A Study on Architechtural Approaches for High Performance Graphics Systems

George Kyriazis

Rensselaer Design Research Center

Rensselaer Polytechnic Institute

Technical Report No: 90041

Contents

List of Figures

1. Introduction

Computer technology has become an integral part of our society today. Computer speed has increased to a point where it is used to simulate natural phenomena with moderate success. The amount of computation that has been performed during the last few years is much larger than all the calculations that mankind has managed to make during its development.

We are at a point in the history of computing, where the manufacturing of complex computing machinery has immediate effects in society, and computer science in not considered a purely theoretical discipline anymore.

The amount of data that is produced from such machines needs a certain amount of preprocessing to be fully comprehended by humans. That preprocessing is required since the data was produced for them to use in the first place. The only human sense that is able to easily comprehend such enormous amounts of data is vision. Computing equipment is traditionally trying to use visual feedback to improve interaction. Front panel displays were followed by ASCII terminals which were followed by graphics and windowing systems.

For a computer interface to be effective, it has to communicate with its user at a rate fast enough to give him a feeling of interaction, but not too fast in order to avoid communication overload.

This paper will not deal with the human-machine interface problem, but with the internal structure of the computing machine that produces the visual representation of the users' data. Although modern graphics equipment is very fast, it does not have the flexibility required for high-speed interactive simulations.

A review of current graphics architectures will first be presented. Emerging future needs will then be evaluated. Finally a solution for the coupled problem of graphics and simulation will be presented.

2. Today's Hardware

Most of the graphics engines available today have very good performance at one application, with mediocre performance at others. The concept of the graphics pipeline is used in all of them, mainly because all graphics standards available today utilize it.

In this section I will describe several of those machines. The discussion will emphasize the data flow through the architecture and the algorithms that each architecture executes efficiently.

2.1. The Alliant GX4000

This architecture was designed to deliver high performance using the proposed PHIGS+ graphics standard. PHIGS+ uses the concept of a *structure*, which is a serial list of primitives, attributes or non-recursive calls to more structures. The building of these structures is performed at runtime, so no compile-time parallelization can be performed. The GX4000 tries to parallelize the rendering of the PHIGS graph traversal at runtime. It is an attached processor that resides on several VME bus slots of a Sun-3 or Sun-4 computer.

The machine is divided into 3 main sections: The Display List Manager, the Geometry Arithmetic Processors and the Image Memory Unit. The Display List Manager (DLM) keeps its own copy of the PHIGS hierarchy and updates the screen when it is needed, by traversing the tree and sending commands and data to the Graphics Arithmetic Processors (GAPs) using a specialized bus called the CMD bus. The GAPs perform all transformation and clipping operations on the primitives sent to them. All the functions of the graphics pipeline are performed at each processor. These processors, in turn, send their results to the Image Memory Unit (IMU) which acts as the renderer for the architecture. The number of GAPs in the system can range from 1 to 8. The Image Memory Unit is able to render about 1 million vectors per second or 330,000 rendered triangles per second [Tor87].

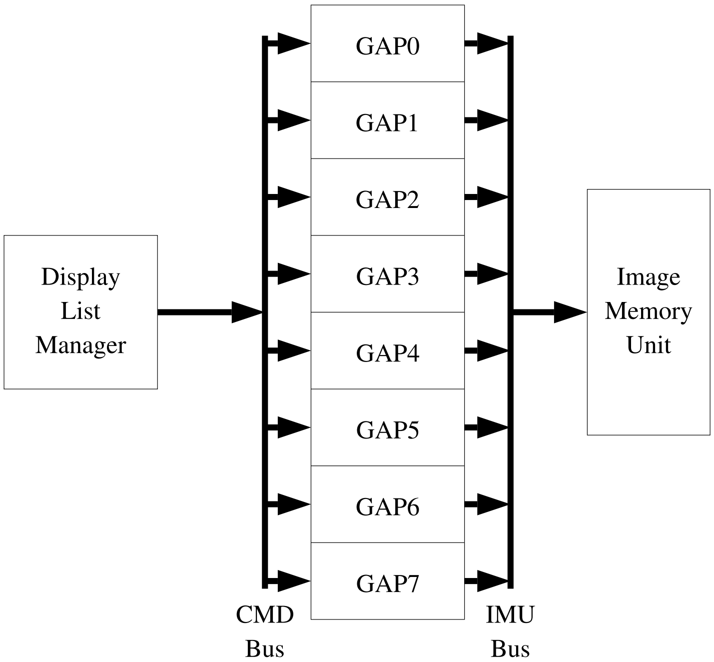

Figure 1. The GX4000.

There are two types of instructions that can be sent to the GAPs. *Primitives* and *Commands*. Primitives include lines, triangles and polygons and can be rendered in any order provided that they all have the same attributes. Commands include transformation matrices, light source information and line types and have to be broadcasted to all the GAPs because they affect all subsequent primitives.

When a large number of commands corresponds to a small number of primitives in a PHIGS structure, the efficiency of a multiple-GAP machine can be lowered to the performance of a single-GAP machine. The quoted GAP utilization figures range in the low 90%'s for complex models. It has been observed that the processor utilization can drop to much lower levels; in fact the average utilization of each processor is less than 90%.

The GX4000 tries to solve the graphics problem by parallelizing the graphics pipeline. Due to probabilistic effects, the GAPs will not produce results at the same time, so a bottleneck at the IMU bus probably won't occur. Compared to the standard pipeline approach, this method is much more flexible, since it neither limits each processor to perform only one operation, nor restricts the algorithm to be executed using a rigid pipeline. For example, efficient adaptive tessellation curve rendering algorithms [Abi89], need a feedback loop on the rendering path, which can be achieved using the GX4000.

On the other hand, the GX4000 is limited in performing interactive graphics, with no future in image processing or ray-tracing algorithms. The GAPs themselves do not contain enough memory and there is no way to feed back their results to DLM memory. Windowing operations, on the other hand, can be easily performed using the IMU's bitblt hardware. It should be noted though, that usually the IMU is a bottleneck for primitive drawing.

2.2. The Silicon Graphics IRIS

The IRIS was designed to perform fast real-time display of shaded polygons. The initial design goal was to obtain a speed of 100,000 shaded polygons per second, 10 frames per second and window support. An analysis concluded that 46.5 MFlops are needed for that task to be successful. The resulting architectural solution fulfills the goal. It contains four pipelined subsystems: The Geometry subsystem, the Scan Conversion subsystem, the Raster subsystem and the Display subsystem[AJ88].

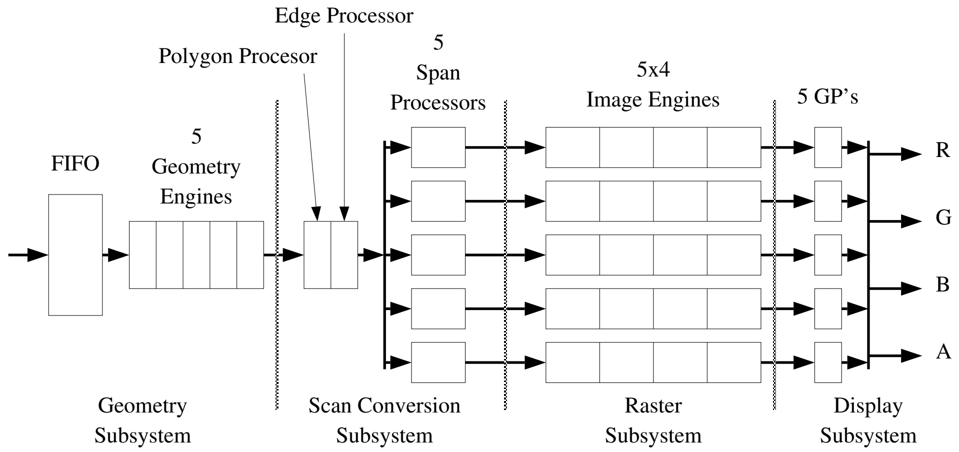

Figure 2. The Silicon Graphics IRIS.

The Geometry Subsystem gets the data from the host CPU and performs all floating point calculations to produce the 2–D points necessary for line and polygon rendering. Polygons are represented by their vertex points. Complex surfaces have to be tesselated into polygons by the host processor. Five geometry processors can sustain up to 100MFlops performance; about double of what is needed for the desired result. The FIFO queue at the input of the pipeline tries to keep the geometry pipeline full. The graphics tasks are distributed among the 5 geometry engines as follows: Matrix and normal vector calculations(25 Mflops), lighting calculations(11 Mflops), clip testing(2.5 Mflops), perspective transformations and clipping(4.5 Mflops), and finally viewport transformation, color clamping and depth cuing(3.5 Mflops). It can be easily seen that the first geometry engine, which performs matrix and normal calculations, is a system bottleneck.

In the Scan Conversion subsystem, polygons are translated into color points in the plane. The polygons processor decomposed polygons into convex triangles and performs slope calculations for y, z, r, g, b, and α (not for x) between the vertex points. The resulting horizontal spans are fed into the edge processor that interpolates these values on the x-axis to produce horizontal spans of actual pixel values. Interpolation is also done on the color values resulting in Gouraud shaded[1] polygons. Phong shading is not supported in hardware, and, if required, has to be simulated in software in the main processor lowering performance significantly. The pixels are then delivered to the 5 span processors, each one serving every other 5th column. Because of the interleaving method used, load balancing is achieved. The span processors then produce the actual pixel values from the spans.

The Raster subsystem contains 4 image engines for each span processor, interleaved every forth scanline. The frame buffer is distributed among the 20 image engines. There are 96 bits per pixel used. Two banks of 32 bits produce double buffering with 24 bits color plus α—mask for each buffer. 24 additional bit planes are used for Z-buffering. 4 more bits constitute an overlay/underlay mask and the final 4 keep the window ID for that pixel. Each image engine is capable of performing graphical replace (80 Mpixels/sec), Z-buffer store (40 Mpixels/sec), Z-buffer blend for transparency effects (10 Mpixels/sec) and high-speed clear (160Mpixels/sec).

[1] In Gouraud shading, the color of the pixel is determined by interpolating the color values between the corner points of the polygon.

Finally, the Display subsystem performs color lookup operations and routes appropriate bit planes to the output based on the overlay mode and/or window ID. This allows each window to have different characteristics, which increases the flexibility of the system.

Details on the operation are omitted, but the reader is encouraged to refer to [AJ88] for further information.

Although the Silicon Graphics IRIS is an excellent machine for real-time interactive windowing environments, its rigidity prevents many algorithms from taking full advantage of its power.

2.3. The AT&T Pixel Machine

This machine was built to give a solution to the graphics problem using a brute force approach. It assumes that the bottleneck in graphics computing is the floating point calculations. Ample computing power is provided using 82 Western Electric DSP32 Digital Signal Processors, each providing a peak performance of 10 MFlops. If all processors are utilized 100%, we get a peak performance of 820MFlops, very close to the performance of a Cray 2. In reality though, sustained performance can be in the range of 5–6 mips per processor (both integer and floating point) giving an overall average speed slightly less that 500 mips, which is still very respectable.

The machine is an attached processor connected to a VME bus. The organization of the CPUs internally reflects that design. 64 out of the 82 processing elements are organized in an 8x8 array. That array is fed by a broadcast bus from the remaining processors, which are organized either as one pipeline of 18, or two pipelines (in parallel) of 9 CPUs. The processors that make up that array are called the "pixel" nodes, while the processors that are used in the pipeline are called the "pipe" nodes. The VME bus has the capability to write data to any processor using programmed handshakes or DMA, or to send data to the top of the pipeline.

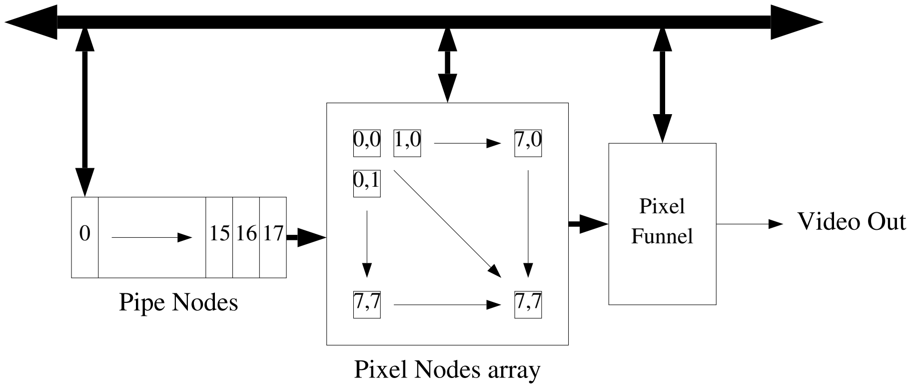

Figure 3. The AT&T Pixel Machine.

The pixel array acts as the "renderer" engine and the pipe nodes are used for a graphics pipeline. This is true, whether or not the Pixel Machine is performing real-time Z-buffer rendering or ray-tracing. The detailed operations on each node differ depending on the task. Two system libraries are included. One of them (PIClib) is meant to perform PHIGS-like functions, while RAYlib is used for ray tracing. The function calls for both libraries are extremely similar, with the main difference being that PIClib functions are executed interactively, while RAYlib calls are not, due to the nature of the ray-tracing algorithm.

When using PIClib, the pipeline is usually configured as parallel (2 pipes with 9 nodes each) with each processor executing a function of a typical graphics pipeline system, such as primitive transformations or clipping. The pixel nodes are just used as a very powerful rasterizing engine. Since the processors are commercial DSP's, they do not perform communication functions efficiently, which is vital in a pipelined system. When many vertices are involved, performance degradation is strongly noticeable, even if the total performance of the pipe nodes is significant (typical computational power of 100 mips).

On the other hand, RAYlib shows the full potential of the machine. Demo programs include close to real-time ray-tracing of a 128x128 pixel array using double buffering. With RAYlib, data is preprocessed using the pipe nodes (bounding volumes are calculated and surfaces are tesselated if required) and then broadcasted to the pixel nodes. To minimize communication overhead, data and code is duplicated in all pixel nodes. Due to the limited memory capacity

(768K per node including the frame buffers, and no shared memory), data has to be often paged in from the host resident in the VME bus which continuously polls the Pixel Machine. Based on the fact that the same page will be requested by many nodes in about the same time, pages to be swapped in are broadcasted to the pixel nodes, increasing paging performance [PH89].

As a conclusion, the Pixel Machine is an efficient ray-tracing machine with low interactive performance due to communication overhead. Each processor is individually programmable, which makes the system flexible. On the other hand, communication between processors is limited to a predefined pattern, allowing only a limited number of algorithms to be directly portable and efficient.

2.4. The Stellar GS1000

The Stellar GS1000 was built with Scientific Visualization in mind. Applications in this area need a lot of floating point performance and rapid changes in the hierarchical structure of the model to be rendered. Its internal design directly reflects that need. General purpose vector hardware is present, together with a specialized graphics subsystem. In fact, the CPU, the Vector Processor and the Rendering Processor are so tightly coupled, that the whole machine is characterized by Stellar as a “Graphics Supercomputer” [ABM88].

This is the first machine so far that was designed with the assumption that special purpose hardware does not make most of the (potential) users satisfied. High performance was therefore achieved using hardware that can be used for other operations apart from typical workstation applications. Since generalized hardware is not as fast as optimized, special purpose machines, very long data bus widths (up to 512 bits) are used internally to overcome the performance loss due to the generality of the architecture.

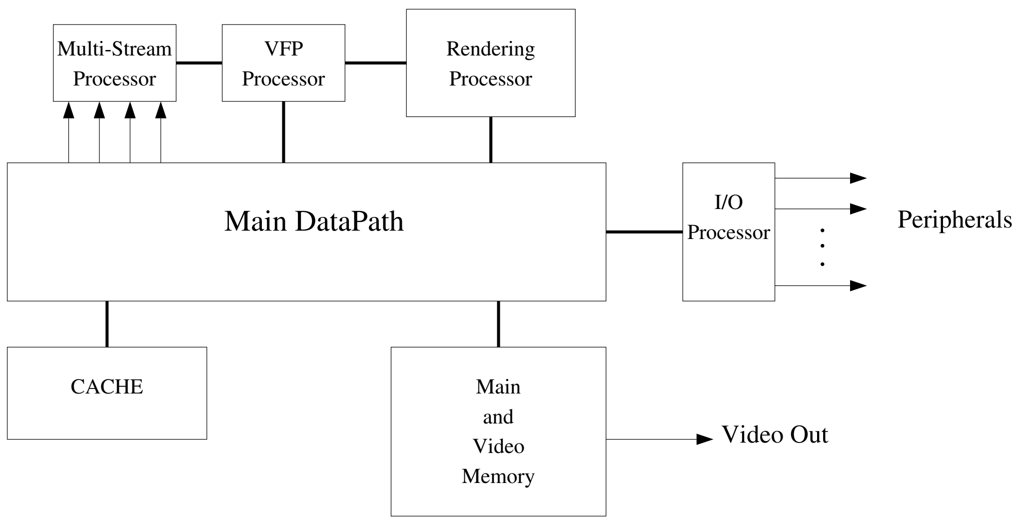

Figure 4. The Stellar GS1000 Graphics Supercomputer.

The Multi-Stream Processor executes instructions from 4 instruction streams simultaneously at a rate of 25 mips. The reference does not specify if this is the throughput of one or all of the instruction streams. The DataPath provides communication between all the subsystems and includes several vector, floating point and integer registers. The switching network in the DataPath is very complicated since it has to route signals on busses with widths of several hundred bits. The cache size is 1 Mbyte and the main and video memory size can range from 16 to 128 Mbytes depending on the configuration. The Vector Floating Point (VFP) Processor performs operations on vector data resident on the main DataPath with a peak performance of 40 MFlops.

What has been described so far is the general purpose computing hardware of the machine. The rendering processor, on the other hand, is the high performance, specialized graphics engine of the GS1000. It gets pixel coordinates from the VFP processor and produces pixel values that are then placed in memory. The internal architecture of the rendering engine will not be described in detail. It consists of three independently microprogrammed engines that operate from their own Writable Control Stores (WCS). The fact that they use a WCS for instruction storage provides some generality to the functionality of these processors, which are implemented in ASICs (Application Specific Integrated Circuits). Their fixed layout though, does not make them available as general purpose computing resources.

The Stellar GS1000 provides a significant improvement over previous graphics architectures by supplying general purpose hardware with a relatively flexible interconnection network. The system is tailored for real-time polygon rendering providing performance figures at the level of 600,000 vectors per second, 150,000 Gouraud-shaded polygons per second, or 40,700 Phong-shaded[2] polygons per second. It should be noted that this is the first machine that quotes performance for Phong-shaded polygons. The jump from Gouraud shading to Phong shading was made possible by the use of the general-purpose VFP processor. The architecture could support other complex time-consuming algorithms, such as ray-tracing, but no reference was made to such rendering algorithms.

2.5. The Pixel-Planes 5

The Pixel-Planes 5 system from the University of North Carolina (UNC) expands the concept of generality introduced with the Stellar system. The concept of the graphics pipeline is abandoned and generalized processors are used to do the transformation and clipping computations. There is still a subsystem that contains hardware which is specialized for rendering, though. The design goal was to be able to render 1 million Phong-shaded polygons per second or a frame rate of >20 frames per second.

[2] In Phong shading, the color at each pixel is approximated by interpolating the normal between polygon corners, instead of just the color. It is clear that Phong shading is more computationally expensive than Gouraud shading.

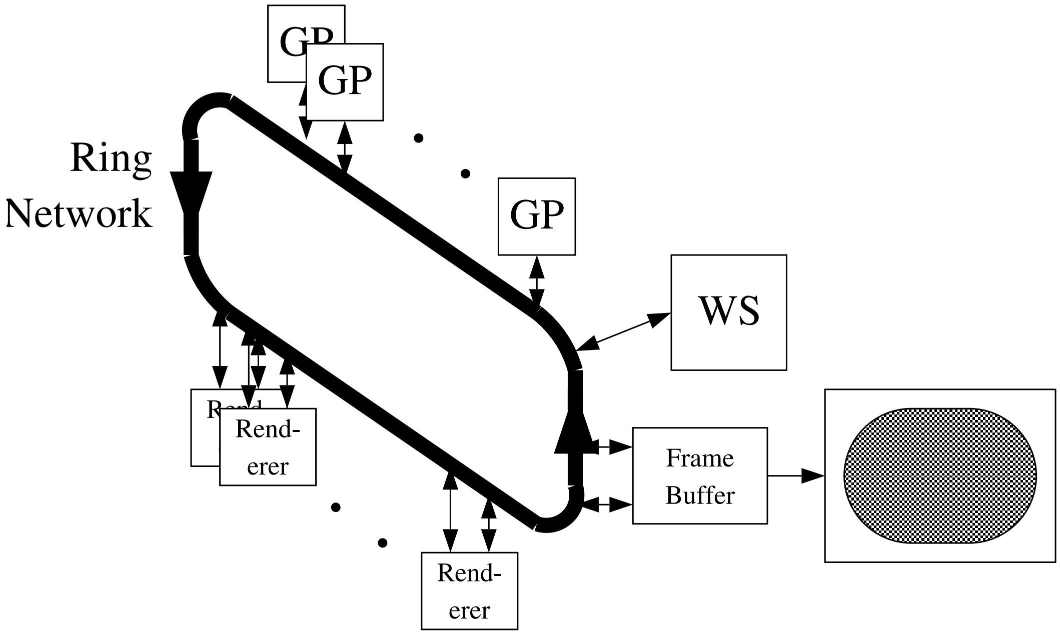

Figure 5. The Pixel-Planes 5.

The design is divided into the Graphics Processors (GPs), the Renderers, the Ring Network, the Frame Buffer (FB) and the Host Interface.

The Graphics Processors are fully programmable processors and perform all the calculations necessary to transform 3–D points to device coordinates. This approach is very close to the GX4000 approach since all the work for the transformation of a given primitive is done by only one physical device. Here, the GPs contain substantial amount of memory to keep part of the graphics database locally. When a command is given to update the display, the data does not have to be fed to the engines; they just perform the necessary calculations using their local data. The program code is duplicated on all the GPs, the same way it is done in the Pixel Machine, providing a true MIMD system. The performance for each GP is 30,000 Phong-shaded polygons per second.

The Ring Network provides high-speed communication between subsystems. The needed throughput is 40 million 32–bit words/second (40MW/sec). The achieved throughput is 160MW/sec using 8 rings in parallel. Each ring node is able to transfer data in and out of the node at 20MHz, while the ring itself has a rate of 40MHz. The performance of the ring is much higher than needed. Future processor upgrades will, therefore, not need a Ring Network upgrade.

Each renderer contains an array of 128x128 SIMD Pixel Processors, each with a quadratic expression evaluator to perform rasterizing calculations for curved surfaces. Its performance is 100,000 Phong-shaded triangles per second. Higher-order polynomial surfaces, though, have to be approximated using quadratic polynomials. When adaptive tessellation is performed, the tessellation program that creates the coefficients has to reside on the GPs. It has to get feedback from the Pixel Processors though (that perform the rasterization) in a manner similar to the one described in [Abi89].

The frame buffer get its pixel data directly from the ring at a rate of 40MW/sec, providing up to 24 full-screen updates per second. No processing hardware is built into the frame buffer. The Host Interface performs communication between the Pixel-Planes 5 and the Sun-4 host. Programmed I/O provides a bandwidth of 4 Mbytes/sec.

For the needed performance of 1 million Phong-shaded polygons per second, 32 GPs and 8–10 Renderers are needed. The silicon utilization is much bigger than in the previous Pixel-Planes design.

The applications that could be run on this system are numerous. A PHIGS+ implementation has been written. Phong-shading models, shadows, texture maps (both image-based and procedural), fonts using conic spline evaluation, radiosity, volume rendering, transparency effects and CSG-defined object rendering are all simulated and will be available on the Pixel-Planes 5 system. This shows that the system is indeed general purpose and can perform almost all of the mostly needed algorithms today.

2.6. Comparison and Trends

Comparison between these architectures can reveal an evolutionary trend in graphics hardware. Certainly, some of the machines lack in performance when compared to others, but those differences makes us able to identify trends.

Machine	Year	Paralleli-zation	Archite-cture	Main Application	Perfor-mance
GX4000	1987	medium, 1-level	rigid	polygon rendering	medium
IRIS	1987	medium	rigid	polygon rendering	good
AT&T PXM	1988	large	rigid	ray tracing	good
GS1000	1988	vector	flexible	polygon rendering	medium
Pixel Planes 5	1990	medium to large	very flexible	Multiple	good

Table 1.

From that simple table, we can observe the following trends:

1. Initially, all the emphasis was given into rendering polygons using special purpose hardware that was left unused when other applications were running.
2. The Pixel Machine followed a totally different path, trying to solve the ray tracing problem. It's still unique for its ray tracing performance. Since it uses commercially available microprocessors, it was meant to run many applications. Its data flow model, though, limits it to ray tracing.
3. The GS1000 was the first actual step towards the generalization of specialized attached graphics processors, which resulted in a drop in performance.
4. Pixel Planes 5, with its general processing architecture approach, continues the powerful trend for future machines that was started by the GS1000.

With the enormous increase in performance of graphics systems in the past few years, we can also observe a shift of the bottleneck from the actual processing, to the communication between subsystems. A perfect example is the Sun GX board for the SPARCstation 1: According to Sun, its line drawing performance is limited by how fast the CPU can send coordinates to it. Some people though continue to produce special purpose graphics engines for polygon rendering [RW89, DWS$^+$88, GGH$^+$88] so that use of hardware can be optimized.

3. Future Needs

Algorithm development and the need for more visualization is constantly increasing. A successful computer architecture must not only be able to fulfill today's requirements, but also tomorrow's needs. The design must therefore be expandable and flexible to ensure that success.

Some algorithms that will probably be needed in the immediate future are addressed here with respect to the hardware architecture that is needed for their realization.

3.1. Scientific Visualization

An estimated 95% of supercomputing time is wasted because of lack of ability to interpret the data that is produced[VIS88]. No matter what the discipline, the problem is that there is "too much data" to deal with. By producing a visual image of the data, we enable the eye, with its complex pattern recognition capability, to extract useful information, which is not easily done by looking at a list of numbers.

Visualization in Scientific Computing (ViSC) is an initiative that started to produce tools that will enable scientists to "visualize" their data with more ease and minimize turnover time. ViSC as a concept that existed for many years, but the bulk of the scientists were (and most of them still are) unaware of the fact that it can help their research and enable them to communicate ideas with greater ease and understanding.

In ViSC, there is too much data, and too much processing to be performed on the data. The typical scenario today is the following: The mathematical problem is solved using a local supercomputer, and then the data is shipped to your personal workstation where the rendering is performed. There is a big communication bottleneck between two ends that can perform extremely fast calculations; the supercomputer is very good at producing the data, and the workstation is very good at displaying it.

Users are finding out the assets of simulation steering (ability to interactively change simulation parameters) and real-time solid modeling, both requiring an even faster communication path between the calculation and the visualization parts of the problem. But, are they really two different and separate entities, or are they two different facets of the same computation? [Wol88]

The graphics standard should aid and complement the simulation process. Almost all simulation programs use a graphics standard to produce graphics output, and do not use any of the processing performed during visualization to simplify the simulation work. Graphics standards today perform many vital operations automatically and efficiently, but they are very clumsy when the user wants to display data that the standard wasn't designed to display. We see therefore that graphics software has fallen to the same pitfall as graphics hardware did: Specialization to certain applications. Hardware started to take the new direction towards generalized machines with the GS1000 and the Pixel Planes 5 systems, but software, as always, is lagging.

3.2. Data Computation and Rendering

In order to fully understand the trend in scientific computing, we first have to know what are the most common problems that a scientist or an engineer solves today using a computer. Then, algorithms have to be mapped on these problems and checked for efficiency. Old algorithms that have been rejected as very computer intensive might be utilizable.

Almost all of the problems solved today come from a mathematical model that is approximated to a reasonable level of complexity for a numerical solution. Due to the nature of the mathematical analysis methods, the end result that has to be numerically solved is a matrix equation, a system of ordinary differential equations (ODEs) or a system of partial differential equations (PDEs)[Pla88]. Whether we are dealing with finite element models[Abe88], or solutions of magnetohydrodynamic equations[WG88], the solution will require the solving of one of these systems. But there are always approximations involved; usually of many different types: Examples include the approximation of the mathematical problem to the solvable model stated above and the difference between the actual solution to the calculated solution. [Hab88] describes the five different kinds of error classified into *Total Solution Error* and *Total Visualization Error*.

The rendering methods that exist today to visualize complex data are insufficient. Interactive display systems are able to efficiently and rapidly display polygons, and if any other type of primitive needs to be displayed fast, it has to be converted to polygons. On the other hand volume rendering techniques are at their infancy, but advancing steadily [DCH88]. Ray tracing and radiosity [CCC$^+$88] try to produce realistic images. Particle systems [Sim89a, Sim89b, Rey87, Rey88] is a new approach to the combined modeling/rendering problem. The natural complexity is $O(n^2)$ since each particle interacts with every other particle. [HB86] describes a way to decrease that natural complexity down to $O(n \log n)$, while [Gre90] further reduces the computational complextiy to $O(n)$, making solutions of large particle systems of millions of particles feasible using today's computing resources.

A particle system approach to solving the n-body problem is an indirect way of looking at the simulation problem. Instead of modeling a macroscopic effect (such as heat transfer) via a system of PDEs, the particle approach models the microscopic interactions of the particles involved.

Physically-based modeling techniques [BB88, Pla89, TW88, WFB87, WK88] is a novel approach to the modeling problem that is more natural. Mathematically, though, it's nothing more than a solution of a system of equations. The difference is that the visual feedback is part of the algorithm.

All of these applications require computing power not available today. Computer architectures have to be devised that accelerate those algorithms. There will definitely be many new rendering algorithms in the future, so a rigid computer architecture at that level automatically destroys long-term success.

3.3. Image Processing

Most engineering computing mainly consists of various methods for altering data of a given form or structure to suit certain needs. Scientific computing, on the other hand, consists mainly of analyzing data of unknown structure. Most of this data is either in image form, or can be translated into such. A main component of scientific research stems from the nature of the human brain to derive patterns from anything that is presented to him though his senses[Nob89]. *Visualization* by definition helps a person "visualize" his problem giving him more visual insight. A tool, therefore, that can perform some preprocessing of the data and present it to the user in a more perceptual form could be very useful.

Image processing and pattern recognition techniques can be applied to any large amount of data gathered over dimensions that are easily perceived visually, i.e. over space and time. During an image processing application, the three-dimensional form of a terrain is extracted when several views of it from a distance are available [MHM88]. This operation resembles the stereoscopic vision of most mammals in existence today. A flight can then be simulated over that terrain, giving the capability of exploring details not clearly visible by just looking at the photographs.

Other types of data cannot be directly sensed conceptually (such as cloud coverage over the entire Earth for a period of several months or years [Hus90]) and new unique ways have to be derived for viewing them.

In many image processing applications, a lot of data shuffling exists from source to destination in a way that does not make the problem trivially parallelizable[HBMS83, YUM88]. This shows that the computing engine should be able to perform almost any process with the same ease in order to be useful.

3.4. Databases

It is known that when new computing equipment arrives, every user thinks that their problem is going to be solved and increases the amount of their data by an amount that cannot be handled by the new machine [Ban89]. The number of computer users has increased exponentially over the last years, so have their complaints about inadequate resources because of the previous tendency. Consequently there is a requirement for the software built for that machine to be more easy to use and efficient.

A problem with traditional relational databases is the fact that the user is given so many ways to access his data that he gets lost in them[Tre88]. Data is definitely organized with a specific scope in mind, but many organization options are given the same semantic weight by the program and therefore making the use of the database difficult.

New object-oriented databases provide us with a classification scheme resembling computer programming. In fact, databases now have the ability to be closely coupled with the application(s) requiring the data. Most scientific applications, usually have to interact closely with some data. Personally, I don't know of any scientific or engineering application that successfully interfaces with a commercial database program. All the necessary data manipulation is done using custom software. A database kernel that is attractive enough to be used by these programs is needed.

It looks that a hybrid application-database program based on the object-oriented programming concept can provide this merge. In such an environment data classification is implicitly required, giving the required functionality.

4. Conceptual Solution

From the previous discussion, it is clear that the computer architecture required for computer graphics needs ample computer power with flexible data flow between its constituent parts. The problem then generalizes into trying to find an easily programmable, parallel, general purpose, computing machine. The system has to be scalable, so it can be fabricated in many cost/performance configurations and easily built so that start-up cost is low.

Almost all of the machines today are based on the von-Newmann architecture. Typically, the machine's subsystems and communication between them is controlled by signals that are independent from the main data path. Von-Newmann machines usually contain a memory unit and a processing unit (ALU).

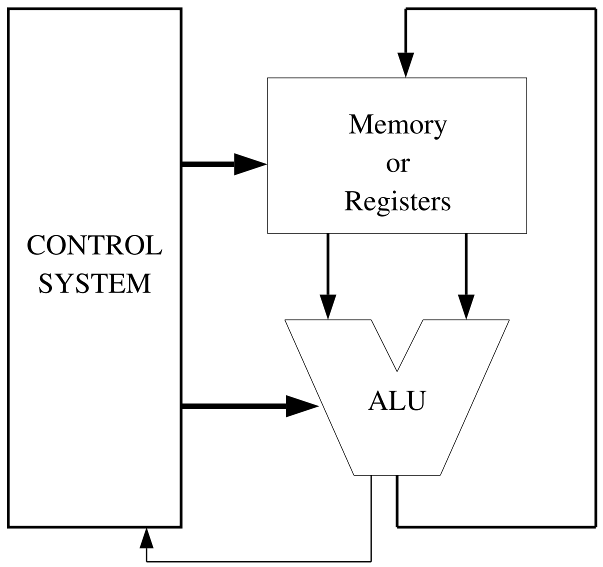

Figure 6. The Von-Newmann architecture.

The control system gets status information back to determine the fate of its actions. This approach can use hardware very efficiently for the following reasons:

1. Using the RISC approach we obtain the highest performance over silicon area ratio. There is minimal hardware involved in communication flexibility, since most of the resources are used in the main data path elements.
2. The control signals for the datapath are limited and almost all of the assignments are made at compile time, leaving only a few for runtime evaluation (simple conditionals).
3. The hardware is optimized for the application it was designed for.

Disadvantages come into play when the system needs to be expanded:

1. The architecture is rigid in terms of hardware and software. On the hardware, new data paths require system redesign. On the software, any alteration requires the reloading of the program.
2. Addition of more processing elements requires the redesigning of the control subsystem and/or the data path. Most of the times the software has to be altered also.

Approaches to parallel computing move from SISD (single instruction, single data) to SIMD (single instruction, multiple data) to MIMD (multiple instruction, multiple data organizations [HB84]. The conceptual difficulty moving along the path increases significantly to the point that some MIMD machines' software is almost impossible to understand simply because conventional programming languages are inadequate when controlling hardware with multiple threads of control; all synchronization has to be explicitly programmed.

The centralized control approach incorrectly assumes that, since it is the fastest, it is the only choice available. Computers don't just need to be fast. They also have to be easy to program, easy to expand, and generally pleasant to operate. Efficient use of hardware is not useful if no application can be run on the machine because of its complexity.

On the other hand, multiprocessing systems with distributed control are both easier to build and expand. At this point, lack of software has not made them popular yet.

Today's fastest machines are built with certain languages and applications in mind. Object-oriented programming has certain requirements from the machine architecture and the operating system design that need to be kept in mind when developing a system. Communication between objects is strongly recommended by the model, and although method execution can be implemented as function calls in sequential machines, message passing is useful when multiple processing nodes are present. Hardware extensions tailored to the object-oriented programming model can therefore greatly enhance efficiency and execution speed.

Rethinking the hardware with object-oriented programming in mind can produce systems that can run the paradigm very efficiently. The following section is an overview of some key ideas.

4.1. Object-Oriented Programming and Dataflow

Object-oriented programming is data-driven instead of control-driven: Data is passed along with instructions on how to use it. This unifies the control and data subsystems to a collection of communicating objects. All of the work done up to now on object-oriented systems assumed a software approach, while the SOAR (Smalltalk On A Risc) project [Ung87] extended traditional von-Newmann architectures to incorporate object-oriented (OO) concepts to RISC, but with not any measurable success.

Another data-driven approach that got initiated from a hardware point of view is dataflow [Boh83]. Most literature is not directly comparing OO and dataflow approaches, even if these two ideas are different facets of the same data-driven concept. Both are leaning towards increased efficiency and ease of programming, one (OOP) from the software engineering point of view, the other (dataflow) from the hardware engineering point of view.

Computers built using the dataflow concept are easily upgradeable by merely installing more processing elements and upgrading the communication network. The processing "code" is distributed among the processing elements, and it executes at its own pace using runtime parallelization with no predetermined user specification [HB84, Boh83, NA89]. The same is true in OOP where the user in unaware of the fact that (possibly) multiple processing units are present.

In dataflow, the completion of an operation signals the initiation of another one. Usually this is done by keeping data tokens in a token store. Each token contains a reference to the required arguments together with the operation that is to be performed to produce a result. The *firing* of an instruction is performed whenever its arguments are available. In OOP, methods are executed by the request of objects. In parallel machines this can be implemented using message passing. The receipt of a message will then initiate execution of the object's method.

From the similarities just listed, it can be observed that dataflow computing does not differ too much from OOP and it is conceivable that their concepts can be merged at the firmware level. Since most of the system work is software development, it is more suitable to use OO concepts and merge them with dataflow ideas to produce a parallel scalable system that will execute OO code efficiently.

4.2. The User Interface

The user interface is yet another integral part of any interactive graphics program. It should not be left unnoticed, since its design can very well affect the whole system. Making the architecture of the user interface compatible with the rest of the system will increase system performance and therefore productivity. An object-oriented approach will be used in both the user interface and the operating system to ensure that compatibility.

Some object-oriented graphics systems follow the Model-View-Controller (MVC) concept [Wis88]. The *Model* contains the graphics database. Viewing transformations to the *model* are performed to produce the *View*, which is the user-observable result on the screen. The *Controller* is responsible for all the feedback from the user. Conceptually, *view* is the output subsystem and *controller* is the input subsystem.

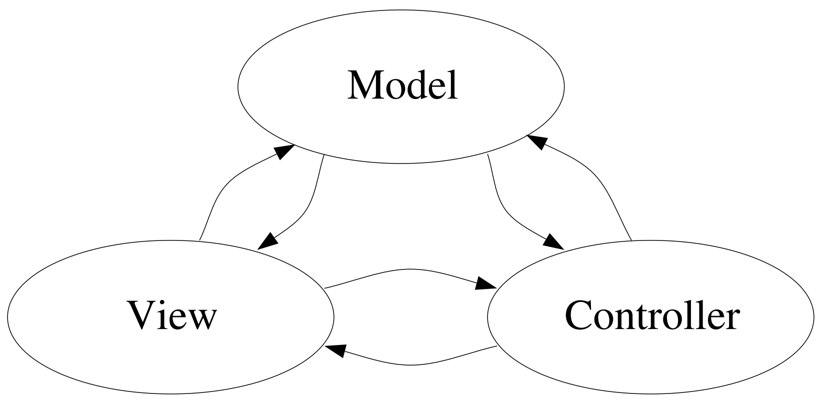

Figure 7. The Model-View-Controller concept.

There are traditionally two ways to build a graphics interface. In both of them, there exists an initiator for all transactions.

1. The order of events is initiated by the application. All user communication is explicitly programmed.
2. The order of events is initiated by the user. After all interactions are set up, it is the user who requests a certain action to be performed by the application.

The MVC approach enables both the application and the user to be masters. This enables changes to be performed by the *controller* independently with other updates in the *view*.

For example, let's assume that we are viewing an oscillating molecule while changing the viewpoint. The *controller* has the ability to update the camera position at the same time the *model* can perform dynamic changes in the *view*. Here we see all three constituents of the user interface acting, and each one of them controls an integral part of the program.

4.3. Is OOP good enough?

Since graphics algorithms are usually easily parallelizable, a graphics system should involve multiple functional units in order to achieve this parallelism. A single sequential machine could be made to obtain the same speed, but the cost will have to be increased significantly.

By making the system object-oriented, we enable more intuitive communication with the user, providing a productive programming environment which is easily parallelizable and upgradeable.

It should be able to perform all of today's graphics algorithms efficiently and also be expandable enough to be able to execute tomorrows algorithms. In previous sections, probable future needs have been presented. We now have to show that an object-oriented system will be able to perform such tasks.

1. *Interactive graphics*. The main concept used today in interactive graphics is the graphics pipeline concept. Pipelines can be simulated using many objects, each one of them providing a function for each step in the pipeline. Moreover, many objects can be generated if the demand for a certain operation is high. If, for example, matrix multiplication is slow, several *matrixXform* objects

can be generated, each one of them serving a different primitive. The system can then take any form between a rigid pipeline to the GX4000 parallel execution approach. In cases where the pipeline approach breaks down, as in the display of B-splines or high-order surfaces [Abi89], the hardware can provide the desired deviations.

2. *Ray Tracing*. Each primitive can be treated as an object, containing certain properties and operations that it can perform. All spheres, for example, have the same intersection routine, and each object has certain shading parameters that are unique to that object. Object-oriented programming is, therefore, directly applicable to ray-tracing.
3. *Numerical Computing*. There have been studies [McD87, McD88] that show that numerical computing can be performed using object-oriented programming while staying as efficient that traditional numerical methods in typical workstations. Such application will run faster, in hardware that is especially optimized for object-oriented programming. Additionally, there are certain advantages in using object-oriented numerical computing. The solution can be well-behaved in ill-conditioned problems and can quickly find alternate ways of solving a problem if a certain approach is inappropriate. On top of that, there is a certain program development ease that is associated with OOP.
4. *Databases*. It is clear from previous discussion that object-oriented databases have advantages over traditional relational databases.
5. *Radiosity*. The basic algorithm consists of form factor creation and their substitution into the radiosity equation to solve for patch intensities [FvDFH]. Form factor formation is characterized by information exchange between patches, while the solution of the radiosity equation is a numerical approximation problem using iterative techniques. Both of these needs can be served by an object-oriented system; the communication by the nature of the OO model, and the numerical computing as stated above.
6. *Physical Modeling*. It can be performed by either solving systems of equations, or by message exchanges between objects that impose physical constraints on each other.
7. *Image Processing*. An image can be classified as one object with all internal details hidden from the application and regarded as system information. Additional image processing hardware can merely redefine the methods internally with no change required in the application itself.

5. Proposed Implementation

Here, an overview of the whole system will be presented together with the description of the scalability and upgrade path of the architecture. Like every new hardware system, an operating system has to be defined for it. The development of an OS based on OOP will greatly simplify application design and improve efficiency. Needs for such an operating system will also be described. A solution concerning the communication network problem, which is inherent in every computing system will also be presented.

5.1. System Overview

The system overview to be described is not limited to hardware. Part of it could be implemented in software. A different mix of hardware and software will allow the design of multiple models with various price/performance ratings, but identical functionality.

It has already been stated that the system involves parallel processing. The structure of each node in the system looks like this:

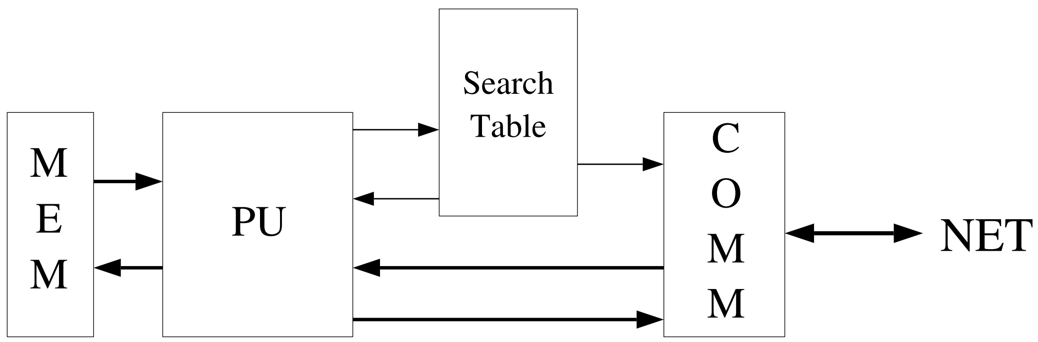

Figure 8. Structure of each processing node.

MEM and **PU** constitute the node's main processing power and storage. **MEM** contains all necessary method code, instance data and all local data needed by any methods. **COMM** is the subsystem that checks incoming network packets for a match. All modern computing systems, regardless of their size and power, contain these three subsystems. What is new is the **Search Table**, that contains all objects and object instances served by the node. For efficiency reasons, it could be implemented using associative memory. When an incoming packet initiates a method search, two things can happen:

1. The object and method are found in the table (a "hit" has occurred). The packet is then forwarded to the **PU** for processing.
2. The search failed (a "miss" occurred). Now the packet will either be ignored or forwarded to another node, depending on the network implementation.

The **PU** keeps track of what objects are present in the node and updates the search table accordingly. If a new instance of an object is created, or a new object is migrated in the node, the new name should be inserted in the search table. Consequently, if an object is destroyed, its name should be deleted from the table. The **PU** keeps track of this activity because it performs all the management of objects, their memory allocation and method execution. **COMM** should only be able to read from the table, since it only uses it as a lookup table to see if the object and method are present in the node. The search table, then, is readable and writable by the **PU** and read-only by **COMM**.

MEM can be conceptually divided into 3 parts:

1. *Method store*: This contains the "code" for each method. It corresponds to the TEXT segment in traditional machines.
2. *Instance store*: The data specific to each instance resides here.
3. *Local store*: Local variables that an object method may need are stored here. This area corresponds to the traditional HEAP space.

Two objects that communicate a lot, should be placed in nodes that have high speed of communication between them. The more high speed channels exist in the implementation, the better. The whole network cannot be built using the same high performance for two main reasons: First, the cost will be prohibitively high. Second, the addition of more high-performance hardware at the same network level will increase the overhead required to route network packets. The same kind of decision involving microprocessor complexity lead to the design of the RISC II chip [Kat85]. The solution is to connect some nodes very tightly, while letting the others to be accessible with lower bandwidth channels.

Message "thrashing" can occur though as an effect of low-bandwidth communication channels. Because of locality of computation, the majority of messages will be exchanged between a small number of neighboring objects. If the node store is sufficiently large, all these objects can be located in the same node, minimizing thrashing. This is the same effect that occurs in traditional cache systems [McD90, Ros90]. If node clusters are placed in close proximity, latency will not be high, so relatively high communication rate can take place. If the objects are capable of migration [OBJ90], the operating system will be responsible for evaluating the need and allow migration to a more efficient physical location to take place.

This discussion is very similar to existing systems, even though the design started using an object-oriented concept.

5.2. Upgrade Path and Object Experts

One main advantage of this system is that it can be used in a broad level of performances, from a low-level PC environment, to a high-end mini-mainframe environment, which does not have to be very expensive.

At the PC level, the whole system can be implemented in software, except (maybe) the rasterization end that could be executed by special-purpose hardware that exists in many PCs today. Since all of the objects might not be able to fit in real memory at once, virtual memory (VM) could be used by the system[3]. If local area networks (LANs) are available, other PCs can be used as processing nodes and for storing information.

A custom-made board may contain one or more general purpose processors with optional additional hardware to control internode communication efficiently. One board, of course, has to be able to handle screen and user-interface I/O.

Heterogeneous environments can also be supported. A SPARC-based node could communicate with an R2000–based node. The **COMM** subsystem (which could be implemented either in software or hardware) should take care of the data translation that might be necessary. Code, on the other hand, cannot be transferred, since the instruction sets are different. The methods should then be compiled

[3] The Multics operating system used the same approach for conventional programs. In OOP, the realization is more natural.

separately for each node type. If certain object methods are not supported in a node, an instance of that object, simply, cannot migrate into that node.

This discussion brings the concept of an object expert [Dal87]: An object that services special hardware has to be "attached" to this hardware. Object methods can therefore be executed very efficiently. This does not necessarily disable the object from creating an instance of itself, since the new instance can be served by a general purpose node. In some cases, though, as in frame buffer devices, this is not possible.

In the case of screen-related operations, most of the functions that can be performed in general purpose processors will migrate to processors "close" to the screen hardware. It makes sense, then, to place several object experts in close proximity to create an "application accelerator". That accelerator could be as special-purpose as we want it; it depends on the placement of the separate object experts and the communication network between them. A graphics accelerator is a perfect example.

Let us now look at the structure of one of those experts. As an example, I am going to use a simple *multiplyAccumulate* object expert:

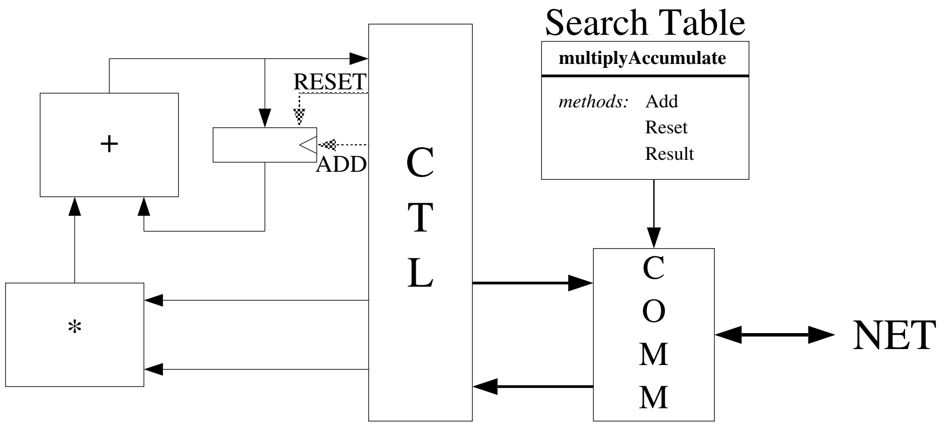

Figure 9. An example *multiplyAccumulate* object expert.

The search table is hardwired in ROM. Since the hardware is limited to executing *multiplyAccumulate* methods, no updates are required to the table. The **CTL** subsystem routes data in and out of the **PU** and also provides necessary control signals (RESET and ADD) to the processing hardware. If that expert did not exist in the system, the emulation could be done in software in another processor, without the knowledge of the application.

An application accelerator's structure heavily depends on the application that it is designed for. A graphics pipeline accelerator might look like this:

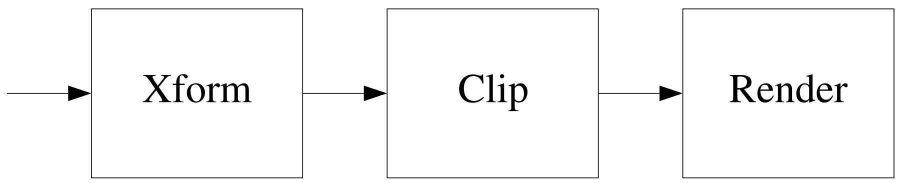

Figure 10. Graphics Pipeline implemented using object experts.

A more flexible and balanced approach could look like this:

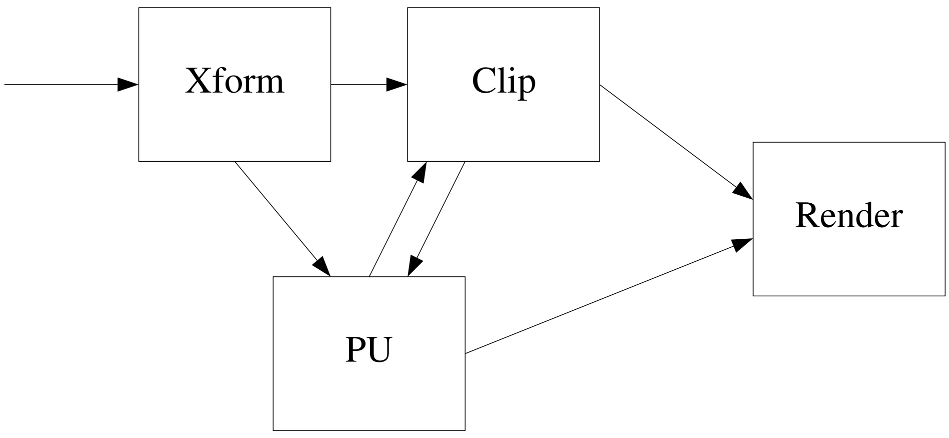

Figure 11. A smarter way to implement a graphics pipeline using object experts.

Here, **PU** is a general-purpose processing element. It contains duplicate methods and objects for all the elements of the pipeline. If one of the object experts is overloaded, messages will be passed to the PU without the knowledge of the application.

As an example, let's assume a loaded pipeline in which both the *Xform* and *Clip* object experts are busy: Primitives sent to *Xform* will be forwarded to the PU where the transformations are now performed. The results are sent to *Clip*. Since *Clip* is also overloaded, the data is sent back to the PU for processing. The result is sent to *Render* for on-screen rendering.

5.3. Network Implementation

The network implementation is very tightly connected to the performance that the system needs to achieve. I will show a network solution that makes the system scalable with minimum additional cost. The minimum upgrade quantity will be one PC board. Many processors can be resident in one board. The interconnection network between the nodes in the same board is fixed and high speed (a crossbar switch, for example) while the external network connections can be built for a fixed network capacity. I will follow Pixel Planes' 5 approach [FPE+89] by placing all the raster hardware in one board.

This "Raster" card should have a high-performance communication subsystem so that realistic speeds can be achieved. Methods served by this *Raster* object may include *readPixel*, *writePixel*, *bitBlt*, *readWindow*, *writeWindow*, etc. Many communication interfaces could be attached to the same card to increase throughput further, the same way the Pixel-Planes 5 nodes attach to the Ring Network. It should also be noted that all the write operations require just high throughput, but read operations also require low latency. This happens because write operations do not need a return value to be sent to the caller, while read operations do. Most graphics systems assume that the hardware does not perform read operations fast. Many times, if read operations are needed, an encoded form of the screen image is kept in main memory to avoid reading the frame buffer. The communication in and out of the board can then be designed assuming high throughput and high latency. This is easier to achieve, since the raster board can now be made highly pipelined.

For pipelined operations, the communication latency is not an important factor, since there is no feedback involved. Several applications, though, include feedback systems, and latency plays an important role in their performance. The interconnection network between the boards should provide both high throughput and low latency, since the architecture has to be general purpose. A hypercube configuration of 2^n nodes, definitely gives low latency (a maximum of n hops between any two nodes in the network) and high throughput (there are n communication busses per node; one for each dimension). Interconnection complexity, though, is high. A backplane of fixed dimension n, can be constructed giving us the capability to wire up to 2^n boards. Additional boards can be added in another backplane by the use of a "cube manager" for each backplane that controls external communication. That network could be a hypercube network itself, or it could be a ring network or a bus.

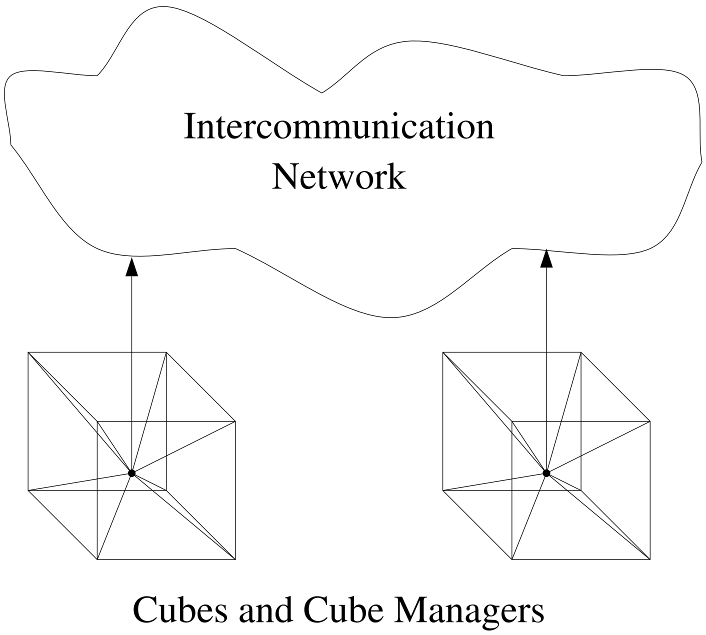

Figure 12. Connecting several hypercube networks.

Upgrading an n-dimensional hypercube (where $n > 3$) in 3 spatial dimensions can be difficult [Dal87]. Wiring space simply does not exist, and delays between adjacent nodes can be unequal, mainly because of unequal wire lengths, unequal line capacitances and inductances, and incorrectly terminated lines. If we prewire a fixed backplane with a dimension larger than needed, and occupy some of the lower dimensions, then we do not lower efficiency significantly. Consider a backplane that is not fully occupied by all 2^n nodes. An example backplane of $2^3 = 8$ nodes, could be occupied by only 5 nodes. In that case, the fifth will be in a different dimension that the rest.

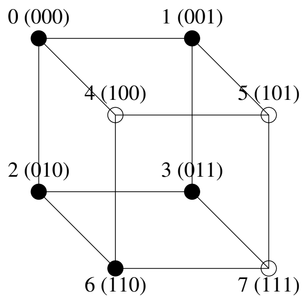

Figure 13. A partially filled 3–dimensional hypercube.

Maximum hop count is 3. Considering the average hop count for each node we find:

$$
\begin{aligned}
node \quad 0: &\quad (0+1+1+2+2)/5 = 1.2 \\
node \quad 1: &\quad (0+1+1+2+3)/5 = 1.4 \\
node \quad 2: &\quad (0+1+1+2+1)/5 = 1.0 \\
node \quad 3: &\quad (0+1+1+2+2)/5 = 1.2 \\
node \quad 6: &\quad (0+2+3+1+2)/5 = 1.6
\end{aligned}
$$

$$
average = \frac{1.2 + 1.4 + 1.0 + 1.2 + 1.6}{5} = 1.28
$$

For comparison, the average hop count for 4 nodes is 1.0, and for 8 nodes is 1.5. We see that the addition of one node that does not complete a dimension has a big effect on the average hop count. For that reason, it is important that dimensions are kept full before adding nodes in higher dimensions.

5.4. An Object-Oriented Operating System (OOOS)

Any hardware discussion should at least be followed by a brief description of the software that is going to be run on it. Some properties of the Object-Oriented Operating System are discussed below:

1. *Object Migration*: If an object or group of objects exchanges many messages with another cluster of objects in a distant location, and if the objects are private (used by only that remote cluster), then the objects' instance data is relocated to a position closer to the communicating objects, if possible. Since the nodes that the objects are executing methods from, might not be the same, the method code might not be able to be moved. An instance can also be marked as "unavailable at this time" in the case it is swapped out to disk.
2. *Multiprocessing*: Because of the parallel nature of the architecture and the requirement that the user is unaware of machine upgrades, multiprocessing capability is required.
3. *Fine Threads of Execution*: Messages are the equivalent of subroutine calls in object-oriented programming. In single processor systems, message passing can be implemented using subroutine calls, but in parallel systems fast thread creation, deletion and context-switching are necessary. An efficient fine-threaded implementation with dataflow extensions is described in [NA89].

6. Summary and Conclusions

The need for newer and better visualization tools in science and engineering will always exist, mainly as a result of the human need for evolution. Unfortunately, the complexity of computing systems has increased tremendously in the process, to a point where only people in the field can use them.

In this paper, I reviewed several computer graphics systems to discover a trend towards general purpose computing in the field of computer graphics. Then, I tried to guess what algorithms will be probably used in the future. After identifying a move towards object-oriented programming, I proposed the implementation of a general-purpose scalable object-oriented system in hardware and firmware, with a promising upgrade path. The "object expert" concept is used to increase performance when needed. The use of a hierarchical user-extensible hypercube network together with the description of the requirements imposed into an object-oriented operating system (OOOS) completes the paper.

References

[Abe88] John F. Abel. Structural Analysis by Finite Analysis Methods. *Computer Graphics*, page D1, August 1988. SIGGRAPH'88 Course 27 Notes: "Developments in Physically-Based Modeling".

[Abi89] Salim S. Abi-Ezzi. *The Graphical Processing of B-splines in a Highly Dynamic Environment*. PhD thesis, Rensselaer Polytechnic Institute, May 1989. Rensselaer Design Research Center Technical Report #89001.

[ABM88] Brian Apgar, Bret Bersack, and Abraham Mammen. A Display System for the Stellar Graphics Supercomputer Model GS1000. *Computer Graphics*, 22(4):255, August 1988. ACM SIGGRAPH'88 Conference Proceedings.

[AJ88] Kurt Akeley and Tom Jermoluk. High-Performance Polygon Rendering. *Computer Graphics*, 22(4):239, August 1988. ACM SIGGRAPH'88 Conference Proceedings.

[Ban89] Gordon Bancroft. Hardware/Software solutions for Scientific Visualization at Large Research Laboratories. *Computer Graphics*, 23(5):137, May 1989. Panel Proceedings, Panel Session Chair: Linnea Cook.

[BB88] Ronen Barzel and Alan H. Barr. A Modeling System Based on Dynamic Constraints. *Computer Graphics*, 22(4):179, August 1988. ACM SIGGRAPH'88 Conference Proceedings.

[Boh83] A. P. W. Bohm. *Dataflow Computation*, volume 6 of *CWI tract*. Centrum voor Wiskunde en Informatica, 1983.

[CCC+88] Michael F. Cohen, Shen Chang, Eric Cohen, John R. Wallace, and Donald O. Greenberg. A Projective Refinement Approach to Fast Radiosity Image Generation. *Computer Graphics*, 22(4):75, August 1988. ACM SIGGRAPH'88 Conference Proceedings.

[Dal87] William J. Dally. *A VLSI Architecture for Concurrent Data Structures*. The Kluwer international series in engineering and computer science. Kluwer Academic Publishers, 1987. Abstract of author's thesis (Ph. D.)–California Institute of Technology, 1986.

[DCH88] Robert A. Drevin, Loren Carpenter, and Pat Haniahan. Volume Rendering. *Computer Graphics*, 22(4):65, August 1988. ACM SIGGRAPH'88 Conference Proceedings.

[DWS+88] Michael Dearing, Stephanie Winner, Bic Schediwy, Chris Duffy, and Neil Hunt. The Triangle Processor and Normal Vector Shader: A VLSI System for High-Performance Graphics. *Computer Graphics*, 22(4):21, August 1988. ACM SIGGRAPH'88 Conference Proceedings.

[FPE+89] Henry Fuchs, John Poulton, John Eyles, Trey Greer, Jack Goldfeather, David Ellsworth, Steve Molnar, Greg Turk, Brice Tebbs, and Laura Israel. Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System using Processor-Enhanced Memories. *Computer Graphics*, 23(3):79, July 1989. SIGGRAPH'89 Conference Proceedings.

[FvDFH] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. Computer Graphics, Principles and Practice. p. 795.

[GGH+88] Nader Gharachorloo, Satish Gupta, Erdem Hokerlek, Peruvemba Balasubramanian, Bill Bogholtz, Christian Mathieu, and Christos Zoulas. Subnanosecond Pixel Rendering with Million Transistor Chips. *Computer Graphics*, 22(4):41, August 1988. ACM SIGGRAPH'88 Conference Proceedings.

[Gre90] Leslie Greengard. The Numerical Solution of the N-body problem. *Computers in Physics*, page 142, March/April 1990.

[Hab88] Robert B. Haber. Visualization In Engineering Mechanics: Techniques, Systems and Issues. *Computer Graphics*, page 89, August 1988. SIGGRAPH'88 Course 19 Notes: "Visualization Techniques in the Physical Sciences".

[HB84] Kai Hwang and Faye A. Briggs. *Computer Architecture and Parallel Processing*. McGraw-Hill series in computer organization and architecture. McGraw Hill, 1984.

[HB86] P. Hut and J. Barmes. *Nature*, (324), December 1986.

[HBMS83] Kevin J. Hussey, Richard J. Blackwell, Gregory J. McRae, and John H. Seinfeld. Environmental Data Display. *Environmental Science and Technology*, 17(2):78A, 1983.

[Hus90] Kevin J. Hussey. Personal Communications. February 1990.

[Kat85] Manolis G. H. Katevenis. *Reduced instruction set computer architectures for VLSI*. ACM doctoral dissertation award; 1984. MIT Press, 1985. Oriniginally presented as author's Ph.D. Thesis, University of California, Berkeley, 1983.

[McD87] John Alan McDonald. Object-Oriented Design in Numerical Linear Argebra. Department of Statistics, University of Washington, June 1987.

[McD88] John Alan McDonald. An Outline of Arizona. Technical Report 131, Department of Satistics, GN-22, University of Washington, July 1988.

[McD90] John F. McDonald. Advanced Computer Hardware Design Course Notes, Spring 1990. Electrical, Computer and Systems Engineering Department, Rensselaer Polytechnic Institute.

[MHM88] H. B. Mortensen, K. J. Hussey, and R. A. Mortensen. Image Processing Methods used to Simulate Flight Over Remotely Sensed Data. *Computer Graphics*, page 77, August 1988. SIGGRAPH'88 Course 19 Notes: "Visualization Techniques in the Physical Sciences".

[NA89] Rishiyur S. NIkhil and Arvind. Can Dataflow Subsume von-Newmann Computing. *Computer Architecture News*, 17(3):262, June 1989. The 16th Annual Internaltional Symposium on Computer Architecture.

[Nob89] Ralph G. Noble. Course Notes on Motivation, Fall 1989. Dept. of Psychology, Rensselaer Polytechnic Institute.

[OBJ90] comp.object. USENET News Discussions, Spring 1990.

[PH89] Michael Potmesil and Eric M. Hoffert. The Pixel Machine: A Parallel Image Computer. *Computer Graphics*, 23(3):69, August 1989. ACM SIGGRAPH'89 Conference Proceedings.

[Pla88] John Platt. Another Introduction to Numerical Analysis. *Computer Graphics*, page C1, August 1988. SIGGRAPH'88 Course 27 Notes: "Developments in Physically-Based Modeling".

[Pla89] John Platt. *Constraint Methods for Neural Networks and Computer Graphics*. PhD thesis, Caltech Computer Science, 1989.

[Rey87] Craig W. Reynolds. Flocks, Herds and Schools: A Distributed Behavioral Model. *Computer Graphics*, 21(4), July 1987. SIGGRAPH'87 Conference Proceedings.

[Rey88] Craig W. Reynolds. Not Bumping Into Things. *Computer Graphics*, page G1, August 1988. SIGGRAPH'88 Course 27 Notes: "Developments in Physically-Based Modeling".

[Ros90] Kenneth Rose. Advanced VLSI Design Course Notes, Spring 1990. Electrical, Computer and Systems Engineering Department, Rensselaer Polytechnic Institute.

[RW89] Desi Rhoden and Chris Wilcox. Hardware Acceleration for Window Systems. *Computer Graphics*, 23(3):61, July 1989. SIGGRAPH'89 Conference Proceedings.

[Sim89a] Karl Sims. Notes on Combining Dynamic and Kinematic Techniques for Animating Particle Systems. *Computer Graphics*, page S1, August 1989. SIGGRAPH'89 Course 30 Notes: "Topics in Physically-Based Modeling".

[Sim89b] Karl Sims. Particle Animation and Rendering Using Data Parallel Computation. *Computer Graphics*, page T1, August 1989. SIGGRAPH'89 Course 30 Notes: "Topics in Physically-Based Modeling".

[Tor87] John G. Torberg. A Parallel Processor Architecture for Graphics Arithmetic Operations. *Computer Graphics*, 21(4):197, July 1987. ACM SIGGRAPH'87 Conference Proceedings.

[Tre88] Lloyd A. Treinish. Discipline-Independent Data Visualization. *Computer Graphics*, page 119, August 1988. SIGGRAPH'88 Course 19 Notes: "Visualization Techniques in the Physical Sciences".

[TW88] Demetri Terzopoulos and Andrew Witkin. Physically Based Models with Rigid and Deformable Compononents. *IEEE Computer Graphics and Applications*, November 1988.

[Ung87] David M. Ungar. *The design and evaluation of a high performance Smalltalk system*. ACM distinguished dissertations. MIT Press, 1987.

[VIS88] The Visualization Roundtable. *Computers in Physics*, page 16, May/June 1988.

[WFB87] Andrew Witkin, Kurt Fleisher, and Alan Barr. Energy Constraints on Parameterized Models. *Computer Graphics*, 21(4), July 1987. ACM SIGGRAPH'87 Conference Proceedings.

[WG88] Robert S. Wolff and Jeffrey Goldsmith. The Global Deformation of Solid Primitives: Application to the Simulation of Astrophysical Magnetohydrodynamic Systems. *COmputer Graphics*, page 31, August 1988. SIGGRAPH'88 Course 19 Notes: "Visualization Techniques in the Physical Sciences".

[Wis88] Peter Wisskirchen. Object-Oriented Graphics. *Computer Graphics*, August 1988. SIGGRAPH'88 Course 22 Notes.

[WK88] Andrew Witkin and Michael Kass. Spacetime Constraints. *Computer Graphics*, 22(4), August 1988. ACM SIGGRAPH'88 Conference Proceedings.

[Wol88] Robert S. Wolff. Visualization In the Eye of the Scientist. *Computers In Physics*, page 28, May/June 1988.

[YUM88] Larry Yaeger, Craig Upson, and Robert Myers. Combining Physical and Visual Simulation - Creation of the Planet Jupiter for the film "2010".

Computer Graphics, page 216, August 1988. SIGGRAPH'88 Course 19 Notes: "Visualization Techniques in the Physical Sciences".