TigerSHARK: A Hardware Accelerated Ray-tracing Engine

Greg Humphreys
Princeton University
humper@cs.princeton.edu

C. Scott Ananian
Princeton University
cananian@ee.princeton.edu

Abstract — The current state of the art in graphics rendering algorithms and hardware is surveyed, and it is shown that ray-tracing, despite larger computational requirements than conventional algorithms, is more amenable to massive parallelism. The TigerSHARK ray-tracing architecture is then presented as an extremely cost-effective means to exploit this fundamental parallelism. A competing approach is analyzed, and the TigerSHARK SIMD-variant architecture shown superior due to its use of low-cost high-performance reprogrammable computing elements and specialized architecture. Prototype hardware is briefly described. The system produces extremely high-quality output, efficiently using low-cost hardware to rival the rendering speeds of systems three orders of magnitude more expensive.

Introduction

The success of the Disney/Pixar film *Toy Story* has catapulted photorealistic computer graphics to the headlines, but the recent hype has, for the most part, obscured the technical details of the feat. Despite the farm of Sun workstations employed, the huge amounts of compute time involved forced Pixar to use decade-old techniques that sacrifice image realism and fidelity for the sake of speed. We propose a low-cost parallel architecture for ray-traced graphics to help eliminate the processing bottleneck. Ray-traced graphics produce much higher-quality output than scan-converting renderers (the technology used for *Toy Story*) but are currently too slow for wide-spread commercial use. Most existing hardware research has therefore concentrated on scan-converting renderers, but fundamental algorithm and hardware limitations affect the amount of parallelism obtainable. Ray-tracing, on the other hand, is amenable to massive parallelism, given the proper architecture. We propose a distributed DSP[1]-based architecture which promises scalability to thousands of processors. Each DSP could process over half a million ray-triangle intersections per second, and a PCI card hosting 16 DSPs would be capable of 8 million ray-triangle intersections per second.

Rendering Techniques

The diverse applications of computer graphics allow a large variety of different algorithms and techniques to be used. For example, the proliferation of 3D game engines has given rise to new classes of high-speed algorithms which sacrifice output resolution and quality for real-time display. More demanding are CAD/CAM applications, which relax the real-time constraints but require the ability to view and manipulate complex environments easily. Computer graphics' high end is filled by entertainment industry companies like Pixar and Industrial Light and Magic, who require extremely realistic output, regardless of the cost in rendering time or computational power.

Ray Tracing

One of the first successful image synthesis methods was ray-tracing [16]. Just as lensmakers would plot the path of light rays through a lens, the computer was used to compute the paths of light rays entering the eye. The light rays were followed backwards from the eye until they hit an object in the scene: it was then known that the light ray must have emanated from the object hit. The color of the ray entering the eye could then be computed, based on the color of the object the ray had been emitted from.

Although such ray-tracing systems are conceptually very simple, they occupy the high end of computer graphics applications. The large number of rays which must be traced backwards into the scene creates a huge compute cost. Ray-tracing produces extremely realistic images because it is based on a physical model: reflections, shadows, motion blurs, and other effects are generated easily from the basic ray-tracing technique. However, ray-tracing is prohibitively slow for production-work.

Scan Conversion

Scan converting renderers, on the other hand, can be made to run extremely quickly, at the cost of decreased realism. Instead of tracing light rays through the scene until they hit the model, scan converting renderers work directly with the model primitives. These primitives (often polygons or triangles) are transformed first into 2D space and then projected onto the viewing plane. Thus every primitive only has to be considered a maximum of once, rather than many times as necessary with iterative ray-tracing techniques.

Scan-converting renderers, although widely used for CAD/CAM applications and in the film industry, cannot produce scenes as visually appealing as can a ray-tracer. Most implementations of scan converting renderers deal only with triangles or simple polygons because of the difficulty of scan converting complex primitives. While it is possible to decompose an arbitrarily complex primitive into component triangles, this results in a huge increase in the number of primitives in the scene, and makes accurate representation of curved surfaces difficult. Because scan-conversion and ray-tracing are based on fundamentally different algorithms, it is much easier to add complex primitives to a ray-tracer.

More fundamentally, scan conversion systems have no inherent ability to represent reflection or refraction. Addition of such features generally amounts to embedding a ray-tracing system into the scan converter. This drawback is most obvious when trying to create accurate shadows, vital for realistic imagery; such 'simple' effects spring naturally from the ray-tracing algorithm, but are very difficult to synthesize in a scan converting renderer.

One notable exception is the REYES[2] system [1], which Pixar used as part of PhotoRealistic RenderMan to create *Toy Story.* REYES can synthesize realistic scenes, but it does so at the expense of speed. It is not unreasonable to expect a RenderMan scene to take days to render on a modern workstation.

Renderer Acceleration

The large amount of parallelism inherent in rendering tasks makes multi-processing a natural solution for graphics acceleration. However, the type of parallelism and architectures best suited to exploit it are very different for scan conversion systems and ray-tracers. Most extant work in the field has focused on scan converters [2,7]. In scan conversion, parallelism is, in general, achieved by distributing primitives to different processors. Several different stages in the rendering pipeline have been proposed as the appropriate place to do the communication [3], but all suffer from very poor worst-case performance. In addition, most existing parallel rendering systems require a shared memory or

1. Digital Signal Processor

2. Renders Everything You Ever Saw

message passing MIMD computer, which is an extremely expensive investment. This paper proposes parallel ray-tracing as a means to avoid the communications bottlenecks of scan conversion, and proposes a new architecture tailored for ray-tracing.

Task Description

The computational task of a scan converter differs substantially from that of a ray-tracer. The fundamental operation of a ray-tracer is computing ray-primitive intersections. The basic task is to find the first object that intersects each ray from the eyepoint. More intersections are computed to create reflection, refraction, shadow, and other effects in the image. Over 95% of the compute time of a ray-tracer is spent computing ray-primitive intersections [4,16], therefore this is the task first parallelized. Amdahl's law states that the other 5% of the problem will become increasingly important as the amount of parallelism utilized increases; a hierarchical approach will be introduced later which allows us to deal with this increase.

The basic task, then, is to simply intersect all primitives in a scene with a set of rays, outputting the (small) set of intersecting primitives. For a given ray projected into a scene, we can expect only about three to five intersections to be found [14,16]. From the nearest intersecting object, we will generate new rays to cast into the scene to compute shadows, reflections, refractions, and other natural effects.

Existing Approaches

There are a number of software acceleration techniques which often influence parallel implementations, primarily various partitioning methods. Partitioning allows us to reduce the number of ray-object intersection calculations by excluding various objects from consideration, sight unseen. Lin [10] mentions two widely used techniques: *hierarchical bounding*, which partitions a scene into a tree of enclosing volumes [6,9], and *space subdivision*, which divides the objects among a tree of uniformly sized spatial cells [5]. Only objects within the cells or bounding volumes through which a ray passes are tested against that ray. Scherson and Caspary [14] analyze the performance of such algorithmic improvements in depth; the performance gains realized are significant.

Parallel ray tracing is done either asynchronously using MIMD[1] architectures or synchronously using SIMD[2] architectures. MIMD computers are more general, and there is evidence that they are more efficient for scan converting renderers (see, for example, [2,7]) and so most research on parallel ray-tracing has focused on MIMD architectures. We contend, however, that the hardware resources of MIMD machines are not used efficiently for ray-tracing. SIMD architectures, such as the one we propose, perform the task as effectively with much less hardware cost and overhead.

Survey of Ray Tracing Architectures

MIMD Ray Tracing

Lin [10] describes two categories of MIMD parallel ray tracers: those that use *image-space partition algorithms* and those that use *object-space partition algorithms*. In image-space partitioning, the pixels in the output image are divided among the processors, with a single processor responsible for the entire ray-tracing task for those pixels. This obviously requires the entire image database to be accessible to all processors. Lin claims that shared memory is essential to this approach; however, multi-computers can implement this scheme equally well without any kind of shared memory protocol if the image primitives are simply broadcast to all nodes at initialization. The image components can be reassembled at completion. For complex images, such as those rendered for *Toy Story*, the communication penalty will be negligible.[3] Instead, the chief disadvantage of this method is the large hardware overhead. The distributed or shared memory used to store the image database, disk storage, operating system overhead, etc., are needlessly replicated. However, this solution can be implemented off-the-shelf with general-purpose hardware (as Pixar did for *Toy Story*), which is an advantage in commercial environments. Cost estimates for TigerSHARK, however, indicate that special purpose hardware of equivalent performance should cost a fraction of the cost of the general purpose multi-computer network. The hierarchy of parallel techniques utilized in TigerSHARK incorporates image-space partition as its topmost level.

Object-space partitioning derives from the software spatial subdivision technique described earlier. Each processor is responsible for all ray-tracing tasks within a spatial cell, and the object database is distributed among the processors according to that spatial division. All rays entering the cell are tested against the objects residing in the cell. Rays leaving the cell are passed off to the processor responsible for the cell it will be entering. Although shared-memory and memory replication are not issues, unbalanced loads, ray propagation overhead, and object fragmentation caused by objects spanning more than one cell create problems for this approach.

SIMD Ray Tracing

The graphics community has at times doubted the feasibility of using SIMD processors for ray tracing [11], but it has been shown [10] that SIMD approaches can be as efficient as MIMD designs. Most published architectures, however, are unable to take advantage of the performance benefits of the various partitioning schemes, and instead use a 'brute-force' approach, blindly comparing every ray against every object [11,13]. Lin [10] gives the speed-up of existing brute-force SIMD architectures using N processing elements as

$$N\left(\frac{(1+k)\,nf}{m}\right)$$

where n is the average number of objects tested against each ray by the optimal sequential algorithm; m is the number of objects in the scene; f, $f<1$, is the degradation caused by SIMD processing, and k is the overhead due to space traversal, for those architectures which use it. Since n approaches m as m decreases, this approach works well for simple scenes, but fails for the complex scenes more common in practice, where m/n can be greater than 1,000.

Clearly, a practical SIMD approach needs to be able to use space-partition methods to reduce the number of intersection calculations. This is more difficult with SIMD processors: one would think that rays could not be tested against different partitions of the object database without multiple instruction streams. In fact, several methods for accomplishing this are available. Lin describes a method using a partially data-driven architecture in [10]. We propose another method, using a hybrid architecture which takes advantage of spatial coherence.

The TigerSHARK Architecture

Our goal in designing a parallel ray-tracing accelerator is to tune the architecture to match the problem. SIMD designs for this task use hardware more efficiently, and existing supercomputer architectures [12] have demonstrated the performance benefits of using large numbers of very simple processors. We strive for very low cost-per-node, eliminating excess generality, to facilitate the use of large numbers of nodes.

---

1. Multiple Instruction stream, Multiple Data stream
2. Single Instruction stream, Multiple Data stream

3. For real-time processing of simple scenes, however, the bandwidth required for this approach is excessive; a shared-memory system (with higher inter-node bandwidth) will have to be used

Overview

Our proposed hybrid architecture consists of a collection of synchronous processing nodes connected to a single data memory, which they access in lock-step. The processing nodes are more complicated than the typical SIMD ALUs; in fact, they are full-featured digital signal processors, with a small amount of on-board memory and serial communications ports.

All memory accesses must be performed together; each processor reads the same memory location at the same time. Writes are disallowed for all but one processor, termed the master. Rays are distributed via the individual serial communications ports, daisy-chained together, and object intersections are collected the same way. The single memory holds the object database.

Note that this is not a pure SIMD architecture. The DSPs can copy their program information from the shared memory to their small internal memories, so that they do not require an external bus access for an instruction fetch. They can then branch and loop independently from the other processors, on the condition that the next external bus access must again be synchronized. A hardware primitive is provided to perform the resynchronization.

In practice, the first operation performed by the DSPs is to copy their programs to internal memory. They then use the daisy-chained serial bus to determine unique node ids, designating node 0 the write-privileged master. A host processor loads the shared memory with the object database, and then sends rays down the serial bus. In the simple case, each processor gets a different ray, and the rays are tested in parallel against the entire object database. Objects are, of course, fetched synchronously.

We saw above that such brute-force approaches (testing every ray against every object) suffer very poor comparative performance on complex scenes. This is avoided by utilizing the spatial coherence of the input rays and a simple voting primitive. 'Spatial coherence' refers to the fact that consecutive rays tested are extremely likely to be close together, spatially. For example, consecutive rays might belong to an adjacent pair of pixels in the output image. This spatial similarity means that they will also have similar intersection properties: if the first ray intersects a primitive or bounding volume, it is very likely the next ray will, as well. Obviously, some forethought is required to ensure that most of the spatial coherence present in the original task (adjacent pixels, etc.) is preserved in the ray ordering as seen by the DSPs.

If we can guarantee this spatial coherence, however, we can also implement bounding-volume partitioning. Each ray is tested against a boundary volume, and then tells its neighbors whether or not it intersected. If none of the rays intersect the boundary volume then we need not examine the objects inside it. Otherwise, all the processors continue to examine the objects inside the bounding volume, since one of them may find an intersection. The degree of ray spatial coherence determines how probable a unanimous vote is.

This architecture requires very little hardware to implement a parallel system. Obviously, if the number of processors is greater than the amount of spatial coherence present, efficiency is poor. Anti-aliasing and stochastic sampling are two common techniques to improve image quality which utilize multiple rays per pixel. High resolution images (where the pixels are 'smaller' with respect to image variations) and multiple rays per pixel both provide large amounts of spatial coherence which can be utilized. Full implementation of the prototype system will provide a definitive measure of the amount of spatial coherence which can be exploited, but 16 to 32 processor systems should certainly be possible without much performance degradation. We use a hierarchical organization to obtain further parallelism.

Hierarchical Parallelism

The prototype system is implemented as a PCI card hosted in a personal computer. The second level of parallelism is to add multiple processor 'cards' to the host. Each card contains a single object database memory, and a number of DSP processing elements. To improve latency times and reduce the amount of on-board memory required, we split up the object database among the PCI cards in the system. We then provide the same rays to each card (we need not waste any of our potential spatial coherency here), reading and merging the intersections found by the two cards. The limiting factor now is the host processor power. As previously shown, that over 95% of the processing task is ray-primitive intersections; the remaining part of the task falls on the host processor. Obviously, as we continue to expand the system, we will want to provide multiple host processors, to avoid a bottleneck. The third level of parallelism is then to add multiple hosts, each with multiple PCI cards holding multiple DSPs. The task is divided up among the hosts using an image-space partition: each host gets a section of the final output image and the entire object database, renders independently of the other hosts, and assembles the final output image when the hosts have all completed.

So the complete system contains three separate levels of parallelism: ray-vector parallelism at the lowest level, followed by object database distribution above it, and capped by an image-space partitioning scheme.

TIGERSHARK SYSTEM HARDWARE

Design Process

The first design parameter we set was the target performance level. We wanted to be able to render complex realistic scenes, on the scale of those created for *Toy Story*, at a better price-performance ratio than available with multi-computer arrangements. Complex scenes meant that we needed support for object database partitioning, and the recent announcement of Advanced Rendering Technologies' AR250-64 ray-tracing system [15] set our performance goal: to be competitive, we should be able to compute close to 10^9 ray-triangle intersections per second. An initial estimate placed this as the compute power of a 512-DSP array; subsequent architecture design thus called for scalability to 512 processors. As we will discuss, actual system performance is not directly comparable on the ray-primitive level, since the AR250-64 system and TIGERSHARK use different sets of primitives; instead, scalability to 512 processors replaces raw compute speed as our performance target.

Digital signal processors seemed a good match to the low component-cost, high floating-point[1] performance required. General-purpose processors typically either required too much support hardware, cost too much, or provided too little floating-point performance to be viable options.

Various system bus options were then considered, starting with traditional shared-memory MIMD systems. The shared-memory hardware overhead and SIMD/MIMD issues did not generally justify the standard shared-memory architecture, but we examined a few variations on the architecture before discarding the idea. The Analog Devices SHARC DSP was considered as a processing node: it provides built in support for global-memory architectures, and includes substantial on-chip local memory (reducing necessary shared-memory bandwidth), but the high cost of this processor was a heavy disadvantage. Dual bus systems, like the Motorola DSP96002 processor, were evaluated as possible low-cost solutions to providing shared memory; however, the required support hardware and high processor cost prompted a search for other options.

Once the SIMD architecture was decided upon, it was fairly easy to select the Texas Instruments TMS320C32 60-MFLOP DSP on the basis of its extremely low cost (less than $10 each in quantity [8]) and excellent performance.

---

1. Investigation was also done into the applicability of fixed-point processors: the results strongly favored floating-point; the reasons are outside the scope of this paper.

Hardware Description

An array of TMS320C32 DSPs are supplied with identical clock signals via a low-skew driver for synchronization. The model database is stored in zero-wait state SRAM, and the ray data input and intersection data output are via a high-speed serial port on the TMS320C32.

Only one DSP is connected to the address lines of the memory, since all DSPs are guaranteed to be driving the same address on any given cycle. Output from the SRAM is via a high-speed buffer to enable the SRAM to drive the data-bus inputs of a large number of DSPs.

The DSP array is hosted on a PCI bus system; for the prototype, this is a Pentium PC. The PCI interface is via the AMCC S5933 'Matchmaker' chip; the host processor can directly load object database information into card SRAM, and a 32-word FIFO is used to buffer ray and intersection data coming to and from the device. The PCI interface is capable of bus mastering for added performance.

The TMS320C32 DSPs can execute 60 MFLOPS peak; estimated ray-tracing performance is on the order of half a million ray-triangle intersections per second (estimating worst-case 100 instructions per ray-primitive intersection).

Cost/Performance Issues

In order to keep the architecture inexpensive and uncomplicated we decided not to share the DSP bus with PCI. Originally, the model database was double-buffered to enable the PCI interface to update the model primitives without interrupting the DSPs. However, calculation showed that typical performance loss due to database updating remained under 1% for typical processor loadings,[1] so in the interest of reducing costs the double-buffering was eliminated. PCI access to the SRAM now requires the DSPs to yield the bus, but system costs are reduced almost 30% to 40%.[2]

Our calculations also verified that the bandwidth available on the PCI bus was enough to sustain our target 512-processor system. Bus mastering and burst writes were employed to utilize the full bus bandwidth.

The size of the SRAM for model database storage was minimized subject to bandwidth and performance constraints. An overly small SRAM would create large amounts of paging traffic as parts of a bigger database are paged into local memory, but large amounts of fast SRAM are very expensive. It was found that 1Mb of SRAM was the smallest practical size for the target 512-processor implementation. The serial communication bandwidth turned out to be the limiting factor: a smaller SRAM would finish iterating rays through the object database before new rays could be transmitted to the DSPs.

Parallelism Issues

The TIGERSHARK system can achieve near linear speedup on ray-primitive intersections. However, processing the output intersections must be done by the host processor. TIGERSHARK uses a hierarchical structure to ensure that this processing will not become a bottleneck for the system.

At the top level of the hierarchy is the image to be rendered. This is split up among several host computers, in the way described for multicomputer systems. Each host in turn distributes the model primitives among the different PCI boards it contains; each PCI board has its own shared memory. The task is further subdivided by the PCI board; each DSP on the board tests the primitives against a single ray. Bottlenecks can be eliminated by changing the ratio of DSPs to PCI boards, or the number of boards per host. Research will certainly be done on the prototype system to identify the best ratios here.

System Evaluation and Comparison

To evaluate the effectiveness of the TIGERSHARK architecture, the system is compared against Advanced Rendering Technologies' AR250-64.[3] Advanced Rendering Technologies (ART) is a Cambridge-based company that announced on 2 October 1995 its plans to build a custom VLSI processor for ray-tracing applications. First samples are anticipated 4Q 1996. Although the goal of both TIGERSHARK and ART is to accelerate ray-tracing, the relevant architectures are very different. It is unwise to make claims about the full 512-DSP TIGERSHARK system until scalability can be empirically verified, so instead we will compare a single TIGERSHARK PCI board to a single module of ART's AR250-64.

The AR250-64 is made up of 64 AR250 ray tracing engines. Each AR250 is claimed to be able to perform 80,000,000 ray intersection tests per second, "roughly 16 times the performance of the best graphics workstations." We compare this single AR250 with a TIGERSHARK board containing 16 TMS320C32 DSPs.

ART's AR250 clearly holds the lead in raw speed. The AR250's custom silicon allows it to perform almost 4 GFLOPS, about 4 times more raw floating-point performance that TIGERSHARK, and the AR250 can perform an order of magnitude more ray-triangle intersections per second than can TIGERSHARK.

TIGERSHARK makes up the deficit in flexibility, however. ART's decision to go for custom silicon entails sacrificing some complexity: in particular, the AR250 is only able to implement triangle primitives on-chip. All other primitives must be decomposed into triangles before they can be rendered. TIGERSHARK's reprogrammable DSPs allow it to perform any type of ray-primitive intersection, which allows a lot of relative performance gain. For example, the AR250 must decompose a sphere in the object database into hundreds of component triangles, while TIGERSHARK can process the sphere in one operation, as a single primitive. Moreover, for TIGERSHARK ray-sphere intersections are actually easier to perform than ray-triangle intersections. TIGERSHARK is therefore an order of magnitude *faster* than the AR250 on this task. On the assumption that most interesting scenes are fairly complex, with lots of non-flat surfaces that the AR250 will need to triangulate, a TIGERSHARK board may very well equal or surpass the AR250's performance.[4]

In addition, TIGERSHARK has a much better price/performance ratio. ART is using a low-volume custom ASIC, with the price penalty that that entails. TIGERSHARK, on the other hand, uses high-volume TI DSPs to achieve a very low node cost.

Overall, we are optimistic that the TIGERSHARK system will be able to match the performance of ART's custom silicon while maintaining a much lower price. Neither system has been fully prototyped yet; scalability is likely to be key to a comparison of operational systems. The results are certainly positive enough to justify a closer look at hybrid SIMD architectures for ray-tracing applications.

Conclusion

TIGERSHARK was designed to achieve the three goals most prized by professional computer graphic artists: speed, cost, and quality. A high-end Silicon Graphics machine is fast, at the expense of cost and quality; a software ray-tracer can produce high quality output but renders extremely slowly; and hardware approaches to ray-tracing have

1. 100,000 rays iterated through object database between updates; 512-processor system
2. RAM typically accounts for 60% of the cost of an average workstation [12]; the percentage is even higher for the TIGERSHARK board, since the processor cost is so low. Eliminating double-buffering allows us to use half as much RAM per board.
3. All data on Advanced Rendering Technologies' products is from their World Wide Web site [15].
4. Furthermore, it is not clear that ART has implemented or can implement a partitioning scheme in its hardware to avoid brute-forcing the object database.

thus far been extremely expensive due to their use of general-purpose hardware.

The TIGERSHARK system provides extremely high quality output, scales well due to the task's inherent parallelism, and uses low-cost hardware extremely efficiently. We believe that a fully expanded TIGERSHARK system would be able to rival the rendering speed of the fastest graphics systems available, at approximately three orders of magnitude less cost. Our hybrid architecture shows great promise for ray-tracing applications, challenging even custom silicon. In addition, the flexibility of the general purpose DSP chips enables us to add new primitives and even procedural shading techniques to achieve rich, complex, and realistic ray-traced scenes at speeds previously unattainable. We believe that TIGERSHARK will set a new price/performance benchmark for graphics systems, and will push ray-tracing towards the realm of real-time realistic image synthesis.

Work is in progress on the construction of a 4-DSP TIGERSHARK prototype to verify performance, feasibility, and scalability.

REFERENCES

[1] Robert L. Cook, Loren Carpenter, and Edwin Catmull, "The Reyes image rendering architecture," in *Computer Graphics (SIGGRAPH '87 Proceedings)*, volume 21, pages 95--102, July 1987.

[2] Michael Cox, *Algorithms for Parallel Rendering*, PhD thesis, Princeton University, 1995.

[3] Michael Cox, Steven Molnar, David Ellsworth, and Henry Fuchs, "A sorting classification of parallel rendering," in *IEEE Computer Graphics and Algorithms*, pages 23--32, July 1994.

[4] Andrew Glassner, *An Introduction to Ray Tracing*, Academic Press, 1989.

[5] Andrew Glassner, "Space subdivision for fast ray tracing," *IEEE Computer Graphics and Applications*, 4(10):15--22, 1984.

[6] J. Goldsmith and J. Salmon, "Automatic creation of object hierarchies for ray tracing," *IEEE Computer Graphics and Applications*, 7(5):14--20, 1984.

[7] Greg Humphreys, *Parallel network objects and graphics*, Junior Independent Work, Princeton University, 1995.

[8] Texas Instruments Incorporated, *New TI floating-point DSP breaks cost barrier*, Texas Instruments ON-BOARD, 1(2), Spring 1995.

[9] T. L. Kay and J.T. Kajiya, "Ray tracing complex scenes," *Computer Graphics (SIGGRAPH '86 Proceedings)*, 20(4):269--278, 1986.

[10] T. T. Y. Lin and M. Slater. *Stochastic ray tracing using SIMD processor arrays*,The Visual Computer, 7:187--199, July 1991.

[11] J. Owczarczyk, "Ray tracing: a challenge for parallel processing," in *Proc Parallel Processing for Computer Vision and Display*, Leeds, 1988.

[12] David A. Patterson and John L. Hennessy. *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, 1996.

[13] David J. Plunkett and Michael J. Bailey, "The vectorization of a ray-tracing algorithm for improved execution speed," *IEEE Computer Graphics and Applications*, 5(8):52--60, August 1985.

[14] Isaac D. Scherson and Elisha Caspary, *Data structures and the time complexity of ray tracing*, The Visual Computer, 3(4):201--213, 1987.

[15] Advanced Rendering Technologies, World Wide Web site: `http://www.art.co.uk`.

[16] Turner Whitted, "An improved illumination model for shaded display," *CACM*, 23(6):343--349, June 1980.