter 1992.

[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.

[Ellsworth 90] Ellsworth, D., H. Good, B. Tebbs, "Distributing Display Lists on a Multicomputer, " *Computer Graphics* 24(2) (Proceedings of the 1990 Symposium on Interactive 3D Graphics), March 1990, pp 147-154.

[Ellsworth 96] Ellsworth, D., *Polygon Rendering for Interactive Visualization on Multicomputers*, PhD Dissertation, University of North Carolina, Chapel Hill, NC, 1996.

[Eyles 88] Eyles, J., J. Austin, H. Fuchs, T. Greer, J. Poulton, "Pixel-Planes 4: A Summary," *Advances in Graphics Hardware II, Proceedings of Eurographics '87 Second Workshop on Graphics Hardware* (August 1987), eds. A.A.M. Kuijk, W. Strasser, book series: Eurographics Seminars, Springer-Verlag, 1988, pp. 183-207.

[Foley 90] Foley, J., A. van Dam, S. Feiner, J. Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley, Reading, MA, 1990.

[Fuchs 77] Fuchs, H., "Distributing a Visible Surface Algorithm over Multiple Processors," *Proceedings of the ACM Annual Conference*, 1977, pp. 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., J. Poulton, "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, "Developing Pixel Planes, A Smart Memory-Based Raster Graphics System," *Proceedings of the 1982 MIT Conference on Advanced Research in VLSI*, Artech House, Dedham, MA, 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.

[Fuchs 89] Fuchs, H., J. Poulton, J. Eyles, T. Greer, J. Goldfeather, D. Ellsworth, S. Molnar, G. Turk, B. Tebbs, L. Israel, "Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor-Enhanced Memories," *Computer Graphics*, 23(3), (Proceedings of SIGGRAPH '89), pp. 79-88.

[Gardner 85] Gardner, G., "Visual Simulation of Clouds," *Proceedings of SIGGRAPH '85*, (San Francisco), 1988, pp. 297-303.

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

[Gharachorloo 89] Gharachorloo N., S. Gupta, R. F. Sproull and I. E. Sutherland: A Characterization of Ten Rasterization Techniques, *Proceedings. Siggraph 89*, pp. 355-368.

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

[Goldfeather 89a] Goldfeather, J., S. Molnar, G. Turk, and H. Fuchs, "Near Real-Time CSG Rendering using Tree Normalization and Geometric Pruning," *IEEE Computer Graphics and Applications*, 9(3), May 1989, pp. 20-28.

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

[Hanrahan 90] Hanrahan, P., J. Lawson, "A Language for Shading and Lighting Calculations," *Computer Graphics* 24(4) (Proceedings of SIGGRAPH 90), pp 289-298.

[IVEX 94] Ivex Corporation, "VDS-2000 Image Generator", *Image Society Annual Image Generator Survey*, 1994, pp. IV1-IV5.

[Kumar95] Kumar, S., D. Manocha, A. Lastra, "Interactive Display of Large-Scale NURBS Models," *Proceedings 1995 Symposium on Interactive 3D Graphics*, (Monterey, CA), April 1995, pp. 51-58.

[Lam 90] Lam, K., L. Dennison, W. Dally, "Simultaneous Bidirectional Signalling for IC Systems," *Proceedings of ICCD '90: VLSI in Computers and Processors*, Cambridge, MA, September 1990, pp 430-433

[Lastra 95] Lastra, A., S. Molnar, M. Olano, and Y. Wang, "Real-Time Programmable Shading", *Proceedings 1995 Symposium on Interactive 3D Graphics*, (Monterey, CA), April 1995, pp. 59-66.

[Levoy 89] Levoy, M., "Design for a Real-Time High-Quality Volume Rendering Workstation," *Chapel Hill Workshop on Volume Visualization* (Chapel Hill, North Carolina, May 1989).

[Levoy 90a] Levoy, M., "Volume Rendering by Adaptive Refinement," *The Visual Computer* 6(1), February 1990, pp. 2-7.

[Levoy 90b] Levoy, M., "Efficient Ray Tracing of Volume Data," *ACM Transactions on Graphics* 9(3), pp. 245-261.

[McMillan 95a] McMillan L., G. Bishop, "Head-Tracked Stereo Display Using Image Warping," 1995 IS&T/SPIE Symposium on Electronic Imaging Science and Technology, *SPIE Proceedings #2409A*, (San Jose, CA) February 5-10, 1995, pp. 21-30.

[McMillan 95b] McMillan L., G. Bishop, "Plenoptic Modeling: An Image-Based Rendering System," *Proceedings of SIGGRAPH 95*, (Los Angeles, CA) August 6-11, 1995, pp. 39-46.

[Mead80] Mead, C., L. Conway, *Introduction to VLSI Systems*, Addison-Wesley, 1980.

[Molnar 90] Molnar S., H. Fuchs, "Advanced Raster Graphics Architecture," Chapter 18 in *Computer Graphics: Principles and Practice* by James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes, Addison-Wesley, New York 1990, pp 855-922.

[Molnar91] Molnar, S., "Image Composition Architectures for Real-Time Image Generation", Ph.D. Dissertation, University of North Carolina, Chapel Hill, also available as UNC-Computer Science Technical Report TR91-046, 1991.

[Molnar 92] S. Molnar, J. Eyles., and J. Poulton, "PixelFlow: High-Speed Rendering Using Image Composition", *Proceedings of SIGGRAPH 92* (Chicago, IL) July, 1992, pp 231-240.

[Molnar94] Molnar, S., M. Cox, D. Ellsworth, H. Fuchs, "A Sorting Classification of Parallel Rendering," *IEEE Computer Graphics and Applications*, **14**, 4, July 1994, pp 23-32.

[Molnar 95] Molnar, S., "The PixelFlow Texture and Image Subsystem", *Proceedings of the Tenth Eurographics Workshop on Graphics Hardware*, (Maastricht), August 1995, pp 3-13.

[Neumann 92] Neumann, U., "Interactive Volume Rendering on a Multicomputer", *Proceedings of the 1992 Symposium on Interactive 3D Graphics*, Cambridge, Massachusetts, 1992, pp 87-94.

[OpenGL92] OpenGL Architectural Review Board, "OpenGL Reference Manual", Addison Wesley, 1992.

[Parke80] Parke, F., Simulation and Expected Performance of Multiple Processor Z-Buffer Systems," Computer Graphics **14**, 6, *Proceedings of SIGGRAPH 1980*, pp. *48-56.*

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

[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, "Pixel-Planes: Building a VLSI-Based Graphic System," *Proceedings of the 1985 Chapel Hill Conference on VLSI*, Computer Science Press, Rockville, MD, 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.

[Poulton 92] Poulton, J., J. Eyles, S. Molnar, H. Fuchs, "Breaking the Frame-Buffer Bottleneck with Logic-Enhanced Memories," *IEEE Computer Graphics and Applications*, Nov 1992, pp 65-74.

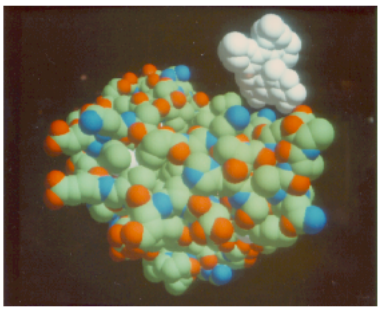
[Rhoades 92] Rhoades, J., G. Turk, A. Bell, A. State, U. Neumann, A. Varshney, "Real-Time Procedural Textures", *Proceedings of the 1992 Symposium on Interactive 3D Graphics*, (Cambridge, MA), 1992, pp 95-100.

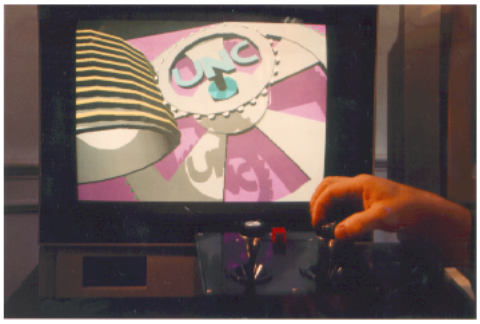
[Regan 94] Regan M., R. Pose, "Priority Rendering with a Virtual Reality Address Recalculation Pipeline", *Proceedings of SIGGRAPH 1994*, (Orlando, FL), July 1994, pp 155-162.

[Tebbs 92] Tebbs, B, U. Neumann, J. Eyles, G. Turk, D. Ellsworth, "Parallel Architectures and Algorithms for Real-Time Synthesis of High Quality Images using Deferred Shading," UNC CS Technical Report TR92-034 (originally written and submitted for publication in 1990).

[Upstill 90] Upstill, S., *The RenderMan(TM) Companion: A Programmer's Guide to Realistic Graphics*, Addison-Wesley, Reading, MA, 1990.

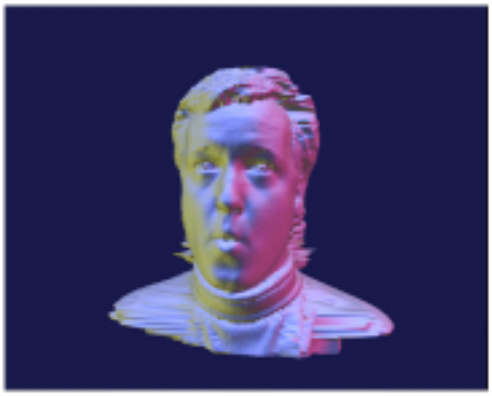
[Yoo 91] Yoo, T., U. Neumann, H. Fuchs, S. Pizer, T. Cullip, J. Rhoades, and R. Whitaker, "Achieving Direct Volume Visualization with Interactive Semantic Region Selection," *Proceedings of IEEE Visualization '91*, San Diego, CA. IEEE, October 1991.



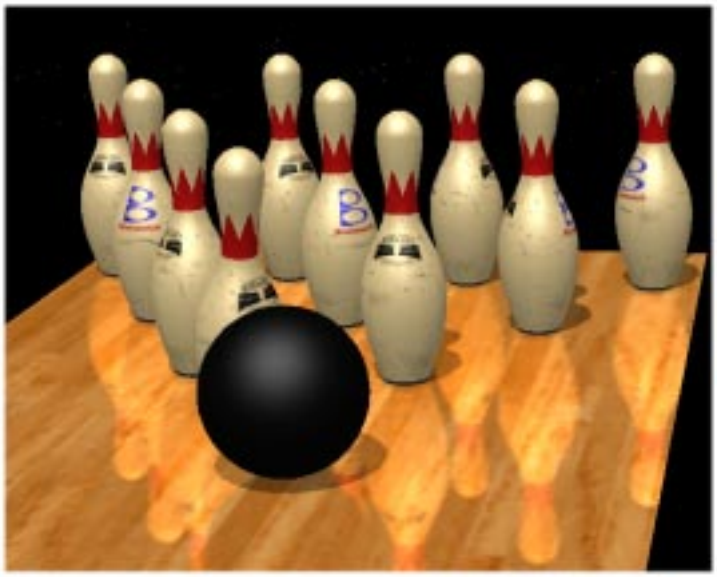


Images from Pixel-Planes 4 illustrating direct rendering of spheres and shadows.

Procedural texturing on Pixel-Planes 5. Brick, ceiling and painting are all rendered procedurally.



Pixel-Planes 5 set a world's record on the Graphics Performance Characterization Committee's "head" benchmark in 1992.



Procedural shading on the PixelFlow functional simulator. This image illustrates shadows, reflections, bump mapping, image-based texturing, and effects such as procedural "dirt" in the bumps on the pins.