Graphics Hardware (2002), pp. 1–9
Thomas Ertl, Wolfgang Heidrich, and Michael Doggett (Editors)

Dependency Graph Scheduling in a Volumetric Ray Tracing Architecture

S. Frank and A. Kaufman

Center for Visual Computing (CVC)
and Department of Computer Science
State University of New York at Stony Brook, USA
Stony Brook, NY 11794-4400

Abstract

We propose a volumetric ray tracing PCI board which uses FPGA components and on chip memory. In a multi-board system a super volume (i.e., one that is larger than on-board memory) can be either distributed or shared. In a single board system it must be fetched from main memory as needed. In any case the volume is sub-divided into cubic cells and process scheduling has a major impact on the rendering time. There is not generally a scheduling order which would allow each sub-volume to be read from memory only once. We introduce instead a new, compact representation of the cell ray-spawning dependencies of all rays, called Cell Tree. We use this Cell Tree to determine a good processing schedule for the next frame based on the ray dependencies from the previous frame. Experimental results show an average miss reduction of 30%. *The main contribution of this paper is the generation of a Cell Tree for ray tracing which collects coherent bundles of rays with very little overhead. This is used to decrease overall memory access in sequences where there is good inter-frame coherence.*

Categories and Subject Descriptors (according to ACM CCS): I.3.1 [Computer Graphics]: Hardware Architecture–Graphics Processors; I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling–Geometric algorithms, languages, and systems; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism–Raytracing

Keywords: Inter-frame coherence, Load Balancing, Super volume, Volumetric Ray Tracing

1. Introduction

Although commercial volume rendering accelerators are now available [11], ray tracing systems are still under investigation. Several architectures implement parallel ray casting [2, 8, 12]. In contrast, our system performs volumetric ray tracing. Fine grain resolution and additional properties such as density or temperature increase the volume data storage requirements. A super volume is one which does not all fit in memory. In a multiprocessor system, the volume is either distributed between processors with the ray data being transferred among machines, or it is accessed from shared memory as needed. We can not use data replication [4] on super volumes.

Volumetric ray tracing allows high quality rendering of complex scenes that include translucent and amorphous objects with perspective projections and realistic lighting. A major obstacle for volumetric ray tracing is the memory required to store the volume and accessing that memory in an efficient manner. Pharr et al. [13] discuss rendering complex scenes with memory-coherent ray tracing. The scene is divided into a scheduling grid. The scheduler selects the next group to trace based on which part of the scene is already in memory and the degree to which processing the rays will advance the computation. Our system uses a similar volume subdivision and ray queue scheme, but scheduling is improved in our system by using ray dependency information.

A study of memory configurations is required for a good architecture design. Doggett et al. [3] compare the cost of three memory schemes for the VizardII design. They propose a buffering scheme that prevents a second memory stall when a ray crosses into a new sub-cube. In order to use off-the-

shelf memory boards they propose using four Dual Inline Memory Modules (DIMMs) and replicating data. Two consecutive voxels are stored instead of one. The results show that the cheaper four DIMMs solution has a slightly slower rendering rate which is improved by the buffered memory scheme. L1 and L2 Cache are used to improve both texture [5], and volumetric ray casting [10], memory performance. Our scheduling scheme can be applied at any level of the memory hierarchy.

The Kilauea system is a two-tier parallel processing system utilizing a cluster of a multi-CPU machines [9]. It uses a distributed memory scheme with parallel execution of rays. Unlike our system, there is apparently no attempt to reorder the processing schedule to reduce computation. Reinhard, et.al. present a hybrid system which distributes the geometry among the processors, while reflected and refracted rays are processed by available processors on demand [15]. Data distribution is made by gathering coherent primary and shadow rays in a pyramid. A pyramid is constructed around a bundle of rays and intersected with an octree subdivision of the volume. Each processor stores the entire octree, but only the cells assigned according to the data distribution will actually contain geometry data. The pyramid clipping overhead took an average of 20%. Our Cell Tree is an all-inclusive, concise ray dependency description that is constructed as rays are generated with very little overhead.

We present an architecture based on GI-Cube [1] with a reduced memory cost that can efficiently render and illuminate super volumes. Space leaping and early ray termination are implemented on cubic cells. This idea was also proposed for a ray casting system by Ray and Silver [14]. The GI-Cube design supports Phong shading and local illumination, as well as global illumination including shadow casting, reflections, glossy scattering and radiosity. While GI-Cube uses four RDRAMs and four processors on the PCI board, our system uses a single RDRAM and single processor. We use dependency graph based scheduling to reduce the number of times sub-volumes must be loaded into the system. It can be extended to any ray tracing system, even one that can render overlapping volumes, polygons, point clouds and implicit surfaces [6].

We introduce an algorithm that produces a description of all ray dependencies called Cell Tree. It gathers clusters of eye, shadow, reflected and refracted rays, and is several orders of magnitude smaller than the number of dependencies represented. We use it to determine a processing schedule for the next frame. We compare our algorithm with the Max Work algorithm, which always loads the cell with the most pending rays. For the data-sets studied we obtained an average decrease of 30% in the number of memory loads. We also provide a means for studying how much the memory schedule could be improved using ray coherency. Our data structure opens the door for the study of memory configurations in ray tracing systems. It can be applied at any level of the memory hierarchy and implemented in hardware or software with very low time and memory overhead. In Section 2, we discuss the target architecture. The Cell Tree creation and the scheduling algorithm that utilizes it are introduced in Section 3, with results shown in Section 4.

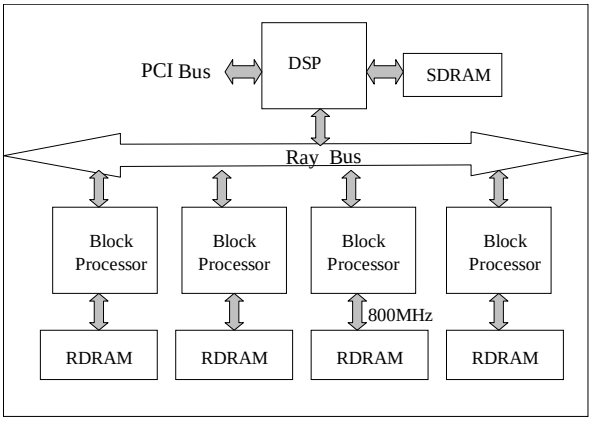

Figure 1: *Block diagram of the GI-Cube system.*

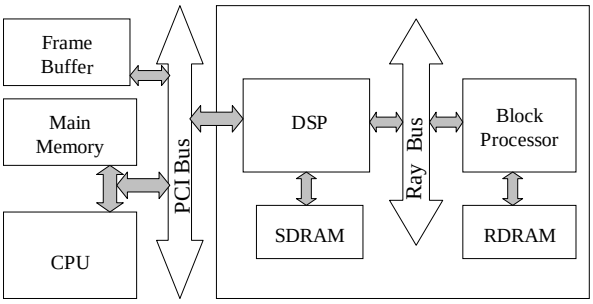

Figure 2: *Block diagram of our system.*

2. Architecture Description

Our research is a component of the Cube-5 architecture design. Cube-5 is a PCI board based volume ray tracing system designed for high quality processing [7] and rendering of very large volumes from various data source types with realistic lighting. Our current research is focused on improving memory performance for super volumes using both ray coherence and inter-frame coherence. We utilize the digital signal processor (DSP) to create a schedule for the next frame based on the cell dependencies of the current frame.

Our system is derived from the GI-Cube architecture. The GI-Cube design has three major components: the DSP, the processors and the memory (see Figure 1). The DSP is directly connected to the frame buffer and has its own SDRAM. It loads the data set, generates lighting and viewing rays, controls processor I/O and sends the result over

the PCI interface. The processors maintain and sort a group of fixed size hardware queues of rays. Each queue is implemented as a pipelined insertion sorter on a separate eDRAM and can hold up to 256 rays. The active queue is processed until it is empty. The queue with the most rays is selected when a processor becomes available. Interactive frame rates are achieved for volumes that fit into on-board memory. Rambus DRAM is used to store the volume locally. Due to its high bandwidth, an 800MB/s RDRAM is able to supply the volume data to the processors so that the only memory stalls occur during the final 0.4% of processing, when ray queues tend to be shorter than the latency of the pipeline. Newly retrieved voxels are written to both the cache and the resampling unit.

We propose a scaled down, less expensive version using only one RDRAM and one processor (see Figure 2). We utilize the idle cycles in the DSP to determine a processing schedule for frame $i+1$ based on ray coherency information of frame i. As the DSP generates initial rays, the Cell Tree is initialized with one node for each cell that contains any eye rays. The DSP56311 can perform 255 MIPS, and has three megabits of on-chip static RAM [16]. The DSP handles data set loading, generates lighting and viewing rays, controls processor I/O, and sends the final image to the frame buffer. It has a dedicated SDRAM to hold the ray queues of cells not currently in memory and the ray dependency data used for scheduling. The queues for cells that are currently in on-board RDRAM memory are kept in processor eDRAM, as in GI-Cube. When the volume currently in RDRAM memory no longer has any pending rays, the DSP fetches a different sub-volume from disk memory. For the first frame, the cell with the most pending rays is selected. This is the algorithm used by GI-Cube as well as most designs based on ray queues. For subsequent frames the cell is selected according to a schedule, which is based on ray dependencies of the last frame.

This schedule is based on ray dependency information which is gathered as rays are placed in queues. Each time a ray exits the cell currently in memory, a ray packet is sent over the ray bus to the DSP. The bus frequency of 100MHz allows one ray packet to be sent per processing cycle. The processor can generate either an exit ray or a neighbor ray in a cycle, but not both. This means that in any cycle at most one ray is sent to the DSP. The DSP consolidates the ray dependencies into a Cell Tree during idle duty cycles. The DSP computes the schedule for loading the sub-volumes for the next frame. This is done between frames. Alternatively, if a slower DSP is used, the schedule can be created during idle cycles during the next frame. A new schedule can be calculated for each frame in order to maximize the exploitation of inter-frame coherence. The algorithm used by the DSP is further explained in the next section.

3. Scheduling Algorithms

If the whole volume doesn't fit into memory, it must be divided into sub-volumes, called cells. Here the term *memory* refers to the memory level being scheduled because the algorithm presented can be used at any level of the memory hierarchy. We also make the assumption that each cell is equal to one memory unit, such as cache size. A queue of pending rays is maintained for each cell. Eye rays which intersect the volume are placed in the queue of the intersected volume cell. When a cell is in memory, all of the rays on the queue are processed. This includes rays which intersect the volume in the same cell that they were spawned from. When a ray exits the cell, it can either exit the volume or enter an adjoining cell. Exit rays are added to the image buffer.

Rays entering an adjacent cell are added to the queue of that cell. If a ray is reflected back into the same cell after it has exited, that cell must be read from memory an extra time. Since memory fetches can be orders of magnitude higher than processing speed, we would like to find a cell processing schedule based on the behavior of rays in one frame that can be used in the next frame to reduce the total number of sub-volume fetches. If the time required to fetch the cell is significant with respect to the processing time, the order in which the cells are processed greatly affects the frame rate.

Ray tracing a sub-divided volume results in a much more complex dependency graph than ray casting. As a volume is ray-traced, one or more rays are cast from each pixel into the volume. Each ray accumulates opacity until either it reaches a predefined maximum opacity, it has reached a predefined maximum number of bounces, or it exits the volume. As a ray intersects the volume it can spawn additional reflection and/or refraction rays. These are dependent on the parent ray, as well as on all of its predecessor rays. The problem is that with ray tracing, a cell may be revisited by the same ray or its descendants any number of times.

The problem can be stated formally as follows: We define a job to be a group of rays requiring processing. For every job j there is an associated cell, c_j. If cell c_j is not available in local memory at the start of job j, then there is a cost of p for fetching it from a higher memory level such as main memory. Given a partial order of jobs, find the schedule that obeys the partial order and has the lowest overall cost.

We use two algorithms for scheduling of cells to be processed. The first one is the Max Work algorithm and the second one is the Cell Tree algorithm. In our experiments, using the Cell Tree algorithm reduced the number of memory fetches compared with using the Max Work algorithm.

3.1. Max Work Algorithm

The Max Work algorithm solves the *on-line* version of the scheduling problem described above. This means it solves the problem without knowing a priori what the ray depen-

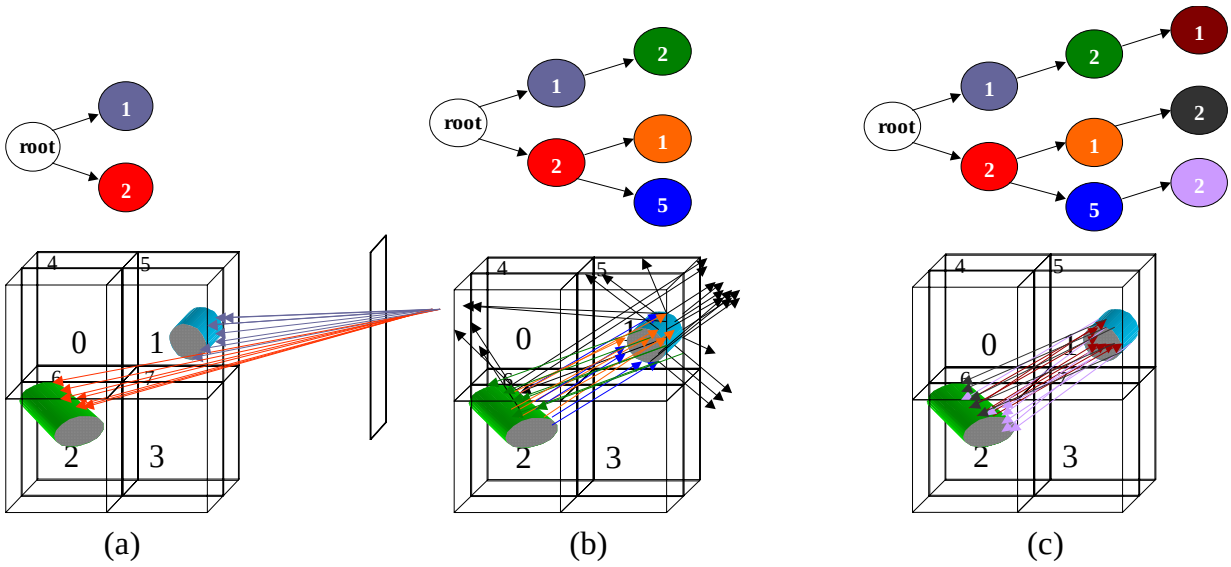

Figure 3: *Cell Tree Construction: (a) Eye rays intersect volume in cells* 1 *and* 2*; (b) Reflection, refraction and shadow rays intersect volume in cell* 2 *from cell* 1 *and cells* 1 *and* 5 *from cell* 2*; (c) Secondary reflection rays enter cell* 1 *from cell* 2*, cell* 2 *from cell* 1 *and cell* 2 *from cell* 5*.*

dencies are. The Cell Tree algorithm solves the *off-line* version, where the dependencies are known. For the first frame, there is no dependency information at all, so the Max Work algorithm is used. The queue with the most rays is selected when a processor has completed all rays in the queue of its current cell. As rays are generated, a Cell Tree is constructed. This is used to determine a better cache schedule for the next frame.

Any time the viewing parameters or the scene itself changes, the dependencies actually become unknown. However, if there is sufficient inter-frame coherence, the dependencies remain the nearly the same from one frame to the next and the Cell Tree algorithm schedule improves performance. If all processing cannot be completed by following the schedule predicted by the Cell Tree algorithm, insufficient coherence is indicated and the Max Work algorithm is followed for the rest of the frame. Since the Cell Tree algorithm takes very little time and memory as discussed in Section 3.4, it can easily be run for each frame, which increases the chances of having sufficient inter-frame coherence. Depending on the implementation platform this can be done either between frames at very high processing speeds, or during frame *i* to be used in frame *i+1*.

3.2. Cell Tree Construction

The dependency graph information gathered in one frame allows the Cell Tree algorithm to generate the cell processing schedule for the next frame. Each ray has an ordered sequence of cells which it depends on. We call this its cell ray-spawning dependencies, or ray dependencies for short. If ray *i* is spawned from ray *j*, *i* is defined to be a child of ray *j*. The ray dependencies of ray *i* are the same as ray *j* with the addition of any new cell ray *i* enters. For each primary ray, there is a tree that describes all of the dependencies spawned from that ray. For a 512^2 image, there are over $25,000$ ray trees, each with its own set of dependencies. We gather these ray dependencies into coherent groups to create a single, consolidated, Cell Tree.

Cell Tree construction proceeds by tagging each ray with a Cell Tree node ID, and adding new dependencies to the Cell Tree as rays are spawned. This process is illustrated in Figure 3. In addition to the Cell Tree, a separate list of Cell Tree nodes is maintained for each cell.

In Figure 3c, there are two branches that contain the dependency from cell 2 to cell 1. In Figure 3a the rays in cell 1 are initial eye rays, and in Figure 3b the new rays in cell 1 are reflection and shadow rays that are generated during cell 2 processing. In order to have a complete dependency graph these cases must be distinguished. This is accomplished by tagging each ray with its corresponding place, or node, in the Cell Tree. As long as a spawned ray remains in the same cell, it is tagged with the Cell Tree node of its parent ray. When a spawned ray exits a cell, the Cell Tree node corresponding to its parent is checked to see if the spawned ray dependency is represented by one of its child nodes. If it isn't, then a new

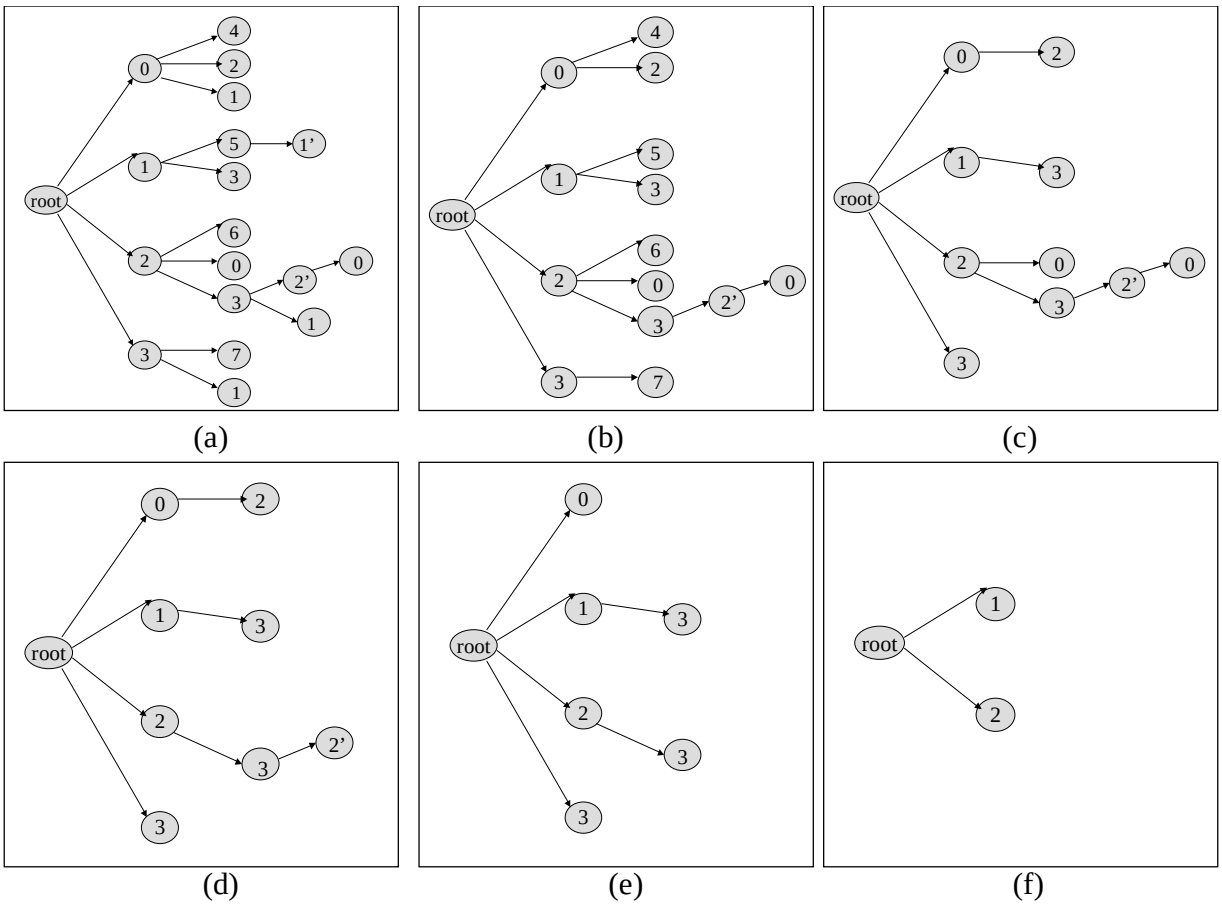

Figure 4: *Cell Tree algorithm example: (a) Completion peel of cell* $1'$*; (b) Completion peel of cells* 4, 5, 6 *and* 7*; (c) Split peel of cell* 0*; (d) Completion peel of cell* $2'$*; (e) Completion peel of cells 0, 3; (f) Completion peel of cells 1 and 2.*

node is added to the Cell Tree. A formal description of Cell Tree construction follows:

1. Initialize: add one node to the Cell Tree for each cell entered by any initial rays, and tag each ray with the node corresponding to its initial cell.
2. When a ray exits a cell, determine if the exit cell is a child of its Cell Tree node.
3. If it is, then tag the exit ray with that child node.
4. If not, create a cell dependency by adding a child to its Cell Tree node and tag the exit ray with the new child node.

3.3. Cell Tree Algorithm

The cell-processing schedule for the next frame is determined in reverse order from the Cell Tree. The tree is processed starting with the roots by peeling the nodes of a selected cell and adding it to the reverse schedule. An interim tree is maintained by keeping a list of nodes that haven't been included in the schedule. The *generation* of a node is one plus the number of nodes with the same cell that are ancestors to that node. We call the maximum generation of a cell on the current tree *maxGen*. If all of the maxGen nodes of a cell are leaf nodes on the current tree, we call it a *ready cell*. In this case, we do a *completion peel* on that cell by adding it to the end of the schedule, peeling all of its leaf nodes from the current tree and decreasing the cell's maxGen by one. No extra cache misses are incurred for these nodes. If no cell has all of its maximal generation nodes ready, the schedule must include an extra memory fetch for some cell. We call this a *split peel*; the peel is unable to gather all remaining nodes of a cell's highest generation on the current tree. We define the *current schedule* as the schedule found by peeling the original Cell Tree to obtain the current sub-tree.

(a) (b) (c)

Figure 5: *Reflective images: (a) CT brain with mirrors; (b) Clouds and moon reflecting in a lake; (c) CT lobster reflected in several mirrors to give a tunnel appearance.*

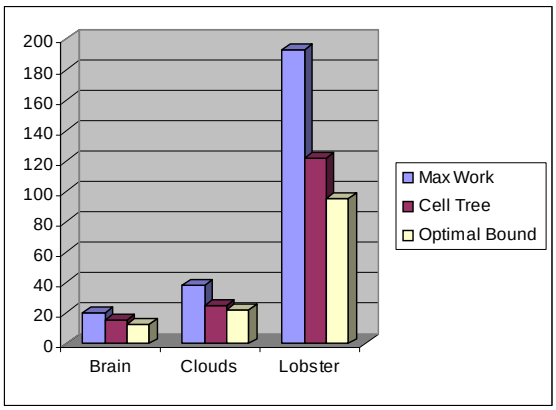

Figure 6: *Memory fetches with the Max Work and the Cell Tree algorithms, and the lower bounds of the optimal schedule.*

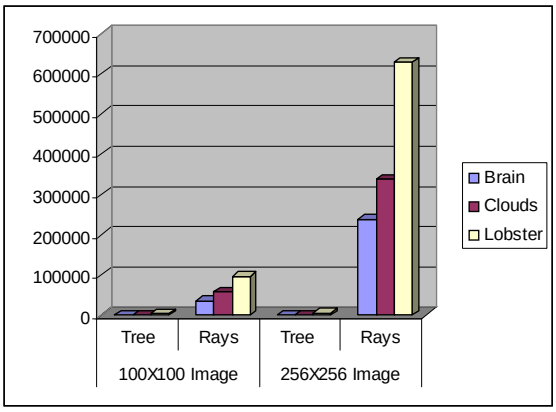

Figure 7: *Cell Tree sizes compared with rays represented.*

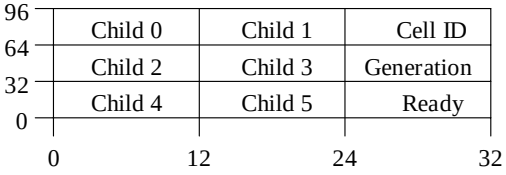

Figure 8: *Cell Tree node configuration.*

If there are no completion peels for some sub-tree, then we know that whatever cell we peel at this level will need to be fetched from memory at least twice for its current maxGen as explained in Section 3.4. If the current schedule has been obtained strictly by completion peels, then it is optimal and the overall optimal schedule can be determined recursively as follows: If we compare the optimal schedule of each sub-tree that is obtained by peeling each cells' leaf nodes from the current sub-tree, one cell at a time, he optimal for the original tree is the current schedule concatenated with the cell whose peel results in the sub-tree with the shortest optimal schedule, together with the optimal schedule of that sub-tree. The recursion can be ended when any sub-tree is found to have all completion peels.

If each iteration reveals another split, then a recursive call is made for each cell until the root is reached, and each of these peels may involve only a single node. This means that the recursive algorithm has a worst time bound of O(n!), where n is the number of nodes in the cell tree. Furthermore, a clone of the current tree must be kept at each stage of the recursion. Both of these problems make the recursive algorithm impractical. We have developed a simpler version,

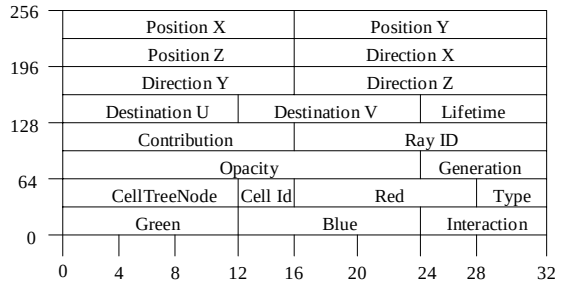

Figure 9: *Bit widths in the 32 byte ray packet.*

called the the Cell Tree algorithm, with polynomial worst time bounds, which is close to optimal. Instead of determining the best choice for the split peel recursively, we simply choose the cell with the most leaf nodes. Alternate criteria include maximum generation or maximal tree level. The Cell Tree algorithm will produce an optimal schedule if the best choices of split peels are always made. The expected amount of sub-optimality increases with the number of split peels. Empirical results indicate that the completion peels tend to dominate, thus the algorithm is close to optimal. The Cell Tree algorithm proceeds as follows until all nodes have been peeled from the tree:

1. Determine if any cell is in the ready state.
2. If so, add that cell to the reverse schedule, peel all of that cell's leaf nodes and decrement that cell's maxGen.
3. If not, choose the cell with the most leaf nodes.

Figure 4 shows an example of the Cell Tree algorithm for creating a schedule. The maximum generation nodes for cells 1 and 2 are designated as $1'$ and $2'$, respectively. In Figure 4a, all $1'$ nodes of this type are ready, so we get a completion peel. After this peel, in Figure 4b, cells 4, 5, 6 and 7 each has only one node which is a leaf node. Thus, these are all completion peels. The resulting sub-tree after these steps is shown in Figure 4c. There is no ready cell, so Cell 0 is selected for a split peel since it has the most leaf nodes. As a result, in Figure 4d, all $2'$ nodes are exposed as leaf nodes, so cell 2 is now in a ready state and it is completion peeled. The resulting sub-tree after these steps is shown in Figure 4e. All nodes for cells 0 and 3 are leaf nodes, so they are completion peeled. This exposes the remaining nodes for cells 1 and 2, which are completion peeled and the algorithm is finished.

3.4. Algorithm Performance

The images rendered for testing the algorithm are shown in Figure 5 and described in Section 4. Figure 6 shows the schedules found by the Cell Tree algorithm compared with the lower bounds of the optimal schedule. The minimal schedule is equal to the sum of each cell's maximum generation plus the minimum number of split peels. This can be shown as follows: The minimum number of memory fetches for a cell is the largest generation number of that cell. This follows from the definition of generation. If cell i is part of a completion, then peeling cell i would result in no extra memory fetches compared with an optimal schedule. When a completion peel is made, exactly one generation, g, of the cell is completed. Although some nodes that are peeled may be from a smaller generation, f, all of the nodes from f cannot be peeled because each chain leading to the generation g node also contains a generation f node which cannot be peeled at the same time. Thus cell i must be cached in at least once for generation g and a separate time for generation f.

The Cell Tree algorithm always finds a feasible solution. A feasible schedule is one where every ray dependency is included in some subsequence of the schedule. First, we must show that the Cell Tree contains a sub-tree for each ray dependency tree. After that we show that the construction of the schedule guarantees that the partial order is not violated on the Cell Tree, and that every node in the tree is included in the schedule. The Cell Tree is constructed by considering every ray dependency. The rays are marked with their current place in the tree, and when a ray (original or spawned) enters a new cell, the dependency created is added to the Cell Tree by adding the new cell as a child node of the rays current Cell Tree node unless it already exists. This guarantees that the Cell Tree represents all ray dependencies. Hence, all ray paths exist as sub-paths of the Cell Tree. Since the reverse schedule is produced by traversal from the Cell Tree leaves to the root, no two jobs j and j-1 will be placed on the reverse schedule with job j-1 preceding job j. This means that the partial order of ray dependencies is never violated. No dependency is ignored since the schedule is not complete until all nodes are peeled.

Each iteration of the Cell Tree algorithm removes at least one node from the tree. Each node removal requires checking each node in the tree at most once. Thus, the worst time case for the Cell Tree algorithm is $O(n^2)$ where n is the number of tree nodes. The upper bound of the Cell Tree size is the total number of rays created. However this is an unrealistic bound in that it would mean that each ray is independent of every other ray. In our simulations, the Cell Trees had an average of 1797 nodes (see Figure 7), which is more than 100 times smaller than the average number of rays, which is $199,919$. 96KBytes of the DSP RAM is used for storing the Cell Tree. This is sufficient to hold $8,000$ 12-Byte Cell Tree nodes (see Figure 8). In addition to the Cell Tree itself, each ray contains a Cell Tree node id, a cell id and a ray id. These fit in the GI-Cube ray packet by decreasing the size of some items without any resulting image quality degradation (see Figure 9).

Cell Tree creation is done during idle DSP cycles. When a ray needs to be queued by the DSP because it is exiting the cell currently in memory, the ray packet is sent over the

ray bus. The cell dependency information, included in this packet, is gathered by the DSP. Each Cell Tree node creation requires one write. Each ray packet is used for a single decision and at most one write to the Cell Tree. The Cell Tree node ID is updated before the ray packet is placed on a queue in the DSP.

Schedule creation is done in the DSP between frames. In our experiments the average number of peels, including split peels, was 55, which is much less than the worst case, which would be one peel per node. This is because peels usually result in the removal of several nodes, and because in most iterations there is a completion peel. In our tests the average number of nodes examined for each completion peel was 85. Based on our experiments, we estimate the time for schedule creation in a 255 MIPS DSP to be between 5 and 10 microseconds. If, instead, the schedule for frame $i+1$ is created while the processor is rendering i, a copy of the Cell Tree from frame $i-1$ must remain on the DSP, thus doubling the DSP memory requirement.

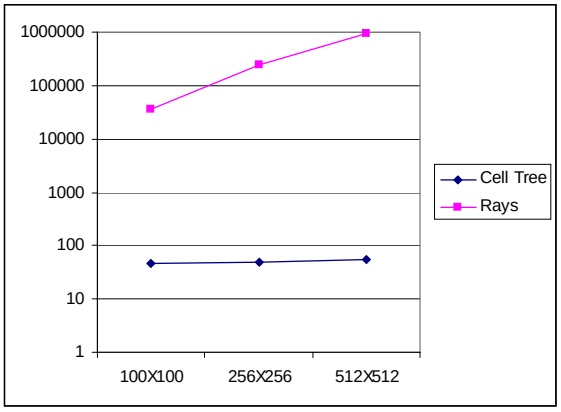

Figure 10: *Cell Tree sizes for the brain image.*

4. Testing and Results

We simulated our system in C++ on an SGI O2/RISC10000 with 128MB of RAM. Three scenes are shown, each rendered with a 256×256 image size. The first scene is an MRI brain reflected in multiple mirrors for a complete look. It includes one $128 \times 128 \times 84$ by eight bits/voxel volume and three planes. The second scene is two clouds and a moon reflected in a lake. It includes two $120 \times 120 \times 120$ volumes, a geometric sphere and three planes. The third is a lobster reflected several times to give a tunnel appearance. It includes one $256 \times 254 \times 57$ volume and five planes. All of the volumes had eight bit voxels. In order to test our Cell Tree construction and schedule creation algorithms, we constrained the on-board memory to be 1/8 and $1/27$ of the volume for each scene.

The sizes of the Cell Trees did not grow nearly as quickly as the number of rays. Figure 10 shows the number of ray dependencies for the brain scene at resolutions of 100×100, 256×256 and 512×512 pixels. We use a logarithmic scale because the Cell Tree sizes grow much more slowly than the number of ray dependencies. The biggest factor in the cell tree sizes was the number of reflections. Image and volume sizes each had a relatively small influence on Cell Tree sizes.

The relative number of misses with the Cell Tree schedule compared with the Max Work algorithm schedule are shown in Figure 6. The memory fetches decreased an average of 30% by using dependency graph based scheduling, with a range of 20% to 37%. Also shown are the lower bound sizes of the optimal schedule. This is a very conservative lower bound. It assumes that the only necessary split peel is the first one, which is not likely to be the case. The Cell Tree schedule was always within 33% of this lower bound. By comparing the number of split peels to the number of completion peels we can get a feeling of how close to optimal the schedule is. The algorithm performs best when there are relatively few split peels.

The performance improvement is greatest when there are more inter-reflections, which is also when it is needed most because the cells must be fetched from memory more frequently. The resulting frame rate increase depends on the relationship between disk fetch time and on board memory cache-in time. The on-board RDRAM memory bandwidth of 0.8GB/sec in our system is sufficient to avoid nearly all stalls due to RDRAM fetches, so the only overhead results from retrieving data from disk. If smaller and/or cheaper memory is used, our algorithm could be applied between the on-board and the cache and would have an additional impact on rendering time.

5. Conclusions and Future Work

We have demonstrated the effectiveness of using ray dependency information in reducing the total number of memory fetches for a ray tracing system where the volume is subdivided into cells. The Cell Tree can be constructed easily in hardware with virtually no cost. This data structure can be utilized in several ways. The Cell Tree algorithm could be extended for dynamic load balancing in both distributed and shared memory systems. Simulations have indicated that volume sub-division could be improved using ray dependencies.

The Cell Tree introduced in this paper allows us to study the best way to split up a volume. For example, when a ray is being reflected between two cells it is likely that the load balance would be improved by a new subdivision which places the parts of the volume being reflected into the same subdivision. Allowing multiple cells per memory unit allows greater flexibility in the schedule at the cost of further complexity. This trade-off should be studied further.

An extension of the algorithm should prove useful in mul-

tiprocessor scheduling. If p cells are to be processed at the same time, and cell i is repeated in the schedule within p processes, it may be handled by the same processor, but it will stall while waiting for other cells which the second occurrence of cell i is dependent on (and the first one isn't), to complete processing. The tradeoff of stalling compared with fetching the cell again, needs to be considered.

There is a wide variety of memory types, with a wide range of bandwidth, size and speed combinations commercially available. When a ray tracing system is being designed, a thorough study of memory configurations is needed. A data-structure containing a compact description of cell-dependency information gives us a scientific means to compare several configurations. We plan to explore these areas further and to proceed with VHDL simulation and synthesis.

Acknowledgements

This work is partially supported by Office of Naval Research Grant No. N00140110034, CES Computer Solutions Inc., and New York State Office of Science, Technology, and Academic Research (NYSTAR) Grant No. COD0057. We would like to thank Kevin Kreeger, Frank Dachille, Michael Bender and Nan Zhang for their assistance and helpful discussions.

References

1. F. Dachille and A. Kaufman. GI-Cube: An architecture for volumetric global illumination and rendering. In *Proceedings of the 2000 SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 119–129, August 2000. 2
2. M. de Boer, A. Gröpel, J. Hesser, and R. Männer. Latency- and hazard-free volume memory architecture for direct volume rendering. In *Proceedings of the 11th Eurographics Workshop on Graphics Hardware '96*, August 1996. 1
3. M. Doggett, M. Meißner, and U. Kanus. A low-cost memory architecture for pci-based interactive ray casting. *1999 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pages 7–13, August 1999. 1
4. T. Günther, C. Poliwoda, C. Reinhart, J. Hesser, R. Männer, H.-P. Meinzer, and H.-J. Baur. Virim: A massively parallel processor for real-time volume visualization in medicine. *Computers & Graphics*, 19(5):705–710, September 1995. 1
5. H. Igehy, M. Eldridge, and P. Hanrahan. Parallel texture caching. In *Proceedings of the 1999 SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 95–106. ACM, August 1999. 2
6. K. Kreeger and A. Kaufman. Hybrid volume and polygon rendering with cube hardware. *1999 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pages 15–24, August 1999. 2
7. Kevin Kreeger and Arie Kaufman. Interactive volume segmentation with the pavlov architecture. *Symposium on Parallel Visualization and Graphics*, pages 61–68, October 1999. 2
8. M. Meissner, U. Kanus, and W. Strasser. VIZARD II, A PCI-card for real-time volume rendering. In *Proceedings of the 1998 SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 61–68, August 1998. 1
9. H. Nishimura, T. Endo, T. Maruyama, J. Saito, and P. H. Christensen. Parallel rendering and the quest for realism: The KILAUEA massively parallel ray tracer. In *SIGGRAPH'01 Course Notes*, pages 1–59, August 2001. 2
10. M. E. Palmer, S. Taylor, and B. Totty. Exploiting deep parallel memory hierarchies for ray casting volume rendering. *IEEE Parallel Rendering Symposium*, pages 15–22, November 1997. 2
11. H. Pfister, J. Hardenbergh, J. Knittel, H. Lauer, and L. Seiler. The volumepro real-time ray-casting system. In *Computer Graphics (SIGGRAPH'99 Proceedings)*, pages 251–260, August 1999. 1
12. H. Pfister and A. Kaufman. Cube-4 - a scalable architecture for real-time volume rendering. *1996 Volume Visualization Symposium*, pages 47–54, October 1996. 1
13. M. Pharr, C. Kolb, R. Gershbein, , and P. Hanrahan. Rendering complex scenes with memory-coherent ray tracing. In *Computer Graphics (SIGGRAPH'97 Proceedings)*, pages 101–108, August 1997. 1
14. H. Ray and D. Silver. The Race II engine for real-time volume rendering. In *Proceedings of the 2000 SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 129–137, August 2000. 2
15. E. Reinhard, A. Chalmers, and F. Jansen. Hybrid scheduling for parallel rendering using coherent ray tasks. *In Proceedings of the IEEE Parallel Visualization and Graphics Symposium 1999*, pages 21–28, 1999. 2
16. Motorola Semiconductor Products Sector. Digital signal processors: Product specification. 2002. http://e-www.motorola.com/webapp/. 3