Parallelism in Rendering Algorithms

Franklin C. Crow
Xerox Palo Alto Research Center
Palo Alto, California

Abstract

The need for increased computing power for rendering computer imagery is motivated and the sources from which it might de derived surveyed. Increased power can come from parallelism, as well as faster technology, implying new machine architectures and new algorithms. Surveys sketch early attempts at using available parallel architectures and look at the limitations and strengths of some existing architectures. Finally, potential parallel algorthms for the rendering process are sketched including rough analyses of potential performance bottlenecks.

Keywords

Parallel architectures, Parallel rendering algorithms, SIMD, MIMD, ray tracing, texture, visible surface algorithms, shading.

Introduction

Choosing a computational base for doing computer graphics is not getting simpler. In addition to networked workstation computers of various sorts, a wide variety of special-purpose graphics processors, more general-purpose vector machines, multiprocessors, and general-purpose supercomputers are now commercially available. An even wider variety of computational bases have been promised or proposed, suggesting that the choices will be even more varied in the future.

Historically, use of concurrency in rendering systems has been the province of special-purpose hardware, chiefly that used in the flight simulator business. However, it has always seemed clear that more general-purpose massively parallel processors could easily be applied to rendering computer-generated imagery.

Furthermore, as more potent computing resources have become widely available, it has been discovered that, by squandering processor cycles (using teraflops per image), spectacular imagery comes relatively easily. The result is that it has become acceptable, even fashionable in some circles, to make images which require days of computing time. Furthermore, the resulting images demonstrate that orders of magnitude *more* time could be used effectively in improving the realism or richness of the images.

Motivations for more power

We have seen a repeating pattern recently in the field of computer graphics. (1) a new technique for improved realism is developed. (2) the technique proves to be extraordinarily expensive. (3) the conferences for the next few years are replete with papers on how to improve the performance of the technique. We have seen this happen with surface texturing, radiosity, and most noticeably with ray-tracing

There appears to be no end to the computational power that can be expended in pursuing ever-increasing realism. Computation times on the order of a week on a superminicomputer for a single image have been seen [Wallace87]. Furthermore, the models from which images are made today are still vastly short of the detail needed to approach the richness of natural images [Snyder87]. Our techniques for modeling and rendering surface texture and complicated surfaces such as fur, forests, or clouds are still woefully primitive

To make imagery practical, a general rule of thumb has been that an image should be produced in 3-6 minutes. That allows 10-20 frames per hour, or 3-8 seconds of animation overnight. Computation time per frame has been increasing lately as more resources are poured into commercial computer graphics. However, it is necessary in the limit to be able to produce animation in a small fraction of the total amount of time available to a given project in order to complete anything on time. There are always reasons why modifications are needed in mid-stream.

Given that we wish to be able to do in minutes computation that now takes days to months, how much faster do we have to go? There are 720 hours in a 30-day month. Therefore, we need 3-4 orders of magnitude more power to realize in a few minutes images that we would like to be able to make on a regular basis. To make such images interactively (once every few seconds) we need to speed up by yet another 2 orders of magnitude and to do them in real time (30 frames a second) we need another 4 orders of magnitude. To make, in real time, images that we can now actually produce, we need to speed up our computations by up to 8 orders of magnitude.

Commercial approaches to increased power

Inevitably the most cost-effective way to render images will be with an architecture dedicated to the purpose. If all you want are images of a few thousand polygons using only bilinear smooth shading, you're in luck. Workstations with special hardware are available for 50 to 150 thousand dollars from several sources which can smoothly animate such images.

The drawback, of course, is that if you want more elaborate imagery from a special-purpose machine, you must wait much, much longer, since there will be no help from the hardware. A more general-purpose machine of the same cost cannot be expected to deliver the same imagery as rapidly. However, the

compensating advantage is that the greater general-purpose power of the machine may be brought to bear on a wider variety of styles of imagery.

The current economics of hardware make the most cost-effective general-purpose system, in terms of MIPS per dollar, that which can be made from a small number of commodity chips sold in the millions. Thus the computational component (processor and memory) of machines based on, say, the 68000 family or 80x86 family is very inexpensive for the power provided.

Why not just use a flotilla of standard workstation computers for all computing then? If you have 1000 images to compute by next week and everything has been worked out so that it can all run automatically, then that may be the right choice. On the other hand, if you are trying to develop the image and need feedback, or you are trying to interact with the imagery, then you will be severely limited in the scope of imagery possible.

Furthermore, workstations come with a separate chassis, power supply, network connection, etc., and even a keyboard and display, for each processor. Clearly, a machine with many processors could be packaged more inexpensively. However, in a personal survey, I found a wide selection of parallel machines ranging from roughly 1.5 to 10 thousand dollars per 32-bit processor or equivalent. These figures bracket the roughly 5 thousand dollar cost of a minimal low-end workstation with a standard 32-bit processor. The claimed cost per MIPS of the machines surveyed varies, but stays within half an order of magnitude. Unfortunately, multiprocessor machines are not yet made in the high volume which brings out the cost advantages of denser packaging.

If general-purpose computing power is needed at all costs, the fastest available scalar processors are the supercomputers. However, it is still unusual to find a Cray or similar machine used for interactive graphics.

The need for supercomputer power for interactive graphics is spawning a new generation of "crayola" [Curran88, McLeod88, Methias88, Rashid88] computers designed to mix a bit of supercomputer technology (eg. floating point vector pipelines, scoreboarding) with powerfull graphics I/O. These machines will fill a niche and drive the level of the practical up to an order of magnitude higher for well-matched applications.

Another developing class of computers are the "massively parallel" machines. These include: (1) a specialized frame buffer architecture with a a quarter million single-bit processors, one per pixel [Fuchs85], (2) a SIMD architecture with up to 64k single-bit processors intimately connected in a cleverly designed network [TMC87], (3) a MIMD architecture with up to 256 32-bit processors communicating over a multi-stage network to shared memory modules [Rettberg86], and (4) a handful of MIMD architectures with up to a thousand or more 32-bit processors communicating over networks with multiple connections per processor (supplied by NCube, Ametek, Meiko, and Intel Scientific Systems). At the moment, there is no way to get more than about 3 orders of magnitude more speed over a standard workstation just by spending more money.

Power to be expected from unavoidable technological improvements

Twenty years ago, in the mid sixties, the fastest scalar computers available ran at about 10 MIPS [Thornton64]. Just over a decade later the Cray 1 [Russell78] was running nearly an order of magnitude faster. After another decade, today's supercomputers have yet to achieve an additional half order of magnitude speedup. By comparison, today's high-volume single-chip processors are approaching the 10 MIPS of the fastest computers of the mid sixties. Industry forecasters see an order of magnitude improvement over that speed in less than ten years. However, there is little being said about when we will manage the second order of magnitude.

The preceding indicates that we can expect, at best, 2 orders of magnitude improvement in computational speeds from our onrushing semiconductor technology within the forseeable future. We can easily make use of another 2-6 orders of magnitude speedup which will have to come from parallelism.

As seen above, commercial parallel machines are offering up to 3 orders of magnitude potential speedup through parallelism. Moreover, reports of real problems attaining very close to 3 orders of magnitude speedup on such machines are beginning to surface. Proposed machines with serious chances of actually being built are intended to involve over a hundred thousand 32-bit processors [Ranade88, Pfister85, Gottlieb83].

There seem to be no good technical reasons (ignoring organizational and economic issues) why a million-processor machine won't be realized within 5 or 6 years. Therefore, the possibility exists of coming very close to the raw computational power needed to realize today's most ambitious images in real-time. Whether that power can be harnessed to the task of generating images quickly enough is a more complicated question.

Parallel architectures

There are many ways to cause more than one operation to take place simultaneously. Today's processors often overlap the fetching of one instruction with the execution of the previous one, for example. Supercomputers have used such pipelining techniques to compute floating point operations on vectors for 2 decades. However, predictions of faster commodity processors generally assume increasingly heavy usage of pipelining techniques.

To get 3-6 orders of magnitude speedup, we will need to use massive parallelism. What architectures are capable of providing this? In the grossest terms, the architectures can be divided into three groups: (1) SIMD (Single Instruction stream Multiple Data stream) machines, in which all processors execute the same instruction in lock-step, (2) Machines in which all processors share access to a global memory, and (3) Machines or groups of machines in which processors communicate by sending messages to each other over interprocessor links. The last two types actually represent a continuum of architectures ranging over the amount of globally shared state in the system.

SIMD machines have a long history as massively parallel architectures. Experimental machines such as Illiac IV [Slotnick71], MPP [Batcher80], and most recently, Pixel Planes [Fuchs85] have demonstrated increasing numbers of active processing elements.

Two reasons are usually given for pursuing SIMD designs. (1) A SIMD architecture provides more computational power for a given cost at any given moment because the instruction fetch and decode hardware need not be replicated for each processor. (2) SIMD machines are conceptually easier to program since all processors are doing the same thing. Furthermore, since the state of the machine changes in lock-step, debugging is relatively easy.

The difficulty with SIMD processors is that for many algorithms it is very difficult to keep all processors doing useful

work all the time. Processor utilization may be low enough to offset the cost advantage due to the shared instruction unit. The difficulty arises from the fact that in order to execute conditional code, all processors satisfying the condition must go idle while those who failed the condition execute the appropriate code, and vice versa.

Fortunately, many of the operations in computer graphics can be done without conditional instructions in which the condition varies across the processors. 3-d vector and matrix operations supporting transforms have this characteristic.

In MIMD (Multiple Instruction stream Multiple Data stream) machines each processor operates independently. However, there are vastly different ways in which to use the processors. The concept of task *granularity* is a useful way to rank different usage styles.

Large grain parallelism is characterized by large tasks which run using little or no communication with other tasks. Each computing element might compute a separate frame from an animation, for example. Fine grain parallelism is characterized by very small tasks and much higher rates of intertask communication. A computing element might, as its task, input a vector, transform it, then pass the result to another computing element.

The spectrum of MIMD architectures useful for graphics spans a wide range of examples. The following is a list of architectural examples ranging from those useful for coarser grained parallism to those useful for finer grained parallelism:
(1) mainframe computers connected by wide area networks,
(2) workstations connected by local area nets,
(3) multicomputers consisting of processors each with its own memory, communicating over multiple serial links, and
(4) multiprocessors consisting of processors communicating through shared memory.

At the finest end of the granularity spectrum are numerous experimental and proposed architectures such as dataflow machines, neural networks, multiple functional unit machines, etc. These machines are designed for parallel execution at the machine instruction level. While very interesting, these machines are intended either to translate existing algorithms automatically or to use completely new programming techniques. They have little relevance to this discussion.

Parallel algorithms

Granularity is a useful concept in terms of parallel algorithms as well. A fine grained task might accumulate surface data for a single pixel, doing depth comparisons and other operations to determine its color. A coarse grained task might do the entire rendering task for a section of the image, or generate all the pixels for a given object.

Thinking simplistically, one might want to organize the computing tasks by one or more of the following items:
(1) pixel or group of pixels,
(2) vertex or control point,
(3) polygon,
(4) patch,
(5) object or subobject, or
(6) frame or subimage.

The lack of pixel-to-pixel coherence in the typical ray-tracing algorithm and the simplicity of its implementation has made it a popular early application for massively parallel architectures. It is relatively easy to broadcast the description of a simple scene and accompanying ray tracing program to a large collection of processors, then have each processor compute a pixel or group of pixels [Crow88].

At the other end of the spectrum it has been easy to use a collection of computers, preferably networked, to compute individual frames for an animation (see below).

To a large extent, these examples span the range of grain sizes for existing applications of parallel rendering, the individual pixel (or subpixel sample) being the smallest computable unit and the single image (or contiguous group of images) being the largest.

It is interesting that the simplest implementations of parallel rendering lie at these extremes. In both cases, all computing units use identical programs and scene descriptions, only a few input parameters need be varied from unit to unit. This simplicity has its costs, however. If the unit of computation is an entire frame, then the first frame is completed no more quickly with many computing units than with one. If the unit of computation is just a pixel or group of pixels, you would expect to get the first frame much more quickly. However, each pixel must be computed more or less independently of its neighbors, preventing use of efficient display algorithms.

To compute each pixel independently, a process must execute with access to the entire scene description. Therefore, either all processors must have the scene stored in their memories (limiting the complexity of the scene) or the data must be in a shared memory (limiting the number of independently-executing processors). In practice this has meant that ray-tracing images may be finished in minutes instead of hours. But real-time imagery still eludes the general-purpose machine.

Other simplistic approaches have adapted scan line algorithms to compute scanlines or groups of scanlines independently [Kaplan79, Schachter83] or have broken the image into slices [Crow86]. Machines have also been proposed and built to operate on a polygon or rectangle at a time [Rougelot69, Sutherland74, Locanthi79, Whelan82].

Traditional approaches to rendering have broken the rendering process down to a fairly standard pipeline of tasks. (1) input data delivered to the head of the pipeline, (2) vertices transformed, (3) polygons clipped to the image limits, (4) polygons scan converted, and (5) scan segments shaded. This is especially true in hardware designs where pipelines may even be replicated for higher performance [Schumaker80, .Clark82, Torborg87].

There is clearly a wide range of choices for mapping the rendering process onto multiple processors. Ideally, once the process is broken down to small enough tasks, a standard algorithm, which rightly should be considered part of the operating system, will manage distribution of the tasks to processors and forwarding of the results to succeeding tasks. Under such a system, performance should be primarily a matter of how many processors are available. Partitioning algorithms into such tasks and finding algorithms which thrive under such partitioning are current challenges.

Early use of general-purpose architectures for parallel rendering

Given the proliferation of human effort and computer resources devoted to computer graphics, it is inevitable that some efforts will be made to use multiple machines to get images made more rapidly. In recent years a number of researchers and practitioners have made use of modest levels of paralleism to make images faster and some have used substantial numbers of machines to produce animation. The following represents a personal survey of activity in the field. Note that this survey is by no means exhaustive and, like all such attempts, is out of date the moment it is printed.

Very Large-Grain Parallelism at Apollo

Animation lovers lead by James Arvo at Apollo have been making movies by distributing the computation over as many (up to a few hundred) idle computers as they can find on their net. This is a very efficient way to make animation, especially when each frame may take several hours to complete. Where a frame takes too much time to compute in the available time (usually overnight), frames have been divided, each computer getting a subset of the scan lines to compute.

Ray tracing on the Arpanet

Mike Muus at the Ballistic Research Laboratory in Maryland has been doing ray tracing using multiple machines on the Arpanet. in a manner similar to the Apollo effort. However, these images are being made on a smaller number of large time-sharing machines.

Ray tracing with the Connection Machine

Ray traced images were one of the first applications of the 16k-processor Connection Machine at the MIT Media Lab. Karl Sims has implemented an algorithm [Crow88] in which each processor handles a pixel. A simple scene description is repeatedly delivered to the processors in the instruction stream. Each processor goes idle when its initial or reflected ray fails to hit anything. Therefore, processors with simple pixels spend a good deal of time idle while waiting for the most complex pixel to complete. Grayscale non-antialiased images of a few spheres over a plane take a few minutes to produce.

More recently, Hubert Delaney has built a space division ray tracer [Delaney88] in which space is divided by a cubic tesselation with tags for objects intersecting each cube. Objects are also stored one per processor and chains of tags associate all the objects intersecting a cube. Each processor traces a ray through the subdivided space. Scenes with many thousands of spheres and polygons can be ray-traced in minutes.

The Connection Machine goes to Hollywood

Whitney/Demos productions is using a Connection Machine for commercial animation and special effects. Their emphasis is on scenes modeled with large numbers of polygons. Rendering algorithms are currently being developed and implemented [Crow88].

Ray tracing on the Meiko Computing Surface

For the last few of years, Meiko has appeared at SIGGRAPH using a Computing Surface (an MIMD machine described below) to make ray traced images. Again, the scene is very simple (spheres over a checkerboard). In this case, each computer holds the scene description and program in its own memory and a supervisory processor distributes small blocks of pixels to be computed.

Pixel blocks queue up behind each processor. It is the job of a controller processor to keep the queues full. All processors are kept optimally busy until the frame is close to completion. Many processors are then idled for a short time while the last few pixels, which had been enqueued behind very expensive pixels, are computed. Full-color anti-aliased images are produced in on the order of a minute.

Many Dorados make one picture at PARC

I have been experimenting with using multiple computers on a local area network to generate optimally chosen subimages for merging at a specific workstation. The limited bandwidth of the network limits the practicality of this approach to fairly expensive images; each pixel has to take a good deal more time to compute than it does to transmit across the network [Crow86].

Scan conversion on a hypercube at Bell labs

Researchers at Bell Labs have used a 64-node Intel Hypercube to scan convert polygons. Each processor runs a z-buffer tiler on a 64x64 pixel subimage. Performance is limited to about 300 triangles per second by input/output constraints. A coarser grained algorithm would be more appropriate for the particular machine used [Potmesil87].

Four pixels at a time at Pixar

The Pixar Image Computer uses an architecture which can operate on a few pixels at a time in lockstep execution. Thus the power of the processor is well matched to the size of datum that the bus delivers. Low-level operations, like linearly interpolating between pixel colors and background colors to achieve transparency and antialiasing can be be done simultaneously for all color primaries.

Pixel Planes at University of North Carolina

While Pixel Planes is hardly a general-purpose architecture (see below), it is certainly a general-purpose pixel cruncher. A full-scale realization is able to scan convert, do depth buffer calculations and even shadow priority calculations much faster than the host workstation is able to feed polygons to it. Images involving on the order of a few hundred polygons have been manipulated in effectively real time [Fuchs85].

AT&T Pixel Machines sticks processors in the frame buffer

Another semi-special-purpose machine has been demonstrated and is shortly to be available from AT&T Pixel Machines. A variety of imagery has been produced ranging from relatively simple real-time imagery to elaborate ray tracing in a few minutes.

Demonstrated parallel architectures and their limits

There are now a number of more or less massively parallel processors, some commercially available. Each has its limits as a rendering engine. The following is an attempt to survey at least the general classes of machines available and point out what those limitations might be.

Pixel Planes

Pixel Planes [Fuchs85] uses lock-step execution to allow a processor at each pixel to make a decision about its state with respect to a linear function (non linear functions have been proposed). Thus it can be used to scan convert by evaluating containment in a convex polygon. Containment is determined

by evaluating the edge equation for each bounding edge of the polygon in turn. The current realization of Pixel Planes is a 512 by 512 array of single-bit processors.

Pixel planes is a great machine for scan-converting large polygons. Furthermore, it is fast enough to scan-convert thousands of reasonably sized polygons in real time. However, processor utilization is very inefficient with complicated imagery involving lots of very small polygons. The bottleneck has been moved from the scan conversion process to the polygon feeding process.

The capacity of the machine could be greatly increased if it could operate on non-overlapping polygons in parallel. However, until a host machine can feed it fast enough there is little motivation to move the architecture in such a direction. On the other hand, the more severe limitation of the current design is the inability to calculate non-linear shading functions.

AT&T Pixel Machine

This machine is based on a very fast signal-processing chip manufactured by AT&T for its switching systems, among other purposes. It consists of a front-end pipeline of 9 or 18 of the chips, each independently programmed. This in turn feeds an array of back-end processors (16 to 64) which are hardwired to interleaved subsets of the pixels in a frame buffer. In the case of 64 processors, each processor operates on every 64th pixel. The pixel processors may communicate with their 4 neighboring processors, allowing messages to be passed across the array.

For today's imagery, this is a successful organization. It is limited by its dedicated architecture to algorithms which can be distributed across the image pixels; but that is a large class of algorithms. The system is well-balanced for situations in which most of the processing is associated with pixels rather than surface elements. However, the system is not expandable to greater numbers of processors.

Connection Machine

The connection machine has a very large number of very slow processors which must execute in lock-step [Hillis85, Hillis86, TMC87]. Therefore it is at its best when the problem can be divided into at least tens of thousands of identical tasks. For complicated environments with at least tens of thousands of surface elements, it promises to be quite successful. However, so far it appears poorly matched for interactive or real-time image rendering.

It is necessary to be cautious about characterizing a radically new architecture such as the Connection Machine. Often, cleverly designed algorithms can bring out capabilities that aren't initially apparent. For example, new algorithms which can chop up surface elements into uniform pieces to make better use of the processors may change the apparent utility of the machine.

Computing Surface

The Computing Surface is an MIMD machine made of an array of Inmos transputers. Each computer has 4 links to other computers. The ray-tracing program often shown by Meiko runs the same program at each processor (homogeneous code). However the array allows much wider possibilities in that each processor can run different code (heterogeneous code). For example, the array could be organized into multiple interlocking pipelined data streams for transforming, shading and scan converting.

The fact that Meiko chooses to use homogeneous code points up the difficulty of optimizing use of such an array with heterogeneous code. It could be a delightful challenge, but mapping a rendering algorithm optimally to a given array size will be difficult. The data flow must be carefully mapped onto the processor interconnection since only 4 direct links to another processor are available to each processor.

Even more difficult will be the problem of designing a general scheme which will scale over different size arrays. On the other hand, assuming adequate bandwidth to the frame buffer, there is a great deal of potential. The ultimate limitation in this architecture may lie in the relatively low interconnection level. To get a message across a large Computing Surface requires that it be passed from processor to processor in several stages. At each stage a processor must interpret the destination of the message and pass it along the next link.

Binary n-Cubes (Hypercubes)

A few companies (Intel, N-Cube, Ametek, etc.) are now offering binary *n*-cube systems based on the Cosmic Cube [Seitz85]. A binary *n*-cube has 2^n processors interconnected so that any processor can route a message to any other in n steps. Similarly, each processor has n routing connections. Thus a *3*-cube would be topologically equivalent to a cube. Eight processors (vertices) would each have three connections (edges) and any processor could be reached by traversing no more than three connections. Hypercubes have the disadvantage that they are not easily extended. Increasing the size means doubling the number of processing elements and incrementing the connections to each one.

Given routing support in the operating system, messages should be sent equally easily to any processor. However each connection involves processor involvement as in the Computing Surface. The advantage here is that the connectivity is higher and thus the number of links lower.

Butterfly

The BBN Butterfly uses a multistage crossbar net to allow access from any of up to 256 processors (M68020+M68881 each) to any of up to 256 memory modules (4 MBytes each). Interprocessor communication is done by leaving messages in shared memory.

The Butterfly is less easily extended than the Computing Surface (an additional stage must be added to the routing network for each doubling of the number of processors). Otherwise the two architectures offer similar capabilities. It should be easier to impose a multiple pipeline structure in the Butterfly since messages can be directed between any two processors equally easily. The fact that results have to be passed through shared memory increases memory contention. However, that is claimed not to be a serious problem [Rettberg86].

Routed Arrays

Two of the binary *n*-cube manufacturers (Intel, Ametek) are beginning to offer second-generation machines with dedicated routing chips which avoid the problem of involving the node processor for every interprocessor link which must be traversed. Given this hardware assistance, the binary n-cube can be abandoned for a more easily expanded grid

organization. This appears to be the way of the future. Other proposed machines are similarly organized [Ranade88].

Local-Area Networks

The limitation on local networks is clearly in the speed of the network. Everything must travel over the same path. Fiber optic networks promise adequate bandwidth for transmitting images in real time. A 1 gigabit/second path can handle up to 40 million 24-bit pixels per second while real-time video-resolution images require only 10 million pixels per second. Currently, the technology to get messages on and off the network is not quite up to the speeds the optical fibers can handle. However, this is currently an area of rapid development.

The limit on performance of such a net would then lie in the bandwidth of the frame buffer's connection to the network and on the level of interprocess communication. Subdividing the job so that each processor produces a subimage pretty much independently, the limit would appear to lie in the number of processors which can be attached to the net.

Getting back to reality, current local area networks typically offer 10 megabits per second. This will remain the major limitation for the immediate future.

Conclusion

The SIMD (lock-step execution) machines such as Pixelplanes and the Connection Machine are successful partially because they are easy to program. They provide orders of magnitude more processors than other existing machines. However, they offer very little power per processor and often very low processor utilization.

Existing MIMD (independently executing) machines offer orders of magnitude greater power per processor but orders of magnitude fewer processors. Roughly equivalent performance (within an order of magnitude) has been demonstrated for simple ray tracing using homogeneous code.

There appear to be substantial untapped possibilities for MIMD machines using heterogeneous code. However, the operating system support needed to ease the task of loading and controlling a complex program is still relatively primitive. Arranging a hundred processors and code segments by hand will not be a welcome chore. Arranging thousands by hand may be effectively impossible. Massively parallel MIMD machines will not achieve their potential until better programming environments emerge.

Parallelizing by subdividing the image

Where each node in a system is large and fast enough to run the whole rendering process quickly, simple division of the image and distribution over a number of independent processors running the same code is a strategy worth considering. This is especially true at the moment, when operating system support for many small tasks is lacking and the shape models making up the scene are relatively simple.

Given a few dozen processors, each processor must produce on the order of ten thousand pixels. For optimum load-balancing, a given processor's pixels should be interleaved with the others as in the AT&T Pixel Machine. If the pixels are not interleaved, unevenly distributed detail will cause unevenly distributed computation.

On the other hand, efficient implementation of proper antialiasing frequently involves sharing computed color values between neighboring pixels. For example, when ray-tracing, antialiasing is often implemented by calculating multiple rays per pixel. The best results are obtained by a weighted average of all rays over an area which overlaps neighboring pixels by at least one-half. Therefore, each pixel must be able to use its neighbor's ray values or be stuck with doing 4 times as much work.

Also, most faster rendering algorithms make use of area coherence, reducing the calculation of the shade of each pixel to an increment from that of its neighbor. It is considerably more efficient, in such cases to scan convert 100 pixel polygons than 10 pixel polygons. Therefore, having each processor operate on a contiguous set of pixels is preferable. For larger polygons or more elaborate shading, however, this argument holds less weight.

It is possible to balance the processor loads when dividing into subimages with contiguous sets of pixels. An analysis of the scene is required to determine the average amount of work needed in each area. Such an algorithm is described in [Crow86]. Having done this, the image can be adaptively divided based on the analysis. However, this imposes a limit due to the speed with which controlling processors can analyze the image and parcel out subimages.

Another approach is to divide the image into much smaller cells and parcel out cells to processors as they need work, the approach taken by Meiko in their ray tracing demonstration. This maintains the advantages of area coherence while balancing the load as long as the cell size is small enough. Again, too small a cell size loses the advantages of coherency.

Another argument against having each processor execute the whole rendering process is that many operations will have to be replicated in each processor. Transformations and clipping to remove items falling outside the field of view are two obvious examples. The Pixel Machine minimizes this problem by providing an input pipeline on which to carry out this sort of shared computation. However, there are cases, such as rendering adaptively subdivided patches, where the bulk of the computation would have to be done in the pipeline. Furthermore, schemes using very small cells, as described in the previous paragraph, will suffer from replicated computations.

In the face of much more complicated scene descriptions, parallelizing by subdividing the image looks like a bad choice. When a scene is described by millions of small details, the memory of a single processor will be inadequate to store all of it. Furthermore, since only a few pixels will be produced by each *surface element* (polygon, patch ,etc.), the percentage of the effort going into scan conversion and shading will be reduced. It will be necessary to parcel out the work of the entire rendering pipeline.

One further complication arises out of texture mapping. Texture maps use a great deal of storage. Furthermore, a scene may require a number of different textures. Replicated storage of several texture maps in every processor's memory is at worst, impractical, at best, wasteful. Storing the texture maps centrally or in a smaller number of distributed locations brings on algorithmic complexity which was heretofore avoided and imposes a substantial interprocessor communication overhead.

In summary, parallelizing by subdividing the image makes sense where the scene description is simple, the bulk of the computation is centered on operations concerning pixels (scan conversion and shading), and only limited communication is required between neighboring pixels or subimages. It is not a

good idea for very complicated imagery or cases where large amounts of shared data are used.

Parallelizing the traditional rendering pipeline

Taking clues from special-purpose graphics architectures, it makes sense to parcel out the stages of rendering into a pipeline. At the same time, the pipeline stages requiring the heaviest computation may be replicated in order to smooth the pipeline flow. In an ideal larger system, all stages of the pipeline would be heavily replicated with computation flowing to the least heavily loaded appropriately coded processor.

Given the wide variety of parallel architectures possible, it is clearly necessary to look more closely at the nature of the computation involved in the pipeline stages to see what kinds of algorithms map easily on to what kinds of architectures.

The conventional rendering pipeline can be broken down into the following stages for most purposes: (1) data input, (2) coordinate transformation, (3) clipping, (4) visible surface determination, (5) scan conversion, and (6) pixel shading.

Different algorithms place different emphases on these operations. For example, ray tracing doesn't require a conventional clipping process. Current very expensive images devote so much computation to pixel shading that the other elements of the pipeline become relatively insignificant in cost. However, future images depending on very complex data will find that all stages of the pipeline need attention. Furthermore, no stage of the pipeline can be ignored where real-time animation is the goal.

Data input

Models of very complicated scenes remain painful to produce. This has helped avoid the issue of data input for most applications, the primary exceptions being in flight simulator applications. However, data exists which can swamp any existing system. For example, large amounts of terrain data for visual simulation are available on tape from the U. S. Defense Mapping Agency.

Complex construction projects are depending to ever greater extents on computer-aided design and manufacturing. Thus immense data bases are being created for large buildings, airplanes, spacecraft, etc. There are abundant reasons for making imagery from such databases. For example, an application such as an interactive repair manual based on computer generated imagery could require instant access to immense amounts of data.

Massively parallel machines and supercomputers in general have massive I/O problems. If the data to be rendered is relatively constant, then it would be reasonable to consider storing it in a distributed manner in the primary memory of the machine. Remember, the primary memory of a multi-thousand processor machine will of necessity be huge. However, constantly changing data, more typical of today's use require staging the data through secondary storage.

The advent of smaller, less expensive disk drives has led to innovations similar to the "disk striping" techniques used in supercomputers. Clearly, greater bandwidth can be achieved by reading from many disks simultaneously. Thinking Machines has developed the "DataVault" following this principle.

The DataVault is an array of 39 disk drives, providing 32 bits in parallel with single-bit error correction (the data can be reconstructed if one drive fails). The bandwidth matches that of a Connection Machine I/O port, 40 megabytes per second.

Clearly, innovative I/O of this or a similar nature will be necessary. For really large machines, it is likely that I/O must be distributed over the processor interconnection network, perhaps physically as well as logically.

Coordinate Transformation

Coordinates must be transformed to: (1) assemble primitive shapes into complex articulated objects, (2) locate and orient objects in a scene, (3) orient the scene to a particular view, and ultimately (4) provide locations on the display. Happily, this is a very straightforward computation requiring a single simple data structure, needing no interaction between different coordinate vectors, and not even involving conditional branches. The basic operation is a matrix multiply.

The simple nature of the computation makes it ideally suited for SIMD machines, or anything else for that matter. Vector units and SIMD machines exhibit a "sawtooth" performance profile for applications like these. If there are n execution units then execution speed will be proportional to the ceiling of c/n, where c is the number of coordinate vectors to be transformed. For most efficient utilitization of such a machine, the number of coordinate vectors should be many times the number of execution units.

MIMD machines may run with maximum efficiency on coordinate transformations since no communication is required between processors and the work may be parceled out in chunks of arbitrary size. The only communication problem lies in getting the coordinate vectors into the memory of the processors executing the transformations. For interactive use or real-time animation it should be safe to assume that the vectors can reside in the appropriate memory.

Transformations involve multiplying a sequence of matrices representing the operations described above and then applying the result to a large number of coordinate vectors. However, for less than four vertices per processor, it is more efficient to apply each matrix in turn to the vertices (16 multiplies, 12 adds for a 4x4 matrix), than to concatenate the matrices (64 multiplies, 48 adds). Therefore, where the number of processors available falls in the same order of magnitude as the number of coordinate tuples to be transformed, it is prudent to rethink the way in which transformations are computed.

Clipping

Clipping, where required, involves determining whether surfaces lie wholly inside or wholly outside the image or need to be divided at the image edge. Techniques for polygons are well established but higher order surfaces and algorithmic shapes may require specialized methods.

In any event, clipping starts with determining the state of the defining points of the surface elements (e.g. vertices, in the case of polygons). Like transformations, these operations can be carried out straightforwardly on most any parallel architecture.

Given the state of all the defining points, some surface types can be trivially accepted or rejected. If all vertices of a polygon lie within the image then the entire surface of the polygon does. If all control points of a bicubic Bezier patch lie within the image then the entire surface of the patch does. Similarly if all points lie off the image across the same edge, polygons and some curved patches can be trivially rejected.

Clipping, requires gathering all the information about a surface element together at one processing element. However, since defining points are frequently shared between neighboring surface elements, it is generally more efficient to do

transformations and initial determination of the clipping state on an array consisting of all the defining points alone. On the other hand, to find out if a polygon, for example, lies within the image, it is necessary to retrieve transformed vertices from wherever the transformation process left them. This can be very demanding on the processor interconnect, if it is to be done for thousands of polygons simultaneously.

For machines designed with an interconnect intended to be used this way, such as the Connection Machine, interdependent data structures like this can work successfully. However, in a machine intended for coarser grained programming styles it would clearly be preferable to keep all the information about a surface element together, even at the expense of having to replicate some computations.

The process of actually dividing polygons which cross the bounds of the image has traditionally been done with algorithms involving substantial numbers of conditional branches. These clearly will not run very efficiently on SIMD machines. Only a small portion of the polygons in a complicated image will be clipped by an image boundary. Of these, a smaller percentage will be clipped by a corner of the image. Furthermore, polygons may have different numbers of vertices. All this nonuniformity suggests that clipping is best done on an architecture with independently executing processors.

Clipping is an insignificant expense in images where the surfaces are elaborately shaded. Therefore, inefficiencies can often be tolerated. However, where extremely complicated scenes or high frame rates are involved, clipping must be considered carefully. Furthermore, if image subdivision is used for increased parallism, an increased clipping load will result.

Visible Surface Determination

Visible surface determination is basically a sorting problem. The sorting can be reduced to finding the closest surface where surfaces are opaque. A depth buffer will often suffice. Similarly, a ray-tracer based on space division needs only to traverse the space until the first surface is reached. A true sort is needed where multiple transparent surfaces are involved or, equivalently, where surface colors must be blended for purposes of anti-aliasing.

Traditional hidden-surface algorithms have depended heavily on global sorting schemes [Sutherland74]. However, most of these schemes become impractical for very complicated scenes. Furthermore, only certain fine-grained parallel architectures, such as the Connection Machine, are very good at global sorting. Generally, parallel algorithms will probably work better by replacing global sorting with local or at least distributed sorting.

Distributed sorting can be done by calculating object priorities and then sorting within objects [Sutherland74, Clark76, Crow82]. If a scene can be easily modeled as a reasonably uniform hierarchy of shapes this approach could be successful. In such a hierarchy, shapes are collected into articulated objects, which in turn are collected into object clusters, which in turn make up the scene. The hierarchy has to be uniform enough that no step in the hierarchy requires significantly more time to sort than the others.

Local sorting lies at the opposite extreme. All surfaces may be scan converted independently and each pixel collects the color, depth and other salient information on each surface affecting it. Once all surfaces are processed, the surfaces at each pixel may be sorted independently and the proper color computed from the resulting order.

Note that it is now necessary to do a full sort of all surfaces covering each pixel. Traditional algorithms use pixel-to-pixel coherency to avoid this for most pixels. Local sorting creates more work. On the other hand, most images have very few surfaces per pixel, making trivial sorting schemes possible.

Of course, the depth buffer algorithm is the first thing one would expect to try in a parallel environment. Given adequate memory the more robust techniques of *a-buffers* [Carpenter84] can be employed. These methods are trivially adapted to parallel environments of all kinds.

Scan Conversion

Scan conversion offers a more difficult challenge than other stages of the pipeline. The obvious algorithms are woefully inefficient or lead to poor load balancing. For example, if processors are assigned to one or more pixels it is straightforward to broadcast each surface element to all processors and let each determine what pixels are affected. However, as scenes become more complicated, putting all surface elements through a single broadcast process becomes a bottleneck.

Similarly, surface elements can be parceled out one or more to a processor. However, a large bicubic patch will take much longer to scan convert than a small triangle. Unless (1) there are many more surface elements than processors and (2) a way is provided of dynamically allocating work to the less busy processors, then the more difficult surface elements will dominate the execution time.

Even where the entire scene is composed of a single surface element type, such as polygons, there is enough variance among polygon sizes and shapes to cause load balancing problems. Especially in SIMD architectures, scan conversion is inefficient without substantial additional algorithmic complexity to smooth out the computational flow [Crow88].

Other suggestions have included assigning a processor to each scanline (or group of scanlines) or other subdivision of the image. All such simplistic task assigments fall heir to the same flaw. The computation is not uniformly distributed over any aspect of the image or the shapes from which it is derived. This argues heavily for a more loosely structured architecture in which a pool of processors are assigned work as required and all tasks are small enough that none takes a significant portion of the total frame time.

Efforts at splitting larger polygons and subdividing patches to more uniform sizes may pay dividends in better performance. However, the best results will come from making sure that task times are short relative to the time needed to make the image, and that tasks can be distributed over the processors dynamically in quick response to changes in processor loading .

Patch subdivision makes a good example of another kind of parallel algorithm. Instead of parceling out surface elements and having a processor convert the whole thing to pixels, each processor would subdivide its patch, spawning new patch tasks to be handed to other processors. The process then recurses on each of the newly created subpatches until the patches are small enough to be scan converted as polygons, or until some other termination criterion is met.

In this case an exponentially expanding number of tasks is generated very quickly. This turns out to be a very successful way in which to use the Connection Machine since its interconnect is well-matched to this kind of algorithm. However, other architectures may have difficulty with the

comunications overhead inherent in the rapid spawning of new tasks.

Another approach would be to do the later levels of the subdivision within one processor then spawn tasks for doing polygon scan conversion. This can cut back on communications since the description of a patch is considerably larger than that for a polygon. Savings are limited, however, since half of all processes spawned occur at the last step.

For the sake of this categorization, I'll treat ray tracing as a scan conversion technique. Ray tracing started out as an inherently parallel algorithm. Speed enhancements to avoid unproductive rays can compromise that quality. However, the recent moves towards space division techniques pose an interesting alternative.

It would appear that scene complexity would be a problem for parallel ray tracing in that each processor needs to see the entire description of the scene. However, for straightforward ray tracing, complex scenes are intractable in the first place. Certainly, modest levels of parallelism where each processor has substantial memory pose no problem for storing the scenes made today.

The space division techniques require a good deal more storage [Glassner84, Kaplan85, Fujimoto86]. However, the storage can be distributed across the processors without having to be replicated everywhere. The Connection Machine implementation described above uses a uniform division of space to allow SIMD execution. Other architectures might be better suited to a more efficient adaptive division.

Pixel shading

Even after it has been determined which surfaces contribute to which pixels, calculating the color for each pixel can consume the vast majority of the total computation in scenes with textured or reflective objects, or otherwise non-trivial shading requirements.

Happily, expensive shading algorithms often use little or no coherence between neighboring pixels. Therefore, the shading computation for pixels can proceed completely independently, making the application of massively parallel architectures relatively straightforward. A processor may be devoted to a pixel or group of pixels.

However, images with surfaces having widely varying shading characteristics (the expected case) can greatly complicate the task of programming for SIMD machines. To avoid idling the majority of the processors on conditional branches, pixels with similar shading requirements must be accumulated until there are enough to compute efficiently. This is clearly a circumstance where an MIMD machine can proceed more efficiently and with simpler algorithms.

For simpler shading, the similarity of neighboring pixels is used to reduce the computation to very inexpensive incremental operations. Here, it is clear that neighboring pixel shades should be calculated in a sequential process which can take advantage of the much faster algorithms based on pixel-to-pixel coherence. Organizations that devote a processor to each polygon make sense here.

Given the surface represented at a pixel, the position in the defining space of the surface given by the intersection of the surface with a ray through the center of the pixel can be used to drive an interesting class of texturing functions. Textures based on space-filling functions [Perlin85, Peachey85] and fractal expansions [Haruyama84] have been used. This general technique has also been used to displace surfaces [Hiller87].

The advantage of these position-based functions is that they require insignificant storage and can be computed independently for each pixel. Therefore, they are well-suited to most parallel architectures. Algorithms using pixel-to-pixel coherence to speed up these functions may be discovered, but none have been published as yet.

The more problematic texturing techniques are those which require an image to define the texture. These techniques (image mapping, bump mapping, environment mapping, displacement mapping [Cook84]) have been very successful for at least two reasons: (1) photographs of natural textures are far easier to come by and more successful than mathematical models, (2) it is less expensive to retrieve pixels from an image of a scene, in general, than to compute the equivalent color directly from the scene description.

The problem with texture maps is that they take up a lot of space. If they are replicated for every processor which needs them, they take up unacceptably large amounts of storage. If they are not replicated then they become bottlenecks in a parallel system as many processors try to access them at once. Clearly, successful use of texture maps in parallel environments will require a careful balance of replicated maps and orchestrated access patterns. This is an area in which innovations are needed.

Conclusion

This is an exciting time for those interested in parallel algorithms. New architectures are becoming commercially available at a rapid pace and there will clearly be more innovation in the near future. Just about every machine introduced does something very well (otherwise they wouldn't have been developed). However, no parallel machine yet matches the universal applicability of the traditional Von Neumann uniprocessor.

Most newly introduced machines are too expensive for interactive (personal computer) applications. However, that will change. The basic cost of producing and packaging a substantial parallel machine could be quite low if the volume were high enough. Volume at that level, however, requires widely available parallel systems software and applications support.

As interest in these new architectures builds, the systems and applications software which can make them useful to a wider community will be developed. No one can deny the benefits of inexpensive, very powerful machines when it comes to extending the use of rich, dynamic computer imagery.

References

[Batcher80] Kenneth E. Batcher, Design of a massively parallel processor, *IEEE Trans. on Computers*, **C-29**, 9 (Sept. 1980), pp. 836-840

[Carpenter84] Loren Carpenter, "The A-buffer, an Antialiased Hidden Surface Method", Proc. Siggraph '84, *Computer Graphics*, **18**, 3, July 1984, pp. 103-108.

[Clark76] James H. Clark, "Hierarchical Geometric Models for Visible Surface Algorithms", *Communications of the ACM*, **19**, 10, October 1976, *pp.* 547-554

[Clark82] James H. Clark, "The Geometry Engine: A VLSI Geometry System for Graphics", Proc. Siggraph '82, *Computer Graphics*, **16**, 3, July 1982, pp. 127-133.

[Cook84] Robert L. Cook, "Shade Trees", Proc. Siggraph '84, *Computer Graphics*, **18**, 3 (July 1984), pp. 223-231.

[Crow82] Franklin C. Crow, "A More Flexible Image Generation Environment, Proc. Siggraph '82, *Computer Graphics*, **16**, 3, July 1982, pp. 9-18.

[Crow86] Franklin C. Crow, "Experiences in Distributed Execution: A Report on Work in Progress", *ACM Siggraph '86 Course Notes #15*, August 1986.

[Crow88] Franklin C. Crow, Gary Demos, Jim Hardy, John McLaughlin, and Karl Sims, "3D Image Synthesis on the Connection Machine", in *Proceedings of the Conference on Parallel Processing for Computer Vision and Display*, University of Leeds, UK, January 1988.

[Curran88] Lawrence Curran, "Surprise! Apollo Unveils a 'Desktop' Supercomputer", *Electronics*, **61**, 5, March 3, 1988, pp. 69-70.

[Delaney88] Hubert Delaney, Personal Communication, January 1988.

[Fuchs85] Henry Fuchs, et al, "Fast spheres, shadows, textures, transparencies and image enhancements in Pixel-planes", Proc. Siggraph '85, *Computer Graphics*, **19**, 3, July 1985, pp. 111-120.

[Fujimoto86] Akira Fujimoto, Takayuki Tanaka, and Kansei Iwata, "ARTS: Accelerated Ray-Tracing System", *IEEE Computer Graphics and Applications*, *6*, 4, April 1986, pp. 16-26.

[Glassner84] Andrew S. Glassner, "Space Subdivision for Fast Ray Tracing", *IEEE Computer Graphics and Applications*, **4**, 10, October 1984, pp. 15-22.

[Gottlieb83] A. Gottlieb, R. Grishman, C. Kruskal, K. McAuliffe, L. Rudolph, and M. Snir. The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer, *IEEE Transactions on Computers*, C-32:175-189, February 1983.

[Haruyama84] Shinichiro Haruyama and Brian A. Barsky, "Using Stochastic Modeling for Texture Generation", *IEEE Computer Graphics and Applications*, **4**, 3, March 1984, pp. 7-19.

[Hiller87] Martha Jean Hiller, *Three Dimensional Texture Trees*, Ph.D. Thesis, MIT, Cambridge, June 1987.

[Hillis85] W. Daniel Hillis, *The Connection Machine*, The MIT Press, 1985.

[Hillis86] W. Daniel Hillis, and Guy L. Steele, Jr., Data parallel algorithms, *Communications of the ACM*, **29**, 12, December 1986, *pp.* 1170-1183

[Kaplan79] Michael R. Kaplan and Donald P. Greenberg, Parallel Processing Techniques for Hidden Surface Removal, Proc. Siggraph '79, *Computer Graphics*, **13**, 2, August 1979, pp. 300-307.

[Kaplan85] Michael R. Kaplan, "The Uses of Spatial Coherence in Ray Tracing", *ACM Siggraph '85 Course Notes #11*, July 1985.

[Locanthi79] Bart Locanthi, Object-Oriented Raster Displays, *Proc. Caltech Conference on Very Large Scale Integration*, January 1979, *pp.* 215-225.

[McLeod88] Jonah McLeod, "Ardent Launches First 'Supercomputer on a Desk'", *Electronics*, **61**, 5, March 3, 1988, pp. 65-68.

[Methias88] C. Methias, "Stellar Personal Supercomputer", *Proceedings 33rd IEEE Computer Society International Conference (Spring COMPCON 88)*, February 1988.

[Peachey85] Darwyn R. Peachey, "Solid Texturing of Complex Surfaces", Proc. Siggraph '85, *Computer Graphics*, **19**, 3 (July 1985), pp. 279-286.

[Perlin85] Ken Perlin, "An Image Synthesizer", Proc. Siggraph '85, *Computer Graphics*, **19**, 3 (July 1985), pp. 287-296.

[Pfister85] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K.P.McAuliffe, E. A. Melton, V.A. Norton, and J. Weiss, "The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture", in *Proceedings of International Conference on Parallel Processing*, pp. 764-771, 1985.

[Potmesil87] Michael Potmesil and Eric M. Hoffert, "FRAMES: Software Tools for Modeling Rendering and Animation of 3D Scenes", Proc. Siggraph '87, *Computer Graphics*, **21**, 3 (July 1987), pp. 85-93.

[Ranade88] Abhiram G. Ranade, Sandeep N. Bhatt and S. Lennart Johnsson, "The Fluent Abstract Machine", TR-573, Dept. Computer Science, Yale U., January 1988.

[Rashid88] Rick Rashid, Chair, "Supercomputer Graphics Workstations - The Advent of the Crayola", *Panel Session at IEEE 2nd Conference on Computer Workstations*, Santa Clara, CA, March 1988.

[Rettberg86] Randall Rettberg. and Robert Thomas, "Contention is no Obstacle to Shared Memory Multiprocessing", *Communications of the ACM*, **29**, 12, Dec 1986., pp. 1202-1212

[Russell78] Richard M. Russell, "The CRAY-1 Computer System", *Communications of the ACM*, **21**, 1, Jan 1978, pp. 63-72

[Rougelot69] Rodney S. Rougelot, "The General Electric Computer Color TV Display", in M. Faiman and J. Nievergelt, Eds., *Pertinent Concepts in Computer Graphics*, University of Illinois Press, Urbana, 1969.

[Schachter83] Bruce J. Schachter, *Computer Image Generation*, John Wiley & Sons, New York, 1983.

[Schumaker80] R. A. Schumaker, "A New Visual System Architecture", Proceedings, 2nd Interservice/Industry Training Equipment Conference, 1980, pp. 94-101.

[Seitz85] Charles L. Seitz, "The Cosmic Cube", *Communications of the ACM*, **28**, 1, January 1985, *pp.* 22-33.

[Slotnick71] D. L. Slotnick, "The Fastest Computer", *Scientific American*, Feb. 1971, pp. 76-87.

[Snyder87] John M. Snyder and Alan H. Barr, "Ray Tracing Complex Models Containing Surface Tesselations", Proc. Siggraph '87, *Computer Graphics*, **21**, 3 (July 1987), pp. 119-128.

[Sutherland74] Ivan E. Sutherland, Robert F. Sproull, and Robert A. Schumaker, A Characterization of Ten Hidden-Surface Algorithms, *Computing Surveys*, **6**, 1, March 1974, *pp.* 1-55

[TMC87] "Connection Machine Model CM-2 Technical Summary", Tech. Report HA87-4, Thinking Machines Corp., April 1987.

[Thornton64] James E. Thornton, "Parallel Operation in the Control Data 6600", *AFIPS Proceedings of the Fall Joint Computer Conference*, **26**, pt. 2, 1964, pp. 33-40.

[Torborg87] John G. Torborg, "A parallel Processor Architecture for Graphics Arithmetic Operations", Proc. Siggraph '87, *Computer Graphics*, **21**, 3 (July 1987), pp. 197-204.

[Wallace87] John R. Wallace, Michael F. Cohen, and Donald P. Greenberg, "A Two-Pass Solution to the Rendering Equation: A Synthesis of Ray Tracing and Radiosity Methods", Proc. Siggraph '87, *Computer Graphics*, **21**, 3 (July 1987), pp. 311-320.

[Whelan82] Daniel S. Whelan, A Rectangular Area Filling Display Architecture, Proc. Siggraph '82, *Computer Graphics*, **16**, 3, July 1982, pp. 147-154.