POSTER COMPENDIUM

2006 IEEE SYMPOSIUM ON INTERACTIVE RAY TRACING

UNIVERSITY OF UTAH
SALT LAKE CITY, UTAH

18-20 SEPTEMBER 2006

SCI INSTITUTE
IEEE COMPUTER SOCIETY
IEEE vgtc
THE UNIVERSITY OF UTAH

Poster Compendium

The 2006 IEEE Symposium on Interactive Ray Tracing

University of Utah
Salt Lake City, Utah

18-20 September 2006

Credits:
Cover image rendered using the Galileo Renderer (http://sourceforge.net/projects/galileorenderer), courtesy of Solomon Boulos
Cover design by Nathan Galli

Table of Contents

III. Sampling

IV. Scientific and Engineering Applications

Ray Tracing on a Cell Processor with Software Caching

Jeremy Sugerman* Stanford University
Tim Foley† Stanford University
Shigeatsu Yoshioka‡ Sony Corporation
Pat Hanrahan§ Stanford University

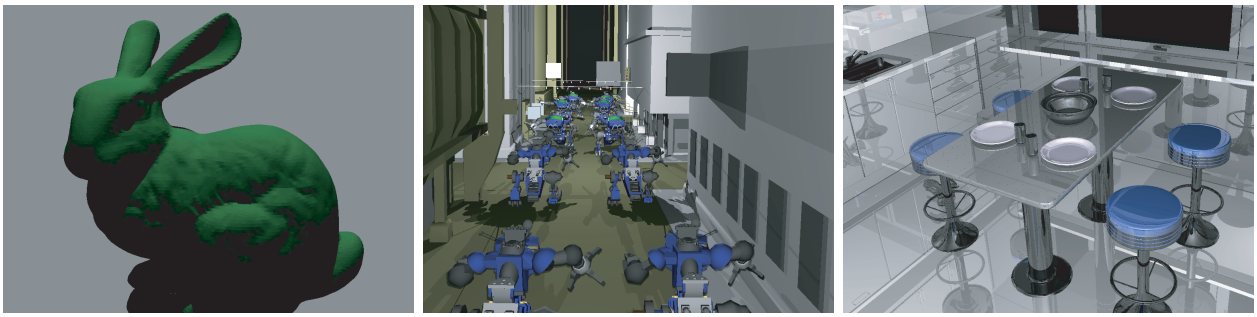


Figure 1: Scenes rendered using a Cell processor for ray-scene intersection tests. From left to right: the Stanford bunny and BART robots (with shadows), and the BART kitchen (with specular reflection).

Over the past few years, advances in both software techniques and hardware performance have spurred increased research into ray tracing images in interactive time. The combination of ray packets and microarchitectural tuning has established k-D tree based tracing as the de facto standard [7, 6] with some specialialized techniques going even further [5]. At the same time, a number of alternative architectures with compellingly high computational capacities have been evaluated for ray tracing ranging from GPUs [4] through custom hardware [8]. In this work, we investigate the performance of the Cell Broadband Engine [2, 1, 3]. We treat Cell as essentially an eight core symmetric multiprocessor with an extra core for coordination and initialization. We find this treatment appealing because it offers a simple model that still exploits the rich ray-level parallelism of ray tracing and, with a good implementation, still scales well across all cores. Our investigation focuses on the core task of ray-scene intersection and considers primary, shadow, and reflection rays.

The major challenge of the Cell architecture is its unusual memory system. Unlike conventional processors, the Cell's eight SPUs lack both hardware caches and even the ability to directly address system memory. Instead, programs must explicitly DMA any code and data they need into per-SPU local memory before accessing it. Our initial naive implementation issued a DMA and stalled on every access to k-D tree or scene data. It was a trivial port of our existing conventional implementation, but performance suffered from the frequent DMAs.

We then augmented our naive implementation to use straightforward software managed caches for k-D tree and triangle data. Surprisingly, this dramatically reduced the DMA performance penalty and kept the cost of tag checks plus cache hit to about 15 instructions per 32-ray packet. Even with the limited size of SPU local store, single-SPU execution improved more than a factor of 2.5 with high scalability up to 8 or 16 SPUs. We examined the complexity of our scenes and the impact of varying cache parameters to understand when software managed caching is most feasible and discovered that our most complex example, with almost 15MB of k-D tree nodes, performs well with a node cache that is only 32KB.

*e-mail:yoel@cs.stanford.edu
†e-mail:tfoley@cs.stanford.edu
‡e-mail:syoshiok@stanford.edu
§e-mail:hanrahan@cs.stanford.edu

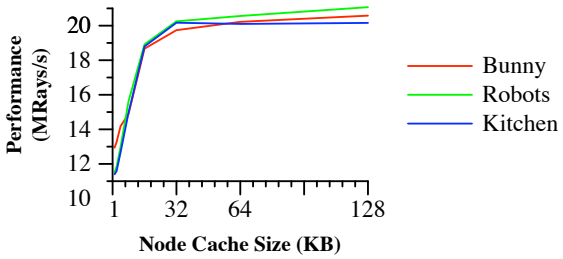


Figure 2: Performance as a function of node cache size when rendering on 8 SPUs.

References

[1] Brian Flachs et al. A streaming processing unit for a CELL processor. In *Proceedings of the IEEE International Solid-State Circuits Conference*, 2005.

[2] Dac Pham et al. The design and implementation of a first-generation CELL processor. In *Proceedings of the IEEE International Solid-State Circuits Conference*, 2005.

[3] Michael Gschwind, Peter Hofstee, Brian Flachs, Martin Hopkins, Yukio Watanabe, and Takeshi Yamazaki. A novel SIMD architecture for the CELL heterogeneous chip-multiprocessor. In *Hot Chips 17*, 2005.

[4] Tim Purcell. *Ray Tracing on a Stream Processor*. Ph.D. thesis, Stanford Unviersity, March 2004.

[5] Alexander Reshetov, Alexei Soupikov, and Jim Hurley. Multi-level ray tracing algorithm. *ACM Trans. Graph.*, 24(3):1176–1185, 2005.

[6] Gordon Stoll. Optimization techniques. In *Introduction to Real-Time Ray Tracing - SIGGRAPH 2005 Course 38*. 2005.

[7] Ingo Wald. *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Saarland University, 2004.

[8] Sven Woop, Jörg Schmittler, and Philipp Slusallek. RPU: a programmable ray processing unit for realtime ray tracing. *ACM Trans. Graph.*, 24(3):434–444, 2005.

Ray Tracing on Asynchronous Supercomputing Stream Processors

Gennette Gill and Montek Singh
Dept. of Computer Science
Univ. of North Carolina, Chapel Hill, NC 27599, USA
{gillg,montek}@cs.unc.edu

1 Introduction

Changes in modern semiconductor technology are making computation so inexpensive that individual chips will soon contain thousands of processing units (PUs). Asynchronous design techniques can make multiprocessor chips more efficient by allowing *idle processors to consume no energy*. Under the asynchronous supercomputing model, transistors—and therefore computing power and memories–are essentially free and idle processors are no longer wasteful.

This work re-explores the domain of ray tracing hardware under this new asynchronous supercomputing model. Since bandwidth is the limiting factor, an effective way to improve speed is to schedule processing tasks, or kernels, across PUs in a way that minimizes bandwidth. Two competing theories in the area of kernel scheduling are time-multiplexing and space-multiplexing. In this work, we present the performance tradeoffs of each of these theories. We then analyze current ray tracing hardware systems and show that they schedule kernels in a way equivalent to time-multiplexing. As an alternative, we offer a sketch of a space-multiplexed ray tracing system and discuss its benefits over time-multiplexed systems.

2 Time-Multiplexing vs. Space-Multiplexing

The speed of a stream processor is limited by several types of time overheads. *Kernel-switching overhead* is the time needed to change which kernel is assigned to a PU. *Data-transmission overhead* is the time spent sending data from one PU to another. *Stalling* occurs when a PU waits for data to become available.

Time Multiplexing A time-multiplexed system assigns different kernels to the same processing unit over time. For example, the Imagine system from Stanford [1] uses time-multiplexed kernel scheduling. Data produced by a PU is retained for use by the next kernel on the same PU, thereby eliminating both data-transmission overhead and stalling. However, task-switching overhead is incurred often. Therefore, time-multiplexing is most beneficial when the average task-switching overhead is low.

Space Multiplexing A space-multiplexed system assigns each kernel to a different PU [2]. Because each PU maintains the same kernel throughout operation, no task switching overhead is incurred. However, the data transmission overhead can be high, depending on the spatial arrangement of the kernels and the amount of data transmitted. Stalling can occur when a fast kernel must wait for data from a slower kernel. In addition, some of the PUs may receive a much higher workload than the others, which can further degrade performance. Space-multiplexing should be used only when the average task-switching overhead is high compared to these other overheads.

3 Application to Previous Work

One important kernel in ray tracing is the *intersection kernel*. In special-purpose ray tracing hardware, such as the SaarCOR family of RPUs [3], one kernel handles intersections with all of geometry in one leaf node of the BSP tree. Essentially, a *different* intersection kernel must exist to handle geometry for each leaf node. When viewed in this way, SaarCOR is a time-multiplexed system. A PU changes to a different intersection kernel each time ray (or packet of rays) arrives. Other GPU based solutions [4] also switch intersection kernels over time. Kernel-switching overhead in these time-multiplexed systems is very high because a large amount of geometry must be sent to the PU during each switch. As described in Section 2, a space-multiplexed system may be more efficient when kernel-switching overheads are high.

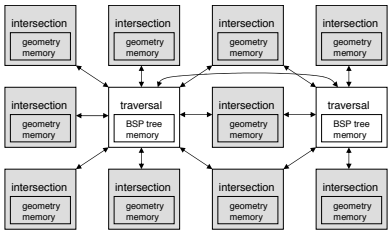


Figure 1: A kernel to PU mapping for a space-multiplexed ray tracer

4 Proposed Ray Tracing System

System Description Our proposed space-multiplexed ray tracer maps each intersection kernel to one PU. Each PU has a local memory which stores the entire set of geometry for that leaf node. Traversal kernels perform BSP tree traversals for arriving rays. Each PU that implements a traversal kernel has a local memory that stores the BSP tree information for the scene.

Figure 1 shows one way of mapping these kernels to PUs. Each box represents a PU, and each arrow represents the transmission of a ray between PUs. Traversal kernels act as routers: they send ray data to the appropriate intersection kernel or to another traversal kernel. For best performance, leaf nodes that are adjacent in the 3D scene should be mapped to PUs that are close to each other.

System Analysis Our approach reduces kernel-switching overhead by eliminating switching altogether. It minimizes the data-transmission overhead by sending only small amounts of data (rays) over short distances. Whereas time-multiplexing approaches have high bandwidth requirements between the memory and the PUs, our space-multiplexed approach only loads the geometry data to each PU *once* for each scene, greatly reducing required memory bandwidth.

Like all space-multiplexed systems, the proposed ray tracing system may have load imbalances. However, increased processing power prevents load imbalances from degrading performance as much as they did in the past. For example, consider the unfortunate case in which one PU handles all of the eye ray intersections. If the scene has 1 mega-triangles and there are 1024 PUs available for intersection in the system, each PU holds only 1 kilo-triangles. With this relatively small number of triangles, it is reasonable to expect one PU to produce acceptable frame rates (*i.e* 10-20 fps) for scene resolutions up to 1 mega pixel. The unused PUs will remain idle and waste no energy. In more typical situations, the ray intersection load will be more balanced, resulting in higher frame rates.

References

[1] W. J. Dally, U. J. Kapasi, B. Khailany, J. H. Ahn, and A. Das, "Stream processors: Progammability and efficiency," *Queue*, vol. 2, no. 1, pp. 52–62, 2004.

[2] M. Budiu, G. Venkataramani, T. Chelcea, and S. C. Goldstein, "Spatial computation." [Online]. Available: citeseer.ist.psu.edu/budiu04spatial.html

[3] J. S. Sven Woop and P. Slusallek, "Rpu: A programmable ray processing unit for realtime ray tracing," in *Proceedings of ACM SIGGRAPH 2005*, July 2005. [Online]. Available: http://www.saarcor.de/

[4] T. J. Purcell, "Ray tracing on a stream processor." [Online]. Available: citeseer.ist.psu.edu/purcell04ray.html

GPU-based rasterization-raytracing hybrid

Daniel Reiter Horn*
Stanford University

Mike Houston
Stanford University

Tim Foley
Stanford University

Pat Hanrahan
Stanford University

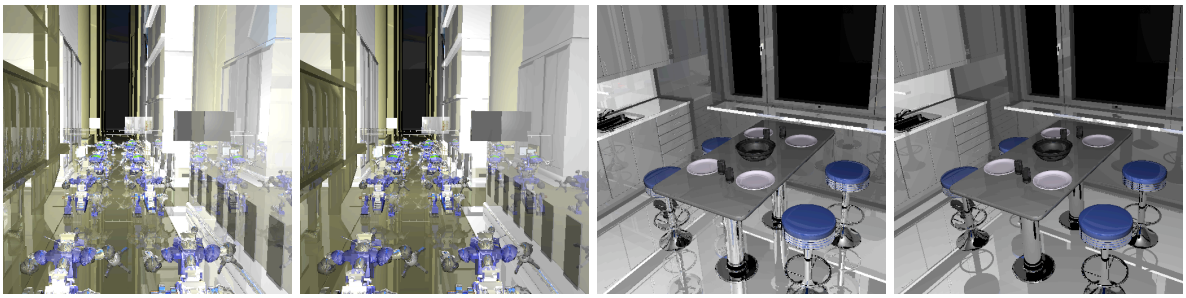


Figure 1: An ATI X1900XT generated the 512x512 images at 3.8, 7.0, 4.9, and 9.0 fps respectively, utilizing 2, 1, 2 and 1 bounces respectively.

With the advent of highly parallel programmable graphics processing units (GPUs) with a large number of floating point units and a custom memory system with a high peak bandwidth, many researchers have investigated the possibility of applying this abundant computation power to data-parallel problems like raytracing. Purcell et al.[4] wrote the first raytracer on a GPU to utilize an acceleration structure, in this case a uniform grid. Foley et al.[2] applied a kd-tree acceleration structure to GPU-raytracing, and he showed that on graphics hardware, there are scenes for which a kdtree yields far better performance. Given the lack of addressable temporary storage within a fragment shader, the fastest programmable unit on the GPU, Foley et al. had to invent two stackless kdtree traversal methods: kdtree-restart and kdtree-backtrack. Likewise Thrane and Simonsen[5] invented a fixed-order bounding-volume traversal method for ray intersection on the GPU to obviate the needs for a stack, and Carr et al.[1] used a similar BVH structure to create a raytracer suited to dynamic geometry.

We have created a raytracing-based graphics pipeline on GPUs using the Kdtree-Restart method presented by Foley et al. Our implementation is an order of magnitude faster than previous implementations of KDtree traversal on the GPU due to our use of 4-wide ray packets [6], looping instructions [3] and hardware-level commands on the ATI graphics hardware. Thus we get the advantage of tracing ray packets, while avoiding the cost of invoking the kdtree-traversal function many times as well as of OpenGL, and DirectX abstractions. Additionally we have a toggle to make use of the rasterization hardware on the board to perform primary ray intersection and produce first hits on the scene before raytracing all subsequent bounces.

Our system performs the entire rendering pipeline on the GPU itself. The CPU merely orchastrates memory movement, passes in scene and camera data, and invokes fragment shaders on that data. Since the GPU is highly suited to computing first-hit and shading steps of the raytracing pipeline, the entire system runs at interactive rates on a variety of scenes as shown in Table 1.

Rasterization hardware is efficient at drawing and clipping geometry, but it is difficult or impossible to use for reflections, refraction, and complex lighting. Raytracing provides a much better solution to complex optical effects, but it is currently not nearly as fast as the best rasterization hardware for primary rays. By combining rasterization and raytracing within a single application, we can leverage the benefits of rasterization and raytracing together.

Scene	Raster -ization	Eye Rays	1^{st} Bounce	2^{nd} Bounce	Shadow Rays
	pixels/s (millions)	rays/s (millions)	rays/s (millions)	rays/s (millions)	rays/s (millions)
Cornell Box	617.9	41.1	48.3	39.4	41.8
Platonic Solids	500.6	22.4	13.4	11.7	14.7
Stanford Bunny	156.1	5.9	4.6	3.4	2.7
Robots	119.4	6.5	4.0	3.6	4.6
Kitchen	95.9	6.7	6.5	4.3	8.3

Table 1: We ran our raytracer on a selection of standard scenes at a resolution of 1024x1024 utilizing an ATI x1900XT. We do not include shading time in the table, but for the bunny, robots and kitchen, shading takes less than 17% of the total time.

References

[1] Nathan A. Carr, Jared Hoberock, Keenan Crane, and John C. Hart. Fast gpu ray tracing of dynamic meshes using geometry images. In *Proceedings of Graphics Interface 2006*, Toronto, Ontario, Canada, 2006. Canadian Information Processing Society.

[2] Tim Foley and Jeremy Sugerman. Kd-tree acceleration structures for a gpu raytracer. In *HWWS '05: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 15–22, New York, NY, USA, 2005. ACM Press.

[3] Microsoft. Pixel shader model 3, 2003. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/directx9_c/Shader_Model_3.asp.

[4] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. *ACM Trans. Graph.*, pages 703–712, 2002.

[5] Niels Thrane and Lars Ole Simonsen. *A comparison of acceleration structures for GPU assisted ray tracing*. M.S. thesis, University of Aarhus, Aarhus, Denmark, August 2005.

[6] Ingo Wald, Philipp Slusallek, Carsten Benthin, and Markus Wagner. Interactive rendering with coherent ray tracing. *Computer Graphics Forum*, 20(3):153–164, 2001.

*e-mail:danielrh@graphics.stanford.edu

Offloading ray processing onto the GPU using cooperative worker threads

Stephan Reiter*

Johannes Kepler University of Linz

ABSTRACT

The processing power and flexibility of recent rasterization hardware permits to speed up image generation through raytracing. This paper introduces a method that performs ray processing on the GPU in parallel to shading operations on the CPU. It effectively exploits CPU-GPU parallelism in a way that can be integrated into existing implementations utilizing the standard recursive formulation of the problem.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing

Keywords: Cooperative threading, GPGPU, hardware acceleration, ray batching, ray tracing

The latest releases of graphics hardware offer new concepts, such as branching, looping and higher instruction count limits for shader programs, into the rasterization world that are crucial for using the GPU as a general purpose processor [3]. Yet, using the GPU as a coprocessor usually requires substantial modifications to traditional, CPU-only code.

Many standard implementations of raytracers employ the concept of materials for the shading of objects: The color of a given ray is computed by invoking the material program of the respective object, which has been hit by the ray. In addition, secondary rays can be traced for advanced effects such as shadows, reflections, or refractions [2]. This direct implementation of the recursive formulation of the raytracing algorithm is incompatible with GPU-based acceleration techniques, because only a single ray is processed at a given time and the involved trace-function is expected to return immediately. In order to use hardware accelerated raytracing, where batches of rays are processed on the GPU [1], a way has to be found to separate shading operations from ray processing.

To solve this problem, we propose the following software architecture: Each of the primary rays is processed in parallel by a set of cooperative threads. Upon calling a trace-function, the ray is inserted into the processing queue and the current thread is put on hold by passing control to the next "worker" in the set of threads. When the size of the ray queue exceeds a given threshold, a batch is issued to the GPU, which begins processing the contained rays asynchronously to the CPU and delivers the results in a texture. Once all cooperative threads are in waiting mode, the available results are retrieved from the GPU and the affected threads are woken up, where control flow resumes after the call to the trace-function. This way trace-calls still behave like in the traditional implementation, but multiple rays can be processed in parallel on the GPU. Therefore no changes to material programs are necessary, although additional speedup may be gained by passing an array of rays to the trace-function at once, e.g. for smooth shadows.

The poster will present an overview of the approach together with results in the form of pictures and benchmarks of a proof-of-concept implementation. It will also deliver images to explain and illustrate the control flow in detail for a sample scene.

*e-mail: stephan.reiter@students.jku.at

REFERENCES

[1] Nathan A. Carr, Jesse D. Hall, and John C. Hart. The ray engine. In *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 37–46, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[2] Andrew S. Glassner, editor. *An introduction to ray tracing*. Academic Press Ltd., London, UK, UK, 1989.

[3] Aaron Lefohn, Ian Buck, John D. Owens, and Robert Strzodka. Gpgpu: General-purpose computation on graphics processors. In *Tutorial #3 at IEEE Visualization*, October 2004.

Interactive Ray Tracing on Modern Graphics Hardware

Andrew Adinetz

Moscow State University

Interactive ray tracing has been a topic of ongoing research for some time. With the growth of the computational power available on commodity PCs interactive ray tracing has become possible not only on expensive supercomputers, but also on commodity machines.

Most of the effort in interactive ray tracing research, however, has been concentrated on doing ray tracing on contemporary SIMD CPU architectures. With the emergence of GPGPU, however, graphics processors have also become target platforms for interactive ray tracing. As GPUs are highly parallel and have been inherently designed for accelerating graphics computations, they are likely to be a good choice platform for interactive ray tracing. And as GPU computational powers are considerably higher than that of typical CPUs, the former become even more attractive platform.

In practice, however, GPU ray tracing turned out not as simple. First GPUs did not support a wide range of control flow instructions (e.g., loops and branches) crucial for implementing ray tracing. These instructions, therefore, had to be modeled, and the entire ray tracing algorithm to be implemented in a streaming programming paradigm. This involved breaking ray tracing in a number of kernels (traversal, intersection etc.) and implementing each in a separate shader and storing intermediate data in textures.

While this implementation allowed performing ray tracing, it introduced a number of bottlenecks, the most significant being the need to read and write the data to and from the textures each pass. As the number of passes typically amounted to thousands, this need created huge video memory bandwidth, which slowed the computations considerably even on modern graphics boards with fast video memory bus.

With the emergence of PS 3.0 graphics hardware, looping and branching instructions needed to perform ray tracing in a single pass have become available. Our approach, therefore, takes advantage of such instructions and performs grid-based ray tracing in a single pass in a single shader instead of involving multiple passes. The algorithm used and the ray tracing code, in fact, resemble much a code written for a CPU ray tracer, with the exception of using vector instructions available on GPUs instead of scalar ones.

Single-pass ray tracing allowed for elimination of memory bottleneck, as the test results have shown, rather moving the bottleneck to the computations. This one is much more tolerable, as the number of graphics pipelines and the computational power of GPUs increase faster than memory bandwidth. On a set of test scenes ranging from 10 to 50 thousand triangles, at 512x512 resolution, 7 FPS has been achieved, with the number of rays per second of about 3-4 million on ATI X1800 GPU.

We're currently working on incorporating BSP subdivision and coherent ray tracing in our framework.

Packet-Primitive Intersection Method

Kazuhiko Komatsu * Yoshiyuki Kaeriyama * Daichi Zaitsu * Kenichi Suzuki * Nobuyuki Ohba †
Tadao Nakamura *

Graduate School of Information Sciences, Tohoku University *
IBM Research, Tokyo Research Laboratory IBM Japan, Ltd. †

Basic ray tracing algorithms were founded in 1980s, and since then many implementation methods have been proposed so far. To accelerate the ray tracing operations, three major approaches have recently been used. The first one is to use a comprehensive data structure. This drastically reduces the number of ray-primitive intersection tests, which dominates the speed of ray tracing. Above many proposed data structures, the kd-tree is most widely used now [2] [3]. The second one is to group two or more rays together into ray packets to make use of the coherency of the adjacent rays. This also raises the efficiency of the SIMD engines in the recent CPUs and GPUs [3]. The third one is to use the frustum traversal [2]. The rays with spatial coherency in a ray group traverse a kd-tree at once. The combination of these three methods dramatically boosts the speed of the ray tracing processing.

In this paper, we propose an intersection method using ray packets. This method reduces the number of ray-primitive intersection computations by using the ray planes in a packet.

After a ray traversed the kd-tree and reached a leaf, an intersection test is performed to see if the ray actually intersects the primitives in the leaf. When every ray in the ray packet has found its nearest primitive, the test proceeds with the next ray packet. If at least one ray cannot find the nearest primitive, the ray packet continues to traverse the kd-tree and iterates intersection tests. In conventional methods, the intersection tests are sequentially performed for each ray in the ray packet. To shorten the time for each intersection test, the kd-tree is inclined to be deep, and thus each leaf has only a few primitives. A deep tree makes the traversing time longer and requires a large data space. In addition, it takes long time to build the tree itself. For these reasons, a deep tree is not suited for dynamic scene generation.

Our proposed method is based on the 'Ray Planes' intersection method [1]. Ray Planes including an origin are formed by each row and column of rays in a packet, as shown in Fig. 1. The intersection tests are done by using the bounding volumes involving primitives and the planes involving rays. When a pair of two orthogonal planes is found to intersect a bounding volume, intersection tests between the primitives in the bounding volume and the ray included by the two planes are performed. This method drastically decreases the number of intersection tests. Let us assume that the size of ray packets is $N \times N$. Conventional methods require $N \times N = N^2$ intersection tests per bounding volume. The proposed method, on the other hand, requires only $N + N = 2N$ intersection tests. Thus, the larger the sizes of ray packets are, the greater number of intersection tests the proposed method can save.

Another advantage of the proposed method is that it no longer requires a deep tree, because it requires fewer intersection tests. Leaves in the tree can have more primitives than a conventional method. As a result of decreasing the tree depth, the proposed method makes the traverse faster and reduces the data space for storing the tree structures.

*e-mail: {komatsu,yoshi,zaitsu,suzuki,nakamura}@archi.is.tohoku.ac.jp
†e-mail:ooba@jp.ibm.com

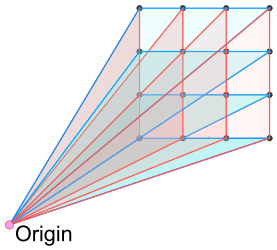


Figure 1: Ray Planes in a packet

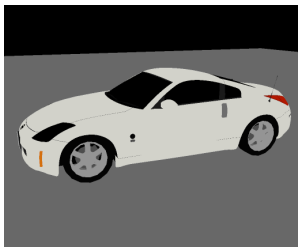


Figure 2: test scene (61828 tri.)

Table 1: Total execution time (sec.)

Packets	Depth of Kd-tree					
	16	18	20	22	24	26
1x1	27.77	8.32	4.43	2.84	2.16	1.98
2x2	123.64	56.54	23.02	10.49	7.86	7.11
4x4	21.51	8.32	4.83	7.92	2.81	2.67
8x8	5.16	2.87	1.97	1.65	1.53	1.59
16x16	2.21	1.60	1.34	1.29	1.35	1.52
32x32	1.64	1.44	1.28	1.35	1.53	1.89
64x64	2.02	1.81	1.58	1.68	2.10	3.02

We performed experiments on generating images of 1024×1024 pixels to evaluate our proposed method using a commodity PC with 3.4GHz Intel Pentium4 and 2GB memory. Fig. 2 is one of the test scenes for the evaluation. We measured the total execution time with varying the number of ray packets and the depth of the kd-tree. The result is shown in Table 1. For this scene, the fastest operation was achieved with a 32×32 packet size using a 20 deep kd-tree. The reasons are that the number of primitives in a leaf of the 20 deep kd-tree is suitable for reducing the intersection tests using our proposed method and the 32×32 packet size could efficiently utilize the coherency of the tree.

In conclusion, our proposed packet-primitive intersection method can reduce the depth of kd-tree and exploits the coherency of rays. As future work, we need a more detailed implementation with SSE and performance evaluations.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing;

Keywords: global illumination, ray tracing, ray packet, Kd-tree

REFERENCES

[1] Yoshiyuki Kaeriyama, Daichi Zaitsu, Kazuhiko Komatsu, Kenichi Suzuki, Nobuyuki Ohba, and Tadao Nakamura. Hardware for a ray tracing technique using plane-sphere intersections. In *Eurographics Symposium on Parallel Graphics and Visualization (Short)*, pages 9–12, May 2006.

[2] Alexander Reshetov, Alexei Soupikov, and Jim Hurley. Multi-level ray tracing algorithm. In *ACM Transaction of Graphics (Proceedings of ACM SIGGRAPH)*, volume 24, pages 1176–1185, 2005.

[3] Ingo Wald. *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Saarland University University, 2004.

Tessellating Planar Quadrilaterals into Triangles to Meet a Maximum-Edge-Length Threshold while Minimizing Grid Size

Peter Djeu* University of Texas at Austin
Gordon Stoll Intel Corporation
William R. Mark University of Texas at Austin

The ability to tessellate surfaces to the right granularity allows rendering systems to provide a variety of useful features, such as efficient rendering of curved surfaces, support for displacement mapping, and economical shading. We present an algorithm for quickly finding the appropriate tessellation rate for a planar quadrilateral which minimizes the number of triangles while ensuring that all edges in the tessellation are shorter than a given edge length. The algorithm makes use of the observation that determining the longest edge in a tessellated quadrilateral can be done in constant time by examining only a fixed number of critical edges along the border and at the corners of the quad.

*djeu@cs.utexas.edu

Interactive Iso-Surface Ray Tracing of Massive Volumetric Datasets

Heiko Friedrich* Saarland University
Ingo Wald† University of Utah
Johannes Günther‡ MPI Informatik
Gerd Marmitt* Saarland University
Philipp Slusallek* Saarland University

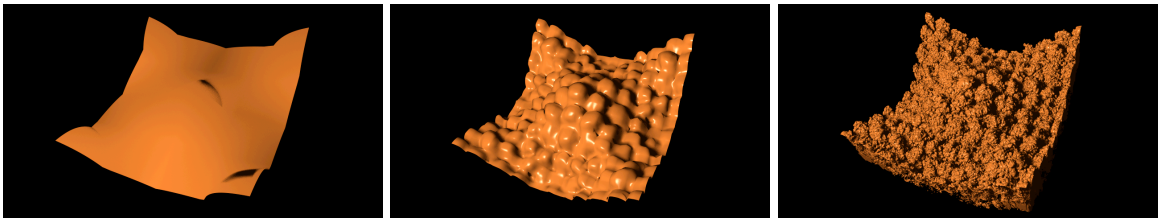


Figure 1: Three images of the Lawrence-Livermore National Laboratory (LLNL) simulation of a Richtmyer-Meshkov instability at different LODs. While data are loaded asynchronously in the background it is possible to fully interact with the scene. A kd-tree based LOD structure is used to bridge-over loading times while allowing interactive ray tracing at several frame per second on a custom PC.

1 Introduction

The interactive visualization of iso-surfaces from volume data is an important tool in many visualization applications. We propose a kd-tree based data structure that allows to ray trace iso-surfaces of large volumetric data sets in the giga-bytes range at interactive frame rates on a single PC with short loading times and the possibility to interact with the scene while not all data are present. To do so, a LOD technique is used to bridge-over loading times of data that are fetched asynchronously in the background.

2 Method Overview

In our approach we combine some previous existing techniques with the goal to get a highly flexible and fast rendering system that only requires commodity PC hardware to allow interactive iso-surface ray tracing of large data sets with short loading times. We first create a LOD hierarchy of the volume. This LOD data is solely used for rendering while not all data from the lowest level is present. Then for each LOD level we build an implicit min/max kd-tree and merge them together such that we obtain a single kd-tree which is valid for all LOD levels. This kd-tree and the LOD data are then decomposed into *treelets* (see Figure 2) and saved together in a *page-based* data structure. During rendering, a background thread

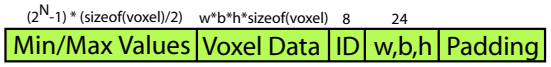


Figure 2: The structure and memory requirements of a treelet. Min/max values, voxel data, an ID, and the dimensions of the voxel data are grouped together. If the size of a treelet is small enough multiple treelet structures can be placed in one page on the hard disc. Width w, breadth b, and height h include an outer ring of voxels in order to calculate the central difference for shading.

loads in a breadth-first-search (bfs) order all relevant treelets from the hard disc that are required for rendering the desired iso-surfaces. Whenever a new LOD level is completely loaded, the render threads are notified such that they can use a finer LOD for the next frame.

*e-mail:{friedrich,marmitt,slusallek}@graphics.cs.uni-sb.de
†e-mail:wald@sci.utah.edu
‡e-mail:guenther@mpi-inf.mpg.de

3 Results

In order to evaluate the efficiency of our out-of-core data structure we measured performance numbers for two data sets: Time-step 270 of the LLNL Richtmyer-Meshkov instability simulation with a voxel resolution of $2048^2 \times 1920$, and a synthetic data set Attractor with 2048^3 voxel resolution (see Figure 3). The test system is a dual-core Opteron 880 PC with 32 GB RAM. The overall rendering performance is between 1.8 and 3.5 fps for the finest level in our hierarchy using two render threads and a simple diffuse shading. Furthermore, the required in-core memory is reasonable with approximately between 2.1 GB (Attractor) and 6.1 GB (LLNL), especially if we consider that we use quantized min/max values in the treelets.

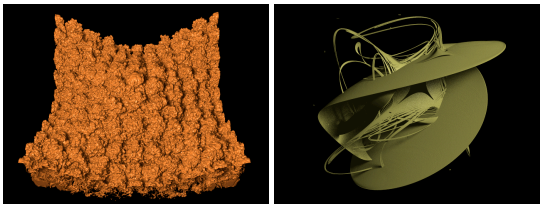


Figure 3: Example images of our test scenes: LLNL and Attractor rendered with progressive soft-shadows and phong shading between 1.2 and 1.4 fps at 640×480 image resolution.

In comparison to the approach of DeMarle et al. [1] we achieve almost the same rendering performance using only a single PC rather then a 32 PC cluster setup. This reduces the hardware requirements significantly. Additionaly we achieve a loading time which is up to a factor of three faster compared to Wald et al's. [2] and a rendering performance that is twice as fast.

References

[1] David E. DeMarle, Steve Parker, Mark Hartner, Christiaan Gribble, and Charles Hansen. Distributed Interactive Ray Tracing for Large Volume Visualization. In *Proceedings of the IEEE Symposium on Parallel and Large-Data Visualization and Graphics (PVG)*, pages 87–94, 2003.

[2] Ingo Wald, Heiko Friedrich, Gerd Marmitt, Philipp Slusallek, and Hans-Peter Seidel. Faster Isosurface Ray Tracing using Implicit KD-Trees. *IEEE Transactions on Visualization and Computer Graphics*, 11(5):562–573, 2005.

Fast ray tracing with Intel® IPP

Alexei Leonenko* Sergey Perepelkin†

Intel Corporation

ABSTRACT

Recent advances in interactive global illumination are mainly based on efficient spatial indexing with k-d trees and optimized tracing of ray packets [3, 2]. As been essentially primitive and common these two operations are suitable for low-level optimization and so IPP team decided to extend functionality of Intel® IPP [1] library and to develop **Realistic Rendering** domain with ray-tracing support functions. In this report we will provide a brief overview of a planned API and will give some preliminary performance results for the reference version of k-d tree construction primitive.

Keywords: kd-trees, ray tracing, low-level API, optimization

API DESCRIPTION

Currently there are four distinctive groups of functions in RR domain: shader and auxiliary support functions, core ray-scene intersection functions and acceleration structures handling functions. Auxiliary support functions include various functions for linear interpolation of normals, texture coordinates or arbitrary vertex properties using barycentric coordinates, fast boundary box evaluation, and explicit intersection coordinates evaluation. Shader support functions are basically limited to various masked arithmetic operations on vectors of FP32, several functions for dot product evaluation and ray casting primitives. In the core of RR domain lays three optimized ray-triangles intersection primitives: *ipprIntersectMO*, *ipprIntersectEyeSO* and *ipprIntersectAnySO*.

These primitives answer on typical visibility queries. For given block of rays defined by origin coordinates *O* and directions *D* function *ipprIntersectMO(O[3], D[3], Dist, UV[2], tId, Ctx, Sz)* search for nearest intersection with group of triangles defined by context *Ctx*. Output *Dist* parameter specifies distance along each ray between origin and hit position, *UV* specify hit coordinates inside triangle *tId*. For rays which do not intersect any scene geometry corresponding *tId* is set to -1, serving as mask for consequent operations. For primary rays function *ipprIntersectEyeSO(o, D[3], Dist, UV[2], tId, Ctx, Sz)* which traces a block of rays from a single origin *o* should be used instead of generic one. To answer on simple visibility queries, like tracing shadow rays, function *ipprIntersectAnySO(o, D[3], Occl, Mask, Ctx, Sz)* should be used. It traces block of rays from a single origin *o* to specified directions looking for any intersections with scene. For further operations logical result of this test is written to *Mask* parameter, whereas *Occl* parameter is used to speedup the intersection. All three intersection functions operates on similar scene context, which includes scene bounding box, internal acceleration structure and k-d tree. Rebuild of internal acceleration structure with help of *ipprTriangleAccelInit* is rather simple procedure and can be performed for moderate scenes in real-time.

*e-mail: alexei.leonenko@intel.com

†e-mail:sergey.perepelkin@intel.com

[1] Intel® Integrated Performance Primitives (Intel® IPP) is a trademark or registered trademark of Intel Corporation or its subsidiaries in the United States or other countries

model	#tris	time	SAH	N_L	E_T	E_L	E_I
bunny	69k	1.7s	966	270k	55.5	16.2	6.6
armad.	345k	4.8s	866	504k	52.1	15.0	4.2
dragon	871k	12.5s	1356	1.43m	79.8	22.7	7.9
buddha	1.1m	16.2s	1490	1.83m	87.2	24.7	9.1
blade	1.7m	17.8s	1764	2.22m	105.3	29.7	9.2
thai	10m	256s	1354	22.3m	75.3	21.6	11.2
erw6	804	16ms	284	3.15k	13.2	4.24	4.3
conf.	282k	7.5s	851	1.04m	43.2	11.9	10.1

Table 1: Reference kd-tree construction performance.

While user may provide its own k-d trees, RR domain includes special function for tree construction. It implements variant of suboptimal $O(nlog^2n)$ SAH construction algorithm with optional fine-grain threading. Table 1 summarizes performance of the reference (ANSI C, algorithmic optimizations only) builder implementation on server with Intel® Xeon 3.6GHz processor. For detailed description of statistical measures refer to [4]. Note though what in Table 1 $SAH = 15 * E_T + 20 * E_I$. User may control tree building algorithm explicitly specifying maximum tree depth or by implicitly changing termination criterion with help of QoS parameter ranging from 0.0 (fastest) to 1.0 (deepest). Our experiments demonstrated that along with early empty space cutoff heuristic [2] it is useful to prioritize boundary singularities cutoff by 70%.

CONCLUSION AND OPEN QUESTIONS

We believe what it is possible to achieve interactive rendering rates using Intel® IPP implementation of core real-time ray tracing operations. But we understand what usage model for current API is limited by plain scene representation, sparse shader support and lack of dynamics. Unfortunately, absence of industry level standards in the area of realtime ray tracing complicates low-level development and many features available in in-house software remains relatively inaccessible for general public. Several steps in this direction were taken with recent OpenRT introduction [1]. Another important issue is lack of universal benchmarks. It is really hard to compare different approaches and choose the best one without uniform and widely supported performance indicators.

Overcoming these issues along with further algorithmic improvements is a key to prosperous future of real-time ray tracing.

REFERENCES

[1] Andreas Dietrich, Ingo Wald, Carsten Benthin, and Philipp Slusallek. The OpenRT Application Programming Interface - Towards a Common API for Interactive Ray Tracing. In *Proceedings of the 2003 OpenSG Symposium*, 2003.

[2] Alexander Reshetov, Alexei Soupikov, and Jim Hurley. Multi-level ray tracing algorithm. In *Proceedings of the ACM SIGGRAPH*, pages 1176–1185, 2005.

[3] Ingo Wald. Realtime Ray Tracing and Interactive Global Illumination. *PhD thesis, Saarland University*, 2004.

[4] Ingo Wald and Vlastimil Havran. On building fast kd-trees for ray tracing, and on doing that in O(N log N). In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, 2006.

Realtime Ray Tracing of Animated Meshes using Fuzzy KD-Trees

Heiko Friedrich* Saarland University
Johannes Günther† MPI Informatik
Ingo Wald‡ University of Utah
Hans-Peter Seidel† MPI Informatik
Philipp Slusallek* Saarland University

Figure 1: Several dynamic example scenes ray traced between 4 (including shadows) and 14 frames per second (with a simple shader) on a single CPU and 1024^2 resolution. The first two images show predefined animations while the last three images demonstrate interactively skinned meshes.

1 Introduction

Kd-trees have been proven to be an efficient spatial index structure to speed up the computation of the closest hit of a ray with a scene. However, due to its costly preprocessing, ray tracing applications using kd-trees have been quasi limited to static scenes. The only exceptions are until today affine transformations which can be handled by a two-level kd-tree [2].

2 Fuzzy KD-Trees

Fuzzy kd-trees allow for ray tracing animated objects without the necessity to rebuild the spatial index of animated objects. To do so, fuzzy kd-trees are not directly built over the time dependent scene primitives p_i, e.g. triangles, but over their corresponding axis aligned *fuzzy boxes* $\mathscr{FB}(p_i)$. Fuzzy boxes bound the space where p_i can be located during the animation. Simply using this strategy for a complete animated object would of course yield huge overlapping fuzzy boxes, and thus a disastrous ray tracing performance. The key is to split the animated object into several sub-meshes of *coherent motion*, find an affine transformation for each sub-mesh approximating this coherent motion, *subtract* this transformation, and finally bound the residual motion by fuzzy boxes. Because this residual motion is usually much smaller the overlap of the fuzzy boxes $\mathscr{FB}(p_i)$ in the transformed space is minimized.

In the following sections we propose two methods to decompose the deformed mesh into sub-meshes and to find the corresponding affine transformations for each animation step.

2.1 Motion Clustering for Predefined Animations

In [1] we propose a novel clustering algorithm which identifies coherent motion in an animated mesh. The resulting sub-meshes exhibit small residual motion and thus the fuzzy boxes of the clusters are minimized. This clustering – based on a generalized Lloyd relaxation – is started with one cluster and we iteratively add new clusters until the overall residual motion does not decrease significantly anymore. A new cluster is seeded by the triangle with the largest residual motion. During the relaxation of the current clusters we alternately find optimal transformations for each cluster and reassign the triangles to the cluster where their residual motion is smallest. The needed affine transformations are determined by solving a linear least squares problem that minimizes the L^2-norm of the residual motion using the positions of the vertices contained in the respective cluster.

To be applicable, this method requires that all animation poses are known in advance. However, only the positions of the vertices are needed and no additional information such as modeling or pose parameters are necessary.

2.2 Bone Model Evaluation for Skinned Meshes

An extension to this approach is based on leveraging additional information of the animated mesh. For skinned meshes we use the present skeleton to avoid the costly clustering. Vertices are simply assigned to the bone with the largest influence resulting in coherently moving clusters. Additionally, we can get the affine transformations directly from the current pose of the bones. In order to bound the residual motion for the fuzzy boxes we sample the space of all possible poses.

This extensions not only allow us to significantly accelerate and simplify the clustering process, but also to interactively animate the skinned mesh. This means that it is possible to interpolate and blend different animation cycles.

3 Results and Conclusions

With our implementation we achieve rendering frame rates up to 14 frames per second on a single 2.4 GHz Opteron processor. As measurements show, the performance penalty for realistic datasets of our fuzzy kd-tree compared to optimized static kd-trees (for every frame) is with a factor of two to three reasonable low. However, avoiding the high per-frame rebuild costs of static kd-trees we are able to ray trace dynamic scenes interactively.

References

[1] Johannes Günther, Heiko Friedrich, Ingo Wald, and Philipp Slusallek. Ray Tracing Animated Scenes using Motion Decomposition. In *Proceedings of Eurographics*, 2006.

[2] Ingo Wald. *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Computer Graphics Group, Saarland University, 2004.

*e-mail: {friedrich,slusallek}@graphics.cs.uni-sb.de
†e-mail: {guenther,hpseidel}@mpi-inf.mpg.de
‡e-mail: wald@sci.utah.edu

Towards Triple Product Sampling in Direct Lighting

David Cline
Brigham Young University

Parris K. Egbert
Brigham Young University

Kenric B. White

1 Introduction

State of the art sampling methods for direct lighting generate samples according to the product of the incident light and BRDF [1, 2, 3]. None of these methods includes a visibility term as part of their sampling distributions, however. This work presents several techniques that extend existing product samplers to include an approximate shadow term as either a pre- or post-processing step.

2 Adding a Visibility Term to Product Sampling

2.1 Visiblity in Preimage Space

Most product sampling algorithms for direct lighting begin with a set of uniformly-distributed points in $[0,1)^2$ which are transformed to the product distribution. Ray samples are then created from the transformed points, and light source visibility is tested for these rays. Conceptually, however, we can replace the shadow rays with visibility maps defined over $[0,1)^2$. We call this kind of a visibility map a *preimage visibility map* because it is defined over the domain of input numbers rather than the range of output directions. The benefit of defining visibility maps in this way is that occluded points can be culled before they go through an expensive transformation.

In practice, it is not feasible to produce an accurate visibility map for each primary ray in a ray tracer, but it is possible to create approximate visibility maps at a sparse grid of locations over the image plane. We keep the size of the visibility maps on the order of the pixel spacing between them so that the total cost of creating the maps is only a few rays per pixel.

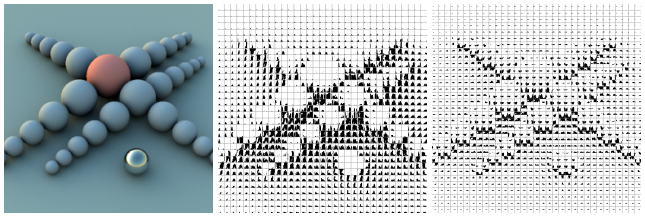
Our preimage space triple product sampler proceeds as follows: First, we generate a number of uniform points in $[0,1)^2$. Next, we discard points that are occluded in all nearby visibility maps. The remaining points are then transformed to the product distribution (BRDF $\times$ incident light). Visibility is assumed for samples not occluded in any nearby visibility map, and shadow rays are cast to determine visibility for uncertain samples.

2.2 Visibility in World Space

A second variant of our algorithm relies on visibility maps defined in world space instead of preimage space. The second algorithm works as a post-process to product sampling rather than a preprocess. The algorithm proceeds as follows: first, uniform points are sent through a product sampler as usual. The resulting samples are then checked against nearby world space visibility maps in the map grid. Once again, samples in the occluded regions of the maps are discarded, samples within visible regions are assumed to be visible, and samples within uncertain regions must cast shadow rays to determine visibility. Figure 1 shows an the preimage and world space visibility maps created for an example scene.

2.3 Unbiased Rendering

Since the visibility maps are never completely accurate, the triple product samplers just described are biased. We can make them unbiased by treating the visibility maps as a visibility guide rather than absolute truth. Instead of eliminating all samples in regions deemed to be occluded, we only cull some of them (by Russian roulette), and reallocate the culled samples to regions thought to be visible, increasing the sample density there. Final visibility must be calculated for all samples.



(a) (b) (c)

Figure 1: (a) An example scene with (b) the preimage visibility maps and (c) the world space visibility maps generated for it.

3 Example

Figure 2 compares a standard product sampling algorithm to a biased and unbiased version of our preimage space triple product samplers. The scene consists of a bunny enclosed in box with a small hole in the top to let light through. Most of the light from the environment map light source (grace cathedral) is blocked, so that the product distribution does not form a good importance function for the scene. By providing an approximate shadow term, our triple product samplers produce much smoother results in the penumbra regions of the image.

(a) (b) (c)

Figure 2: "Bunny in a box" scene tone mapped to bring out details in dark regions of the image. (a) Two stage importance sampling, (b) biased triple product sampling and (c) unbiased triple product sampling.

References

[1] Petrik Clarberg, Wojciech Jarosz, Tomas Akenine-Möller, and Henrik Wann Jensen. Wavelet importance sampling: efficiently evaluating products of complex functions. *ACM Trans. Graph.*, 24(3):1166–1175, 2005.

[2] David Cline, Parris K. Egbert, Justin F. Talbot, and David L. Cardon. Two stage importance sampling for direct lighting. In T. Akienne-Möller and W. Heidrich, editors, *Rendering Techniques 2006 Eurographics Symposium on Rendering*, pages 103–113, Aire-la-Ville, Switzerland, 2006. Eurographics Association.

[3] Justin F. Talbot, David Cline, and Parris K. Egbert. Importance resampling for global illumination. In Kavita Bala and Philip Dutré, editors, *Rendering Techniques 2005 Eurographics Symposium on Rendering*, pages 139–146, Aire-la-Ville, Switzerland, 2005. Eurographics Association.

Evaluating Multidimensional Samples Using 2-D Projections

Dave Edwards Solomon Boulos Peter Shirley Ingo Wald

School of Computing, University of Utah
{edwards|boulos|shirley|wald}@cs.utah.edu

In theory, Monte Carlo rendering is an infinite-dimensional integration problem; in practice, the rendering equation is evaluated over a finite-dimensional space. In order to efficiently produce high-quality rendered images, certain dimensions must be sampled more carefully than others. In this paper, we present an algorithm for ordering the dimensions of a multi-dimensional sample set by applying quality metrics to two-dimensional projections of the sample points. Our method orders the dimensions from highest to lowest quality; as a result, important dimensions in the rendering integral can be sampled using the earlier dimensions in the ordering to efficiently produce high quality renderings.

Improved Adaptive Frameless Rendering Using Edge Respecting Filters

Ewen Cheslack-Postava* University of Virginia
Abhinav Dayal† Northwestern University
Abe Stephens‡ University of Utah
David Luebke§ NVIDIA Corporation
Ben Watson¶ North Carolina State University

ABSTRACT

Multiple techniques have been proposed to exploit temporal coherence of samples for real time ray tracing [4, 5]. These techniques store samples, reproject them for the current frame, and guide sampling based on the sparsity and age of samples. A common problem with these techniques is that, as the sampling rate is reduced edges become more difficult to resolve and become blurred or, even worse, misshapen.

Bala et al. [1] propose a solution to this problem by explicitly calculating where edges occur in each frame and construct an edge and point image. Samples are only used in the reconstruction of a pixel if they can be reached without crossing an edge. This edge respecting reconstruction makes a striking visual difference and allows for sparse sampling which is necessary if we wish to include expensive shading effects such as global illumination.

We propose an extension and modification of edge and point images and combine that technique with Adaptive Frameless Rendering (AFR) [4] to achieve high quality, ray traced, animations at very low sampling rates. At the core of our technique is a new primitive, the space time plane, which represents edges in the image-time volume.

First we shortly review the AFR method. Because AFR is frameless samples are taken continuously, without regard for frames. AFR is adaptive in sampling by focusing samples in regions of high spatial or temporal variation. It is also adaptive in reconstruction by varying the reconstruction filter size based both on the sample density in the region and the variation detected in each dimension. We find this technique to be successful in many cases, but it has difficulty, even with the adaptive sampling, resolving edges well at low sampling densities. Our new technique specifically targets this issue.

Next we see how edge point images extend to animations. Consider the concept of a video cube [3]. A video cube is simply all the data contained in an animation set up in the expected form - two image dimensions and one time dimension. To generate frames of the animation slices are taken from this volume. If we consider the edges used in edge and point images in the context of the video cube it is simple to see that edges carve out surfaces in the volume. These surfaces can open and close, for example due to occlusions events, intersect, merge and split.

We note an important feature of the surfaces formed by edges: they are always closed surfaces (possibly using the edges of the video cube). Because of this feature the surfaces form a segmentation of the video cube. We can use these surfaces to maintain this segmentation. This segmentation can then in turn be used to tag both samples and pixels with a single region. Using these tags we can efficiently discard samples which should not be used in reconstructing a pixel (i.e. only samples with tags matching the tag of the pixel we are reconstructing will contribute).

Our system uses the ray tracer to detect these surfaces instead of calculating them explicitly as in [1]. Currently we detect these surfaces by clamping a fraction of samples to an edge of the triangle they hit and sample on either side. If the samples differ significantly in color then a short time later the new location of the edge is combined with the old position to generate a space time plane. This space time plane represents a tangent to the surface and is reported in the sample stream.

With a set of space time planes in hand we use a level set based segmentation scheme to generate the regions. This segmentation scheme is closely related to [2], but our cost function is based on our space time planes instead of on image colors. Tagging samples and pixels using this segmentation is trivial, as is adding tag checking to the reconstruction step.

Note that the cost has not increased significantly over traditional AFR. Sampling remains largely the same, some samples are simply shifted to be space time planes instead of normal samples. Also, edge detection is not dependent on the amount of geometry in the scene, as it is for Bala et al. [1]. The most expensive step we have added is the segmentation. However, level set techniques are highly parallelizable and could be performed on the GPU, so this is not a significant weakness. The cost of the final reconstruction has remained the same since the only addition there is the tag check.

Our initial results are very promising. We see a significant reduction in RMS error compared to traditional AFR. We are focusing on improving both space time plane sampling and the segmentation technique. Space time plane sampling currently only finds edges formed by silhouette edges. We are considering techniques both to detect arbitrary edges in images as well as specific types of edges, such as shadows and edges in textures. Finally, we are also investigating ways to reproject space time planes and hope doing so will allow for fewer space time plane samples and will improve reconstruction.

*e-mail: elc5d@cs.virginia.edu
†e-mail: abhinav@cs.northwestern.edu
‡e-mail: abe@sci.utah.edu
§e-mail: dave@luebke.us
¶e-mail: bwatson@ncsu.edu

REFERENCES

[1] Kavita Bala, Bruce Walter, and Donald P. Greenberg. Combining edges and points for interactive high-quality rendering. *ACM Trans. Graph.*, 22(3):631–640, 2003.

[2] T. Brox and J. Weickert. Level set based image segmentation with multiple regions. pages 415–423, 2004.

[3] Michael F. Cohen, Alex Colburn, Adam Finkelstein, Allison W. Klein, and Peter-Pike J. Sloan. Video cubism. Technical report, Microsoft Research, 2001.

[4] Abhinav Dayal, Cliff Woolley, Benjamin Watson, and David Luebke. Adaptive frameless rendering. *Eurographics Symposium on Rendering*, pages 265–275, 2005.

[5] Bruce Walter, George Drettakis, and Steven Parker. Interactive rendering using the render cache. In D. Lischinski and G.W. Larson, editors, *Rendering techniques '99 (Proceedings of the 10th Eurographics Workshop on Rendering)*, volume 10, pages 235–246, New York, NY, Jun 1999. Springer-Verlag/Wien.

BRL-CAD: Ray Tracing for Scientific and Engineering Applications

Lee A. Butler

US Army Research Laboratory

The BRL-CAD package boasts one of the first distributed, parallel and vectorized ray tracers. The package has been in active development since 1985 and has been ported to most POSIX-like systems in existence in the last 20 years. In December, 2004, it became an open source project. Today the system is a hybrid CSG and BREP modeling environment, supporting a wide variety of implicit and explicit geometric representations. Its geometry engine and ray tracing capabilities are used by numerous software packages in the global defense community to support engineering analysis and simulations.

Typical BRL-CAD models are vehicles with geometric detail down to the individual wire, nut and bolt. This level of detail can also be extended to the surrounding environment. Individual blades of grass, leaves on trees, road surfaces and buildings in vast tracts of terrain have been modeled.

Unlike many optical rendering systems which typically stop and shade at the first ray/object intersection, most BRL-CAD based applications pass rays through the entire depth of the scene. These applications trace rays for a variety of purposes including: multispectral and optical rendering, ballistic analysis, nuclear transport, presented/exposed area computation, mass/weight/volume calculation, and interference/overlap detection.

In the last five years, the package has been adopting recently developed algorithmic enhancements in ray tracing including cache coherency, packet tracing and performance oriented space partitioning. This has delivered substantial improvement in key analysis applications.

Expressing Blast Sensitivity Envelopes with Implicit Surfaces

Erik Greenwald

US Army Research Laboratory

Scientists and engineers in the field of blast mechanics express sensitivity envelopes of objects in terms of a center of damage and a Lethal miss distance. Currently engineers produce 2D drawings and extrapolate an approximation of the shape back into 3D for analysis purposes. Isosurfaces are blended together (usually by hand). Interest has been expressed by the DoD community in providing a mechanism and tools to describe, visualize, and interrogate these envelopes via software.

The concept of center of damage and lethal miss distance seems to map directly to the isosurfaces produced by blobs or metaballs. This would allow the scientist/engineer a familiar notation for expressing blast envelopes. It also allows simulations to directly query the envelope as specified.

The US Army Research Laboratory is implementing metaball capability in the BRL- CAD open source modeling package to support the blast modeling community.

A Coherent Grid Traversal Approach to Visualizing Particle-Based Simulation Data

Christiaan P. Gribble Thiago Ize Andrew Kensler Ingo Wald Steven G. Parker

Scientific Computing and Imaging Institute, University of Utah
{cgribble|thiago|aek|wald|sparker}@cs.utah.edu

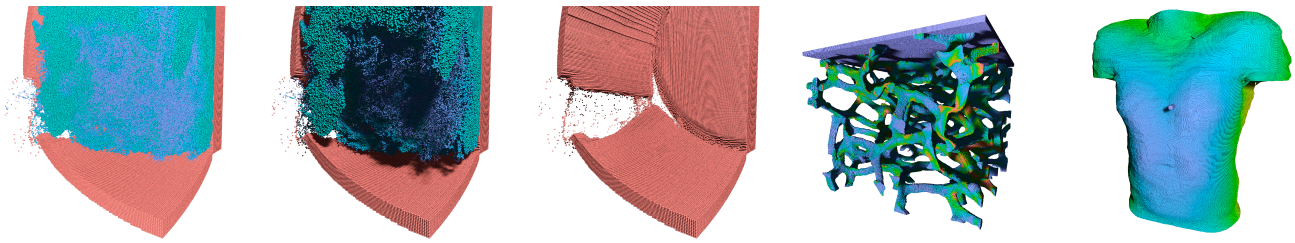


Figure 1: *Visualizing particle-based simulation data with efficient ray tracing.* We describe several optimizations that tailor the coherent grid traversal algorithm for efficient and effective visualization of particle-based simulation data. Our approach renders images of datasets with millions of particles at highly interactive rates and also provides run time control of several advanced visualization features, including color mapping, illumination effects from soft shadows, and parameter range culling. The interactive performance of our approach compares favorably with other particle visualization systems.

1 Introduction

We describe an efficient algorithm for visualizing particle-based simulation data using fast packet-based ray tracing and multi-level grids. In particular, we introduce optimizations that exploit the properties of these datasets to tailor the coherent grid traversal (CGT) algorithm [5] for particle visualization, achieving both improved performance and reduced storage requirements.

2 Coherent Grid Traversal for Particle Datasets

Several observations about glyph-based particle visualization permit optimizations over and above those employed by the original CGT algorithm. First, a sphere S with center C and radius r is symmetric, so determining whether S overlaps a frustum F is analogous to testing whether C is in the r-neighborhood of F. In particular, we can test whether the distance from C to any of the bounding planes of F is less than r. Second, testing whether the distance from C to the planes of F is less than r is the same as testing whether C is inside another frustum F_r that has been enlarged by r. By traversing the grid using the enlarged frustum, only those spheres whose centers lie inside that frustum must be tested for intersection, and therefore each sphere must be stored only in the cell in which its center is located. We call this approach the *sphere-center* method.

To facilitate a more efficient traversal, we leverage the macrocell hierarchy described by Parker et al. [2]. Each level in this hierarchy imposes a coarser grid over the previous level, and each macrocell corresponds to an $M \times M \times M$ block of cells in the underlying level. We currently use a simple two-level hierarchy: one level of macrocells imposed on top of the actual grid.

SIMD shaft culling, which is used in the original CGT algorithm, relies on primitives that posses planar edges, a property which spheres do not exhibit; hence, this fast culling technique is not appropriate for our application. However, if the distance from the center of a given sphere to any of the planes of the bounding frustum is greater than the radius of the sphere, the rays bounded by the frustum cannot intersect the sphere. We use this test to quickly cull non-intersecting spheres.

In addition to its position and radius, up to four values representing properties from the simulation can be stored with each particle. To gain additional insight into the behavior of a simulation, investigators often isolate particles with parameters that take on a particular value or that lie within some range of values. We cull particles whose range of values do not overlap the currently valid range, thereby avoiding unnecessary intersection tests. To support this interaction, macrocells must store the minimum and maximum values of each data variable across all of the particles they contain.

Dataset	Our CGT	RTRT Bigler et al. [1]	GPU-based Gribble et al. [3]
Cylinder	100.20	12.60	5.78
JP8	99.88	6.90	17.40
Bullet	126.93	11.90	2.56
Thunder	40.86	14.90	8.10
Foam	14.33	6.40	2.04
BulletTorso	18.31	13.50	1.56

Table 1: *Comparison of particle visualization methods.* Frame rates achieved using our modified CGT algorithm and two state-of-the-art interactive particle visualization systems.

Soft shadows from area light sources provide important visual cues about the relative position of objects in complex datasets. Packets of coherent shadow rays are generated by connecting the hit point corresponding to a given primary ray with some number of samples on an area light source [4]. Secondary ray packets then traverse the grid using the modified CGT algorithm in a manner identical to that used for primary ray packets.

3 Results

We compare the performance of our approach with two recent systems that, to our knowledge, represent the current state-of-the-art in interactive particle visualization. The first is based on an optimized single ray grid traversal algorithm [1], while the second leverages programmable graphics hardware and software-based acceleration techniques [3]. The results in Table 1 show that our modified CGT algorithm compares favorably with these systems.

References

[1] J. Bigler, J. Guilkey, C. Gribble, S. Parker, and C. Hansen. A Case Study: Visualizing Material Point Method Data. In *Eurographics/IEEE Symposium on Visualization*, pages 299–306, May 2006.

[2] S. Parker, M. Parker, Y. Livnat, P.-P. Sloan, C. Hansen, and P. Shirley. Interactive Ray Tracing for Volume Visualization. *IEEE Transactions on Visualization and Computer Graphics*, 5(3):238–250, 1999.

[3] C. P. Gribble A. J. Stephens, , J. E. Guilkey, and S. G. Parker. Visualizing Material Point Method Datasets on the Desktop. In *British HCI 2006 Workshop on Combining Visualization and Interaction to Facilitate Scientific Exploration and Discovery*, pages 1–8, September 2006.

[4] I. Wald, C. Benthin, M. Wagner, and P. Slusallek. Interactive Rendering with Coherent Ray Tracing. *Computer Graphics Forum*, 20(3):153–164, September 2001.

[5] I. Wald, T. Ize, A. Kensler, A. Knoll, and S. G. Parker. Ray Tracing Animated Scenes using Coherent Grid Traversal. *ACM Transactions on Graphics*, 25(3):485–493, July 2006. (Proceedings of Siggraph '06).