A Simple, Flexible, Parallel Graphics Architecture

John Amanatides †, Edward Szurkowski ‡

† Dept. of Computer Science, York University
North York, Ontario, Canada M3J 1P3
amana@cs.yorku.ca

†‡ Computer Systems Research Lab
AT&T Bell Labs, Murray Hill, NJ, USA 07974

ABSTRACT

Traditional graphics hardware architectures, with their emphasis on the graphics pipeline, are becoming less useful. As graphics algorithms evolve and grow more capable, it becomes much harder to implement them in silicon. By using general-purpose hardware technology effectively, one can build powerful graphics hardware that is very flexible, yet inexpensive. In this paper we would like to discuss one such architecture that allows for both traditional interactive graphics (polygon scan conversion) as well as more advanced graphics (ray tracing and radiosity).

KEYWORDS

computer graphics hardware

INTRODUCTION

The graphics pipeline has had a long history in computer graphics [1]. Consisting of a front end that performs simple, repetitive floating point calculations on short vectors (for transformations, clipping, perspective) and a back end that scan converts primitives into pixels and determines visibility, it easily became a candidate for hardware acceleration [2, 3, 4]. However, as graphics algorithms evolve and become more capable the utility of this specialized hardware is reduced. The required functionality can no longer be incorporated easily in hardware and instead must increasingly be performed in software on the host system.

VLSI technology is squeezing more and more onto a chip. However, designing a special-purpose chip is getting harder as more functionality is added. This is especially true of custom graphics chips. Most effort in semiconductor houses is now being put into creating more powerful microprocessors and ever larger DRAMs and VRAMs.

In this paper we want to explore an architecture that tries to take advantage of the growing power of VLSI by concentrating on these general-purpose products and using them effectively. We wanted an architecture that minimized graphics hardware. The result, we believe, is an inexpensive, powerful and flexible system.

As well, we will describe a design that utilizes this architecture. The origin of this design came from our experiences with the AT&T Pixel Machine. We wanted to produce a low-cost next-generation machine. Let us first review architecture of the Pixel Machine.

THE PIXEL MACHINE

The Pixel Machine (PXM) was designed as a programmable computer subsystem with pipeline and parallel processing closely coupled to a display system [5]. Built by Pixel Machines Corp, a subsidiary of AT&T, it was launched in 1987. An important design goal was flexibility. Graphics algorithms were not hardwired into the design. Instead, digital signal processors (DSP32) were the basic building blocks (nodes) of the PXM. This allowed for a lot of flexibility as new functionality could be programmed in afterwards. In fact, most of the algorithms were written in C with only the critical sections written in assembler. This resulted in a product that was ideal for research and development.

The PXM consisted of a large box (containing up to 20 VME boards) which was connected to the host computer via a series of registers in the memory address space of the host computer. Data and commands reaching the PXM would first be sent through a pipeline board consisting of nine DSP32 *pipe nodes*. For interactive graphics, the pipe nodes would do transformations, clipping and lighting calculations for the various graphics primitives. A second pipeline board could be added to the PXM to increase performance.

Next, the primitives would be broadcast to 16-64 (depend-

ing on options) DSP32 *pixel nodes* whose job it was to rasterize the primitives. Each pixel node contained an interleaved portion of the frame buffer (every eighth pixel of every eighth row in the 64 processor version). As well, it had 32KB SRAM (for program storage) as well as 512KB of VRAM (double-buffed frame buffer, texture map, accumulation buffer) and 256KB of DRAM (z-buffer). Finally, the pixel nodes were each connected to their four neighbours via a serial link.

For ray tracing and image processing, the pipe nodes were mostly unused; the work was done by the pixel nodes. When running ray tracing programs we found that we could get two orders of magnitude performance improvement over the workstations that the PXMs were connected to (in our case, a Sun 3/260).

Unfortunately, the PXM was expensive. A fully configured machine could cost up to to $150K; it was not something every scientist could have in their office. It had several other limitations. The DSPs had a limited address space. This fit into their original purpose (signal processing typically has small code) but we quickly ran into limitations when writing graphics programs with sophisticated shading. More memory was needed for both programs and data. The two-dimensional interleaved design of the PXM had drawbacks. Scan converting small polygons meant that each processor only drew a few pixels per polygon. However, every processor has to do all the edge setup calculations and amortizing this over only a few pixels is expensive. When performing interactive graphics the bottlenecks in the system would change depending on the types of rendering involved. If there was texture mapping, the pixel nodes were the bottleneck. If only simple shading was performed, the pipe nodes were the bottleneck.

DESIGN GOALS

In the summer of 1989, after extensively using the PXM for a variety of graphics research projects at AT&T Bell Labs, we decided to design a similar machine that we could afford to place in each of our offices. As well, Pixel Machine's next generation machine was ambitious and late and we wanted to offer them an option. The stress was on low cost, flexibility and power.

Like the PXM we wanted the new machine to be flexible; we wanted something to perform both interactive graphics and ray tracing fast. Our interactive performance goal was over 100K independent polygons/sec.† Our ray tracing goal was a performance level over one order of magnitude faster than the workstations that it would be connected to.

† Performance for independent polygons was important as some of our graphics research on BSP trees generated them.

It had to be cheap; the target was a list price of $25K. Ideally, it would be a single board that would fit into our workstations.

It needed lots of memory; we had already run out of memory for programs and needed more for textures and models.

Finally, we wanted to explore a minimalist design; Gordon Bell has said that "the cheapest, fastest and most reliable components of of computer system are those that aren't there." We wanted to see if we could apply this to graphics hardware.

DIFFERENT ARCHITECTURES

Figure 1 illustrates the typical graphics subsystem [4, 5].

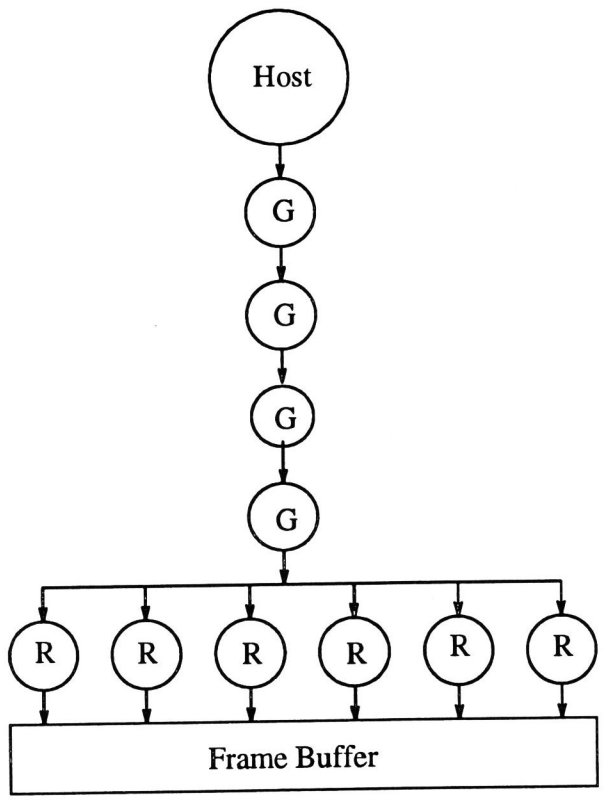

Figure 1

The host traverses the model data base and sends the graphics primitives to the graphics pipeline. The front end, which performs a series of geometric operations, is typically implemented as a pipeline of floating-point ALUs (labeled G). (The FIFOs typically found between stages are left out of the diagrams) Afterwards, the transformed/shaded primitives are sent to the back end which performs rasterization and visibility operations. Because interactive rasterization requires a great deal of pixel

throughput, parallel access to the frame buffer is required; the back end is typically implemented with multiple rasterization processors (labeled R) each being responsible for a portion of the frame buffer.

A variant approach, illustrated in figure 2, is increasingly being explored [6, 7, 8].

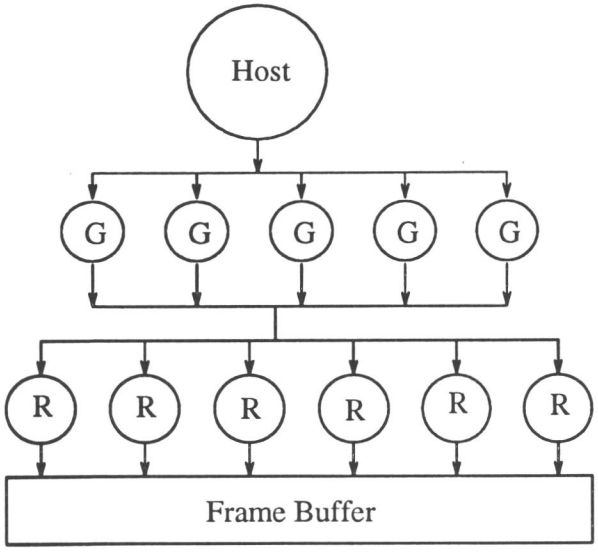

Figure 2

Here, the front end, instead of being a pipeline of very simple processors, has become populated with more powerful processors, each capable of performing all the required front-end operations on a single primitive. Graphics primitives are distributed in a round-robin manner to each front-end processor, processed, and then broadcast to the back end. Because the processor is more powerful, better shading algorithms can be utilized. Also, more powerful graphics primitives can be dealt with (splines, for example) and tessellation can occur further down the pipe. As well, there is less data movement amongst the various stages in the pipe (analysis of the PXM indicated that the pipe nodes spent a significant amount of time on this). Several processors are typically required to keep performance up and care must be taken to make sure primitives stay in priority order.

The general outline of our new minimalist architecture is found in figure 3. Here, the functions of the front and back ends are collapsed onto the same processors. In this case, the host distributes primitives in a round-robin manner to each of the processors. Each processor performs the front-end tasks of transforming/clipping/lighting/perspective for the primitives. It then broadcasts the results to the other processors to perform the back end operations of rasterization. Because, the primitives are distributed in this manner, each processor does the same back-end operations as in previous designs but only 1/*n*th of the front-end operations (assuming there are *n* processors).

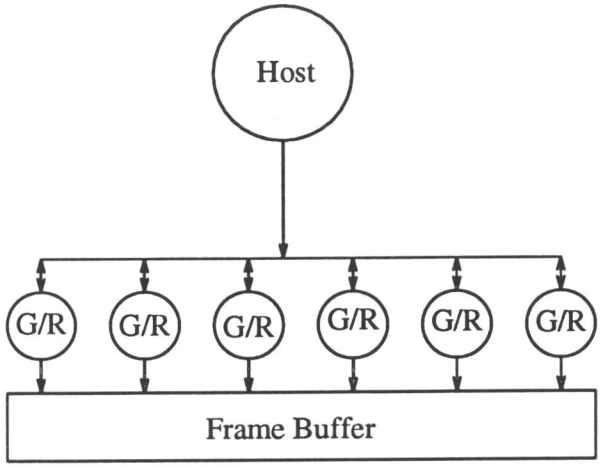

Figure 3

There are several advantages to this approach. First, the design is auto load balancing. For example, if there is a lot of texture mapping, the processors spend most of their time on this. If most of the primitives are simple, then the processors spend the majority of their time on the front end computation. Alternately, if the computation is ray tracing, no front-end processors are wasted idling. Second, the approach tends to be simpler, requiring powerful general-purpose processors instead of special-purpose graphics hardware. Because VLSI technology is making these single-chip microprocessors increasingly available this results in a smaller, simpler board layout. Finally, it is a more flexible approach. Since the hardware pipeline is not optimized for a particular algorithm, there is a lot of freedom to change or enhance capabilities. For example, higher-order primitives can be used (NURBS). If necessary, the model can be distributed amongst the processors (assuming that they have the memory) without introducing inefficient asymmetries.

THE PXMjr DESIGN

As mentioned earlier, we wanted to produce an inexpensive version of the PXM and began in the summer of 1989. We gave the new design the name Pixel Machine Junior (PXMjr). Like the PXM the PXMjr would be a peripheral; this would keep the design simple and would allow it to be connected to a variety of workstations. This would also result in a fast turnaround time from design to production.

We also decided that to keep it simple and small, we preferred fewer, more powerful processors. Thus we could have more memory per processor without incurring high costs.

The new design would require more memory for each processor; a lot of memory is needed as textures and geometric models, if they are stored locally, are space intensive. As well, program space needs to be larger as each processor does more work. To keep costs down, we wanted to eliminate the need for SRAM, and use DRAM instead. DRAM also has the advantage that a lot more memory can be placed per unit area. Unfortunately, DRAM is slower. To get back the speed, we wanted the processor to have an on-chip cache. Because of typical DRAM chip layouts and bus widths (64 bits), we would need at least 16 chips (nibble data paths). This would result in either of 2 MB (1Mb chips) or 8 MB (4Mb chips) per processor. A similar number of VRAM chips would be required for the distributed frame buffer. At the time 4Mb VRAM chips weren't available; otherwise we would have considered just using VRAM. We felt that the simplified design and halfed chip count may have been worth it.

After exploring several DSP microprocessors, we decided on the Intel i860. It is a powerful general-purpose microprocessor optimized for graphics. According to Intel engineers we could expect 20-25K poly/sec from each processor. It has both a powerful floating-point unit (80 MFlops peak) and on-chip caches. We realized that this was a change from the DSP32s in the PXM but since most of our software was written in C, we felt that we could make this change without a big penalty.

The processors would be connected to each other and the host via a bidirectional FIFO connected to a message passing bus. These FIFOs would help keep each processor busy and help perform the necessary multicasting without processor intervention. The message bus has the capability of both high and low priority packets (the need for which we will describe later).

Like the PXM, the host sees a set of registers in its address space which implement a bidirectional FIFO and control the PXMjr. It can tell if its input FIFO is empty, its output FIFO is full and can read and write from the FIFOs. It can also reset the PXMjr and there is a mechanism for syncing with the processors (to synchronize the completion of each frame). The host interface was kept simple so that it could easily be adapted to multiple host types.

A general outline of the new design is found in figure 4. It consisted of 8 i860 processors, each with 2 MB of VRAM and either 2 or 8 MB of DRAM.

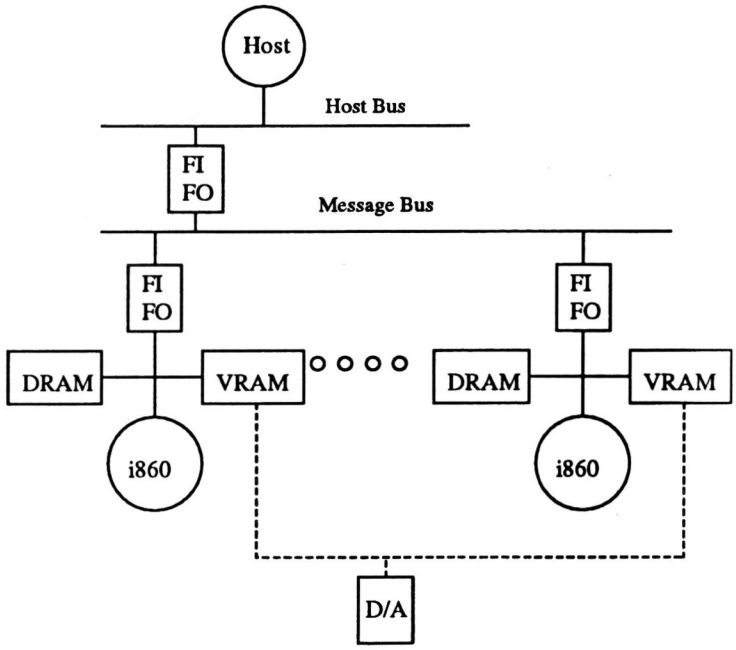

Figure 4

THE FRAME BUFFER

There is a virtual 2Kx2Kx32 frame buffer distributed amongst the 8 processors in a column interleaved manner. The video rate is programmable from RS-170A (NTSC resolution) to 1280x1024 non-interlaced.

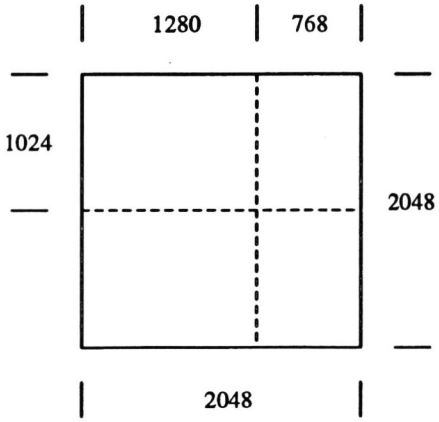

Figure 5

As it is column interleaved, the frame buffer reduces the pixel rate from the individual VRAMs by a factor of 8. Also, it was found that in the original PXM there was significant wasted computation because the edge set-up costs

for polygon scan conversion were not amortized over enough pixels. By interleaving in only one dimension, amortization is increased. Finally, it simplifies anti-aliasing.

Each processor has 2 MB of VRAM (16x1Mb nible-mode chips). The 2Kx2K allows for double buffering at 1280x1024 resolution. As well, the extra 768 pixels in each scan line can store a 16 bit z-buffer (see figure 5). By careful optimization we can use the VRAM serial buffer to reset the frame buffer to the background colour and reset the z-buffer during vertical blanking. This is done in conjunction with the i860 executing a very tight loop to generate the appropriate addresses on the local bus.

THE BUS

The message bus is a conservative design. Originally designed to match the VME bus, and then the S-bus, it is 32 bits wide; running at 10 MHz it provides a raw transfer rate of 40 MB/sec.

The bus transfers fixed-length packets of 32 32-bit words. The size was chosen so that triangles and quadrilaterals would fit into one packet and a constant-size packet simplified the design. The first word in the packet indicates the destination. With 1 bit per processor, it allows for multicasting. There are two types of packets: high and low priority packets. Low priority packets get onto the bus only if no high-priority packets are waiting to be sent in any of the other FIFOs. As well, there is a fair scheduling policy in which a second packet from any FIFO cannot get onto the bus until all other FIFOs are first given a chance to send theirs.

Running at 10MHz, 2 time slots are required for arbitration. This results in a maximum flow rate of 294K packets per second. Since each polygon would have to traverse the bus twice, the maximum number of polygons that can be handled is about 145K. This fits well with the 100K poly/second design goal.

The FIFOs can be implemented with two 1Kx18 bidirectional FIFO chips [9]. These chips can be programmed during reset so that they raise signals at various levels of filling. (This will be used to detect if a full packet is ready and in the deadlock prevention scheme described below). There is room for 32 packets in each FIFO. This allows for considerable incoming work.

INTERACTIVE GRAPHICS

To help understand our design let us describe what would happen when rendering polygons interactively.

The host busy-waits until its output FIFO has space and then starts sending packets containing polygons. These are sent in a round-robin fashion to each of the processors (the i860s). The host continues until it completes traversing the model and then waits until the processors are finished.

Each processor waits for packets in its input FIFO. Packets containing polygons come in two flavors: geometry messages (GM) and rendering messages (RM). The GMs come from the host and their polygons are transformed, clipped, shaded and then sent out as RMs via the output FIFO to the other processors. When a processor receives a RM it scan converts the polygon and draws it in its frame buffer.

Since processors complete their tasks of converting GMs into RMs at different rates it is possible that RMs arrive at processor in the wrong priority order. The processor has to sort this out; it may have to store up to 7 RMs (this is as far off as the ordering can get).

DEADLOCK

Since every processor can write into every other processor's input FIFO at the same time there is a remote possibility of deadlock. Consider the following scenario: The host is so fast that it fills all the input FIFOs of all the processors with GM packets. Each processor converts the GM into a RM and puts it in its output FIFO. Their output FIFOs start to fill. What happens if its output FIFO becomes full? If the input FIFO has a RM it is consumed (good). However, if the next packet is a GM it must eventually be sent out and the processor blocks because there is no more room in its output FIFO. If something like this happens at several processors deadlock occurs.

The best way to handle deadlock is to design it out in the first place. In our design we can guarantee that no deadlock will occur if the output FIFO at each processor is not full. We have to make sure that this never happens. RM packets are consumed; they will not cause trouble. It is the GM packets that must be taken care of.

As was mentioned earlier packets on the bus have two priorities, low and high. low priority packets cannot get onto the bus if high priority packets are waiting. If we map GMs to low priority packets and RMs to high priority packets then the solution is in sight.

The worst possible scenario is when every input queue is full of GMs. As soon as one of these GMs turns into a RM it shuts down further introduction of GMs from the host. The current GMs are slowly turned into RMs and wait in the output FIFOs. As long as the output FIFOs are at least as big as the input FIFOs then each processor can do useful work. Whenever space appears at any input FIFO it is the RMs that are delivered to it. Eventually, all the GMs will be turned into RMs and will be delivered into their appropriate input FIFOs. Now, when all the output FIFOs are empty and the processors are working on the remaining RMs in their input FIFOs can the host begin to deliver GMs again. What will eventually happen is that each processor will oscillate between working on RMs and GMs; in both cases, it will be doing useful work. The long FIFOs (room for 32 packets) make sure that the processors are kept busy.

PACKAGING

The preliminary design called for a single 9VME board (why pay for cabinet/power supplies?). But because of changing environments (fewer people were purchasing large-chassis workstations) a "pizza box" design was chosen. The case, 16"x16"x3", was designed to fit under a SPARCstation, with a ribbon cable extending into an S-Bus slot in the SPARCstation. (Because of the simple host interface, adapters were contemplated for other machines). In the chassis there would be room for one large circuit board, along with power supply and fan. The preliminary design also called for the possibility of a daughter board with 8 more processors but power dissipation and complexity problems stopped this at an early stage. There is analog RGB and sync out and genlock in.

CONCLUSION

In this paper we have introduced an architecture that allows for simple, flexible yet powerful graphics hardware for both interactive graphics (polygon scan conversion) and more advanced graphics (ray tracing, radiosity). This is accomplished by not dedicating hardware to specific tasks but allowing processors to both transform and render polygons. As well, a design, consisting of eight i860s, with local memory, and a double-buffered 1280x1024 frame buffer was outlined. This design was finished in the spring of 1990 and a small prototype was built. However by this time the first author had left AT&T and Pixel Machines Inc. decided not to continue with development.

We would like to thank Don Mitchell and Bruce Naylor for many valuable suggestions.

References

1. J.D. Foley, A. Van Dam, S.K. Feiner, and J.F. Hughes, *Computer Graphics: Principles and Experience,* Addison-Wesley Publishing Co, Reading Mass, 1990.
2. T. Myer and I Sutherland, "On the Design of Display Processors," *Comm. of the ACM*, vol. 11(6), pp. 410-414, June 1968.
3. J. Clark, "The Geometry Engine: A VLSI Geometry System for Graphics," *Computer Graphics*, vol. 16(3), pp. 127-133, July 1982.
4. K. Akeley and T. Jermoluk, "High-Performance Polygon Rendering," *Computer Graphics*, vol. 22(4), pp. 239-246, August 1988.
5. M. Potmesil and E.M. Hoffert, "The Pixel Machine: A Parallel Image Computer," *Computer Graphics*, vol. 23(3), pp. 69-78, July 1989.
6. J.G. Torborg, "A Parallel Processor Architecture for Graphics Arithmentic Operations," *Computer Graphics*, vol. 21(4), pp. 197-204, July 1987.
7. D. Kirk and D. Voorhies, "The Rendering Architecture of the DN1000VS," *Computer Graphics*, vol. 24(4), pp. 299-307, August 1990.
8. M. Mehl and H. Joseph, "GRACE: The Graphics Coprocessor Engine of the EuroWorkStation (EWS)," *Eurographics '90: Proceedings of the Graphics and Interaction in Esprit Sessions*, September 1990.
9. IDT72521, *1Kx18-Bit CMOS BiFIFO, Preliminary Data Sheet,* Integrated Device Technology, Inc, January 1989.