A Cellular Architecture for Ray Tracing

Abdelghani Atamenia, Michel Meriaux, Eric Lepretre, Samuel Degrande and Bruno Vidal

ABSTRACT We propose in this paper a massively parallel machine dedicated to image synthesis by discrete ray tracing techniques. This machine is a four-stage pipeline, the last stage being a bidimensional cellular array with one cell per pixel. Two main phases describe its behaviour:

- Loading into the cellular array of the objects of the scene to be displayed, after having been transformed into sets of planar polygons, and then into voxels.
- Cellular ray tracing over the fully distributed scene.

The first phase allows us to see this machine as a massively parallel (not realistic) rendering unit: at the end of the loading phase: objects are fully identified pixel per pixel in the cellular array. Then, we have only to display the computed visual features (by means of Gouraud or Phong-like incremental methods during the loading phase).

The second phase increases the image quality by executing the ray tracing algorithm in a very special way, i.e., completely distributed all over the many cells of the array. In that phase, objects are seen as split into voxels into a virtual 3D memory space. The machine is an attempt to bring a dramatic answer to the problem of performance, taking into account not only the computational power required for image synthesis by using a massive parallelism, but also the realization costs by using very regular structures, which make it a VLSI-oriented architecture.

1 Introduction

The aim of this paper is to describe a VLSI-oriented machine dedicated to ray tracing. This machine uses the massive parallelism of a cellular network in order to achieve the amount of computation required by computer image generation.

Cellular networks are usually built for solving specific problems, such as linear algebra [6], [9]. Some attempts have appeared in the fields of image synthesis or image processing ([12], [5], [10]), but none of them has dealt with ray tracing yet. Some experimental work is currently being done on the Cube-3 Machine, at the University of New-York.

Let us quote, however, two models which are not so far from what we propose here:

- CM^2 [4] is a multicomputer machine based on a regular subdivision of the object space in order to limit the number of computations of ray-object intersections to the objects belonging to the volume where the ray actually is. But its extension to a very large number of processors would cause a lot of costly repetitions of (parts of) objects in the processors.

delivered by
EUROGRAPHICS DIGITAL LIBRARY
www.eg.org diglib.eg.org

- VOXAR is a Transputer-based multicomputer machine with a hypertorus structure [3]. Every processor handles a metavoxel space, every metavoxel being composed of voxels. The 3D space is subdivided into voxels crossed by rays, until they meet one voxel occupied by an object. Rays in different metavoxels are processed in a parallel way in different processors. Rays are traced analytically (according to the 3DDA algorithm proposed fro ARTS [8]) through the voxels of the same metavoxel.

2 Proposed Architecture

The architecture we propose here aims at using massive parallelism for ray tracing (several thousands of elementary processors), instead of the low or medium parallelism of the previous attempts.

2.1 Global Description

The architecture is a 4-stage pipeline. The first one, namely the 'geometry' subsystem, is concerned with geometric computations, and outputs objects in screen coordinates. The *geometry subsystem* is supposed to be able to, on the one hand, solve the usual geometric problems, mainly those about the complex scene modelling, and on the other hand, be fast enough to feed the downstream states [2].

The 3 following states, *precomputation, distribution* and *RC*, handle the voxel part of the image synthesis process, which has to discretize the objects of the scene into voxels and to load those voxels into a memory space fully distributed all over the network. This will allow further shading computations.

2.2 Precomputation

Precomputation essentially splits the polygons into trapezes with horizontal parallel sides and computes the true normals to their vertices. The choice of this special type of polygon, which has the drawback of artificially increasing the number of faces, is due to the fairly better performances we get under a pipeline mode while loading the polygons into the network ([11]).

2.3 Object Distribution

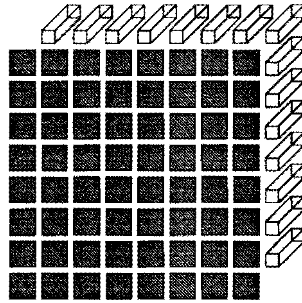
The objects are distributed to the cells of RC by using the left edge of the network. They go across the network like a wavefront, and are at the same time converted into surface elements (voxels), which are stored in the virtual 3D memory space associated with the physical 2D network.

2.4 Cellular Network

RC is the main module of this architecture. It is a 2D matrix with one cell per pixel. RC obviously belongs to the class of image-space partitioning machines.

The aim of RC is:

- To store the objects as surface or volume elements in the virtual 3D memory space.
- To execute the ray tracing algorithm in a cellular way in order to compute the shadows and shadings of all the visible objects.



New problems about programming and control appear when using a cellular network. These are essentially due to the large size of the array (512×512 or 1024×1024) and to the need for an efficient communication tool allowing for easy cooperation between the cells.

The greatest interest of such an architecture, apart from its obvious massive parallelism, lies in the special loading of the scene into the network: it appears to be a kind of splitting of the database, which allows, on the one hand, to suppress the usual bottleneck when accessing at this database, and on the other hand, to drastically reduce the complexity of the ray-object intersection computation, by only making a very easy comparison between one ray coming into a given cell and what this one contains.

For architectural reasons as well as computational complexity ones, the network we use is only 4-connected.

3 Modes of Operation

There are two modes corresponding to the two phases of scene loading and ray tracing.

3.1 Scene Loading

The loading phase makes each cell know what objects, split into voxels, it has to handle, and give it the relevant information about light sources and the viewer's position.

The objects are split into trapezes and normals at the vertices are computed in the precomputation unit. The trapezes are then injected in a distribution pipeline and split into horizontal lines or 'spans'. Lastly, each horizontal span is split into voxels in the multipipelines of the RC network.

Every cell will receive the features of the objects, such as object type, proper color, coefficients for further shading computations, etc., followed by the sequence of polygons belonging to the object.

Xl, Xr	left and right limits of a span
Z, DZx	depth and depth variation along x
N, DN	normal and normal variation along x

3.2 Ray Tracing Execution

It is well known that the ray tracing algorithm can be highly parallelized because the computations are independent in one pixel from another. This remark fully justifies the use of cellular networks for ray tracing.

From a global point of view, a ray tracing algorithm builds and evaluates a forest of intersection trees, each one being associated with a pixel. Cellularly speaking, the forest can be seen as distributed all over the network, with one root per cell. Indeed, the image computation is the evaluation by every cell of the corresponding distributed tree.

The tree construction is started by every cell, which, given the viewer's position, searches its own memory space for an intersection; if it is the case, secondary rays are cast. When a given ray goes out of the memory space of the cell, i.e., x or y changes, a request is transmitted to one of the neighbours in the right direction. This new cell will then have either to route the ray if there is no intersection, or to send again secondary rays, thus building a sub-part of the initial tree. When receiving a ray which intersects some object, a cell has to compute the light intensity which must be sent back to the emitting cell. This computation consists of storing the received ray, computing the new rays, sending them, and waiting for them to come back.

Every cell will thus compute the sub-tree it has generated and will transmit the computed values to the node which has sent it the request (the ray). The execution of the algorithm, started by the cells of the screen (i.e., the cells with z=0), mainly consists of drawing rays in the virtual 3D network and trying to find intersections in the cells crossed by the rays. In order to decide whether there is intersection, one only has to compare the current depth of the ray with the depths of the various voxels composing the objects the cell has to deal with. The ray propagation is exactly an analytic drawing of the half-line in 3D, given its direction ([18], [1]).

However, the rays carrying the intensity information back to the root of a sub-tree already know their destination: so we can use a more straightforward way of routing them through the network. This is also the case for ways toward the light sources.

3.3 Other Problems

RC Size

As the cellular structure is finite, the question arises whether the chosen size (screen size) is correct. Some problems appear, due to the fact that we are only able to discretize that part of the scene which is inside the virtual 3D cube. So we cannot deal with objects or parts of objects outside the cube; moreover rays going outside the cube are lost. A minimal solution to this important problem is to suppose that the whole scene is inside the cube. Another solution would be to add some processors executing a classical ray tracing outside the cube.

Viewer's Position

The viewer's position has only an importance on the primary rays. If the viewer is located at a finite distance, it is necessary to send oblique primary rays. However some aliasing effects may appear when he is close to the screen.

Shadowing

A question arises about the best solution between the two following ones:

- should we cast rays from the voxels toward the sources, every cell being supposed to know their positions?
- should we, in a first step, try to illuminate all the voxels from every source, and keep in every cell information about the sources which can reach it? In the second step, it would thus be unnecessary to send secondary rays toward the sources.

It appears that a good storage/communication load balance leads us to choose the last solution, i.e., pre-casting the source rays.

4 The Basic Cell

A cell is a processor with a special communication tool and some reduced computational power. This is a compromise between a higher integration rate—allowing for a lower cost—a shorter image generation time and a better quality.

4.1 Computational Power

A cell must be able to identify objects and commands it receives. It has to manage coming and going rays. A coming ray may generate 2 rays. The cell has to determine their directions from the coming ray and the local normal, and evaluate the light intensity according to a given shading model, e.g. Whitted's or Phong's. It is clear that the most time- or silicon-consuming parts are the computations of the reflected or transmitted rays, requiring complex operations like square root (for normalization) or division, and the evaluation of the intensity model, requiring exponentiation. it appears that approximations can be used without any important consequence on the resulting image.

We have chosen to use a standard 20 bits ALU with a RAM microprogram in the first implementation of the cell, helped by specialised operators.

4.2 Storage Size

A cell has to store some rays (say M), including the primary one at the beginning of the ray tracing process. But M is rather hard to evaluate; however it is quite important because it could cause the deadlock of the communication system. In case of such a deadlock, we propose to destroy one or more sub-trees and to postpone the corresponding computations.

A cell must store in its memory:

1. viewer's position
2. visible light sources positions and features
3. ambient light intensity
4. each voxel it has to deal with, and its visual features (including true normal).

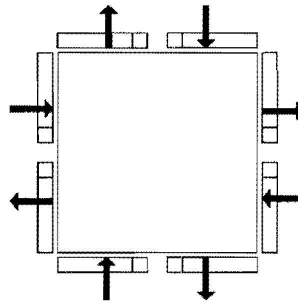
Thus, every cell is able to compute the data necessary for generating new rays or evaluating light intensity.

The most important part is obviously the voxel storage: our study shows that about 40 bytes are necessary per voxel. This will be implemented in the first prototype in standard cell for 16 voxels. In a full custom implementation, we intend to store only the occupied voxels, by using an index table. If the occupancy rate is 10 per cent, the voxel storage would cost about 5 Kbytes if the depth is 1024, which is quite compatible with current full custom technology.

4.3 Communications

Communications are unidirectional and asynchronous. They use buffers shared by neighbouring cells. To every buffer is associated a mutual exclusion flag indicating its current state (full or empty).

A cell has to get a message only when the corresponding output buffer is free in the case of simple routing, or when the message has to be processed locally (i.e., in case of intersection).



5 Simulation

This architecture has been simulated on a 17 Transputer network: the distribution stage and RC are implemented on a 4×4 network, and a first Transputer with more memory acts as precomputation and data construction unit. It appeared quickly that we could not successfully simulate large networks on this array. We are now simulating the machine on a Parstec Multi-Cluster with 32 T800, with 2 Megabytes per Transputer. This allows us to simulate a 128×128 cellular network with 64 depth levels.

6 Conclusion

The fully distributed approach that we propose appears to be realistic: the first simulation results we have got for the ray tracing phase are very positive and go along the same line as the theoretical and practical results we had for the first loading phase. We are now studying more accurately the architecture of the basic cell for a VLSI semi-custom implementation. Such a basic cell should be operating at mid-1990, allowing the building of a 16×16 prototype (but with coordinate range over 0-1023 in x, y, z) by mid-1991. Other theoretical studies remain to be done, e.g., about deadlock cases, correct processing of non trivial light sources, or antialiasing.

References

[1] Amanatides, A., et al: A Fast Voxel Traversal Algorithm for Ray Tracing. *Eurographics '87*, pp. 3-12.

[2] Atamenia, A.: Architectures Cellulaires Pour La Synthèse d'Images. *Thèse de Doctorat*, Lille, June 1989.

[3] Caubet, R., et al.: Le suivi analytique de rayons: un algorithme incremental rapide pour la machine Voxar, *MICAD '89*, Paris, pp. 653-664.

[4] Cleary, J.G., et al.: Multiprocessor Ray Tracing. *Computer Graphics Forum*, vol. 5, no. 1, March 1986, pp. 3-12.

[5] Deering, M., et al.: The Triangle Processor and Normal Vector Shader : A VLSI System for High Performance Graphics. *Computer Graphics*, vol. 22, no. 4, August 88, pp. 21-30.

[6] Drake, B.L., et al.: SLAPP: A Systolic Linear Algebra Parallel Processor. *Computer*, vol. 20, no. 7, July 1987, pp. 45-49.

[7] Eyles, J., et al.: Pixel-Planes 4: A Summary. *Advances in Computer Graphics Hardware II.* EurographicSeminars. Springer, 1988, pp. 1833-207.

[8] Fujimoto, A., et al.: ARTS: Accelerated Ray-Tracing System. *IEEE Computer Graphics and Applications*, April '86.

[9] Kung, S.Y., et al.: Wavefront Array Processors—Concept to Implementation. *Computer* vol. 20, no. 7, July 1987, pp. 18-33.

[10] Lattard, D. and Mazaré, G.: Une nouvelle architecture cellulaire pour la reconstruction parallèle d'images. *Proceedings of PIXIM 88*, Paris, pp. 193-207.

[11] Lepretre, E.: Algorithmes Parallèles et Architectures Cellulaires Pour La Synthèse d'Images. *Thèse de Doctorat*, Lille, June 1989.

[12] Martin, P., et al.: Circuit systolique pour la synthèse d'images. *COGNITIVA 87*, Paris La Villette, 18-22, May 87, pp. 112-117.

[13] Meriaux, M.: Contributions a l'imagerie informatique: Aspects algorithmiques et architecturaux. *Thèse d'état*, Lille, 1984.

[14] Meriaux, M.: A Cellular Architecture for Image Synthesis. *Microprocessing and Microprogramming*, 1984, vol. 13, pp. 179-187.