Parallelization and Hardware Support for Ray Tracing

Alwin Groene and Oliver Renz *

Abstract

Even on the latest workstations ray tracing is still a very time-consuming algorithm. This paper makes a thorough analysis of previous attempts to accelerate ray tracing by means of parallelization with general purpose processors and by means of designing special purpose processors. Since much work has been done concurrently by many researchers, only the most important milestones are mentioned. The conclusions drawn are quite different from those in many other papers.

1 Introduction

Ray tracing is now a well established image synthesis technique and has produced some of the most realistic images to date. Glassner's book [15] gives a very good introduction into the subject. Even though many software acceleration techniques have been devised [3] and fast graphics workstations are now widely available, ray tracing is still too slow for many applications.

This paper makes a thorough analysis of previous attempts to accelerate ray tracing by means of parallelization with general purpose processors and by means of designing special purpose processors. It is an attempt to create a profound basis for promising future research work.

The next section encourages further work by exemplifying the many benefits of ray tracing. Section 3 illustrates its huge computational expense and section 4 presents a long list of characterization criteria. The two following sections carefully analyze previous papers on hardware acceleration techniques for ray tracing. Finally conclusions are drawn and a prospect of our further research work is given.

2 Advantages of Ray Tracing

Following is a list of the most important advantages of ray tracing, especially in comparison to standard scan conversion algorithms:

1. Apart from global diffuse illumination – this is approximated either by an ambient term or by stochastically sampling the environment – the ray tracing algorithm solves the global illumination problem.

*Universität Tübingen, Wilhelm-Schickard-Institut für Informatik, Graphisch-Interaktive Systeme, Auf der Morgenstelle 10/C9, W-7400 Tübingen, Germany, Email: alwin@gris.informatik.uni-tuebingen.de

2. The scene may comprise all kinds of objects for which

   - an intersection with a ray and
   - an object normal vector at the intersection point

   is computable. A few important examples are:

   - polygons,
   - parametric surfaces (Bézier/B-spline/Steiner patches, ...),
   - quadrics (spheres, ellipsoids, cylinders, cones, hyperboloids, paraboloids),
   - algebraic surfaces,
   - procedurally defined surfaces (surfaces of revolution, sweep surfaces, CSG objects, ...),
   - fractals, etc.

   There is no need to approximate everything with polygons. This means smaller model databases with more accurate scene descriptions.

3. Hidden surface removal, shadows, reflections, and transparencies are in inherent part of the ray tracing algorithm. No extra calculations are necessary.

4. Perspective transformation and clipping calculations are not necessary. Again these problems are automatically solved by the overall ray tracing idea.

5. Penetrating objects can easily be handled. There is no need for intersection calculations between objects.

6. Shades need only be computed for visible object points.

7. Distributed ray tracing [9] is able to render a whole range of fuzzy phenomena, such as gloss (blurred or hazy reflections), translucency (blurred transparency), penumbras, depth of field, and motion blur.

8. The usefulness of the hemi-cube algorithm [8] for computing form factors is limited due to aliasing artifacts caused by uniform sampling. Wallace et. al. [45] propose a ray tracing algorithm as a robust approach to perform the numerical integration of the form factor equation.

3 Computational Expense

The major reason why ray tracing is not much more widespread is its huge computational expense. Here is a typical example:

500×500 pixels	$\longrightarrow$ 250.000 pixels
4 primary rays per pixel	$\longrightarrow$ 1.000.000 primary rays
Average ray tree consists of 10 rays	$\longrightarrow$ 10.000.000 rays
Small scene comprises 2000 objects	$\longrightarrow$ 20.000.000.000 ray-object intersection tests
Exploitation of acceleration techniques	$\longrightarrow$ 1.000.000.000 ray-object or ray-extent intersection tests

One can easily change the numbers chosen above and come to a different solution, but we think it is a reasonable assumption that the overall number of intersection calculations, that have to be done, is at least at the order of one billion.

Thus real-time ray tracing with "reasonable" machines seems impossible for the foreseeable future.

4 Criteria

Ray tracing as understood in this paper was invented by Whitted in 1980 [49]. Since that year hundreds of researchers have tried to diversify and to accelerate this powerful but slow algorithm – the latest cross-indexed guide to the ray-tracing literature [40] mentions more than 500 papers.

In the following a check list to help characterizing these 500 ray tracing techniques and to aid designing new ray tracing systems is given. The list is divided into four parts: different levels of parallelism, use of coherence, hardware considerations, and finally other characterization criteria.

4.1 Different Levels of Parallelism

Parallel execution of the ray tracing algorithm can be performed at different levels. The following list is roughly sorted in order of increasing granularity:

Coordinate level. Three dimensional vector operations can be executed in parallel with respect to the x, y, and z coordinates. These operations occupy a major part of the ray tracing algorithm.

Pixel level. Different processors can shade different pixel areas at the same time. This means a partitioning of the image space.

Ray level. Since different rays are not dependent upon each other, they can be traced concurrently. Pixel level parallelism is the same as ray level parallelism for eye rays.

Subpatch level. Several chips have been developed to calculate the intersection of a ray with a spline patch. All of them use a divide and conquer strategy: they recursively subdivide a patch into four subpatches and intersect the ray with the subpatch bounding boxes until a termination criterion is met. The processing of the four subpatches could be done parallely.

Object level. Many ray tracing acceleration techniques use octree or hierarchical extent schemes to subdivide the object space. If the subspaces are distributed among the processors, the ray-object intersection calculations can be performed independently.

Tree level. In constructive solid geometry (CSG) objects can be represented as binary trees, where the leaves represent primary objects and the internal nodes set operations (union, intersection, difference). Usually every tree node is represented by one processor so that the whole tree can be evaluated in a systolic fashion: one tree level after the other.

Task level. The shading of a pixel involves the following tasks [14]:

- intersection calculations of rays with objects,

- address calculations and database searches, and
- accumulation of contributions from primary/secondary rays to obtain the final pixel color.

Provided that many rays are traced at the same time, all these tasks can be executed parallely.

Frame level. For animation sequences, different frames can be computed at the same time.

Many variations and extensions of these eight parallelism types are thinkable.

4.2 Use of Coherence

Sutherland et. al. [42] define the term coherence as *the extent to which the environment or the picture of it is locally constant.* The proper use of coherence can vastly increase the speed of many graphics algorithms. The following kinds of coherence have been identified in numerous ray tracing papers:

Object coherence. Local neighborhoods of space tend to be occupied by the same object, and distinct objects are likely to be disjoint in this space. This property can be exploited by partitioning the object space and testing only those objects for intersection, that approximately lie along the rays' paths.

Area or image coherence is the property that adjacent pixels on a display device are often covered by the same visible object.

Ray coherence. Rays whose origins and directions are almost equal are likely to intersect with the same objects in the environment.

Temporal or frame coherence is the property that consecutive frames of an animation sequence tend to be very similar, despite small changes in objects and viewpoint. The ray tracer can significantly reduce the processing time for one frame by reusing the results of the previous frame.

Data coherence. This term was introduced in [16]. It is related to the term *locality of reference* in virtual-memory management systems and means that most database references account for only a small subset of the objects. This property can be exploited by using fast data caches.

4.3 Hardware Considerations

Load balancing. An acceleration of factor n can only be achieved if all n processors have the same work load to perform. The equal distribution of the load can be achieved **statically** before the actual tracing of rays starts or **dynamically** during the ray tracing calculations.

Communication overhead. Every execution of n subproblems, especially when performed asynchronously, needs some management overhead to distribute the subproblems, to communicate between the PEs, and to collect the subresults. This overhead should be as small as possible. In addition, deadlocks have to be properly prevented.

Configurability. The machine should be configurable with respect to the number of processors (hardware components) so that each customer can make his or her optimal cost/performance choice.

Special purpose hardware. The use of special purpose hardware often results in better performance. General purpose processors on the other hand are usually cheaper and freely programmable. This latter property is important to support the further development of new algorithms.

Accuracy. In terms of accuracy floating point arithmetic is preferable to fixed point arithmetic. If fixed point arithmetic is used, the input data has to be properly scaled in order to avoid register overflows and underflows.

Software environment. The successful use of any hardware architecture requires a good method for programming the system. The efficient support by device drivers and (existing standard or custom) compilers is essential.

Programmibility. Quite often special architectures could support more applications than formerly intended if they were just slightly more flexible. The lesson one should learn from this is to make even special purpose hardware as much programmable as possible.

I/O speed. Many authors report I/O as the major bottleneck of their architecture. Reduced I/O speed limits the usefulness of any hardware system.

Regularity. The use of iteration to form arrays of identical cells simplifies the design and testing of VLSI circuits.

4.4 Further Characterization Criteria

The different kinds of parallelism, coherence and hardware features provide already a good taxonomy for examining ray tracing machines. Nevertheless there are still many design decisions that can only be characterized by the following list of criteria:

Size of model database. Does the machine support arbitrary large model databases?

Primitive types. A flexible ray tracing machine shouldn't force the application programmer to model the scene with just one type of primitives.

Acceleration. What order of acceleration can be achieved in comparison to a single processor machine? Is it possible to perform interactive or even near real-time ray tracing?

Antialiasing. Ray tracing is inherently a discrete technique because it only traces a finite number of rays. This causes aliasing problems because the infinitely thin rays determine the color of pixel areas. This means that the ray tracing algorithm must comprise some sort of antialiasing in order to be useful for different scenes. Therefore any good ray tracing machine must support antialising.

Speed/quality tradeoff. Since it seems impossible to have real time ray tracing in the foreseeable future, a ray tracing machine should provide the possibility to trade off speed for quality, i.e. the machine should be able to compute a *quick and dirty* as well as a slow and good image. This can easily be done by tracing varying numbers

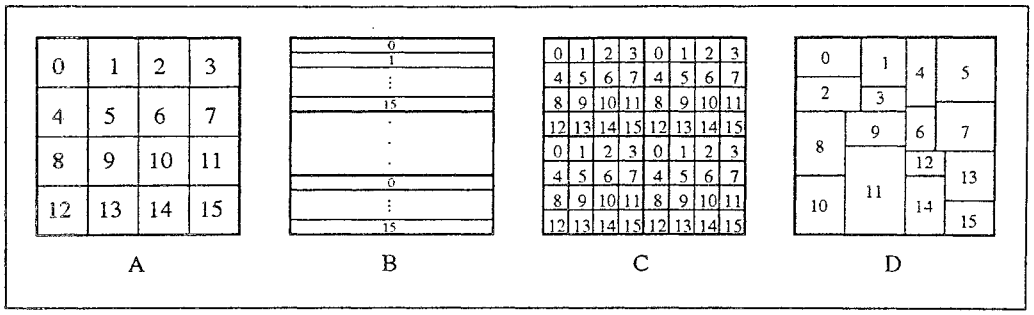

Figure 1: Static subdivision

of rays to varying ray tree depths. The possibility to perform adaptive refinement ray tracing should also be provided.

Flexibility. The hardware should have the flexibility to support different ray tracing techniques. Especially illumination modeling is still a very active research area.

5 Parallelization with General Purpose Processors

5.1 Image Space Subdivision

The main idea behind image space subdivision strategies [31, 10, 36, 35, 50] is to divide the image plane into a number of distinct regions and to distribute these regions among the n processing elements (PEs). Each PE then independently computes the whole ray tree for all the primary rays going through the pixels of its region(s). Two major problems have to be solved: The work load must be distributed evenly among the PEs and each PE has to have access to the whole model database.

Load balancing

The balancing of work loads can be achieved statically or dynamically. Static load balancing means that the image plane regions are determined and distributed in a pre-process. Several choices are possible:

A Divide the image plane into n equal sized regions.

B Give each PE each n-th scanline to compute.

C Each PE is given each $\sqrt{n}$-th pixel horizontally and vertically.

D Let the host computer (or the PEs) perform coarse resolution ray tracing (e.g. every 8th pixel horizontally and vertically) in a pre-processing step. The work load for these pixels is recorded. Assuming that neighboring pixels will have a similar load associated with them, a good static load balancing can be achieved by recursive binary subdivision of the image plane – a region is subdivided such that both subregions have the same work load associated with them.

If only load balancing is taken into consideration, A will be the worst and either B, C, or D be the best choice, because the load will hardly ever be distributed evenly over the image plane. Unfortunately the preprocessing step in D is very time consuming and C makes very poor use of coherence.

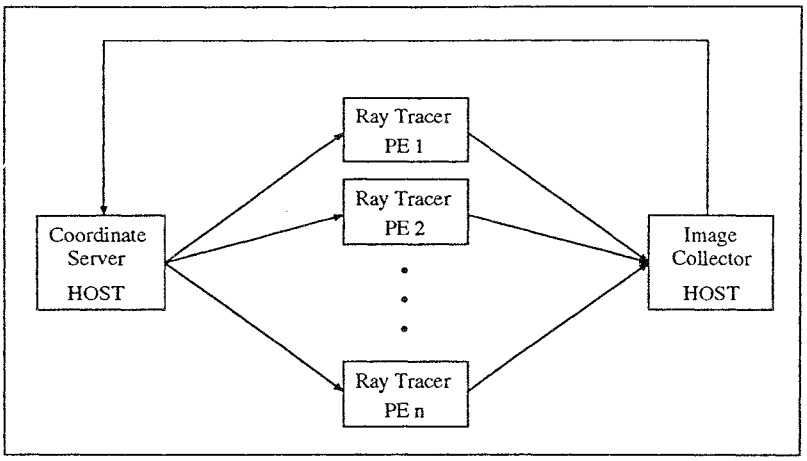

Figure 2: Dynamic subdivision [31]

Dynamic load balancing may be achieved as proposed by Orcutt [31]. At the beginning each PE gets a group of pixels and starts computing its intensities/colors. Whenever a PE has finished, it sends the pixel colors to the image collector and gets a new pixel group from the coordinate server.

Storage of model database

If there is enough local memory available than each PE can store a copy of the whole database. This is the easiest solution and the obvious choice if e.g. the PEs are workstations.

If the computer under consideration is a multiprocessor machine with a large number of processors, then the database cannot be duplicated n times because usually the local memory modules are of limited capacity. The alternative approach here is to store the model database once in global shared memory and to organize the local memories as caches in order to alleviate the access bottleneck problem. Green and Paddon [16, 17, 18, 19] made a thorough analysis of how to efficiently organize main and cache memories. They propose to arrange the processor system into a tree structure with the controller processor placed at its root. This distributes the root fanout over the whole tree.

Distributed memory is out of question here because each PE has to have regular access to each part of the database. This would cause a huge interprocessor communication overhead and could easily end in a deadlock situation.

Overall characterization of image space subdivision architectures

\+ **High flexibility**

Nothing was said about how the PEs trace the rays through their pixels and how the shading calculations look like. Any existing and future software technique for ray tracing is easily supported.

\+ **Near linear speedup**

Since no communication between PEs is necessary, a near linear speedup is possible.

± **Large databases**

There seems to be no efficient solution for extremely large databases. Image space subdivision is best suited for MIMD architectures with relatively few but powerful processors with large amounts of memory.

± **Coherence**

The architecture doesn't directly exploit any form of coherence – especially not object coherence – but the software running on the PEs has every possibility to do so.

\+ **Special purpose hardware**

There is a lot of potentiality to support the PEs with special purpose hardware. A few examples are: intersection calculations [34, 44, 5, 39], 3D vector operations [47, 50], and octree traversal [1].

\+ **Interactive ray tracing**

Interactive ray tracing – one frame in less than one minute even for complex scenes – is possible in the foreseeable future.

5.2 Object Space Subdivision and Hierarchical Trees

Since Whitted's paper was published in 1980 [49], numerous software acceleration techniques have been devised [3]. These techniques can be roughly classified according to the data structure they use: uniform 3D grids of voxels, octrees, BSP trees, and hierarchical trees of extents. Many attempts have been made to map these acceleration data structures onto a suitable multi-processor topology. The general idea is to distribute the objects among the PEs. Each PE then computes intersections of rays with only a small subset of the scene objects. Propagation of rays through object space is achieved by interprocessor communication. These data driven architectures can only be successful if a good load balancing strategy is found and the number of ray messages is minimized.

Nonuniform static subdivision

Kobayashi et. al. [27] use an adaptive division graph based on octrees to partition the 3D object space. Each PE stores all objects of one octree leaf voxel in its local memory. The PEs also store information about location and associated PE addresses of all neighboring voxels. A hypercube (binary n-cube) network is used to propagate the ray message packets.

Hierarchical tree of extents (HTE)

HTEs can be used to speed up ray tracing. The scene objects are first represented as a hierarchy of bounding volumes. Intersection testing is done by a depth-first traversal of the tree, pruning a branch whenever a bounding volume is missed entirely.

Both Salmon and Goldsmith [37] as well as Scherson and Caspary [38, 6] propose similar systems for a parallel implementation of these tree-tracing methods. The general idea is to divide the HTE into an upper tree and a set of lower trees. Each PE stores a copy of the upper tree while each of the lower trees is randomly sent to only one of the PEs. This allocation scheme builds the basis for dynamic load balancing. Any free PE can shoot a ray into the scene by starting its upper tree searching process. When a leaf of the upper tree is reached, a ray message is sent to the corresponding lower tree PE which then continues with the lower tree traversal.

$2\frac{1}{2}$D static subdivision

Another possibility is to use only two dimensions for the subdivision process such that the three dimensional scene bounding box is partitioned into a number of parallel columns. Each PE is responsible for one column. To achieve a good static load balancing, Kobayashi et. al. [28, 26] suggest distribution schemes similar to those in Fig. 1 A and C. Other researchers prefer to use a sub-sampling preprocessing phase in order to have a rough work estimate for a regular grid of columns. This grid is then subdivided similar to Fig. 1D either by recursive binary subdivision [4, 32, 33] or by means of a graph partitioning scheme [22].

Regular 3D grid of voxels

Caubet et. al. [7] suggest to subdivide the object space into a regular grid of $10 \times 10 \times 10$ metavoxels. Each metavoxel processor further subdivides its subspace into $100 \times 100 \times 100 = 1.000.000$ voxels. The major advantage of the VOXAR architecture is that ray tracing is reduced to incremental integer calculations but the machine suffers from insufficient load balancing, severe aliasing artifacts, expensive interprocessor communication, and large storage problems.

Dynamic load balancing

Some early papers [11, 30] discuss dynamic scheduling performed at run time. Whenever there is a load imbalance between neighboring PEs, objects are locally redistributed by sliding subspace boundaries. The overhead to detect and rebalance work load differences inevitably degrades the performance of the system.

Communication overhead

The biggest problem with object space subdivision methods is the huge communication overhead caused by passing rays between PEs. Here is an example of what information a ray message structure typically at least contains:

		bytes
ray origin	3 × float	12
ray direction	3 × float	12
ray length	1 × float	4
ray color	3 × byte	3
ray depth	1 × byte	1
pointer to object	1 × address	4
...		
		$\Sigma = 36$

If we consider again our example from section 3 (10.000.000 rays) and a machine with tens, hundreds, or even thousands of processors, we come to the conclusion that the sum of all ray messages lies in the range of Giga bytes, possibly even Tera bytes.

Overall characterization of object space subdivision architectures

+ **Spatial coherence**

  Object space partitioning systems efficiently exploit spacial coherence by performing intersection tests only for those objects that approximately lie along the rays' paths.

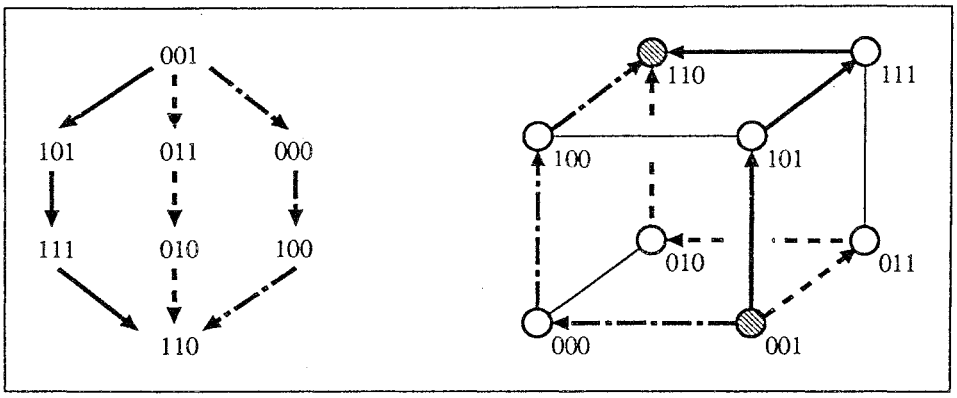

Figure 3: Hypercube message routing

+ **Large databases**

  Because the objects are distributed among the PEs, even very large model databases can be handled.

– **Communication overhead**

  As outlined above, these architectures cause a huge interprocessor communication overhead. The more PEs contribute, the larger is the overhead.

– **Internal fragmentation**

  Objects intersected by voxel boundaries end up in more than one voxel. This potentially requires a ray to be intersected with the same object more than once. Arnaldi's mailbox technique [2] to prevent this cannot be applied here, because it requires each object to be stored exactly once. The more PEs are involved, the finer is the subdivision and the larger becomes the internal fragmentation problem.

– **Deadlocks**

  If any PE can send messages to any other PE, then there is a high possibility of deadlocks. To prevent this, Green [18] proposes a costly handshake protocol.

± **Hypercube message routing**

  Because the hypercube combines many advantageous characteristics (2^n nodes, $n2^{n-1}$ links, dimension = diameter = degree = n, simple message routing, high fault tolerance) most researchers prefer it to other interconnection topologies. Nevertheless a message may have to pass up to $n-1$ PEs before it finally reaches the destination PE, thus ray message sending is slow.

– **Speedup**

  Due to the communication and fragmentation problems, the achievable speedup is only sublinear.

± **Special purpose hardware**

  Most object space subdivision systems reported are intended for a large number of PEs. It would be very costly to support every PE e.g. with intersection hardware.

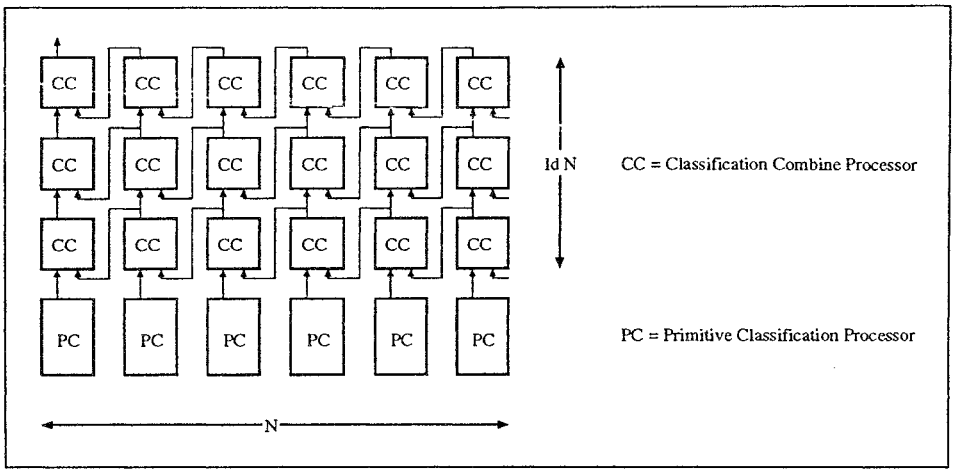

Figure 4: Block Diagram of Raycasting Machine [23]

– **Software**

  In comparison to image space subdivision systems, the software to run on these systems is more determined by the hardware, i.e., existing software may have to be completely redesigned.

6 Special Purpose Hardware for Ray Tracing

We consider the following eight approaches to accelerate ray tracing with dedicated graphics hardware as promising.

Raycasting Machine

The famous Ray Casting Maschine by Kedem and Ellis [23, 25, 24, 12, 13] classifies a grid of parallel lines against a CSG object. It mainly consists of bit serial processors that mirror the CSG tree. The primitive classification processors (PCs) at the leaves of the tree concurrently compute the intersection of a line with the primitive solids. The classification-combine processors (CCs) at the internal nodes of the tree compute the set operations on streams of internal segments. The results come out of the root processor.

Unfortunately, the Raycasting Machine is of limited use for many graphics applications. Currently, the CSG objects are restricted to boolean combinations of linear and quadric halfspaces. This means that even many relatively simple objects (e.g. machine parts) can only be roughly approximated with this form of ray casting. Moreover, the machine only solves the visible surface problem because only parallel eye rays are traced. Consequently there are no shadows, no reflections, and there is no perspective view of the scene. Theoretically light rays for light sources at infinity could be traced in a second pass, but matching of eye ray intersection points and light ray intersection points would require additional calculations.

Ray-Patch Intersections

Pulleyblank and Kapenga [34] developed a VLSI chip for ray tracing bicubic patches in Bézier form. To find the intersection of a patch with a ray, the patch is broken into four

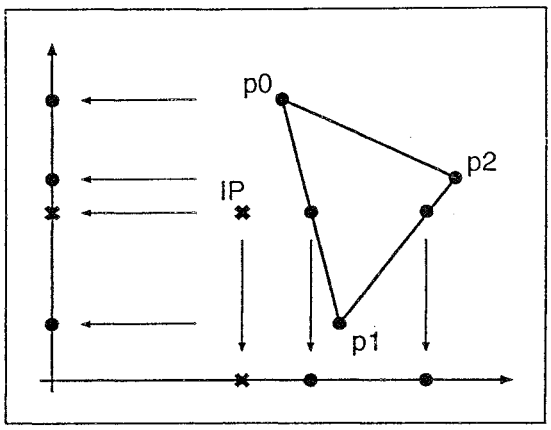

Figure 5: Ray-Triangle Intersection [44]

subpatches, each of whose bounding boxes are computed and tested for an intersection with the ray. If the ray hits the bounding box of a subpatch and the termination conditions are not met, i.e. the patch is not smaller than a specified accuracy requirement and the maximum level of subdivision has not been reached, the subpatch is placed on a stack to be processed further. Because of the large number of additions the implementation is only bit serial but parallel in x, y and z.

Bouatouch et. al. [5] made a similar design to the one described above but gave more implementation details.

Schneider [39] extended the idea of ray tracing splines to the more general rational B-spline patches because non-rational splines cannot precisely define conic sections. He performed subdivision with a modified OSLO algorithm.

Ray-Triangle Intersections

Voorhies and Kirk [44] reported an efficient algorithm to determine if a line segment and a triangle intersect. The algorithm is predominately based on binary subdivision and simple comparisons and therefore extremely favourable for VLSI implementation. The intersection of the line segment with the triangle plane is found by recursively subdividing the line until the midpoint has distance zero from the plane.

Next the triangle vertices and the intersection point are projected to 2D by simply dropping the coordinate, whose absolute value in the plane normal is largest. A preliminary point-in-triangle test is performed by projecting the triangle vertices and the intersection point onto one of the remaining two axes. Only if the intersection point's projection lies within the extent of two triangle edges can the intersection point lie within the triangle. The actual triangle edge locations that project onto the same point as the intersection point are found by binary recursive subdivision along the chosen axis. Finally the same procedure (extent test and binary subdivision) is performed with these two edge locations and the intersection point, but this time along the second axis (see Fig 5).

Given the (u,v) coordinates of the triangle vertices ((0,0), (0,1), (1,0)), the last three binary subdivision processes may additionally compute the (u,v) coordinates of the edge locations and the intersection point. This solution to the inverse texture mapping problem could easily be extended to arbitrary (u,v) coordinates for the triangle vertices, which would be more useful if, as is mostly the case, triangles serve only as approximations for more complex surfaces.

The only problem seems to be the determination of the distances of the initial line endpoints from the triangle plane, which means evaluating the plane equation in floating point arithmetic. Besides the algorithm needs two line endpoints as input instead of a starting point and a direction, which is the usual definition of a ray.

3DP Processor

The 3DP [47] is a parallel architecture that operates on length-3 vectors. Its execution unit consists of three arithmetic-logic units (ALUs) and three floating-point units (FPUs) operating in parallel and is further supported by two crossbar switches. Since 3D vector operations dominate the intersection and lighting calculations of ray tracers, they could be efficiently computed on the 3DP: cross product ($\mathbf{v} = \mathbf{v}_1 \times \mathbf{v}_2$), dot product ($\mathbf{v} = \mathbf{v}_1 \cdot \mathbf{v}_2$), sum/difference ($\mathbf{v} = \mathbf{v}_1 \pm \mathbf{v}_2$), scaling ($\mathbf{v} = a \cdot \mathbf{v}_1$), etc. Unfortunately, there is no square root unit to aid computing normalized vectors.

CORDIC Processor

CORDIC is an acronym for COrdinate Rotation DIgital Computer. It can be used for rotating a planar vector $\mathbf{v}$ as well as computing its length and phase in different coordinate systems m [43, 46]. Since the CORDIC method is an iterative algorithm based on shift and add operations only, it is extremely suitable for VLSI implementation. Kocsis and Böhme [29] report many 3D graphics applications for CORDIC processors, including computations for

- point-line and point-plane distances,
- the angle between two vectors (scalar product),
- point inclusion problems for convex polygons,
- ray-plane intersections,
- ray-quadric surface intersections,
- ray-bicubic surface intersections, and
- intensity calculations.

Octree Traversal

Agate et. al. [1] developed the HERO algorithm for rapid traversal of octree data structures. The algorithm generates the addresses of child voxels in the order they are penetrated by the ray. Since explicit ray-voxel intersection calculations are avoided, only simple arithmetic and logic operations are required. Both software and hardware implementations are discussed.

Item Buffer

The item buffer [48] speeds up ray tracing for eye rays. A z-buffer algorithm is executed to record the identity of the closest object for all item buffer pixels. Ray tracing a primary ray may then reduce to a simple buffer lookup. This algorithm can be supported by scan-conversion and z-buffer hardware.

The light buffer [20], though similar in concept to the item buffer, is more difficult to support because it requires general list handling facilities.

Area Sampling Machine

The most complex ray tracing machine known to us is the Area Sampling Machine (ASM) proposed by Sung [41]. Instead of point sampling the environment by tracing single rays, the ASM performs area sampling by parallely tracing bundles of rays bounded by a viewing frustrum. This is performed by up to 85 Area Samplers (ASs), each of which is a simple z-buffer style rendering pipeline. The scene database is stored once and periodically broadcasted to all ASs. The ASM is a pure visibility determination machine; the actual shading is performed in software on the host computer.

7 Conclusions and Future Work

Even though the basic ray tracing idea is quite simple, the actual software implementation of a ray tracer usually is quite a complex project. Due to different scene description and lighting models as well as different intersection, acceleration, and sampling techniques, numerous quite different ray tracers exist nowadays. Picking just one ray tracing algorithm for hardware implementation would mean ignoring all others and therefore restricting its field of applications. But even then a whole VLSI ray tracing machine would be too complex.

What could and should be supported with special purpose hardware is that part of the algorithm, that is common in all ray tracers and occupies most of the computing time: the actual tracing of rays, i.e., finding the first intersection point of a ray with all scene objects. VLSI hardware therefore is particularly efficient and desirable for intersection calculations. To be really useful for unburdening general purpose processors, four items of information have to be computed:

- the existence of an intersection point, and if so for non-shadow rays additionally
- the coordinates of the intersection point,
- the normalized object normal vector at the intersection point, and finally
- the (u,v)-parameters of the intersection point to solve the inverse texture mapping problem.

Concerning parallelization, image space subdivision architectures are clearly preferable to object space subdivision architectures because of their greater flexibility and smaller communication overhead. Object space subdivision should additionally be done within each PE, but in software.

Our future work will be especially aimed at image space subdivision architectures with relatively few but powerful PEs. Currently a parallel implementation based on a pool of workstations is under way. Since RAM becomes cheaper and cheaper, we don't see any real problem to store the full model database in each PE memory.

8 Acknowledgements

We would like to thank Prof. W. Straßer for motivating us to do this research and for his numerous helpful comments. Our special thanks go to our colleague Philipp Slussalek for many fruitful discussions.

References

[1] Mark Agate, Richard L. Grimsdale, and Paul F. Lister. The hero algorithm for ray-tracing octrees. In R.L. Grimsdale and W. Straßer, editors, *Advances in Computer Graphics Hardware IV*, pages 61–73. Springer-Verlag, 1991.

[2] Bruno Arnaldi, Thierry Priol, and Kadi Bouatouch. A new space subdivision method for ray tracing csg modelled scenes. *The Visual Computer*, 3:98–108, 1987.

[3] James Arvo and David Kirk. A survey of ray tracing acceleration techniques. In Andrew S. Glassner, editor, *An Introduction to Ray Tracing*, chapter 6, pages 201–262. Academic Press, 1989.

[4] K. Bouatouch and T. Priol. Parallel space tracing: An experience on an ipsc hypercube. In Nadia Magnenat-Thalmann and Daniel Thalmann, editors, *New Trends in Computer Graphics, Proc. of CG International '88*, pages 170–188. Springer Verlag, 1988.

[5] Kadi Bouatouch, Yannick Saouter, and Jean Charles Candela. A vlsi chip for ray tracing bicubic patches. In W. Hansmann, F.R.A. Hopgood, and W. Strasser, editors, *EUROGRAPHICS '89*, pages 107–124. Eurographics Association, Elsevier Science Publishers B.V. (North-Holland), 1989.

[6] E. Caspary and I.D. Scherson. A self-balanced parallel ray-tracing algorithm. In P.M. Dew, R.A. Earnshaw, and T.R. Heywood, editors, *Parallel Processing for Computer Vision and Display*, pages 408–419. Addison-Wesley, 1989.

[7] R. Caubet, Y. Duthen, and V. Gaildrat. Voxar: A tridimensional architecture for fast realistic image synthesis. In N. Magnenat-Thalmann and D. Thalmann, editors, *New Trends in Computer Graphics (Proceedings of CG International '88)*, pages 135–149. Springer Verlag, 1988.

[8] Michael F. Cohen and Donald P. Greenberg. The hemi-cube: A radiosity solution for complex environments. *Computer Graphics*, 19(3):31–40, July 1985.

[9] Robert L. Cook, Thomas Porter, and Loren Carpenter. Distributed ray tracing. *Computer Graphics*, 18(3):137–145, July 1984.

[10] Franklin C. Crow, Gary Demos, Jim Hardy, John McLaughlin, and Karl Sims. 3d image synthesis on the connection machine. In Horst D. Simon, editor, *Proceedings of the Conference on Scientific Applications of the Connection Machine*, pages 260–281. World Scientific, 1988.

[11] M. Dippé and J. Swensen. An adaptive subdivision algorithm and parallel architecture for realistic image synthesis. *Computer Graphics*, pages 149–158, July 1984.

[12] J.L. Ellis, G. Kedem, T.C. Lyerly, D.G. Thielman, R.J. Marisa, J.P. Menon, and H.B. Voelcker. The raycasting engine and ray representations. In *ACM Solid Modeling Symposium*, pages 255–267, 1991.

[13] John Ellis, Gershon Kedem, Richard Marisa, Jai Menon, and Herb Voelcker. Breaking barriers in solid modeling. *Mechanical Engineering*, pages 28–34, February 1991.

[14] Severin Gaudet, Richard Hobson, Pradeep Chilka, and Thomas Calvert. Multiprocessor experiments for high-speed ray tracing. *ACM Transactions on Graphics*, 7(3):151–179, July 1988.

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

[16] S.A. Green and D.J. Paddon. Exploiting coherence for multiprocessor ray tracing. *IEEE Computer Graphics & Applications*, pages 12–26, November 1989.

[17] S.A. Green, D.J. Paddon, and E. Lewis. A parallel algorithm and tree-based computer architecture for ray-traced computer graphics. In P.M. Dew, R.A. Earnshaw, and T.R. Heywood, editors, *Parallel Processing for Computer Vision and Display*, pages 431–442. Addison-Wesley, 1989.

[18] Stuart Green. *Parallel Processing for Computer Graphics.* Research Monographs in Parallel and Distributed Computing. Pitman Publishing, London, 1991.

[19] Stuart A. Green and Derek J. Paddon. A highly flexible multiprocessor solution for ray tracing. *The Visual Computer*, 6:62–73, 1990.

[20] Eric A. Haines and Donald P. Greenberg. the light buffer: A shadow-testing accelerator. *IEEE Computer Graphics & Applications*, pages 6–16, September 1986.

[21] M.-P. Hébert, M.D.J. McNeill, B. Shah, R.L. Grimsdale, and P.F. Lister. Marti – a multiprocessor architecture for ray tracing images. Technical report, University of Sussex, VLSI and Graphics Research Group, August 1990.

[22] Veysi İşler, Cevdet Aykanat, and Bülent Özgüç. Subdivision of 3d space based on the graph partitioning for parallel ray tracing. In *Second Eurographics Workshop on Rendering*, Barcelona, 13-15 May 1991.

[23] G. Kedem and J.L. Ellis. The raycasting machine. *Proceedings of the 1984 International Conference on Computer Design*, pages 533–538, 1984.

[24] G. Kedem and J.L. Ellis. The ray-casting machine. In P.M. Dew, R.A. Earnshaw, and T.R. Heywood, editors, *Parallel Processing for Computer Vision and Display*, pages 378–401. Addison-Wesley, 1989.

[25] G. Kedem and S.W. Hammond. The point classifier: A vlsi processor for displaying complex two dimensional objects. *Proceedings of the Chapel Hill Conference on VLSI*, pages 377–393, 1985.

[26] H. Kobayashi, T. Nakamura, and Y. Shigei. A strategy for mapping parallel ray-tracing into a hypercube multiprocessor system. In Nadia Magnenat-Thalmann and Daniel Thalmann, editors, *New Trends in Computer Graphics, Proc. of CG International '88*, pages 160–169. Springer Verlag, 1988.

[27] Hiroaki Kobayashi, Tadao Nakamura, and Yoshiharu Shigei. Parallel processing of an object space for image synthesis using ray tracing. *The Visual Computer*, 3:13–22, 1987.

[28] Hiroaki Kobayashi, Satoshi Nishimura, Hideyuki Kubota, Tadao Nakamura, and Yoshiharu Shigei. Load balancing strategies for a parallel ray-tracing system based on constant subdivision. *The Visual Computer*, 4:197–209, 1988.

[29] F. Kocsis and J.F. Böhme. Some possible applications of cordic processors in computer graphics. In C.E. Vandoni and D.A. Duce, editors, *EUROGRAPHICS '90*, pages 17–29. Elsevier Science Publishers B.V. (North-Holland), 1990.

[30] K. Nemoto and T. Omachi. An adaptive subdivision by sliding boundary surfaces for fast ray tracing. *Proc. Graphics Interface*, pages 43–48, 1986.

[31] David Edward Orcutt. Implementation of ray tracing on the hypercube. In Geoffrey Fox, editor, *Proc. of the Third Conference on Hypercube Concurrent Computers and Applications*, pages 1207–1210. ACM, 1988.

[32] Thierry Priol and Kadi Bouatouch. Experimenting with a parallel ray-tracing algorithm on a hypercube machine. In D.A. Duce and P. Jancene, editors, *EUROGRAPHICS '88*, pages 243–259. Elsevier Science Publishers B.V. (North-Holland), 1988.

[33] Thierry Priol and Kadi Bouatouch. Static load balancing for a parallel ray tracing on a mimd hypercube. *The Visual Computer*, 5:109–119, 1989.

[34] R.W. Pulleyblank and J. Kapenga. A vlsi chip for ray tracing bicubic patches. In W. Straßer, editor, *Advances in Computer Graphics Hardware I*, pages 125–140. Springer-Verlag, 1987.

[35] Norbert Quien and Werner Müller. Der virtuelle steinmetz. *Spektrum der Wissenschaft*, pages 128–133, December 1991.

[36] Owen F. Ransen. The art of ray tracing. *BYTE*, pages 238–242, February 1990.

[37] John Salmon and Jeff Goldsmith. A hypercube ray-tracer. In Geoffrey Fox, editor, *Proc. of the Third Conference on Hypercube Concurrent Computers and Applications*, pages 1194–1206. ACM, 1988.

[38] Isaac D. Scherson and Elisha Caspary. Multiprocessing for ray tracing: A hierarchical self-balancing approach. *The Visual Computer*, 4:188–196, 1988.

[39] B.O. Schneider. Ray tracing rational b-spline patches in vlsi. In A.A.M. Kuijk and W. Straßer, editors, *Advances in Computer Graphics Hardware II*, pages 47–63. Springer-Verlag, 1988.

[40] L. Richard Speer. An updated cross-indexed guide to the ray-tracing literature. *Computer Graphics*, 26(1):41–72, January 1992.

[41] Kelvin Sung. The area sampling machine. In Alan Chalmers and Derek Paddon, editors, *Third Eurographics Workshop on Rendering*, pages 147–160, Bristol, England, 17-20 May 1992.

[42] Ivan E. Sutherland, Robert F. Sproull, and Robert A. Schumacker. A characterization of ten hidden-surface algorithms. *Computing Surveys*, 6(1):1–55, March 1974.

[43] J. Volder. The cordic trigonometric computing technique. *IRE Transactions on Electronic Computers*, EC-5(9):330–334, 1959.

[44] Douglas Voorhies and David Kirk. Ray-triangle intersection using binary recursive subdivision. In James Arvo, editor, *Graphics Gems II*, chapter V – Ray Tracing, pages 257–263. Academic Press, 1991.

[45] John R. Wallace, Kells A. Elmquist, and Eric A. Haines. A ray tracing algorithm for progressive radiosity. *Computer Graphics*, 23(3):315–324, July 1989.

[46] J. Walther. A unified algorithm for elementary functions. *Proc. of SJCC*, pages 379–385, 1971.

[47] Yulun Wang, Amante Mangaser, Partha Srinivasan, Steve Jordan, and Steven Butner. The 3dp: A processor architecture for three-dimensional applications. *IEEE Computer*, pages 25–36, January 1992.

[48] Hank Weghorst, Gary Hooper, and Donald P. Greenberg. Improved computaional methods for ray tracing. *ACM Transactions on Graphics*, 3(1):52–69, January 1984.

[49] Turner Whitted. An improved illumination model for shaded display. *Communications of the ACM*, 23(6):343–349, June 1980.

[50] Masaharu Yoshida, Tadashi Naruse, and Tokiichiro Takahashi. A dedicated graphics processor sight-2. In R.L. Grimsdale and W. Straßer, editors, *Advances in Computer Graphics Hardware IV*, pages 151–169. Springer-Verlag, 1991.