EG UK Theory and Practice of Computer Graphics (2009)
Wen Tang, John Collomosse (Editors)

Design and Evaluation of a Hardware Accelerated Ray Tracing Data Structure

Michael Steffen and Joseph Zambreno[†]

Department of Electrical and Computer Engineering
Iowa State University, USA

Abstract

The increase in graphics card performance and processor core count has allowed significant performance acceleration for ray tracing applications. Future graphics architectures are expected to continue increasing the number of processor cores, further improving performance by exploiting data parallelism. However, current ray tracing implementations are based on recursive searches which involve multiple memory reads. Consequently, software implementations are used without any dedicated hardware acceleration. In this paper, we introduce a ray tracing method designed around hierarchical space subdivision schemes that reduces memory operations. In addition, parts of this traversal method can be performed in fixed hardware running in parallel with programmable graphics processors.

We used a custom performance simulator that uses our traversal method, based on a kd-tree, to compare against a conventional kd-tree. The system memory requirements and system memory reads are analyzed in detail for both acceleration structures. We simulated six benchmark scenes and show a reduction in the number of memory reads of up to 70 percent compared to current recursive methods for scenes with over 100,000 polygons.

Categories and Subject Descriptors (according to ACM CCS): I.3.1 [Computer Graphics]: Graphics Processors

1. Introduction

Ray tracing algorithm performance has improved due to the reduced time spent processing individual rays as well as the abundant parallelism that modern hardware has enabled. One variable that has led to the increased efficiency of ray computations has been the use of custom data structures. Data structures define how geometric data are stored in memory and determine what elements should be tested for a given ray. A data structure that reduces the number of computations and memory reads while providing relative geometry elements for intersection testing will likely result in an overall performance improvement. Since all ray types (visible, shadow, reflection, etc) require intersection tests, a common data structure can be used for all rays. Current data structures determine geometry elements by traversal, which is a process of stepping through hierarchical layers. This results in a recursive search processes and multiple memory reads. In this paper, we propose a method for reducing the number of recursive steps by starting traversal of the tree data structure farther down the tree rather then starting at the top.

Today's graphics hardware supports large numbers of multiple cores for data parallelism in graphics rendering. To further increase the amount of computational parallelism, our approach calls for parts of the traversal to run on additional fixed hardware, freeing up processor time for other computations. User programmability is offered by performing intersection tests on the graphics processor, enabling user-defined code to interact with system memory and setting up the format of the data structure in memory.

We use a custom performance simulator for comparing our data structure implementing a kd-tree and a conventional kd-tree with a stack [Hav00]. The performance simulator reveals the total system memory required for all implementations and the resulting memory reads. Our experimental results using this simulator show a significant reduction in memory reads for scenes with 100,000 or more polygons.

[†] {steffma, zambreno}@iastate.edu

DOI: 10.2312/LocalChapterEvents/TPCG/TPCG09/101-108

The remainder of this paper is organized as follows: Section 2 discusses previous work in data structures and graphics hardware. Section 3 describes our custom data structure and traversal operations. Section 4 outlines an architecture for implementing our data structure and Section 5 shows results of our simulator using six benchmark scenes. Finally Section 6 and 7 concludes the paper with a description of planned future work.

2. Previous Work

To accelerate rendering time, a variety of rendering methods [Whi80] [DBB02] [GTGB84] use accelerated data structures and parallelism offered by hardware. In the past, data structures and the hardware have been developed separately; however in the near future it is expected that graphics hardware is expected to move away from z-buffer rendering and directly incorporate scene data structures for ray tracing [MF05].

2.1. Accelerated Data Structures

Hierarchical Space Subdivision Schemes (HS3) like kd-tree [Hav00] and Bounding Volume Hierarchies (BVHs) [SM03] have been the common data structures used for ray tracing implementations. There popularity comes from their ability to adapt to the geometry of a scene allowing for efficient ray triangle intersection tests. Current-day implementations of HS3 have focused on reducing the number of memory operations and maintaining ray coherency for specific hardware platforms.

Today's kd-tree data structures for GPUs have improved the runtime performance for traversing the data structure and are achieving faster rendering time then CPUs [PGSS07] [HSHH07]. The use of HS3 does result in a recursive traversal method requiring frequent memory reads. Ray coherency [WBWS01] has been used to group similar rays together to improve the locality of memory reads for cache optimization. Still, a significant time is spent in the recursive search of these data structures. Quad-BVH [DHK08] utilizes processor vector-width to convert binary trees into quad based trees, reducing the size of the tree and the number of traversal steps.

Uniform grid data structures [WIK*06] are commonly used because of their ability to quickly step to neighboring grid points. Grid data structures do not require any recursive search method, however, they cannot adapt to the scene geometry and can result in multiple iterations before performing relative ray triangle intersection tests. Because of its non recursive traversal, uniform grid data structures outperform HS3 data structures for scenes that are uniformly distributed. Because most scenes are not uniformly distributed, uniform grid data structures are not commonly used.

2.2. Ray Tracing Hardware

Research in graphic architectures has resulted in fixed function [Han07] [FR03] and programmable multi-core designs [SKK*08] for accelerating ray tracing rendering. Fixed function hardware such as SaarCore [SWS02] [SWW*04] and RPU [WSS05] [WBS06] implement a fully defined rendering pipeline for ray tracing. Both designs have been fully implemented on FPGA and an ASIC model was developed for RPU. Dedicated hardware for traversing tree data structures is included in both designs, but recursion and multiple memory reads are not eliminated. Additionally, legacy rendering methods such as rasterization and others using the current programmability of modern pipelines are not supported.

Other architectures for ray tracing introduce a high number of general multi-core processors and memory systems designed for graphic computations. Intel's Larrabee [SCS*08] is composed of several in-order x86 SIMD cores that can be programmed to support any graphics pipeline in software. Acceleration comes from running large amounts of graphic computations in parallel and running multiple threads on each processor to reduce the latency for memory operations. CUDA [NVI07] and Copernicus [GDS*08] also offer large numbers of cores and can hide memory latency by having large numbers of threads with no switching penalty. Ray tracing implementations on these architectures is accomplished through software kernels that then run on the processors. Direct hardware acceleration is supported for several graphic computations, but none for data structure traversal.

3. Group Uniform Grid

3.1. Overview

To reduce the number of recursive steps required by HS3 to processes any ray type (visible, shadow, reflection, etc), we propose implementing an additional data structure called Group Uniform Grid (GrUG) over the HS3 data structure. GrUG makes rays bypass parts of the tree structure allowing for traversal to begin closer to its final leaf node. GrUG is an axis-aligned subdivision of space consisting of only two hierarchical layers. The top layer is a uniform grid data structure that divides the scene into grid cells. The lower layer consists of groups of top layer cells and corresponds to nodes of the HS3 tree structure. Figure 1 shows a 2-D example of the two layers and a kd-tree.

To traverse this data structure, only the mapping between the two layers and mapping between each ray and its grid need to be addressed. Once this mapping is complete, the HS3 structure can be traversed starting at the node identified by GrUG. Mappings between layers and ray is performed using a hash lookup table. The value of the hash function of a ray origin equals the top layer cell ID corresponding to the ray. By this method, a collision produced by a hash function

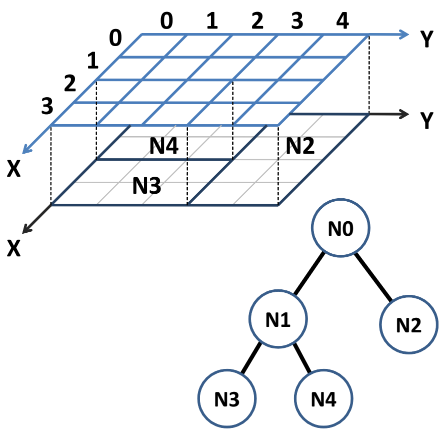
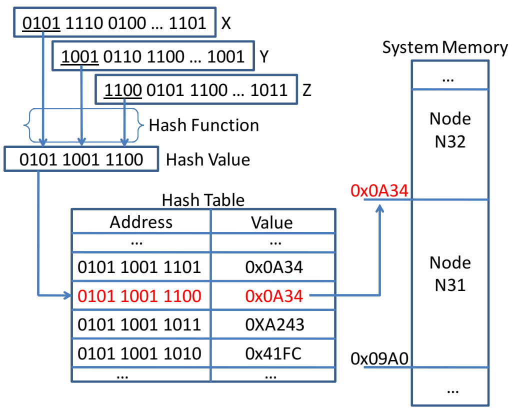


Figure 1: *GrUG data structure composed of a uniform grid for the top layer and a lower layer that maps to a HS3 tree structure.*



Address	Value
…	…
0101 1001 1101	0x0A34
0101 1001 1100	0x0A34
0101 1001 1011	0XA243
0101 1001 1010	0x41FC
…	…

Figure 2: *Hash function starting with X,Y,Z coordinates in integer format and producing the data structure that contains geometry information for intersection testing.*

indicates that the rays are in the same cell. The hash value is then used in a table lookup to determine the memory address of the HS3 node data structure. Figure 2 shows the entire hashing process starting with integer X,Y and Z coordinates.

Using GrUG to reduce recursion and memory reads results in the top data structure resembling a uniform grid. Rays that require additional traversal outside of its original group bounding tree cannot continue using HS3 methods, but require traversing the uniform grid. Stepping to the next grouping can be done by a 3-D Digital Differential Analyzer (DDA) [AW87]. The DDA allows quick stepping between cells because they are on a uniform grid pattern. Stepping to neighboring cells does not guarantee that the lower layer tree node has changed, so this process is repeated until a new tree node is found. This process can only be performed after the initial traversal and is not needed for intersection testing, and therefore can be run in parallel with triangle intersection tests.

To create a hash function with collisions resulting in rays being in the same top layer cell requires an integer format representation for ray origin. A uniform grid data structure maps nicely to integer values as each cell is assigned to an index value. With the DDA implemented on top of the uniform grid, integer arithmetic can be applied to the DDA. With proper setup, the use of integer operations inside the traversal of GrUG still maintains floating point accuracy.

The use of dedicated hardware for GrUG traversal cannot stand on its own, but must be able to interface to the processing cores for intersection testing and traversal of the HS3 tree. While a complete interface is not presented for this implementation, a high level architecture model is shown for performing data structure traversal. This model relies on the software processor to run HS3 traversal, triangle intersection tests and shading while the GrUG hardware performs data structure traversal.

3.2. Hash Table Function

There are two parts to a hash table function: the hash function itself and the table memory. For our approach, the table memory is used to take a grid cell and map it to the tree node of the HS3 structure. HS3 nodes are stored in system memory, so a pointer to system memory is stored inside the hash table. Our table size is then equal to the total number of cells created by the uniform grid. For simplification, only powers of 2 are allowed for grid spacing. The number of bits needed to address the table memory is then $Log_2(number\ of\ cells)$.

The hash function result is used as the address in the table memory and must produce a value between 0 and the $(total\ number\ of\ cells - 1)$. Furthermore, a hash function collision indicates that the two ray origins that produced the collision are in the same uniform grid cell. If the ray origins are represented in integer format, this is a simple operation of concatenating the most significant bits of each axis. The number of significant bits is determined by the grid spacing. The conversion from floating point notation to integer notation is described in Section 3.5

3.3. Creation

Creating a GrUG data structure requires setting up two memory spaces, the hash table and the HS3 tree. Since the hash table memory contains pointers to the HS3 tree, we setup the HS3 tree structure first and then populate the hash table memory with pointers to the appropriate HS3 tree node. GrUG requires only one constraint on building the HS3 creation algorithm. GrUG operates on a uniform grid structure, so splitting locations of the HS3 structure must align with these bounds until a mapping between GrUG and the node is defined. By forcing the HS3 tree structure to align with GrUG, leaf nodes generated with aligned splitting locations are identified as thresholds. Thresholds have spatial bounds corresponding to cells in the uniform grid and are the nodes that get mapped to GrUG. Threshold nodes can then continue being subdivided without any restraints on the splitting locations. The mapping of threshold nodes to GrUG requires that every uniform cell defined in the threshold bounding area gets populated with a pointer to that node. Once this is complete, every node above the threshold node can be removed from memory, resulting in several smaller trees. Each tree is mapped to the GrUG structure.

In addition to storing the resulting trees, bounding values must also be stored with the threshold node. The specific format for storing bounding values and HS3 forests are user-defined and requires the ray intersection code to interpret pointers to system memory generated by GrUG. This allows for grouping of geometric data with rendering parameters such as color and texture coordinates in the same memory location. While the specific implementation is left to the user, two requirements must be met:

- Tree nodes and geometry data inside the scene must be present and accessible by only providing the memory address to the data structure. This allows traversal of GrUG to produce a single memory address and have the tree traversal and ray polygon intersection code be able to test all relative polygons in the cell area.
- Six bounding values of the grouped grid must be contained in the data structure. These values are used for computing the next grouping of ray intersects if no intersection is found in the current grouping. These values must also be accessible by providing the same pointer to system memory.

3.4. Stepping Between Neighbors

To step between uniform grid cells, a DDA method is used. While this method allows stepping between cells, this process repeats until reaching the boundary of the starting group. DDA steps one cell at a time and is a function of both the current cell and the direction of the ray. The absolute position of the ray inside the cell is only needed once for the entire traversal of the ray. Three parameters are needed per axis for stepping: *tmax*, *delta* and *step* (Figure 3). The *tmax* value is the independent variable in the 3-D parametric line equation and is incremented when traversed along that axis. The *delta* value defines how much *tmax* gets incremented by and is the value needed to cross an entire cell. The final value is the *step* value that specifies what direction to step and is discussed in more detail in the next section. Stepping is then done along the axis that has the lowest *tmax* value since it is the closest to the edge of the cell. The *tmax* value is then incremented by the *delta* parameter to represent the

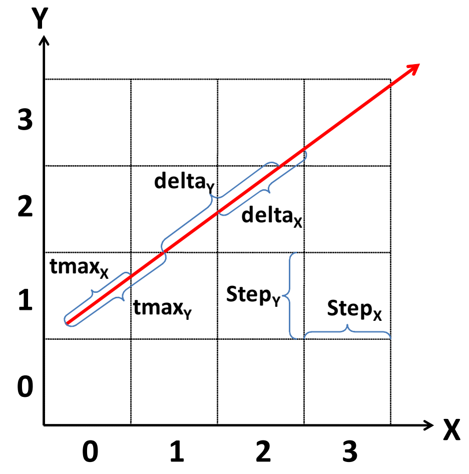


Figure 3: *Variables needed for DDA traversal.*

new distance from the edge of the cell. This process is repeated until the cell index value is outside of the group. The boundaries of groups must be provided to the hardware before this process can begin and is stored in the data structure for the current group.

3.5. Integer Operations

The GrUG hash function uses integer values to represent the entire grid spacing. Conversion of ray origin to integer values should utilize the entire integer range. A fixed size of *N* bits is used for representing the integer value and the grid resolution is specified during initialization, the number of integers that are inside of a cell is then equal to $\frac{2^N}{GridResolution}$. The index of each cell is then equal to the hash function for any points inside that cell. Pre-computed integer scaling values are computed from the maximum and minimum limits of the scene. Linear interpolation can determine the location inside the uniform grid. Once the integer value has been found, it is used throughout the rest of the traversal.

Since the location is now stored as an integer format, integer arithmetic is used for the DDA traversal. Both *tmax* and *delta* are calculated in a similar fashion but the step parameter is then assigned a constant value of $\pm\frac{2^N}{GridResolution}$. This value is the number of integers in a cell. When traversing to a neighbor cell the integer location value must change by that amount to leave the current cell. The $\pm$ determines the direction of the step and has the same sign as the ray direction.

4. Hardware Implementation

An architecture for the GrUG traversal process is presented in Figure 4. Because a fully defined pipeline is beyond the

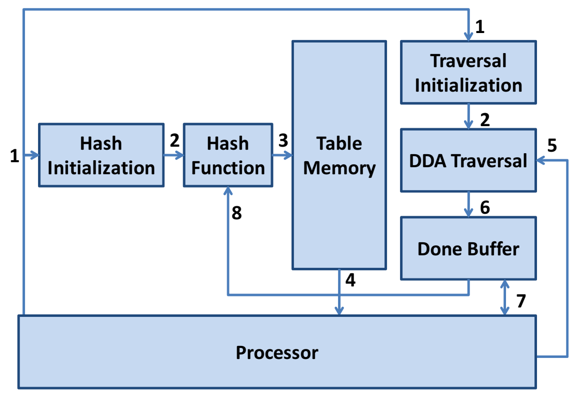


Figure 4: *Architecture of Grouped Uniform Grid*

scope of this paper, only the hardware for GrUG traversal, along with its required interactions with the processor, are shown. For a multi-core implementation, each processor would require its own hardware implementation. The only exception is the table memory that would be shared among all the processors. This purposed architecture breaks down into two main areas, hash table lookup and traversal. Hash table lookup takes two inputs, one for new rays and the second for performing another traversal of the data structure. Since new rays only need to be initialized, returning rays can be sent directly to the hash function. The output of the hash table can then be passed directly to the processor for triangle intersections. The traversal hardware also requires two inputs for initializing new rays and performing further traversal operations. The output of traversal is then sent to a buffer that waits for the processor to indicate if the ray has finished or needs to continue traversing.

The architecture presented is a pipelined implementation and operation is performed in the following order for new rays corresponding to the numbers presented in Figure 4:

1. New rays are passed into the architecture from the processor and are initialized in parallel for both the hash table function and the traversal using DDA.
2. The output of the hash initialization is then inputted into the hash function where the hash value of the ray is computed. The computation results in the ID for the cell that the ray belongs to. The traversal initialization results are also passed into the DDA traversal where it waits for further inputs.
3. The resulting hash value is then used as the address in the table memory. The resulting memory value is then the pointer address for a node/leaf in the HS3 tree.
4. The processor then begins executing the HS3 traversal and intersection kernel and performs a memory read to get the bounding values of the tree node/leaf.
5. The processor outputs the bounding values to the DDA traversal to determine the next grouping the ray intersects. The processor then begins traversal of the HS3 and computing the intersection results in parallel with the DDA.
6. The DDA traversal finishes and outputs the results to the done buffer.
7. The processor finishes intersection tests and tells the done buffer of any intersection results.
8. The done buffer checks the intersection results and the DDA result to determine if the ray requires additional traversing. If the ray must continue traversal the new ray location is passed into the hash function and begins step 3 again. If the ray intersects a valid polygone, it is removed from the pipeline.

Benchmarks	polygons	kd-tree Mem. (MB)	GrUG Memory (MB)			
			Tree	Boundary	Hash-table	Total
Ulm Box	492	0.01	0.01	0.01	512.00	512.02
Stanford Bunny	69451	1.34	0.87	1.41	512.00	514.28
Fairy Forest	172561	2.66	2.32	1.02	512.00	515.34
Cabin	217903	3.12	2.88	0.70	512.00	515.58
Atrium	559992	9.30	8.77	1.61	512.00	522.38
Conference	987522	8.54	8.23	0.93	512.00	521.17

Figure 5: *Benchmark data for each scene and the total memory required by the data structure.*

While a software implementation of GrUG is feasible, A hardware implementation of our GrUG approach will have greater performance improvement by allowing additional operations to perform in parallel that would not be possible with a software implementation. In the hardware method the DDA traversal is able to run in parallel with triangle intersection, allowing for its computational cost to be hidden. In addition to operating in parallel, specific computations can be accelerated including the hash function and the operations performed for each axis. The hash function and axis operations require multiple instructions in software that can be performed by a single functional unit in hardware. This acceleration allows for a hardware implementation of GrUG to have a greater performance over an equivalent software implementation.

5. Performance Analysis

A total of six different scenes, each with different polygon counts, were tested using our custom CUDA-based performance simulator. Figures 5 and 6 list the benchmarks with scene data and overall results.

Our performance simulator implements only the traversal of primary visible rays and does not perform any rendering. The HS3 data structure implemented is a kd-tree used in RADIUS-CUDA [Ben08]. To measure performance of GrUG, only system memory requirements and the number of memory reads are reported. All benchmark scenes were simulated at a resolution of 1920x1080, resulting in a total of 2,073,600 rays, and a GrUG grid size of 512x512x512.

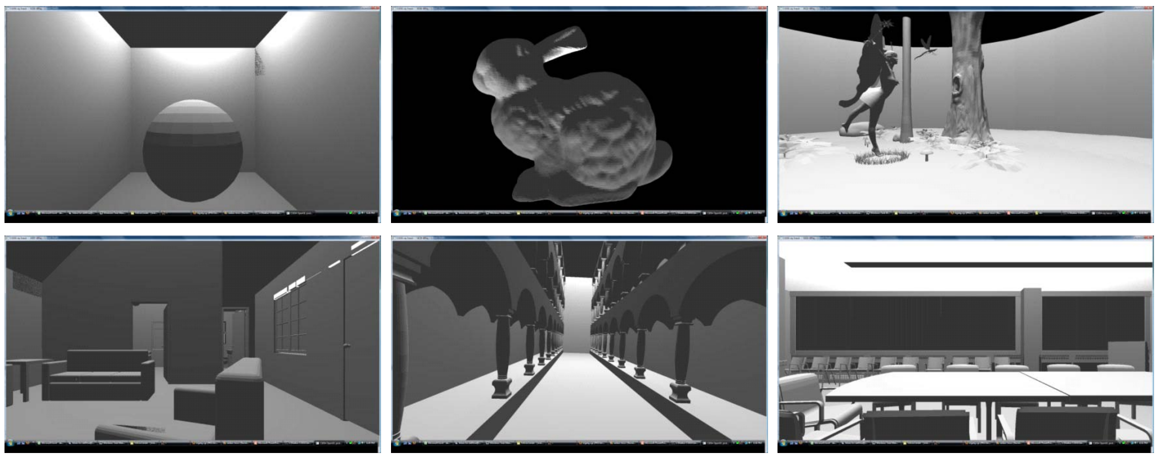


Figure 6: *Benchmark scenes from left to right, top to bottom: UlmBox, Stanford Bunny, Fairy Forest, Cabin, Atrium and Conference. Images rendered at a resolution of 1920x1080 using RADIUS-CUDA.*

5.1. Memory Utilization

Figure 5 shows the total system memory needed for storing the different data structures used by GrUG. The use of a hash table in GrUG results in a significant overhead in memory requirements to store the entire hash table in memory. The required memory is based on the GrUG grid size, 4 bytes per grid cell. A fixed grid size of 512x512x512 will use 512MB for storing the hash table.

In addition to the hash-table, the kd-tree structure and bounding dimensions of all threshold nodes must be stored in system memory. Figure 7 shows the memory requirements for both kd-tree and GrUG. GrUG uses less memory for storing the tree structure and bounding dimensions of threshold nodes since it does not need to store the entire tree structure, but only the tree nodes that are at and below the threshold node. The additional 24 bytes needed for storing the bounding dimensions, equivalent of 3 nodes, is smaller than the tree structure above threshold nodes.

Figure 7 also shows that the memory space required for storing GrUG tree data sets are not linear with respect to the number of polygons in the scene. Kd-tree memory usage scales linearly with polygon count because the number of nodes created is a function of the tree depth, which is determined by the number of polygons in the model. Memory requirements for GrUG are influenced by the density of polygons in a scene. Polygon density is a function of the number of polygons in a scene and how evenly distributed they are in the entire scene. Scenes with higher polygon density will result in more threshold nodes, resulting in deeper trees and additional bounding dimensions being stored.

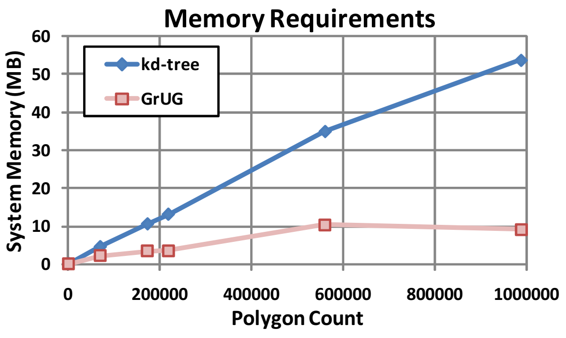


Figure 7: *The memory requirements of GrUG compared to kd-tree.*

5.2. Memory Operations

The resulting memory reads for both GrUG and kd-tree are shown in Figure 8. The number of memory reads are for traversal of GrUG and kd-tree data structure and do not include the memory operations needed for triangle intersection testing. Memory reads from triangle intersection testing are not reported because both implementations resulted in nearly identical intersection tests.

All but one benchmark scene required fewer memory reads using GrUG. UlmBox resulted in higher memory reads due to having a small kd-tree size. While GrUG allowed for traversal to begin farther down the kd-tree, the resulting memory read operations for getting boundary data was larger than the savings of starting lower on the kd-tree. The remaining five benchmarks resulted in varying reduced mem-

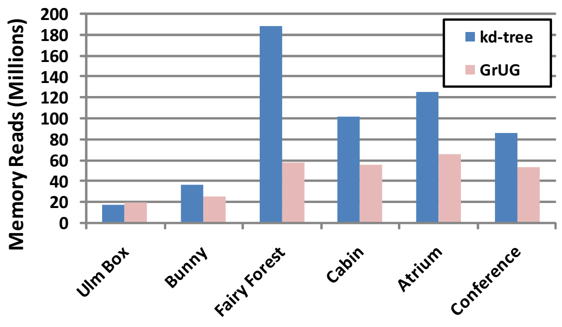


Figure 8: *Number of memory reads required for traversing GrUG and kd-tree.*

ory reads. The number of reduced memory reads needed by GrUG is greatly dependent on the polygon density.

6. Future Work

To further analyze GrUG, a hardware implementation with processor interaction is required. Because the processor architecture will have a large influence on the run time, we plan on evaluating architectures similar to both NVIDIA GPUs [NVI06] and Intel's Larrabee [SCS*08]. Both processor architectures are designed for graphics processing using a large number of multi-core processors and both seem well-suited for GrUG. Once a hardware implementation is accomplished, real-time performance can be measured and memory bandwidth optimization can be investigated.

7. Conclusions

Data structures have had a dramatic impact on the performance for ray-based rendering methods. To further increase performance of data structures, this paper proposed a method for reducing the number of recursive steps while still implementing common HS3 data structures. Our method of reducing the number of recursions in the data structure allows for an accelerated hardware implementation, further reducing the workload of the processor. We used an existing performance simulator for comparing our GrUG implemented kd-tree against a modern day kd-tree implementation. Our experimental results show a significant reduction in system memory reads for large polygon count scenes.

References

[AW87] Amanatides J., Woo A.: A fast voxel traversal algorithm for ray tracing. In *Proceedings of Eurographics* (1987), pp. 3–10.

[Ben08] Benjamin Segovia: Radius-CUDA. http://www710.univ-lyon1.fr/ bsegovia/demos/radius-cuda.zip, 2008.

[DBB02] Dutre P., Bekaert P., Bala K.: *Advanced Global Illumination*. AK Peters, Ltd., July 2002.

[DHK08] Dammertz H., Hanika J., Keller A.: Shallow bounding volume hierarchies for fast SIMD ray tracing of incoherent rays. *Computer Graphics Forum 27*, 4 (June 2008), 1225–1233.

[FR03] Fender J., Rose J.: A high-speed ray tracing engine built on a field-programmable system. In *Proceedings of Field-Programmable Technology* (2003), pp. 188–195.

[GDS*08] Govindaraju V., Djeu P., Sankaralingam K., Vernon M., Mark W.: Toward a multicore architecture for real-time ray-tracing. In *Proceedings of Microarchitecture* (2008).

[GTGB84] Goral C. M., Torrance K. E., Greenberg D. P., Battaile B.: Modeling the interaction of light between diffuse surfaces. *SIGGRAPH Computer Graphics 18*, 3 (1984), 213–222.

[Han07] Hanika J.: *Fixed Point Hardware Ray Tracing*. PhD thesis, Ulm, Germany, 2007. University-Universitat Ulm.

[Hav00] Havran V.: *Heuristic Ray Shooting Algorithms*. PhD thesis, Prague, Czech Republic, 2000. Czech Technical University in Prague.

[HSHH07] Horn D. R., Sugerman J., Houston M., Hanrahan P.: Interactive k-d tree GPU raytracing. In *Proceedings of Interactive 3D graphics and games* (2007), pp. 167–174.

[MF05] Mark W. R., Fussell D.: Real-time rendering systems in 2010. In *ACM SIGGRAPH 2005 Courses* (2005), p. 19.

[NVI06] NVIDIA: NVIDIA GeForce 8800 GPU Architecture Overview, 2006.

[NVI07] NVIDIA: NVIDIA CUDA Compute Unified Device Architecture Programming Guide 1.0, 2007.

[PGSS07] Popov S., Günther J., Seidel H.-P., Slusallek P.: Stackless kd-tree traversal for high performance GPU ray tracing. *Computer Graphics Forum 26*, 3 (2007), 415–424.

[SCS*08] Seiler L., Carmean D., Sprangle E., Forsyth T., Abrash M., Dubey P., Junkins S., Lake A., Sugerman J., Cavin R., Espasa R., Grochowski E., Juan T., Hanrahan P.: Larrabee: a many-core x86 architecture for visual computing. In *ACM SIGGRAPH* (2008), pp. 1–15.

[SKK*08] Spjut J., Kopta D., Kellis S., Boulos S., Brunvand E.: Trax: A multi-threaded architecture for real-time ray tracing. In *Proceedings of Application Specific Processors* (2008), pp. 108–114.

[SM03] Shirley P., Morley R. K.: *Realistic Ray Tracing*. A. K. Peters, Ltd., Natick, MA, USA, 2003.

[SWS02] SCHMITTLER J., WALD I., SLUSALLEK P.: SaarCOR: a hardware architecture for ray tracing. In *Proceedings of Graphics Hardware* (2002), pp. 27–36.

[SWW*04] SCHMITTLER J., WOOP S., WAGNER D., PAUL W. J., SLUSALLEK P.: Realtime ray tracing of dynamic scenes on an FPGA chip. In *Proceedings of Graphics Hardware* (2004), pp. 95–106.

[WBS06] WOOP S., BRUNVAND E., SLUSALLEK P.: Estimating performance of a ray-tracing ASIC design. *Symposium on Interactive Ray Tracing* (2006), 7–14.

[WBWS01] WALD I., BENTHIN C., WAGNER M., SLUSALLEK P.: Interactive rendering with coherent ray tracing. In *Proceedings of EUROGRAPHICS* (2001), pp. 153–164.

[Whi80] WHITTED T.: An improved illumination model for shaded display. *Communications of the ACM 23*, 6 (1980), 343–349.

[WIK*06] WALD I., IZE T., KENSLER A., KNOLL A., PARKER S. G.: Ray tracing animated scenes using coherent grid traversal. *ACM Transactions on Graphics 25*, 3 (2006), 485–493.

[WSS05] WOOP S., SCHMITTLER J., SLUSALLEK P.: RPU: a programmable ray processing unit for realtime ray tracing. *ACM Transactions on Graphics 24*, 3 (2005), 434–444.