The Pixel Machine: A Parallel Image Computer

Michael Potmesil and Eric M. Hoffert

AT&T Bell Laboratories
Holmdel, New Jersey

Abstract

We describe the system architecture and the programming environment of the Pixel Machine - a parallel image computer with a distributed frame buffer.

The architecture of the computer is based on an array of asynchronous MIMD nodes with parallel access to a large frame buffer. The machine consists of a pipeline of *pipe nodes* which execute sequential algorithms and an array of $m \times n$ pixel nodes which execute parallel algorithms. A *pixel node* directly accesses every *m-th* pixel on every *n-th* scan line of an interleaved frame buffer. Each processing node is based on a high-speed, floating-point programmable processor.

The programmability of the computer allows all algorithms to be implemented in software. We present the mappings of a number of geometry and image-computing algorithms onto the machine and analyze their performance.

CR Categories and Subject Descriptors: C.1.2 **[Processor Architectures]:** Multiprocessors - *MIMD processors - parallel processors - pipeline processors*; D.4.2 **[Operating Systems]:** Storage Management - *distributed memories - virtual memory*; I.3.1 **[Computer Graphics]:** Hardware Architecture - *raster display devices*; I.3.3 **[Computer Graphics]:** Picture/Image Generation - *display algorithms*; I.3.7 **[Computer Graphics]:** Three Dimensional Graphics and Realism - *animation - visible line/surface algorithms*; I.4.0 **[Image Processing]:** General - *image displays*.

General Terms: Parallel, Pipeline, Architecture, Algorithms, Geometry and Image Computing, Shared Memory, Distributed Memory, Interleaved Memory, Virtual Memory, Message Passing.

Additional Key Words and Phrases: Active server, passive server, virtual node, virtual shared memory, virtual display lists, virtual volumes, virtual textures, parallel paging.

 ACM-0-89791-312-4/89/007/0069 $00.75

1. Introduction

As computing technology progressed, it became apparent that even the most powerful computers available, built on principles devised by John von Neumann in the early 1940s, are reaching the limits of their speed imposed by the constraints of physical laws. The single-processor model executing only at most one instruction in every machine cycle is beginning to outlive its usefulness. There is no inherent reason why many calculations cannot be performed simultaneously. Computer graphics is a perfect example of such an application area. Pixels can be read, written and processed simultaneously; in fact, most graphics algorithms impose few limits on the amount of parallelism achievable for pixel processing.

With a parallel architecture, a designer hopes that, instead of the typical linear improvement in performance that is inherent in technology evolution, a quantum leap in performance can be obtained. Such a quantum leap has been *demanded* by the various communities using image computing. The recent report *Visualization in Scientific Computing* [13] stresses the need for innovative high-speed architectures to meet the needs of interpreting large amounts of scientific data. Animators require photorealistic rendering of high scene complexity and image quality with quick turnaround times. Doctors and radiologists must see a 3D reconstruction from an NMR or CT device in seconds. For image computing to be a practical tool in these and other areas, it is not feasible to wait for evolutionary improvements in technology. Instead, a break from traditional architectures must occur and be built. In this paper, we describe such an architecture; what motivated its development, how it works and what it portends for the future of image computing.

The design of the Pixel Machine was inspired and influenced by:

- **speed** - the advent of fast RISC-style digital signal processors that offer a large amount of the functionality found in a microprocessor with an integrated floating-point unit at a fraction of the price [10].
- **parallelism** - parallel architectures, in which processing is performed in parallel by nodes on the contents of their local

Authors' addresses:

Michael Potmesil, AT&T Bell Laboratories, Room 4F-625, Holmdel, NJ 07733; Telephone: (201) 949-4826; Email: mp@vax135.att.com

Eric M. Hoffert, Apple Computer Inc., 20525 Mariani Avenue, Mail Stop 60V, Cupertino, CA 95014; Telephone: (408) 974-0493; Email: emh@apple.com

memories and messages can be exchanged between processors [17,3,9] and local memories can be part of a video frame buffer [7,18].

- **interleaving** - the notion of an interleaved frame buffer, distributed among the processors of a parallel image-computing system, to achieve load balancing as originally developed in [6,14].
- **programmability** - the concept of a programmable graphics machine attached to a host computer as introduced in the *Ikonas* frame buffer and graphics processor and later also used by *Pixar* [11] and *TAAC-1* [19].
- **pipelining** - pipelined operations as applied in the *Geometry Engine* [2] to geometry computing.
- **flexibility** - the value of a rendering and modeling programming environment, such as FRAMES [15], where different computing modules following the old software adage *"small is beautiful,"* can be interconnected in different ways to achieve diverse modeling and rendering functions.
- **partitioning** - image-space or object-space partitioning of data among 2D or 3D arrays of asynchronous, independent processing elements as described in [5].

2. System Architecture

The Pixel Machine was designed as a programmable computer with pipeline and parallel processing closely coupled to a display system [16,1]. The Pixel Machine consists of four major building blocks [Figure 1]: (a) a pipeline of *pipe nodes*, (b) an array of $m \times n$ parallel *pixel nodes* with a distributed frame buffer, (c) a *pixel funnel*, and (d) a *video processor*. The pipeline and pixel-array modules can be incrementally added to a system to build a more powerful computer.

The Pixel Machine functions as an attached processor. In the current configuration the host computer is a high-end workstation, but in principle diverse hosts could be supported, ranging from personal computers to supercomputers.

2.1 Computations

The CPU of the computing nodes is a DSP32 digital-signal processor with an integrated floating-point unit [10]. It consists of a 16-bit integer section and a 32-bit floating point section. The integer section with 21 registers is mainly used to generate memory addresses while the floating-point section with four 40-bit accumulators is used to process geometry and image data.

The DSP32 has a RISC-style instruction set and instruction decoding. Unlike a RISC processor which operates only on data in registers and uses load/store register-memory accesses, the DSP32 uses register pointers to point to arrays of data in memory. The pointers are usually post-incremented during the same instruction. In a typical operation, the DSP32 can read two operands from memory and one from an accumulator, perform a multiply-accumulate operation and write the result to an accumulator and to memory.

The DSP32 has a 16-bit addressing capability, allowing it to address directly only* 64 Kbytes of memory. There are 4 Kbytes of RAM memory on board of the chip. Each pixel and pipe node has additional 32 Kbytes of fast static RAM memory. These 36 Kbytes are used for program and scratch data storage.

Pixel nodes also contain a distributed frame buffer and z-buffer. In each pixel node, there are 512 Kbytes of video RAM memory organized as two banks of 256×256 32-bit $rgb\alpha$ pixels and 256 Kbytes of general-purpose dynamic RAM memory which can be organized as a 256×256 32-bit floating-point z-buffer. These additional 3/4 Mbytes of memory are addressed via a memory management unit.

The nodes are running at 5 Mips or 10 Mflops which must really be interpreted as 5 million multiply-accumulate operations per second. In typical applications, programmed in C, the overhead of invoking functions, computing data pointers, etc. can reduce the floating-point operations to about 10-25% of the peak rate.

2.2 Communications and Connections

There are a number of different communication paths in the system. Each pixel and pipe node is connected to the VMEbus via a DMA port (*host-to-node connection*). This port can be used by the host to access all memory-mapped locations in a node and for handshaking and synchronization activities by the node.

Pipe nodes are connected with fifos into nine-node pipelines (*downstream pipe node-to-node connection*). The fifo input to the first node is written by the host via the VMEbus, the fifo output of the last node is either broadcast - via a broadcast bus - to all the pixel nodes (*pipe-node to pixel-node connection*) or written to a fifo read back - via the VMEbus - by the host. The pipe nodes in a pipeline are also connected via a unidirectional serial asynchronous link in the direction opposite to the fifos (*upstream pipe node-to-node connection*). Two pipelines can be placed in a system and configured as two parallel pipes or one long serial pipe.

Pixel nodes are connected to their four nearest neighbors, in a closed-torus network, via serial bidirectional asynchronous links (*pixel node-to-node connection*). These pathways allow flexibility for data movement needed in different algorithms. Some pixel-node operations, such as changing display buffers or exchanging messages with adjacent nodes, require all the nodes to be synchronized: they have to wait for the last node to complete its previous computations. There are two hardware semaphores, shared by all the pixel nodes, which allow global synchronization. Pixel nodes can also be synchronized with vertical and horizontal video retrace periods.

2.3 Pixel Mapping and Display

The frame buffer in the Pixel Machine is distributed into the array of the $m \times n$ pixel nodes. The frame buffer is divided into two or more display buffers. One of these buffers is always displayed by the video system, at the selected size and speed, on the screen. When in double-buffered mode, a second buffer is used to draw the next image. Additional buffers may contain other pixel-oriented data such as texture maps. Pixels in the displayed buffer are read by the video processor and mapped on the video screen. This mapping is determined by the position of each pixel node within the array and is fixed. Each pixel node contains the size of the pixel-node array (m,n) and its position within the array (p,q) where $0 \le p < m$ and $0 \le q < n$. The position (p,q) also serves as a unique identification number of each node. Pixel node (p,q) then displays every m-th pixel starting with pixel p on every n-th scanline starting with scanline q, i.e., a processor-space

* It should be noted that the next generation of this processor has a 24-bit addressing space allowing it to address directly 16 Mbytes of memory.

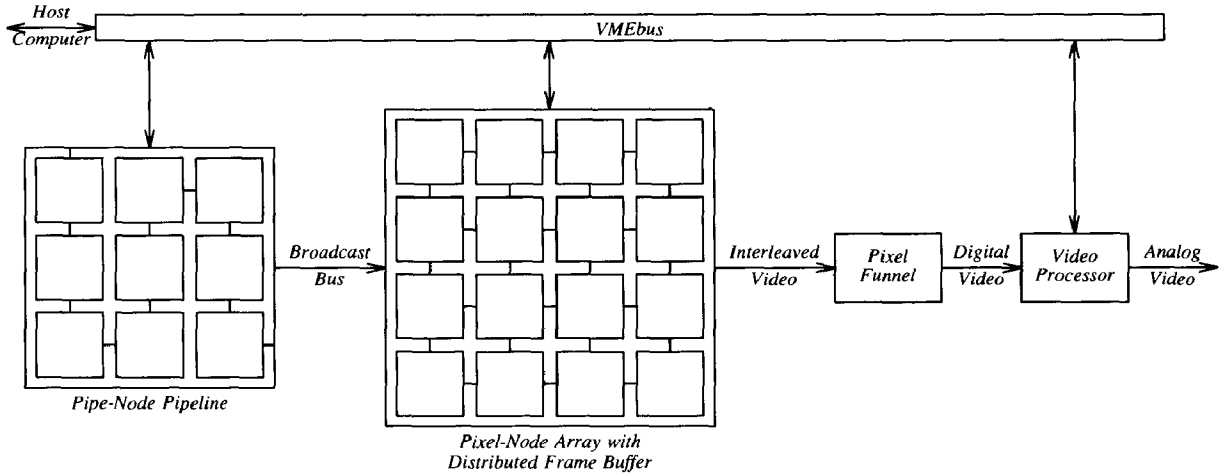

Figure 1 A block diagram of the Pixel Machine.

pixel (i,j) is mapped into a screen-space pixel (x,y) by:

$$x = m\ i + p$$
$$y = n\ j + q \qquad (1)$$

This format requires the display subsystem to collect all of the distributed frame-buffer pixels and assemble them into a contiguous screen image. The device that performs this function is called the video *pixel funnel*. The interleaved format of the frame buffer provides load balancing for image-computing algorithms and matches well the speed limitations of video RAM memories with the speed requirements of a high-resolution display*.

The architecture of the pixel nodes is scalable, using between 16 and 64 nodes [Table 1]. The video processor can be programmed to display two high-resolution formats as well as NTSC and PAL.

To aid in the development of uniform software for all the pixel-node configurations and to allow hardware modularity, the concept of *virtual pixel nodes* was utilized. A virtual node renders into a subset of a buffer, called a *virtual screen*, all within a physical node. The virtual nodes and their virtual screens are also interleaved in an $m' \times n'$ pattern - just as the physical nodes - with each virtual node having a unique screen position (p',q'). The mappings in equation (1) also apply to the virtual nodes. All software is written for one virtual node and is invoked one or more times, depending on the system size, by a physical node. The physical and virtual pixel-node configurations of the Pixel Machine are shown in Table 1.

3. Software Architecture

Software developed to run on the Pixel Machine is always divided into two major conceptual areas: *host software* and *node software*. The latter category is further subdivided into *pipe-node software* and *pixel-node software*. Host software controls interaction with the Pixel Machine, pipe-node software executes sequential-type algorithms and finally pixel-node software executes parallel algorithms.

* A pixel is shifted out of a video memory in ≈40 ns while it is displayed on a 1280 × 1024 pixel screen in ≈9 ns. Therefore, at least 5 parallel banks of video memories are required to shift out 5 pixels in ≈40 ns.

3.1 Host Software

Each node in the Pixel Machine is a small autonomous computer, albeit with a number of limitations. The current processor used in each node does not support interrupts and has limited addressing capabilities. These limitations forced the software designers of the Pixel Machine to come up with a number of creative solutions to difficult problems typically not encountered on a conventional computer. A programming environment had to be developed that simulates much of the functionality taken for granted in a standard operating system.

There are two different types of processes which can run on the host computer and interact with or control the Pixel Machine:

- **passive server**

  This process functions as a data-base server for the Pixel Machine. In this capacity, interaction takes place in a linear fashion: the host sends a stream of commands and data to the Pixel Machine, and the Pixel Machine performs various operations on the received data. There is no interaction initiated by the Pixel Machine, it responds only when it is explicitly requested to do so (e.g., to a command to return the current transformation matrix). This server is employed almost exclusively for traditional polygon rendering, where databases and commands are generated by the host and sent to the machine. In this mode the Pixel Machine acts as slave and the host computer as master.

- **active server**

  This process is responsible for responding to all requests for resources that are made by the Pixel Machine. It polls a user-defined set of nodes (pipe *or* pixel) for messages. When a message is received, the active server initiates a host function that supplies needed resources to the requesting node. We have found this to be a very powerful paradigm for host/Pixel Machine interaction. The host needs the Pixel Machine for certain demanding geometry and image computing, and the Pixel Machine needs the host for contiguous large blocks of memory and for access to a file system (among a number of other potential needs). In this mode the Pixel Machine acts as master and the host computer as slave.

Physical			Virtual			V/P
nodes	$m \times n$	pixels/node	nodes	$m' \times n'$	pixels/node	Ratio
16	4×4	256×256*	64	8×8	128×128	4
20	5×4	256×256**	80	10×8	128×128	4
32	8×4	128×256*	64	8×8	128×128	2
40	10×4	128×256**	80	10×8	128×128	2
64	8×8	128×128* 160×128**	64	8×8	128×128 160×128	1 1

Table 1 Physical and virtual pixel-node configurations.

* Display screen size: 1024×1024 pixels.
** Display screen size: 1280×1024 pixels.

The host process has complete control over all nodes. It can access all memory in each node including program memory and frame-buffer memory in pixel nodes. Such accesses take place, via DMA, even when the nodes are running.

The host software is also responsible for halting, initializing and starting each node as well as for downloading programs into them. It also configures the video processor and accesses the video lookup tables.

3.2 Pixel-Machine Software

Software that runs on the Pixel Machine is quite distinct from software that runs on von Neumann machines. The important distinction from the single-processor approach is that software is mapped to different architectural components, each of which has a different *character* and number of nodes. The pipeline (where each pipe node typically contains a distinct program) executes sequential algorithms and the pixel-node array (where each pixel node typically contains the same program*) executes parallel algorithms. In some cases, our algorithm is entirely sequential; such an algorithm would run only in the pipe nodes. Analogously, we have algorithms that are entirely parallel in nature; such an application might not utilize the pipeline at all. We have found that most applications have components that map onto both the pipeline and pixel-node array.

3.3 Pipe-Node Software

Pipe nodes are employed for operations that are intrinsically sequential in nature. Such operations are those that constrain the efficiency of a parallel algorithm. The use of a pipeline is an attempt to remove as much sequential style processing from the parallel pixel-node array as possible.

Pipe-node software requires algorithm partitioning. Each pipe node acts as a distinct computational element in a pipeline. A separate program runs in each node and messages - commands and data - are passed down a pipeline. The last node in a pipeline has the ability to broadcast messages to all the nodes in a pixel-node array or to return them back to the host.

The FRAMES system [15] contains methods for experimenting with pipeline partitioning and how to achieve maximum flexibility in such a scheme. The same philosophy is employed here. Our experience shows that special care must be taken to ensure that software in the pipeline does not become I/O bound.

Pipe-node software can be written to allow the same program to reside in several consecutive nodes and to operate on alternating input messages (e.g., each instance of an n-node transformation program transforms only every n-th polygon). This allows the same software to run efficiently in longer pipelines and to eliminate or reduce bottlenecks by repeating the slowest program in more than one node.

3.4 Pixel-Node Software

This section describes (a) what actions a pixel node performs as a computational element and (b) the general mechanisms available for increasing the amount of data that a pixel node can directly access. There are two approaches to the issue of memory limitation. The first approach is that of message-passing, where nodes exchange portions of distributed data. This approach exploits the ability of a machine to shuffle large amounts of data among its nodes. The second approach utilizes the memory of the host computer, letting it serve as an adjunct memory device for individual nodes.

Support software in the pixel nodes comprises several categories: screen-space to processor-space coordinate mapping, frame-buffer and z-buffer access to pixel-oriented data, display-list access, and optimized mathematical functions.

Mapping functions transfer coordinates from the (x,y) display screen space to the (i',j') virtual screen space of a virtual pixel node (p',q') by:

$$i' = \frac{x - p'}{m'} = \frac{1}{m'} x - \frac{p'}{m'}$$

$$j' = \frac{y - q'}{n'} = \frac{1}{n'} y - \frac{q'}{n'}$$

where the scale multiplication is the same for all the pixel nodes and therefore is actually computed by a pipe node and

* However, there is no reason why each pixel node cannot execute a different program.

the offset subtraction is computed individually by each pixel node.

There are four basic mapping functions, used in all image-computing algorithms, which transform screen coordinates to processor coordinates. Function $ilo(x)$ returns the *smallest* integer i' such that $m'i' + p' \geq x$:

$$ilo(x) = \left\lceil \frac{x - p' - 0.5}{m'} \right\rceil$$

Function $ihi(x)$ returns the *largest* integer i' such that $m'i' + p' \leq x$:

$$ihi(x) = \left\lfloor \frac{x - p' + 0.5}{m'} \right\rfloor$$

Similarly, function $jlo(y)$ returns the smallest integer j' such that $n'j' + q' \geq y$, and function $jhi(y)$ returns the largest j' such that $n'j' + q' \leq y$.

The mapping from the screen space to the processor space is not one-to-one: there are more pixels in screen space than in processor space. To be certain that processor-space pixel (i',j') is actually screen-space pixel (x,y), these two conditions must be true:

$ilo(x) = ihi(x)$, and

$jlo(y) = jhi(y)$

Each node can independently read or write the contents of its individual frame buffer and z-buffer. Access to these memories is in row and column addressing modes using virtual screens. A 32-bit pixel can be accessed in four instruction cycles (one cycle to read each color component) and a 32-bit z-buffer value in one cycle.

Mathematical functions include routines for frequently used operations in geometry and image computing such as square root (ray-sphere intersection), vector normalization (shading), and dot product (back-face removal). These highly-optimized functions efficiently utilize the floating-point capability of the DSP32 at each node, since many of the operations involve multiply/accumulate instructions.

3.5 Interleave/De-Interleave

Each node in the pixel-node array has a four-way serial I/O switch. This allows a node to communicate directly with its four nearest neighbors. Communications between two nodes occur over a half-duplex serial channel. All nodes must synchronize to exchange data, and message-passing occurs in lock-step fashion, with all nodes sending data in the same direction at the same time. This type of communication scheme is well-suited to problems that map onto a grid or torus architecture.

There are times when it is undesirable to compute on pixels in an interleaved format. Using the current Pixel Machine, this is not possible through hardware due to constraints imposed by video memory access requirements. At this point, the old hardware adage *"do it in software!"* is employed.

Software can take the interleaved frame-buffer format, and using serial I/O message-passing, reconfigure the frame buffer so that each node has a contiguous block of pixels. We call this process *de-interleaving*. Analogously, it is possible to take a frame buffer configured as contiguous blocks and again employing serial I/O message-passing, distribute the pixels so that they are in their correct interleaved position for display. We call this method *interleaving*.

3.6 Virtual Memory

Photorealistic rendering requires large amounts of data. This data is typically geometry information, but can also consist of texture maps, environment maps, etc. Other rendering techniques, such as volume rendering, can also require significant amounts of data storage. We have also found that an efficiently coded implementation of a rendering program (ray or volume tracers, for example) can be very small, in terms of code space. Hence it became apparent that we could develop schemes for virtual memory [4] which would be used only for *data*.

Each node has a page table in its memory along with a set of associated pages. When a memory access is required for data that does not reside in the available pages, a *parallel page-fault* is generated, causing a node to make a request to the host to deliver the required page of memory. The page is broadcast to all nodes in the pixel array from the last node in the pipeline. At this point, the page table in each node is updated, deleting a page based on a page-replacement policy and adding the newly requested page to the table. We call it parallel paging, since typically nodes may request pages from the host concurrently.

The parallel paging scheme is employed for *virtual display lists* in the ray-tracing software implemented on the Pixel Machine. Figure 2 shows a ray-traced image with 17,000 polygons. Each polygon uses 100 bytes, giving a database size of 1.7 Mbytes, substantially more than can fit in one pixel node's local memory. Figure 3 also shows a ray-traced image generated using virtual display lists. This scene contains over 50,000 polygons, area-light sources and is antialiased at 16 samples per pixel.

The active server can store multiple texture maps or volume databases in host's memory. When an individual pixel or voxel is requested by an arbitrary node, the host retrieves a page of adjacent data and routes it to the requesting node. This scheme is especially suitable for either (a) applications with memory requirements that far exceed the collective memory capacity of the pixel nodes, or (b) applications where distribution of memory over the pixel nodes would require an overly complex and/or inefficient algorithm. Because all pixel nodes have access to this memory, we call it *virtual shared memory*.

Figure 4 shows a ray-traced image that uses *virtual texture maps*. There are 13 virtual texture maps requiring a total of 4 Mbytes of texture map data. The scene also contains approximately 2,000 polygons. Figure 5 shows a volume rendering of a nuclear magnetic resonance (NMR) angiography study that uses *virtual volumes*. The size of the data is 256 × 256 × 160 voxels or approximately 10 Mbytes.

3.7 Program Overlays

A node can directly address 64 Kbytes of memory. This constraint coupled with the cost and size of fast static RAM memories dictated the size of program memory at 36 Kbytes in the current Pixel Machine. The solution to this problem of small program size is a classic one, first seen in the early days of computing. If a node does not have enough program or local data memory available for a required function or message processing, we use *program overlays* [8].

A program is manually divided into a static instruction and data segment which resides in a node at all times and several dynamic segments which are swapped-in, one at a time from the host. The host server keeps track of the overlay segments loaded into any of the nodes and ensures that the correct

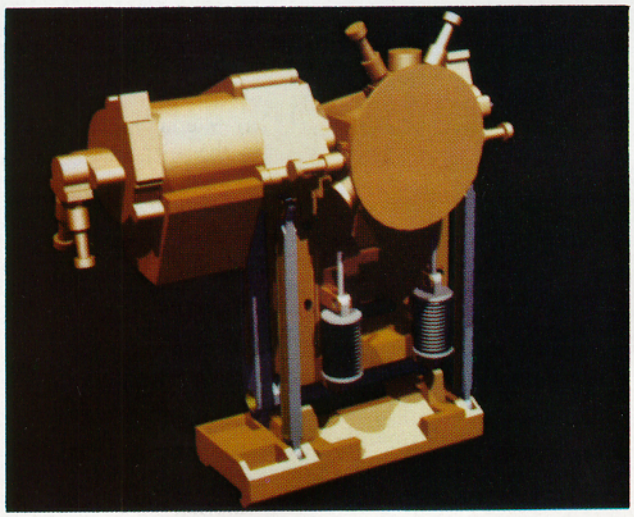

Figure 2 Virtual display lists: *A Stabilized Platform-Deployment Station.*

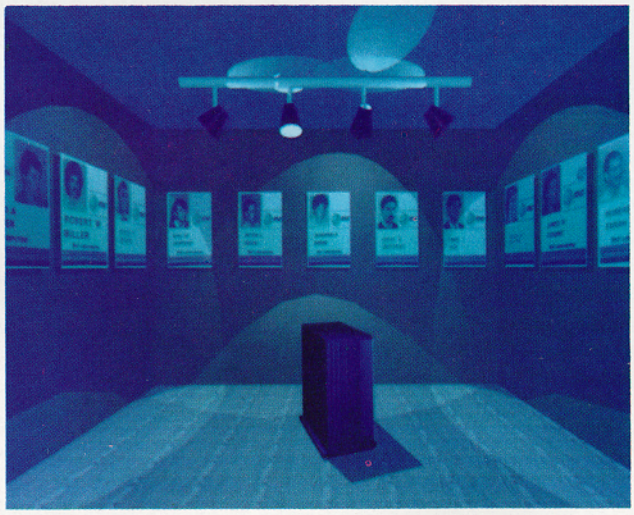

Figure 4 Virtual texture maps: *A Museum Room.*

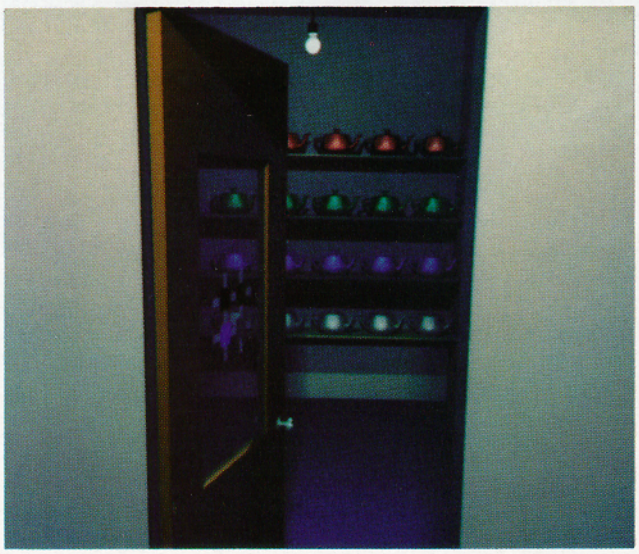

Figure 3 Virtual display lists: *A Tea Room.*

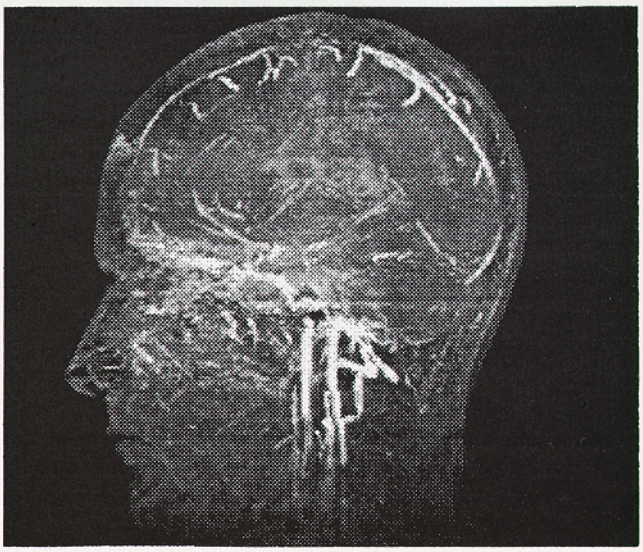

Figure 5 Virtual volumes: *A Sagittal View of NMR Data.*

segments are loaded into each node before data requiring them arrive. The cost of sending overlays from the host and loading them into a node's program memory is small: the bandwidth from the host to the pipeline is on the order of Mbytes/s and the overlay segments are on the order of single Kbytes.

4. Algorithm Mapping

In this section, we describe the mappings of a few well-known geometry and image-computing algorithms to the Pixel Machine architecture:

- **polygonal rendering**

  Points, lines, polygons and other geometric primitives are transformed, clipped, shaded, projected and broadcast by the pipeline nodes. Complex geometric primitives (patches, superquadrics) are also generated or converted into polygons in the pipeline. The pixel-node array is used for raster operations, rendering of geometric primitives, z-buffering, texture mapping, image compositing and antialiasing. For polygonal rendering, the passive server is employed, routing large polygonal databases or multiple frames of animation to the Pixel Machine via the pipeline. Image antialiasing is accomplished by supersampling and floating-point convolution with an arbitrary filter kernel.

- **ray tracing**

  Ray trees are traced in parallel by the pixel nodes, with each node generating ray trees for pixel sampling points in its unique set of interleaved pixels. Each pixel node contains a copy of the display list of the scene being rendered. If the size of the display list exceeds the local pixel-node memory, the display list is paged from the host computer, using the parallel page-faulting method described earlier. The active server is used to service display list page faults and texture

map virtual shared memory requests respectively. The pipeline is used to compute bounding volumes, tessellate geometric primitives and to transform the display list before rendering begins. The floating-point capability of each node is exercised to its maximum for the ray-object intersection tests. Antialiasing is performed by stochastic sampling in multiple passes.

- **volume rendering**

  Rays are marched in parallel [12] by the pixel nodes inside volume data. Each node processes its set of interleaved pixels in the image. At each pixel, a ray is cast into the volume and ray-marching incrementally steps along the direction of the ray, sampling the signal inside. The sampled values of a ray are then converted into image intensity according to the application: thresholding, finding maximum, translucency accumulation and integration can be selected. The volume is stored on the host computer, with each pixel node requesting voxel packets that contain voxels along the path of a marching ray. This procedure is accomplished using virtual shared memory via the active server. The pipeline is not utilized in this mapping. Antialiasing is accomplished by sampling very finely along each ray and by interpolating voxel values adjacent to an intersection point.

- **image processing**

  An image is processed by the pixel nodes in parallel, with each pixel node computing its set of interleaved pixels. If the image is too large to fit in the local pixel-node memory, it can be distributed over the collective memory of all the nodes in contiguous block fashion and redistributed into interleaved format for a final display using the interleave/de-interleave strategy. The pipeline can be used for run-length decoding and other sequential image functions as an image is being sent to the pixel nodes.

5. Performance Analysis

In this section we attempt to analyze the theoretical performance of the Pixel Machine architecture and then look at some of our actual results.

5.1 Theoretical Performance Analysis

The classic recurrence equation for the *divide-conquer-marry* paradigm is as follows:

$$T(n) = g(n) + M\ T(n/M) + h(n)$$

where $g(n)$ is the cost of dividing up a problem into M subproblems (*divide*), $T(n/M)$ is the cost of running the subproblem (*conquer*), $h(n)$ is the cost of combining the results of the subproblems into a final solution (*marry*) and n is the number of data elements. This generic equation is typically applied to a sequential implementation of a recursive algorithm. Interestingly enough, the equation can also be applied to the analysis of algorithms on parallel machines. In this case, the multiplicative term M would drop out, since the *divided problems* or subproblems are being solved concurrently. The modified equation becomes:

$$T(n) = g(n) + T(n/M) + h(n)$$

The ideal parallel algorithm will have minimal $g(n)$ and $h(n)$ terms; these are the *parallel overhead* costs. The algorithm development efforts for parallel architecture are primarily concerned with ensuring that the $T(n/M)$ term will predominate in the expression above. This ensures that adding more processors to a problem yields a linear improvement in performance. A term that has recently entered into the parlance of parallel processing is *Non von Neumann bottleneck*. This refers to the costs $g(n)$ and $h(n)$, which are considered bottlenecks if they predominate in the expression above.

The salient difference between the Pixel Machine and other parallel machines is that there is no $h(n)$ term for displaying or animating the image computed by the pixel nodes. This immediately obviates a large amount of the usual parallel overhead. This term is eliminated because the interleaved frame buffer is assembled into a contiguous scan image by the pixel funnel. Only if we read back the computed image from the frame buffer to the host computer does the $h(n)$ term reappear.

The $g(n)$ term represents the cost associated with the screen space to processor space conversion. As an example of how this term affects efficiency, consider the case of rasterizing a geometric primitive in a pixel node. A simple equation describing the rasterization is as follows:

$$T(p) = g(x) + p\ I(x)$$

where p is the number of pixels rasterized, $T(p)$ is the time required to rasterize these pixels, $I(x)$ is the cost per pixel of rasterization for an arbitrary algorithm x and $g(x)$ is the parallel overhead for that algorithm. Let us also define η, the efficiency of a parallel algorithm implementation, to be the slope of the graph of normalized inverted execution time vs. number of pixel nodes. A unity value of η implies exactly linear improvement in performance for linear increases in the number of pixel nodes. This is what we aspire to for all implementations. Values less than unity indicate sublinear improvement for pixel-node increases. If p is small and $g(x)$ is large so that $g(x) > p\ I(x)$, then the parallel overhead predominates and $\eta \ll 1$. Conversely, if p is large and $g(x)$ is small so that $g(x) < p\ I(x)$, then the parallel overhead is small or negligible and $\eta \approx 1$.

The optimal algorithms for the Pixel Machine are those that require a $g(n)$ term only once per *image* as opposed to once per *object*. An example of the former is ray-tracing and of the latter is vector drawing. It is much easier to amortize the cost $g(n)$ once per image than once per object, since there may be many objects in an image.

5.2 Measured Performance Analysis

We have tested the actual efficiency of the machine on a number of different image-computing algorithms:

- **raster operations**

  A basic pixel-node function is to modify rectangular regions on the screen in various ways. The pixel-node organization allows $m'n'$ pixels to be processed in parallel by $m' \times n'$ virtual nodes during each iteration. Figure 6 illustrates performance of the machine performing raster operations on 128^2, 256^2, 512^2, and 1024^2 pixel regions. The execution times are plotted as solid lines and the normalized efficiency is shown as dotted lines. The efficiency of the machine, as the slopes of the dotted lines indicate, is very high with almost linear improvement and increases as the size of the region increases.

- **point rasterizing**

  A pixel node maps a point into its screen space and then tests if the point actually belongs there and should be drawn. Each pixel node maps and tests all the points but typically draws only $1/m'n'$ of them, giving a very low figure of merit. The graphs in Figure 7 indicate that the sequential part of the algorithm dominates: the bottleneck is a pipe node which converts the host floating-point and integer

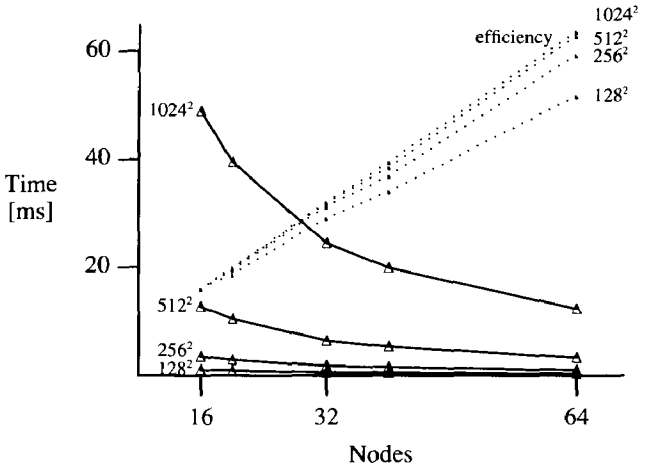

Figure 6 Parallel performance: raster operations.

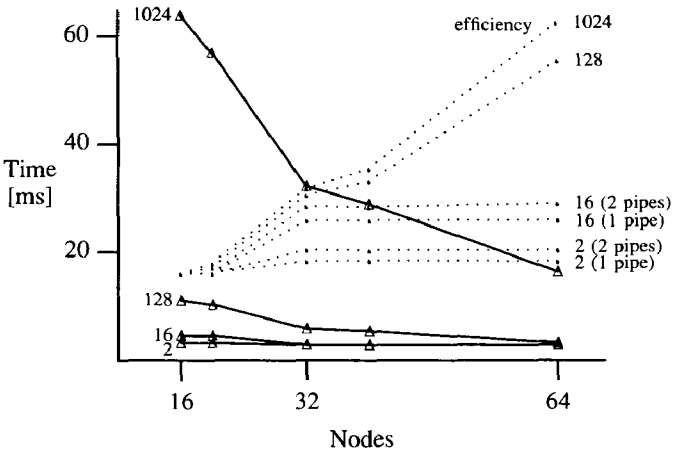

Figure 8 Parallel performance: aliased vectors.

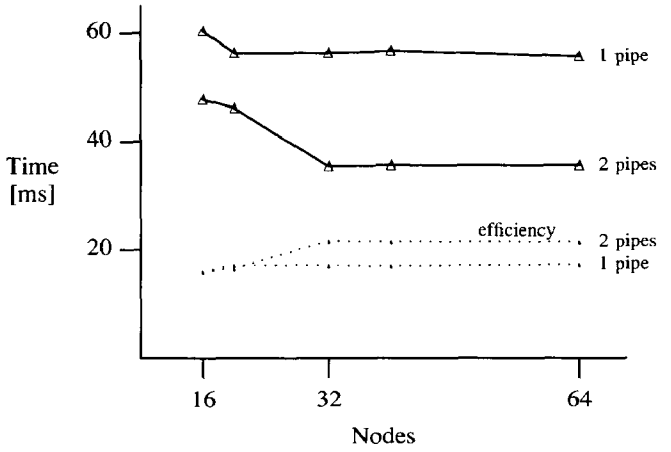

Figure 7 Parallel performance: points.

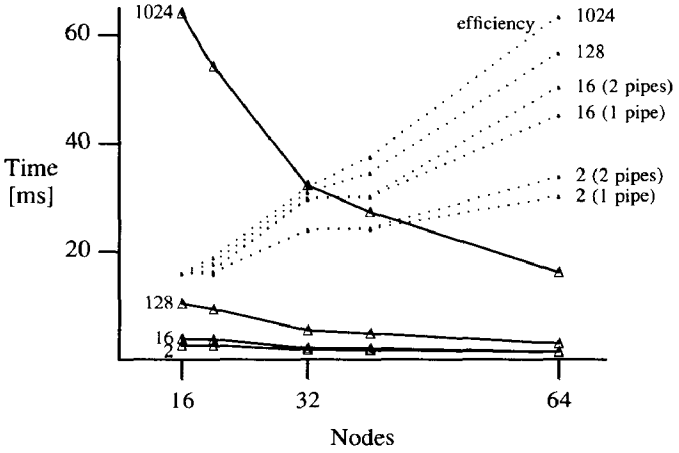

Figure 9 Parallel performance: antialiased vectors.

formats to the DSP32 floating-point format. There is not any speed improvement above 20 pixel nodes when a single pipeline is used. In a system with two parallel pipelines the improvement stops at 32 pixel nodes. Two parallel pipelines improve the speed of this algorithm by about 70% for 32 or more pixel nodes.

- **vector rasterizing**

A parallel version of the Bresenham algorithm rasterizes one-pixel wide aliased vectors. In an $m' \times n'$ array of virtual pixel nodes, the algorithm writes $min(m',n')$ pixels during one iteration. The figure of merit for this algorithm is only $min(m',n')/m'n'$. Line drawing, which is essentially a one dimensional process, cannot be very efficiently implemented on this architecture. Performance for randomly-oriented 2, 16, 128 and 1024 pixel-long vectors is shown in Figure 8. Actual times are again plotted as solid lines while the efficiency of the algorithm is plotted as dotted lines. As expected, the slope of these lines illustrates the low efficiency. For very short vectors the overhead becomes dominant and there is almost no improvement in speed as the number of processors increases.

Antialiased vectors are drawn by a modified version of the above algorithm which computes pixel intensity based on distance from the vector and blends the intensity with the background. Figure 9 shows the relative performance of this algorithm for the same randomly-oriented vectors as in Figure 8. On absolute time scale, aliased vectors are about twice as fast as antialiased vectors. However, because more processors do more useful work per pixel and per iteration, the antialiased algorithm is more efficiently implemented in this architecture than the aliased algorithm. In both algorithms, a small speed improvement is obtained for short vectors when two parallel pipelines are used.

- **polygon rasterizing**

A pipe node converts polygons into triangles, sorts their vertices in y and computes the slopes of the three edges. The pixel nodes transform the slopes into their processor spaces, compute forward difference in y along the edges and scan-convert the triangle by stepping in y along two active edges and filling the span in x between them. Performance for random triangles within 8^2, 16^2, 64^2, 256^2 and 1024^2 bounding squares is given in Figure 10. For large triangles the performance is similar to raster operations. As the size of the triangles decreases the sequential part (executing in the pipe nodes) and the parallel overhead of the rasterizing algorithm dominates and decreases the efficiency of the architecture.

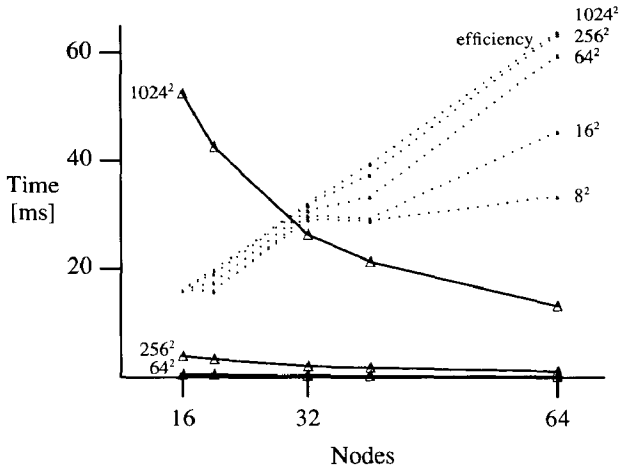

Figure 10 Parallel performance: Gouraud-shaded z-buffered polygons.

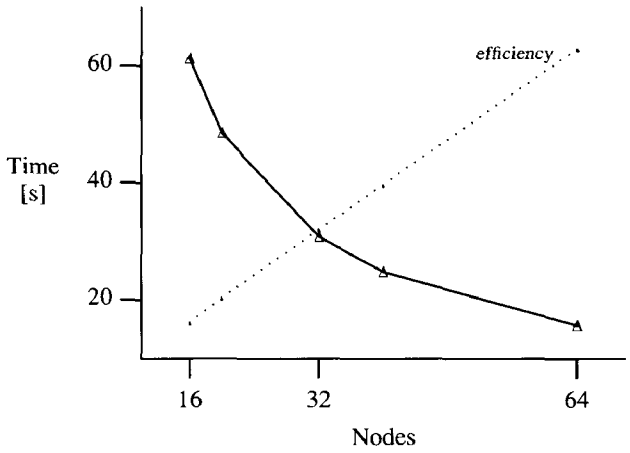

Figure 12 Parallel performance: ray tracing.

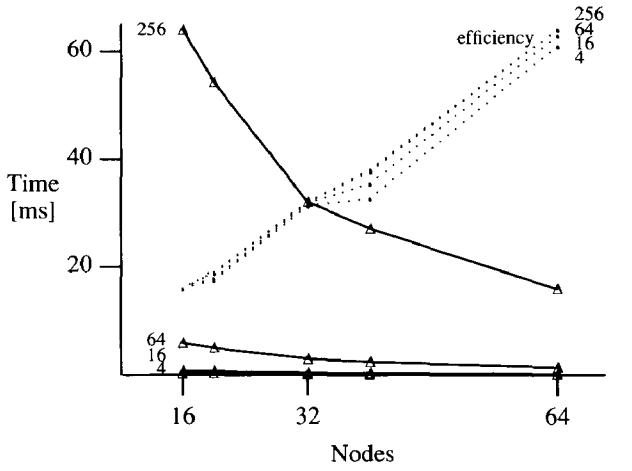

Figure 11 Parallel performance: Phong-shaded z-buffered spheres.

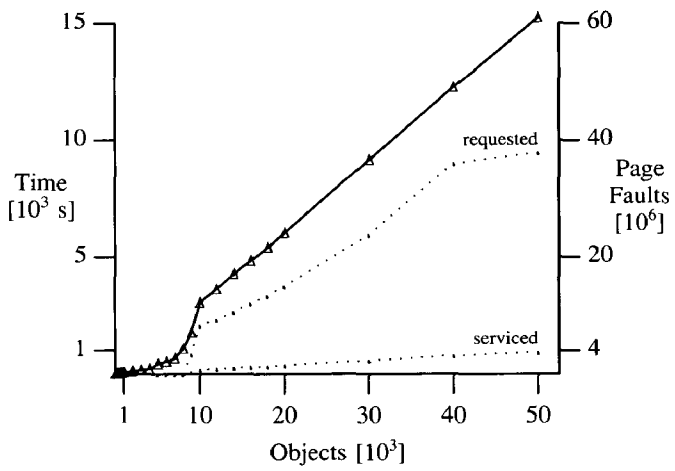

Figure 13 Ray tracing: scene complexity vs. time (solid) and parallel paging (dotted) using virtual data memory.

- **sphere rasterizing**

  Phong-shaded z-buffered spheres are rasterized using the approximations described in [7]. Because many computations have to be done at each pixel to evaluate the inside/outside equation, depth, normal vector, and finally the color, the efficiency of this algorithm is high even for small spheres [Figure 11]. Note that the implementation of this algorithm is more efficient then, for example, Gouraud-shaded z-buffered rasterizing of polygons of similar size.

- **ray tracing**

  Our first objective was to see if the performance would improve linearly with increases in the number of processing nodes for the case where the object database resides entirely in a node's memory. In these tests, the database size is kept constant while the number of pixel nodes in a system is increased. As can be seen from the dotted graph in Figure 12, our objective of linear improvement was met, and we have had similar experiences with many other object databases. In addition, the actual rendering times for the image are plotted (solid graph) and they are two to three orders of magnitude faster than on typical workstations.

Our second objective was to examine what happens to the performance when the parallel paging is used for virtual display lists. This test was run keeping the number of pixel nodes in the system constant, while increasing the number of objects in a scene. It can be seen from the graph in Figure 13 that the paging begins at about 5,000 objects in the display list. The performance degrades exponentially when the paging begins but becomes again linear above 10,000 objects. The dotted graph labeled "requested" shows the number of page faults generated by all the nodes. Since many nodes request the same page at the same time, the dotted graph labeled "serviced" shows that only about one tenth of the generated page faults had to be serviced. The speed of the algorithm when paging a display list is about five times slower than when all of a display list is in the pixel-node memory.

The measured performance confirms our analysis: the architecture of the machine is best suited for image-computing algorithms which require a parallel overhead only once per image (e.g., ray tracing, fractals, 2D and 3D solid textures) and degenerates as the overhead increases and the number of usefully employed pixel nodes decreases (e.g., line and point rasterizing).

6. Summary and Conclusions

We have described a parallel image computer designed for fast geometry and image computing. The computer contains a large distributed frame buffer which allows many computing elements, capable of floating-point operations, to access pixel-oriented data in parallel. We have developed software for standard 3D polygonal graphics, 3D volume display and ray tracing, all based on a common programming environment.

To overcome the problems inherent to the architecture of the machine and its current implementation - particularly the limited amounts of program and data memories in each node - we resorted to using established software techniques found in traditional computers such as program overlays for instructions and virtual memory for data. We also found a method to remove the restrictions of the interleaved frame-buffer design using interprocessor communication capabilities. To simplify the development of software for our scalable parallel architecture we have developed a concept of physical and virtual nodes which makes the size of the machine transparent to the programmer.

We have also used the Pixel Machine for real-time playback of compressed audio and video data and as a general-purpose parallel computer.

Acknowledgements and Credits

We would like to thank Bill Ninke, Kicha Ganapathy and Jim Boddie for providing a fertile environment that allowed the exchange of ideas between people involved in graphics, parallel processing and digital signal processing. Leonard McMillan contributed major ideas to both the software and hardware architecture and should be identified as one of the principal architects of the system. Bob Farah should be credited with the design of the pipeline card and for handling enormous numbers of odds and ends. Marc Howard should be thanked for bringing to life very high-quality, reliable video at 2 A.M. on a Friday night. Jennifer Inman should be thanked for writing a great deal of the pipe and pixel-node software. Pete Segal must be credited with much work on the ray tracer. Jon Leech pulled his hair on message-passing and the interleave/de-interleave code. John Spicer and Tom Rosenfeld contributed towards a nice parallel-programming environment.

Miss Piggy was an immense help in the early days when everything great was done at around 4 A.M. and common sense prevailed. Spouses and lovers are most importantly thanked for being understanding at the worst of times. The controversial ghost-writer Tango A. Scampers wrote the original version of this paper.

We would also like to thank *NASA* for generating the image in Figure 2 and for providing the database for testing purposes. Kamran Manoocheri must be thanked for creating the image *A Museum Room* in Figure 4 and Leonard McMillan for *A Tea Room* in Figure 3.

References

[1] AT&T Pixel Machines, "The Pixel Machine System Architecture," A Technical Report, Holmdel, NJ, November 1988

[2] Clark, J. H., "The Geometry Engine: A VLSI Geometry System for Graphics," *ACM Computer Graphics*, **16**, (3), July 1982, 127-133

[3] DeBenedictis, E. P., *The Bell Laboratories' Hypercube*, personal communication, April 1986

[4] Denning, P. J., "Virtual Memory," *Computing Surveys*, **2**, (3), September 1970, 153-189

[5] Dippé, M., and Swensen, J., "An Adaptive Subdivision Algorithm and Parallel Architecture for Realistic Image Synthesis," *ACM Computer Graphics*, **18**, (3), July 1984, 149-158

[6] Fuchs, H., "Distributing a Visible Surface Algorithm Over Multiple Processors," *Proceedings of ACM 1977*, Seattle, WA, October 1977, 449-451

[7] Fuchs, H., et. al., "Fast Spheres, Shadows, Textures, Transparencies, and Image Enhancements in Pixel-Planes," *ACM Computer Graphics*, **19**, (3), July 1985, 111-120

[8] Heising, W. P., and Larner, R. A., "A Semi-Automatic Storage Allocation System at Loading Time," *Communications of the ACM*, **4**, (10), October 1961, 446-449

[9] Hillis, W. D., *The Connection Machine*, The MIT Press, Cambridge, MA, 1985

[10] Kershaw, R. N., et. al., "A Programmable Digital Signal Processor with 32-bit Floating Point Arithmetic," *Proceedings of IEEE International Solid-State Circuits Conference*, February 1985, 92-93

[11] Levinthal, A., and Porter, T., "Chap - A SIMD Graphics Processor," *ACM Computer Graphics*, **18**, (3), July 1984, 77-82

[12] Levoy, M., "Volume Rendering: Display of Surface from Volume Data," *IEEE Computer Graphics and Applications*, **8**, (3), May 1988, 29-36

[13] McCormick, B. H., DeFanti T. A., and Brown, M. D., "Visualization in Scientific Computing," *ACM Computer Graphics*, **21**, (6), November 1987

[14] Parke, F. I., "Simulation and Expected Performance Analysis of Multiple Processor Z-Buffer Systems," *ACM Computer Graphics*, **14**, (3), July 1980, 48-56

[15] Potmesil, M., and Hoffert, E. M., "FRAMES: Software Tools for Modeling, Rendering and Animation of 3D Scenes," *ACM Computer Graphics*, **21**, (4), July 1987, 85-93

[16] Potmesil, M., McMillan, L., Hoffert, E. M., Inman, J. F., Farah, R. L., and Howard, M., "A Parallel Image Computer with a Distributed Frame Buffer: System Architecture and Programming," *Proceedings of Eurographics '89*, Hamburg, Federal Republic of Germany, September 1989

[17] Seitz, C. L., "The Cosmic Cube," *Communication of the ACM*, **28**, (1), January 1985, 22-33

[18] Sato, H., et. al., "Fast Image Generation of Constructive Solid Geometry Using a Cellular Array Processor," *ACM Computer Graphics*, **19**, (3), July 1985, 95-102

[19] Whitton, M. C., England, N., and DeMonico C., "Manage Design Trade-Offs in High-End Graphics Board," *Electronic Design*, **36**, (6), March 1988, 77-84