Interactive Ray Tracing on Reconfigurable SIMD MorphoSys

H. Du, M. Sanchez-Elez[†], N. Tabrizi, N. Bagherzadeh, M. L. Anido[††], M.Fernandez[†]

Electrical and Computer Engineering, University of California, Irvine, CA 92697
[†]Universidad Complutense de Madrid, Spain
[††]Federal University of Rio de Janeiro, NCE, Brazil
{hdu, ntabrizi, nader} @ece.uci.edu, {marcos, mila45}@fis.ucm.es, mlois@nce.ufrj.br

Abstract - MorphoSys is a reconfigurable SIMD architecture. In this paper, a BSP-based ray tracing is gracefully mapped onto MorphoSys. The mapping highly exploits ray-tracing parallelism. A straightforward mechanism is used to handle irregularity among parallel rays in BSP. To support this mechanism, a special data structure is established, in which no intermediate data has to be saved. Moreover, optimizations such as *object reordering* and *merging* are facilitated. Data starvation is avoided by overlapping data transfer with intensive computation so that applications with different complexity can be managed efficiently. Since MorphoSys is small in size and power efficient, we demonstrate that MorphoSys is an economic platform for 3D animation applications on portable devices.

I. Introduction

MorphoSys [1] is a reconfigurable SIMD processor targeted at portable devices, such as Cellular phone and PDAs. It combines coarse grain reconfigurable hardware with one general-purpose processor. Applications with a heterogeneous nature and different sub-tasks, such as MPEG, DVB-T, and CDMA, can be efficiently implemented on it. In this paper a 3D graphics algorithm, Ray tracing, is mapped onto MorphoSys to achieve realistic illumination. We show that SIMD ray-tracing on MorphoSys is more efficient in power consumption and has a lower hardware cost than both multiprocessors and the single CPU approaches.

Ray tracing [2] is a global illumination model. It is well known for its highly computation characteristic due to its recursive behavioral. Recent fast advancement of VLSI technology has helped achieving interactive ray tracing on a multiprocessor [3] and a cluster system [9,10] for large scenes, and on a single PC with SIMD extensions [4] for small scenes. In [3], Parker achieves 15 frames/second for a 512x512 image by running ray tracing on a 60-node (MIPS R12000) SGI origin 2000 system. Each node has a clock faster than 250MHz, 64-bit data paths, floating-point units, 64K L1 cache, 8MB L2 cache, and at least 64MB main memory. Muuss [9,10] worked on parallel and distributed ray tracing for over a decade. By using a cluster of SGI Power Challenge machines [10], a similar performance as Parker's is reached. Their work is different in their task granularity, load balancing and synchronization mechanisms. The disadvantage of their work is that there are extra costs such as high clock frequency, floating-point support, large memory bandwidth, efficient communication and scheduling mechanisms. Usually, the sub-division structure (such as BSP tree-*Binary Space Partitioning* [7,8]) is replicated in each processor during traversal. As will be seen, only one copy is saved in our implementation.

Wald [4] used a single PC (Dual Pentium-III, 800Mhz, 256 MB) to render images of 512x512, and got 3.6 frames/second. 4-way Ray coherence is exploited by using SIMD instructions. The hardware support for floating-point, as well as the advanced branch prediction and speculation mechanisms helps speed up ray tracing. The ray incoherence is handled using a scheme similar to multi-pass scheme [14], which requires saving intermediate data, thus causing some processors to idle.

The migration from fixed-function pipeline to programmable processors also makes ray tracing feasible on graphics hardware [5,6]. Purcell [6] proposes a ray-tracing mapping scheme on a pipelined graphics system with fragment stage programmable. The proposed processor requires floating-point support, and intends to exploit large parallelism. Multi-pass scheme is used to handle ray incoherence. As a result, the utilization of SIMD fragment processors is very low (less than 10%). Carr [5] mapped ray-object intersection onto a programmable shading hardware: Ray Engine. The Ray Engine is organized as a matrix, with vertical lines indexed by triangles, and horizontal lines by rays represented as pixels. Data is represented as 16-bit fixed-point value. Ray cache [5] is used to reorder the distribution of rays into collections of coherent rays. They got 4 frame/sec for 256x256 images but only for static scenes. Although still on its way to achieve interactivity, all these schemes represent one trend towards extending the generality of previous rasterization-oriented graphics processors [10].

In this paper, a novel scheme that maps ray tracing with BSP onto MorphoSys is discussed. Large parallelism is exploited such that 64 rays are traced together. To handle the problem that the traversal paths of these rays are not always the same, a simple and straightforward scheme is implemented. Two optimizations, object test reordering and merging, are utilized. The overhead caused by them is very small after being amortized in highly paralleled ray tracing. Compared with the other schemes, MorphoSys achieves 13.3 frames/second for 256x256 scenes under 300MHz with a smaller chip size, with a simple architecture (SIMD,

fixed-point, shared and small memory), with power efficient performance (low frequency), and highly parallel execution (64-way SIMD). This justifies MorphoSys as an economic platform for 3D game applications targeted at portable devices.

The paper begins with an architecture overview of MorphoSys. Section 3 describes ray tracing pipeline process, followed by the SIMD BSP traversal in Section 4. Some design issues, such as data structure, emulated local stack, and memory utilization are detailed in Sections 5, 6, 7. Section 8 describes the global centralized control. We give the implementation and results in Section 9, followed by the conclusion in Section 10.

II. MorphoSys Architecture

MorphoSys [1] is a reconfigurable SIMD architecture targeted for portable devices. It has been designed and intended for a better trade-off between generality and efficiency in today's general-purpose processors and ASICs, respectively. It combines an array of 64 Reconfigurable Cells (RCs) and a central RISC processor (TinyRISC) so that applications with a mix of sequential tasks and coarse-grain parallelism, requiring computation intensive work and high throughput can be efficiently implemented on it. MorphoSys is more power efficient than general-purpose processors because it was designed with small size and low frequency yet to achieve higher performance through highly paralleled execution and minimum overhead in data and instruction transfers and reconfiguration time. The detailed design and implementation information of the MorphoSys chip are described in [16]. The general architecture is illustrated in Fig. 1.

The kernel of MorphoSys is an array of 8 by 8 RCs. The RC array is responsible for the parallel part of an application,

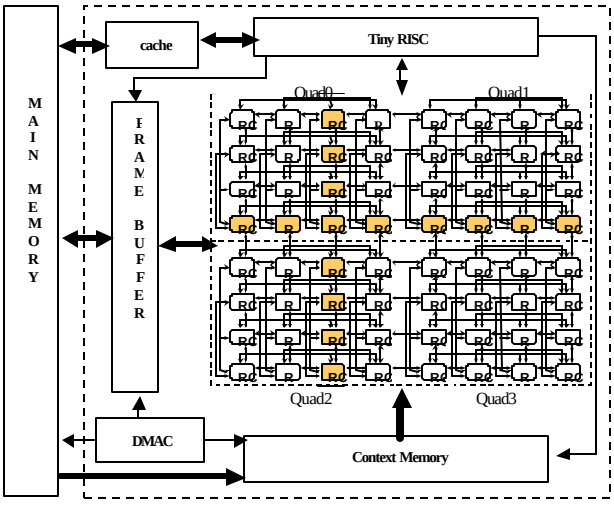

Fig 1. MorphoSys SIMD architecture

RC is a 16-bit fixed-point processor, mainly consisting of an ALU, a 16-bit multiplier, a shifter, a register file with 16 and is organized as SIMD style. 64 different sets of data are processed in parallel when one instruction is executed. Each registers, and a 1KB internal RAM for storing intermediate values and local data.

MorphoSys has very powerful interconnection between RCs to support communication between parallel sub-tasks. Each RC can communicate directly with its upper, below, left and right neighbors peer to peer. One horizontal and one vertical short lane extend one RC's connection to the other 6 RCs in row wise and column wise in the same quadrant (4 by 4 RCs). And one horizontal and one vertical global express lane connect one RC to all the other 14 RCs along its row and column in the 8 by 8 RC array.

The reconfiguration is done through loading and executing different instruction streams, called "contexts", in 64 RCs. A stream of contexts accomplishes one task and is stored in Context Memory. Context Memory consists of two banks, which can be flexibly used to support pre-fetch: while the context stream in one bank flows through RCs, the other bank can be loaded with a new context stream through DMA. Thus the task execution and loading are pipelined and context switch overhead is reduced.

A specialized data cache memory, Frame Buffer (FB), lies between the external memory and the RC array. It broadcasts global and static data to 64 RCs in one clock cycle. The FB is organized as two sets, each with double banks. Two sets can supply up to two data in one clock cycle. While the contexts are executed over one bank, the DMA controller transfers new data to the other bank.

A centralized RISC processor, called TinyRISC, is responsible for controlling RC array execution and DMA transfers. It also takes sequential portion of one application into its execution process.

The first version of MorphoSys, called *M1*, was designed and fabricated in 1999 using a 0.35um CMOS technology [16].

III. Ray Tracing Pipeline

Ray tracing [2] works by simulating how photons travel in real world. One *eye ray* is shot from the viewpoint (eye) backward through image plane into the scene. The objects that might intersect with the ray are tested. The closest intersection point is selected to spawn several types of rays. *Shadow rays* are generated by shooting rays from the point to all the light sources. When the point is in shadow relative to all of them, only the ambient portion of the color is counted. The *reflection ray* is also generated if the surface of the object is reflective (and *refraction ray* as well if the object is transparent). This reflection ray traverses some other objects, and more shadow and reflection rays may be spawned. Thus the ray tracing works in a recursive way. This recursive process terminates when the ray does not intersect with any object, and only background color is returned. This process is illustrated in Fig. 2.

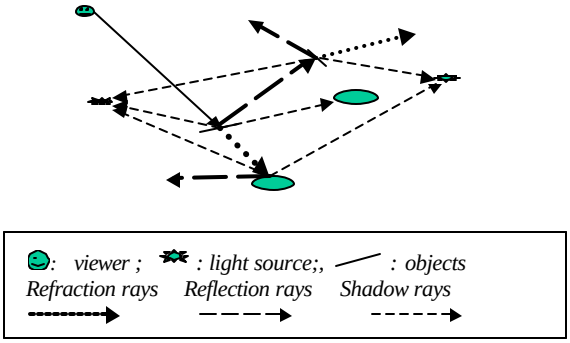

Fig. 2. Illustration of ray tracing

Ray tracing is basically a pipelined process. The algorithm is separated into four steps. First, rays are generated. Then each ray traverses BSP tree to search for the object with the closest intersection point. This is an iterative process in term of programming model, where BSP tree nodes are checked in depth-first-order [11]. Once a leaf is reached, the objects in this leaf are scanned. If no intersection in this leaf or the intersection point is not in the boundary of this leaf, BSP tree is traversed again. Finally, when the closest point is found, the shading is applied, which recursively generates more rays. This pipeline process is illustrated in Fig. 3.

IV. SIMD BSP Traversal

The ray object intersection occupies more than 95% of ray tracing time. Some techniques have been developed to accelerate it. BSP [7,8] is one of them. It tries to prevent those objects lying far away from being tested. BSP recursively partitions a 3D cube into 2 sub-cubes, defined as left and right children. BSP works like a Binary-Search Tree [15]. When one ray intersects with one cube, it tests whether it intersects with only left, right, or both children. Then the ray continues to test these sub-cubes recursively. The traversal algorithm stops when the intersection is found or when the tree is fully traversed.

The efficiency may be sacrificed in SIMD ray tracing. Each BSP-tree ray traversal involves many conditional branches, such as *if-then-else* structure. *Program autonomous* [12] is thus introduced to facilitate them in a SIMD. We implemented *pseudo branch* and applied *guarded execution* to support conditional branch execution [13].

During parallel BSP traversal different rays may traverse along the same BSP tree path ("*ray coherence* [15]") or along different *BSP* tree paths ("*ray incoherence*"). The coherence case is easily handled in a SIMD architecture.

However, the incoherence case demands a high memory bandwidth because different object data and also different object contexts are required concurrently. It can be further observed that two types of ray incoherence exist:

? First type, the ray incoherence occurs in an internal tree node, and not all rays intersect with the same child of the current node. In this case, BSP traversals of all rays are

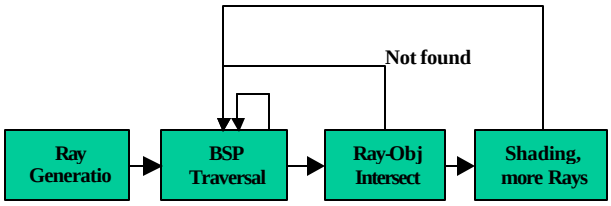

Fig. 3. Ray tracing pipeline.

stopped and all objects under this node are tested.

? Second type, the ray incoherence occurs in a leaf, where not all rays find the intersection points in that leaf. In this case the RCs that find the intersections are programmed to enter the sleep mode (no more computation), while the others continue the BSP traversal. Power is saved in such a way.

Sometimes, some rays may terminate traversal earlier and may start ray-object intersection while others are still traversing. In this case, different context streams are required for different rays. In [6], the extended *multi-pass* [14] scheme is used to address this problem. In their scheme, if any ray takes a different path from the others, all the possible paths are traversed in turn. The disadvantage of this scheme is that the intermediate node information has to be saved for future use.

In our design, SIMD BSP traversal is implemented in a simple and straightforward way. Whenever there is a ray taking a different path than the others ("ray incoherence"), the traversal is stopped and all the objects descended from the current node are tested. Thus, all the RCs process the same data and over the same contexts at the same time. This process is illustrated in Fig. 4.

The advantages of this scheme is that no intermediate node information needs to be saved, thus simplifying control and reducing memory accesses since fewer address pointers are followed.

In this scheme, each ray may test more objects than necessary. Thus some overhead is introduced. However, simulations show that the amortized overhead is very small when 64 rays are processed in parallel, although each ray itself may take more time to finish than those with 4, 8, 16, or

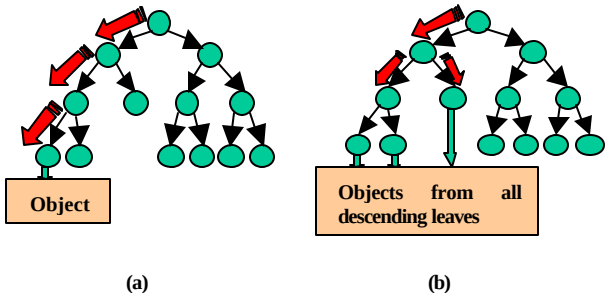

Fig. 4. Novel BSP incoherence handling.(a) All the 64 rays have ray-coherence. Objects in one leaf are tested. (b) Not all the 64 rays have ray-coherence. All the leaf objects under the node are tested.

32 parallel rays.To further remove this overhead, we applied *object test reordering* and *object merging*. We will discuss these optimizations in more details in the next section.

V. Data Structure and Optimizations

We have developed a special data structure to support BSP mapping. This structure reduces data and contexts reload, as we describe in this section.

Our data structure is created so that for each node the objects under it are known immediately. The data structure is illustrated in Fig. 5. In this figure, the item bit-length (e.g., 16 bits) is specified in parenthesis after each item. *Child Address* stands for the address of each child node in FB. The objects descending from this node are grouped by object type. *Sphere*, *cylinder*, *rectangle* and *other objects* stand for vectors where each bit indicates whether or not the object of this type specified by its positions belongs to this node. Fig. 5 gives one example for sphere case. We order all spheres as *sphere 0*, *sphere 1*, *sphere 2*, and so on. If any of these spheres belongs to the node, the bit indexed by its order is set to '1' in the sphere vector. In this example, sphere 1, 3, 4 belong to the current node. TinyRISC tests these vectors to know which geometrical data to operate on when incoherence happens. All the identified objects under this node are tested. In all the cases, the objects are tested only once. This is called "object merging", or *mailbox*" in [6]. If one object is tested, it is labeled as tested. The calculated value is retained in the FB. If one object to be tested has been labeled, this testing is cancelled, and the value is fetched from FB.

This data structure automatically does "object test reordering", which tests the same objects without context reload. For example, we check all the spheres before we continue with other object types. Moreover this data structure allows the object addresses in the FB to be easily calculated, as is shown in the algorithm in Fig. 6. It also facilities pre-fetching since objects are known in advance.

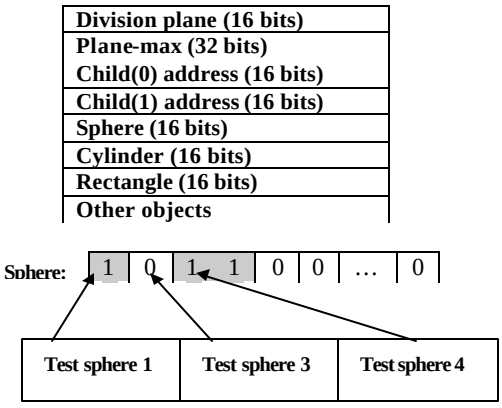

Fig. 5. Data structure for our BSP algorithm.

```
r12:= BSP sphere data;
r13:= First sphere address;
r14:= Sphere size;
r0:= 0;
while r0 = number of sphere in the image do {
  if r12 < 0 then compute_sphere(r13);
          r13 := r14 + r13;
          r12 << 1;
  r0 = r0 +1;
}
```

Fig. 6. Calculation of FB address from data structure in each node.

VI. Local Stack for Parallel Shading

After all rays find the closest intersection points and intersected object, the ray-tracing algorithm calculates the color (using *Phong-Shading* model [2,3]). The shadow and reflection rays are generated and again traverse the BSP tree, as described in Section 3.

However, during this process, the intersection points and intersected objects can be different for different rays. This data cannot be saved to and later fetched from the FB. The reason is that they would have to be fetched one by one for different RCs due to limited bandwidth, which means all but one RC are idle and cycles are wasted.

Local RC RAM is used to emulate a stack to address this problem. Besides stack, each RC has one vector to store the current intersection point and intersected object. During ray-object intersection process, when one object is found to be closer to the eye than the one already in the vector, the corresponding data replaces the data in the vector. Otherwise, the vector is kept unchanged. When new recursion starts, the vector is pushed into the stack. When recursion returns, the data is popped from the stack into the vector. In this way, the object data and intersection point required for shading are always available for different rays. The overhead due to these data saving and restoring is very small compared with the whole shading process. This process is illustrated in Fig. 7.

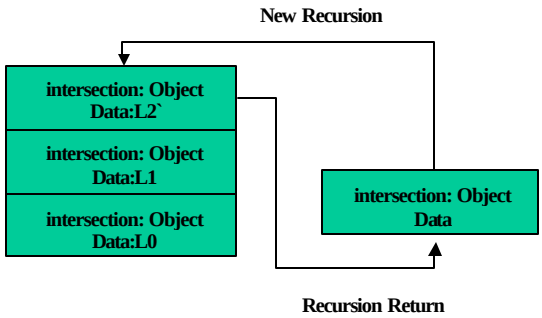

Fig. 7. Local stack and vector to keep track of the current closest intersection point and the intersect object.

SIMD processing of 64 RCs demands high memory bandwidth. For example, up to 64 different data may be concurrently required in MorphoSys. Fortunately, this is not a problem in our design. Our implementation guarantees that all RCs always require the same global data for BSP traversal and intersection, such as BSP tree structure, static object data, etc. Thus, only one copy of this data is needed. The implementation of object merging is also simplified. Since all rays are always required to test the same set of objects, whether they are coherent or not, one copy of the object test history is kept for all RCs.

One parameter that affects all ray tracing mapping schemes is the memory size. For a very large scene, the size of BSP tree structure or object data can be so large that not all of them can fit in the main memory or cache. The result is that processors have to wait for the data to be fetched from the external memory. Also, the context stream may be very large so that not all of them are available for execution. However, this can be easily solved by our double-bank organizations of the FB and Context Memory, as was described in Section 2.

VIII. Centralized TinyRISC Control

Using our ray tracing mapping scheme, all the rays traverse along the same BSP tree path and find the intersection with the same set of objects. Thus one central controller is enough to broadcast data and contexts to all 64 RCs. TinyRISC in MorphoSys plays this role. The same context is loaded and broadcast to all 64 RCs. Traversal starts from the BSP tree root downward toward the leaves. At each internal node, the status of all rays is sent to TinyRISC. TinyRISC loads the corresponding tree node and also contexts and broadcast them to all 64 RCs. This interaction continues until leaves are reached, where the object data are broadcast. In case that the status information indicates incoherence, TinyRISC loads all the objects data descended from the current node and broadcast them for calculation.

VIIII. Simulation and Experimental Results

MorphoSys is designed to be running at 300MHz. It has 512x16 internal RC RAM, 4x16Kx16 FB, 8x1Kx32 Context Memory, and 16 internal registers in each RC. The chip size is expected to be less than $30mm^2$ using 0.13um CMOS technology. Thus MorphoSys is more power efficient than general-purpose processors.

Our targeted applications are those running on portable devices with small images and small number of objects. In our experiments, applications are 256x256 in size. The BSP tree is constructed with maximum depth of 15, maximum 5 objects in each leaf. The recursive level is 2.

Different from the other ray tracing mapping schemes [4,5,6], whose primitives are only triangles, the primitives mapped in our implementation can be any type, such as sphere, cylinder, box, rectangle, triangle, etc. The advantages

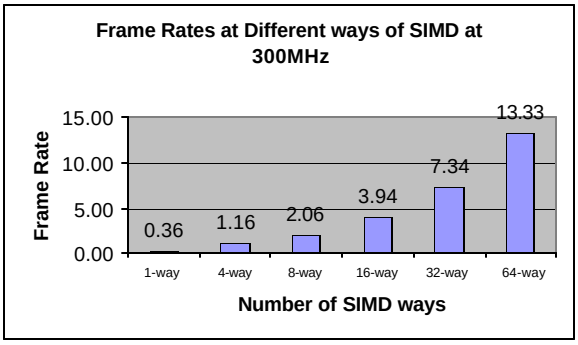

Fig.8. Frame rates under different SIMD ways

are: (1) the object data size is small compared with pure triangle scheme. Much more data is needed to represent scenes using only triangles, and very small triangles are used to get good images. (2) No preprocessing time is needed to transform original models into triangles. However, using pure triangles can simplify BSP traversal since some conditional branches are removed, and also only one ray-object intersection code is needed. Thus, we decided to use a mix of different objects, to attain a better trade-off between algorithm complexity and memory performance.

The algorithm was translated into MorphoSys Assembly and then into machine code. The Simulation is run on MorphoSys processor simulator "Mulate" [16]. We did simulation to get the frame rates for 4, 8, 16, 32, and 64 parallel rays, so that a view of ray tracing performance under different levels of parallel processing is seen. The result is shown in Fig. 8.

This figure shows that the frame rates increase with more paralleled rays. However, the performance of 64-way ray tracing is not twice that of 32-way, but less than that. The reason is that overhead increases as well, although the amortized overhead is actually decreased. This can be formulated and explained as follows. Suppose the processing time for one ray without overhead is T, total number of rays is N, and number of rays processed in parallel is n, and the overhead in processing one ray is OV, the total processing time is:

$$N*(T+OV)/n$$

Thus the frame rate is $C*n*/(N(T+ OV))$, where C is a constant. If overhead is constant, the frame rate is $O(n)$. However, OV increases as n increases. Thus frame rate is sub-linear with n, the number of parallel rays.

X. Conclusion

This paper gives a complete view of how to utilize the simple SIMD MorphoSys to achieve real-time ray tracing with an efficient use of hardware resources. BSP traversal is mapped in a straightforward way such that no complicated decision and intermediate data saving are necessary. Optimizations, such as object reordering and merging, help

simplify the SIMD mapping. The resulted overhead is amortized and is very small when large number of rays are traced in parallel, thus the performance can be very good. Memory is also flexibly utilized. Due to its small size and potential power efficiency, MorphoSys can be used as an economic platform for 3D games on portable devices. Right now, we are further optimizing the architecture so that better hardware supports, such as 32-bit data paths, more registers, etc, are included. Based on what we have achieved (more than 13 frames/second in 300MHz), it is believed that obtaining real-time ray tracing on portable devices is practical soon.

Acknowledgements

We would like to thank everyone in MorphoSys group, University of California, Irvine, for their suggestions and help in our architectural and hardware modifications. We thank Maria-Cruz Villa-Uriol and Miguel Sainz in Image-Based-Modeling-Rendering Lab for giving sincere help in our geometry modeling and estimation. This work was sponsored by DARPA (DoD) under contract F-33615-97-C-1126 and the National Science Foundation (NSF) under grant CCR-0083080.

References

[1] G. Lu, H. Singh, M.H.Lee, N. Bagherzadeh, F. Kurdahi, and E.M.C. Filho. *"The MorphoSys Parallel Reconfigurable System,"* Proc. of Euro-Par 1999.

[2] A. Glassner. "*An Introduction to Ray Tracing*," Academic Press, 1989.

[3] S. Parker, W. Martin, P.P.J.Sloan, P.Shirley, B.Smits, and C.Hansen. "*Interactive Ray Tracing*," In Proc. ACM Symposium on Interactive 3D Graphics, ACM 1999.

[4] I.Wald, P.Slusallek, C. Benthin, and M.Wagner. "*Interactive Rendering with Coherent Ray Tracing*," Computer Graphics Forum 20, 2001 153-164.

[5] N.A.Carr, J.D.Hall, J.C.Hart, "*The Ray Engine*," Tech. Rep. UIUCDCS-R-2002-2269, Department of Computer Science, University of Illinios.

[6] T.J.Purcell, I.Buck, W.R.Mark, and P.Hanrahan, "*Ray Tracing on Programmable Graphics Hardware*," SIGGraphics 2002 Proc.", 2002.

[7] K. Sung and P.Shirley. "*Ray Tracing with the BSP-Tree*," Graphics Gem III, 271-274. Academic Press 1992.

[8] A. Watt. "*3D Computer Graphics. 2nd Edition*," Addison-Wesley Press.

[9] M.J.Muuss. "*Rt and Remrt-Shared Memory Parallel and Network Distributed Ray-Tracing Programs*," In USENIX: Proc. of the Fourth Computer Graphics Workshop, October 1987.

[10] M.J.Muuss. "*Toward Real-Time Ray-Tracing of Combinational Solid Geometric Models*," In Proc. of BRL-CAD Symposium, June 1995.

[11] T.H.Cormen, C.E.Leiserson, R.L.Rivest, C.Stein, "*Introduction to Algorithms, 2nd Edition*," McGraw-Hill and MIT Press, 2001.

[12] P.J.Narayanan. "*Processor Autonomy on SIMD Architectures*," ICS-7, 127-136. Tokyo 1993.

[13] M.L. Anido, A. Paar, and N. Bagherzadeh, "*Improving the Operation Autonomy of SIMD Processing Elements by Using Guarded Instructions and Pseudo Branches*," DSD'2002, Proc. EUROMICRO Symposium on Digital System Design, North Holland, Dortumond, Germany, September, 2002.

[14] M.S. Peercy, M. Olano, J.Airey, and P.J. Ungar. "*Interactive Multi-Pass Programmable Shading*," ACM SIGGRAPH, New Orleans, USA July 2000.

[15] L.R. Speer, T.D. DeRose, and B.A.Barsky. "*A Theoretical and Empirical Analysis of Coherent Ray Tracing*," Computer-Generated Images (Proc. of Graphics Interface'85), May 1985, 11-25.

[16] H. Singh, H., M.H.Lee, G.Lu, F.J.Kurdahi, N. Bagherzadeh, and E.M.C. Filho, "*MorphoSys: An Integrated Reconfigurable System for Data-Parallel and Computation-Intensive Applications*," IEEE Transactions on Computers 49 (5): 465-481 (2000).