Deep Coherent Ray Tracing

Erik Månsson[*] Lund University/TAT AB
Jacob Munkberg[†] Lund University
Tomas Akenine-Möller[‡] Lund University

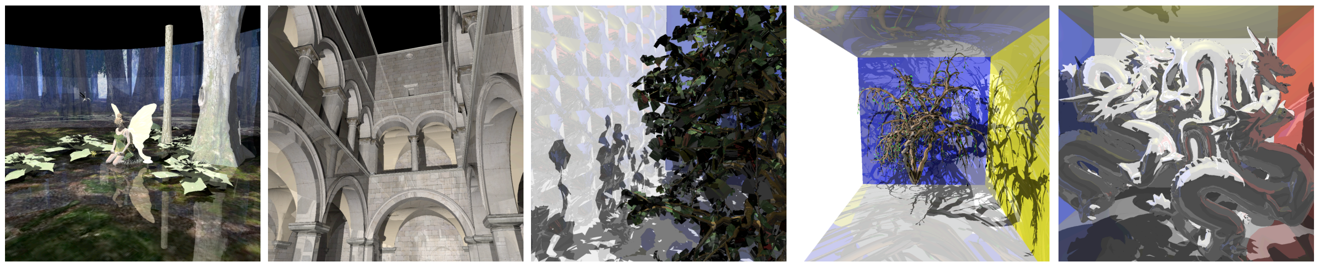


Figure 1: The example scenes used for evaluating our reordering heuristics and coherence measures. All materials in the scenes are reflective in order to study the behavior of secondary rays. *Fairy* is an example of the ”teapot in a stadium” problem with a small detailed model in a simple large environment. *Sponza* is a standard benchmark model. Two tree-scenes with and without leaves (*newtree* and *oldtree* respectively) ensure complex traversal paths for secondary rays. *Dragon* is a scene with four reflective Stanford Dragons in a Cornell box.

Abstract

Tracing secondary rays, such as reflection, refraction and shadow rays, can often be the most costly step in a modern real-time ray tracer. In this paper, we examine this problem by using suitable ray coherence measures and present a thorough evaluation of different reordering heuristics for secondary rays. We also present a simple system design for more coherent scene traversal by caching secondary rays and using sorted packet-tracing. Although the results are only slightly incremental to current research, we believe this study is an interesting contribution for further research in the field.

Index Terms: I.3.7 [Three-Dimensional Graphics and Realism]: Raytracing; I.3.6 [Methodology and Technique]: Graphics data structures and data types

1 Introduction

Current research on real-time ray tracing has been very biased toward optimizing primary and shadow rays. The results are impressive, approaching frame rates of rasterization-based techniques. However, one major motivation of using ray tracing instead of a GPU renderer is the natural extension to secondary rays, giving true reflections, refraction and global illumination effects.

A key to ray tracing performance is grouping bundles of rays in *coherent* packets, and testing these packets against the three-dimensional scene instead of testing each ray individually. Coherent rays here means rays with similar directions and origins. Section 3 will present more formal coherence definitions. For primary rays, sorting rays into coherent packets is straightforward, as all rays has a common origin (the camera/eye), and we can easily create packets by grouping the rays of nearby pixels. For secondary rays, however, we no longer have a common origin, nor coherent directions. Imagine a packet of coherent primary rays traced towards a reflective ball. If the ball is small enough, the rays will diverge in nearly all directions, and some rays of the packet might even miss the ball. The packet will lose its coherent properties already after one bounce, making it useless for further packet-traversal, as the generated secondary rays will take substantially different paths through the scene.

In this paper, we will investigate techniques for increased coherence for secondary rays and evaluate them in a modern packet-based ray tracer. The rest of the paper is structured as follows:

Section 2 presents previous work in this area, while Section 3 introduces more formal definitions of ray coherence. Section 4 discusses our system design and sorting approaches, which are evaluated in Section 5. Finally, we offer a discussion with some thoughts on future work in Section 6.

2 Previous Work

Fast ray tracing has been an active area of research for more than two decades. In this section, we will summarize the research most related to our work.

The idea behind breadth-first ray tracing [12] is to form a set of rays and compare each object in turn against this set. First, all primary rays are traced, followed by all shadow rays from the primary intersections, and then all reflection and refraction rays. Each ray type on each level forms a set that is compared to the geometry.

By tracing packets of coherent rays together, interactive ray tracing has been achieved on standard desktop computers using highly optimized *kd*-trees [18, 20]. The resulting performance is impressive for primary rays and coherent shadow rays, but the algorithms are not easily extendable to secondary rays.

Interactive distribution ray tracing [5] uses ray packets for distributed ray tracing to simulate depth of field, motion blur and soft shadows. Secondary rays are grouped in coherent packets according to ray type (shadow packets, reflection packets and refraction packets), similar to breath-first tracing, but locally, as it is applied to the secondary rays generated from *one* packet. By carefully identifying and controlling the parameters that causes divergent secondary rays, they argue that some coherence can be obtained for secondary rays too. Recently, performance improvements of about $2\times$, compared to single ray tracing, was reported [6].

A survey of algorithms where ray direction information is used

[*]e-mail: erik.mansson@tat.se
[†]e-mail: jacob@cs.lth.se
[‡]e-mail: tam@cs.lth.se

to accelerate ray tracing is presented by Arvo and Kirk [2]. The light buffer algorithm [9] optimizes shadow rays by associating objects to each cell of a direction cube around a light source. A more recent extension uses rasterized orthogonal views created from a set of angles [8]. A more general approach to this problem is to partition secondary rays in coherent voxels, as in the ray classification algorithm [1]. Here, a 5D BSP-tree with three dimensions for ray origins and two for ray directions, is used. Each split will generate 2^5 sub-nodes, so the tree grows rapidly. Effective culling can be performed, but the memory requirements for storing a 5D-tree of rays in a complex scene are huge. A recent variant of this approach is described in the ray engine [7], where an octree is used for geometry and a 5D tree for rays. This configuration is used to efficiently find and trace coherent packets of rays on a GPU. As rays are generated, they are added to a cache, which collects them into buckets of rays with coherent origins and directions.

Phaarr et al. [16] introduce memory coherent ray tracing of complex scenes by grouping rays and geometry into a spatial *scheduling* voxel grid. The voxels are processed one at a time, by tracing the contained rays against the contained geometry. Voxels with geometry currently in the cache have priority. By carefully designing ray-, geometry- and texture caches, they argue that rendering times can be substantially improved. They also proposes a ray grouping approach, where rays are first clustered by position and then sorted by direction. However, they discarded this approach, as it worked only for rays with similar origins, but failed to exploit coherence for rays whose origins were not close together. Navrátil et al. [13] extend the scheduling algorithm and use a simulator to prove that they can dramatically reduce the amount of geometry brought into the caches. The sorting approaches from Pharr et al. [16] has been applied to photon mapping and global illumination [10, 11, 19].

Another approach to handle secondary rays is omnidirectional ray tracing [17], where packet-based *kd*-tree traversal is relaxed to handle packets with more divergent ray directions.

Many approaches exist, but there is still no general algorithm for efficient packet tracing of secondary rays in terms of both reduced memory bandwidth and frame rates. We will borrow ideas from the work described above and evaluate techniques for coherent secondary ray tracing.

3 Coherence Measures

By inspecting frame rates alone, we risk getting biased results from the underlying system, including the compiler, processor and memory architecture. For a more general discussion of performance, we have chosen to include ray coherence measures alongside actual frame rates. For clarity, we present our measures with regards to a *kd*-tree acceleration data structure (ADS), but it is straightforward to adapt these concepts to other data structures, such as bounding volume hierarchies and grids.

The *ray coherence theorem* [2, 15] can be used to compute a bound on the directions of rays which originate at one object and then hit another. However, it is not straightforward to adapt this to a global coherence measure in complex scenes with arbitrary geometry. For a given scene and ray tracing engine, we can easily extract two measures: the number of *traversal steps* in the ADS and the number of ray-triangle *intersection tests*. In the case of a *kd*-tree, the traversal steps is the sum of split-plane intersections over all rays, and the intersection tests are the total number of ray-triangle intersection tests. We can also measure *unique* traversal steps, which is the number of split planes in the *kd-tree* touched by any ray. Unique intersection tests are defined analogously. Note that traversal steps and intersection tests are not independent of each other. By adjusting ADS construction parameters, one can trade between them to some extent.

As suggested by Benthin [3], we define **traversal coherence** (c_t) as the the sum of traversal steps (s_t) divided by the number of unique traversal steps (s_u):

$$c_t = \frac{\sum s_t}{\sum s_u}. \tag{1}$$

The interpretation is: if many rays traverse the ADS in similar paths, c_t will be high, because the number of unique traversal steps will be relatively low. This means that there is coherence to exploit. Conversely, if each ray takes a unique path in the ADS, the traversal coherence will be close to one, and there is no coherence to exploit. We define **intersection coherence**, (c_i) similarly, as the sum of intersection tests (i_t) divided by the sum of unique intersection tests (i_u):

$$c_i = \frac{\sum i_t}{\sum i_u}. \tag{2}$$

We include these measures as they are good indicators of the amount of *available* ray coherence in a certain scene.

3.1 Packet Coherence

The goal of this project is to examine how different ray sorting techniques exploit the available coherence. In order to measure that, we introduce packet coherence.

For a packet-based ray tracer, we define *packet traversal steps* (s_{packet}) as the sum of the split-plane intersections for ray packets, and *packet intersection tests* (i_{packet}), which is the number of packet-triangle intersections. Denote the ray traversal steps as s_{rays}. The **traversal packet coherence** (p_t) is defined as:

$$p_t = \frac{\sum s_{rays}}{\sum s_{packet}}. \tag{3}$$

This measure can be interpreted as the average number of active rays per traversal step, and is a number between one and the packet size. A high number indicates that the bundle exploits the available coherence better.

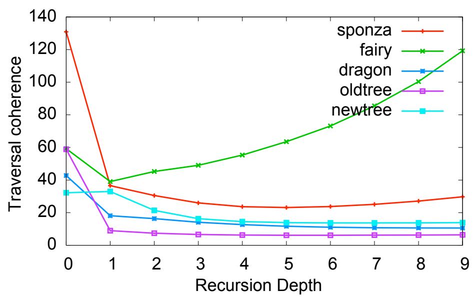
In an analogue way, we define **intersection packet coherence** (p_i) as:

$$p_i = \frac{\sum i_{rays}}{\sum i_{packet}}. \tag{4}$$

As a demonstration of these measures, we will look at traversal coherence and traversal packet coherence for the test scenes of Figure 1 as a function of the ray depth. Intuitively, one expect a drastic decrease in coherence when we allow rays to bounce around in the scene. Figure 2 confirms this, but also indicates that some scenes have a slight increase in traversal coherence for higher ray depths, presumably because some rays get stuck and bounce back and forth between parallel planes. In the lower part of Figure 2, we show the traversal packet coherence for a standard, packet-based, ray tracer, and we can see that it fails to exploit the increase in traversal coherence. In the next section, we will discuss a technique that can handle this situation.

4 Deep Coherent Tracing

The traversal unit in a ray tracer benefits from a coherent packet, with respect to both ray origins and ray directions, in order to effectively cull parts of the three-dimensional geometry. The standard technique to achieve this is to create coherent subgroups in the packet by masking out incoherent rays [3]. The same packet will thus be sent several times to the traversal unit, each time with a different mask of active rays. Inspired by previous work [5, 7, 16], we suggest grouping coherent rays together earlier in the system, in order to reduce the number of calls to the traversal unit, thus reducing the number of packets and bypass the masking procedure.



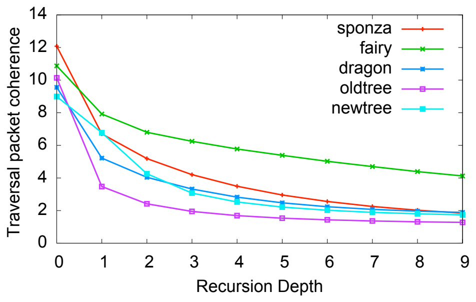


Figure 2: Traversal coherence and traversal packet coherence for specular rays. Rays are grouped in packets of 16 rays in this example. The traversal packet coherence drops exponentially, but several scenes show an increasing traversal coherence, indicating that there is coherence to exploit.

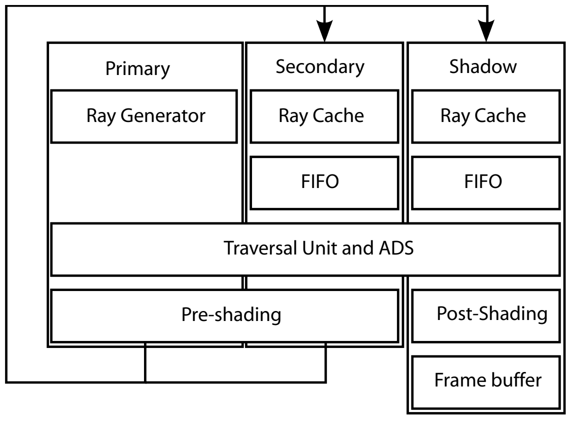


Figure 3: A high level illustration of our ray tracing system. All spawned secondary rays are added to a ray cache, and are sorted before sent to the traversal unit. Primary rays can be sorted when generated, and do not need to be cached.

4.1 Implementation

In a scene with glossy reflections, multiple bounces and soft shadows, the primary rays only represent a fraction of the total number of rays. To improve coherence for secondary rays, we use a system as outlined in Figure 3. In the following, we will describe the functionality of each unit.

Ray Cache This unit holds spawned secondary rays waiting to be ray traced. To improve coherency, secondary rays are sorted using some heuristic, as will be further described in Section 4.3.

When a cell in the cache is filled, the sorted ray packet is sent to a FIFO queue, ready to be traced by the traversal unit. We alternate between sending primary packets and coherent packets of secondary rays until the entire image is rendered. Ray tracing finishes when there are no more primary rays to trace and no more rays waiting in the cache. Please note that all levels of secondary rays will write into the cache, so a packet can be a combination of secondary rays of many different depths and types. Some logic has to be added in order not to overflow the FIFO and to empty the cache of remaining rays. For example, a packet of primary rays can generate a large number of secondary rays, all of which, in turn, spawns new rays. Dequeuing and tracing one packet from the FIFO may generate a large number of new FIFO entries. We can avoid this situation by tracing a tile of primary rays and all its generated packets before proceeding to the next tile. To empty the cache of remaining rays, a sweep at the end forces these rays into the FIFO. A certain ray path can take arbitrary time (within one frame/tile), as a ray will not be moved to the FIFO queue until the associated cache cell is full.

Traversal Unit The input is either a primary packet or a ray packet from the FIFO, consisting of a set of N coherent secondary rays, which are traced through the scene with the help of a suitable acceleration data structure.

Shader Unit This unit executes a shader program for a set of rays and is divided in two parts:

– Pre-shading. Here, one or more output rays are generated and added to the ray cache. Each spawned ray gets an updated weight, which is the weight of the incoming ray multiplied with the shader value at the intersection point. It also inherits the pixel address of the incoming ray. In this way, a ray holds its accumulated shading (all shading in the ray tree starting from the camera ray up to the current ray), as well as the pixel position it affects.

– Post-shading. Outputs color values that are added to the output buffer. In practice, post-shading is handled by shadow rays. If a shadow ray does not hit an occluder, its accumulated weight is added to the corresponding pixel in the output buffer.

4.2 Iterative Ray Tracing

To allow for ray sorting and cacing as described above, the classic, recursive ray tracing algorithm is not suitable. However, by adding a weight and a pixel address to every ray [16], we can reformulate the ray tracing algorithm iteratively, and we do not need to keep track of any ray trees to compute the final shading.

This model is based on the assumption that the shader can be split into a linear combination of weighted ray contributions. That is, we must be able to assign a weight to each ray *before* it is traced, so we can discard the ray tree which generated it. Shaders involving adaptive sampling based on intermediary results can therefore not directly be rewritten iteratively. The weights can be computed using information from the intersection points, including incoming and outgoing ray directions and all material properties, so most shaders are compatible, including analytical and measured BRDFs.

Another drawback is the higher resolution of the frame buffer. As each ray will directly write back its contribution, a higher color bit depth is needed. 1000 shadow rays, each with weight 0.001, will add up to one, but if added thousand times to a frame buffer with standard eight bit resolution, information will inevitably get lost due to quantization. We do not consider this to be a serious problem as a good rendering system would store each color component using at least a 16-bit floating point value, in order to handle, for example, high dynamic range light probes. We expect that all real-time ray tracers will have this capability in the future.

The number of frame buffer accesses will also increase as every (shadow) ray segment will write back a value, instead of as in the recursive case, where a ray tree only writes back the final color once.

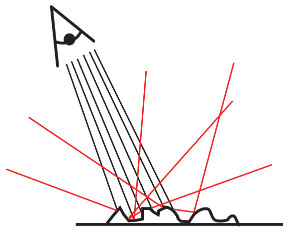


Figure 4: Secondary rays can diverge fast even if the primary rays (and their intersection points) are coherent.

4.3 Regrouping Algorithms

After the first intersections of a packet of primary rays has been found, new rays may be spawned by the surface shaders. These new rays should be sorted in a way that exploits the coherency in the three-dimensional scene. This sorting could be naturally integrated in some shaders. An ambient occlusion shader could, for example, output coherent packets of rays, clustered by direction over the hemisphere. We strive for a more general approach, where we only assume that the shader returns a set of rays with corresponding weights. Given a coherent packet of primary rays, the primary intersections will still be relatively coherent as they intersect the base geometry. These intersections become origins for secondary rays. The directions of the secondary rays can be more or less randomized, as normal maps, displacement maps or other techniques modify the local geometry and normal vectors as shown in Figure 4. This observation is important when designing a ray sorting strategy.

Many ray sorting approaches exist, ranging from a full 5D-ray tree [2], to sorting by ray directions and or positions [10, 11, 16, 19]. We want to combine these ray grouping techniques with packet tracing for real-time rendering.

Keep in mind that we compete with tracing single rays. If we only care about core tracing speed, this relation can be formulated as: $t_{sorting} + t_{packet\ tracing} < t_{single\ tracing}$. In order to improve core tracing speed, we need both a fast sorting step and a substantially faster packet tracing step.

We have tested the following heuristics for sorting rays:

none Rays are not regrouped. Rays are masked out as they exit the scene. Secondary packets may be partially filled, and the contained rays might not be coherent. An *active mask* is updated before each traversal call to ensure that a coherent subset (with respect to ray direction) of the rays are active for that traversal call. To find all intersections, the same packet might be sent several times to the traversal unit, each time with a different active subset or rays. This is the "standard" approach in packet-based ray tracers [3].

dir Eight packets are filled in parallel, each corresponding to one direction/sign configuration, sorting the ray direction into octants $(-,-,-),(-,-,+),\dots,(+,+,+)$. Packets that are sent to the traversal/intersection unit will be completely filled as long as more rays are available.

mdir(x) Extension of **dir** that split the ray directions in x cells of equal size over a direction cube [2] . A higher value x means that each direction cell covers a smaller solid angle.

fastpos The scene bounding box is split into octants. All rays with origins in an octant will be added to a corresponding ray packet. This gives a coarse but fast position sorting approach.

pos(x) A list of packets is maintained. Each packet is assigned a origin p_o. A ray with origin r_o searches for the closest packet where $\|r_o - p_o\|^2 < x$, where x is the search radius. If no packet is close enough, a new packet is created with $p_o = r_o$.

opos All rays are sorted into lists depending on their direction sign, but packet creation is delayed until no more rays are available. Then the first ray is extracted and the list is searched for other rays with closest origin. These rays will form a packet. This algorithm is too slow for real-time ray tracing, but is included as a reference sorting approach.

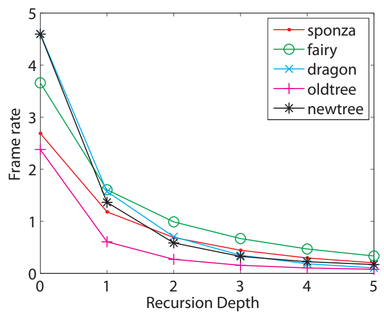


Figure 5: Frame rate as a function of ray depth for the **none** heuristic, using 16 rays per packet. The frame rates are drastically lower for higher ray depths for all scenes. Measured on a 3.0GHz Pentium-4 with hyperthreading.

5 Results

In order to design a system that efficiently traces secondary rays, we have studied the performance of secondary rays in a modern, SIMD optimized packet based ray tracer [3], using an SAH-optimized *kd*-tree as acceleration data structure. We use standard test scenes of but with added reflective materials and we have also designed scenes to stress the performance of secondary rays. See Figure 1 and Appendix A for scene details.

To increase the number of secondary rays in the evaluation, we use a fixed ray depth and do not discard rays because of a too small color contribution. We have rewritten the tracing kernel according to Section 4.1 so that secondary rays are sorted in coherent packets before traversal, and we propagate ray weights forward so each ray directly can write back a value once it is traced and shaded.

5.1 Heuristics and frame rates

When using higher recursion depths for our scenes, Figure 5 shows how the frame rates drastically decrease for higher ray depths when we using the standard way (**none**) for tracing secondary rays. Can a better sorting heuristic help in this situation?

heuristic	none	dir	fastpos	mdir(96)	pos(1.0)
fps	1.9	1.8	1.7	1.7	1.8

Table 1: Performance for different heuristics in the dragon scene. The recursion depth is one and each packet contains sixteen rays. Measured on a 3.0GHz Pentium 4 with hyperthreading.

In our implementation, the answer is unfortunately no. By using the heuristics from Section 3, we get similar or slightly lower frame rates, as seen in Table 1. Boulos et al. [6] report speed-ups of 2× compared to *single* tracing, but as the **none** heuristic is already used for secondary rays in a state-of-the art real-time ray tracer [4], and described as a way of tracing secondary rays in a modern *kd*-tree packet tracer [3], we use this as the base line. Our goal with this system is to exploit more coherence than available in the rays generated from intersections of one packet.

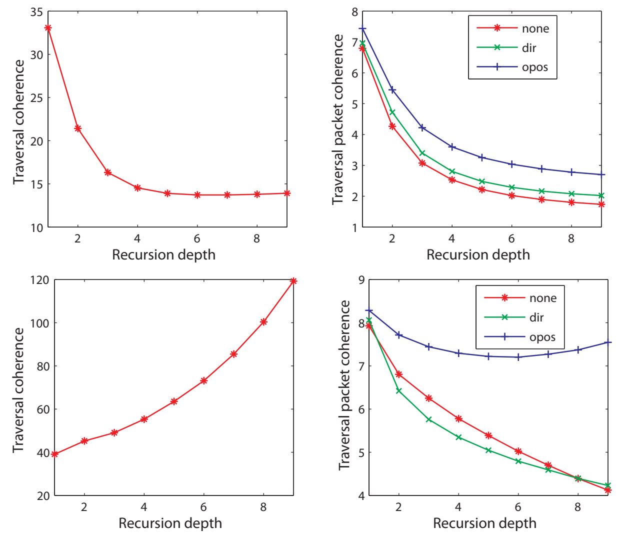


Figure 6: Traversal coherence and traversal packet coherence for the test scenes *Newtree* (above) and *Fairy* (below). To the right, three different sorting heuristics are compared. All heuristics use a packet size of 16 rays in this example.

We have seen performance increases using **dir** and **mdir** in scenes where many secondary rays are generated from one intersection point, e.g. ambient occlusion shaders or diffuse reflections, but as it is relatively simple to output coherent packets locally directly from those shaders, we have not included such test cases in our evaluation.

One "problem" is that the **none** heuristic performs quite well. Packets with different direction signs will traverse the acceleration structure from the top several times, once for each sign configuration. This is actually quite inexpensive, because split planes and triangles that are accessed by rays during the first traversal steps are likely to be in the memory cache if they also are accessed during the other trace passes, and will therefore not degrade performance as much as the ray coherence predicts. In addition, all kinds of ray sorting, although simple, add overhead as well.

All measurements have been made while rendering a 640×480 pixel large image with texture mapping and phong shading. Each intersection spawns one shadow ray and higher order rays until a fixed recursion depth has been reached. We have used recursion depth one unless otherwise stated.

5.2 Coherence Analysis

However, as we will show in this section, interesting and useful information inherited in "deep" ray tracing can be extracted using our coherence measures.

Lets consider how the sorting heuristics affects the coherence measures of Section 3. Note that we allow rays from different levels, e.g., secondary rays from any depth, to be part of same packet and therefore represent measures from *all* secondary rays up to a specified recursion depth. *Newtree* is a typical scene where the traversal coherence declines exponentially and the window of opportunity for good re-orderings slips away. The packet coherence drops for all heuristics, and **dir** has only slightly better values than **none**. The upper row of Figure 6 shows this behavior.

The *Fairy* scene shows a different behavior. In the lower row of Figure 6, we see how the traversal ray coherence increases almost linearly with increasing ray depth. The traversal packet coherence from the **opos** heuristic is almost constant and the **dir** heuristic performs worse than **none**. The explanation for this behavior can be found in the backdrop of the scene which is constructed from three transparent layers. Most rays will miss the scene after a few bounces since it is open, and the only rays that bounce nine times are the rays stuck between the layers of the backdrop. This "extra" scene coherence was captured by our measures, but could not be exploited by the simpler sorting heuristics.

Table 4 presents coherence measures for all five test scenes for the sorting heuristics at recursion level 1, 2 and 3. The table shows that the sorting heuristics succeeds in exploiting the inherent coherence of the scene, but the differences are relatively small. The results also indicates that the direction-based heuristics performs better overall. These results by themselves will not justify rewriting the ray tracing kernel on a current CPU, as the **none** approach is competitive, but the analysis is still interesting for future ray tracing research.

5.3 Packet Size Statistics

In this section, we study the packet size influence. We have seen earlier that the **none** algorithm performs surprisingly well, partly because many packets are half-filled due to masking, with smaller frustums. Therefore, we have studied performance and coherence for various sizes of secondary ray packets. Generally, smaller packets means that we fill coherent packets faster (and more locally), but also that we can not exploit as much coherence as a bigger packet. Figure 7 shows how the packet size affects the frame rate for the dragon scene. Looking at Table 2, which lists packet coherence for different packet sizes, we see highest numbers for small packets. However, when looking at the total frame rate, packets with 8 or 16 rays perform best for our current implementation.

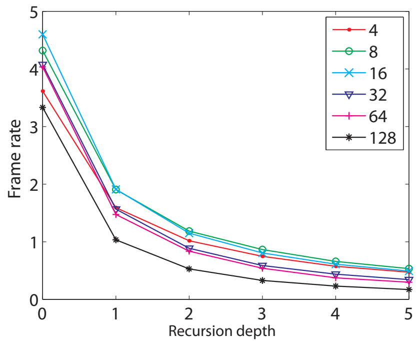


Figure 7: Varying packet sizes for the dragon scene plotted in a frame rate versus ray depth chart for the **none** heuristic. Measured on a 3.0GHz Pentium4 with HT.

5.4 Other Traversal Schemes

Interval ray tracing [18] uses extremal properties of the ray packet, which results in fewer instructions in the inner traversal loop since we do not need to inspect every ray for each step through the *kd*-tree. For primary rays, we note how the packet coherence drops, but each traversal step is faster. In our implementation, interval ray tracing without *Entry Point Search* [18] is on par with the **none** algorithm. Interval ray tracing will visit more nodes than what is strictly needed. Before intersecting with the triangles of a leaf, we therefore intersect the rays with the bounding box of that leaf to avoid unnecessary primitive tests. Consequently, we perform the same number of intersection tests as when doing standard packet tracing. Secondary packets, however, diverge far too much for interval ray tracing to be feasible; the packet coherence drops below one, which means that the traversal scheme visits many nodes that would not be touched during standard (single-)ray tracing.

scene	coherence type	packet size 4	8	16	32	64
dragon	c_t	17.8	18.8	18.8	18.8	18.8
	p_t	2.6	3.9	5.5	7.0	8.8
	$p_t/size$	0.65	0.49	0.44	0.14	0.14
fairy	c_t	37.0	39.1	39.1	39.1	39.1
	p_t	3.1	5.1	7.9	11.2	15.0
	$p_t/size$	0.78	0.64	0.49	0.35	0.23
newtree	c_t	31.6	33.1	33.1	33.1	33.1
	p_t	2.9	4.5	6.8	9.2	12.4
	$p_t/size$	0.72	0.56	0.43	0.28	0.19
oldtree	c_t	8.3	9.0	9.0	9.0	9.0
	p_t	2.0	2.7	3.5	4.1	4.8
	$p_t/size$	0.5	0.34	0.22	0.13	0.075
sponza	c_t	33.1	36.6	36.6	36.6	36.6
	p_t	2.7	4.3	6.7	9.0	12.5
	$p_t/size$	0.68	0.54	0.42	0.28	0.20

Table 2: Traversal coherence analysis for different packet sizes at recursion level 1, using the standard **none** heuristic. We list traversal coherence (c_t), traversal packet coherence (p_t) and packet coherence divided by the packet size. ($p_t/size$)

		primary		secondary	
scene	type	c_t	p_t	c_t	p_t
newtree	none	32.2	9.0	33.1	6.8
	interval	32.2	5.0	33.1	0.087
	omni	32.2	9.2	33.1	7.3
oldtree	none	58.8	10.1	9.0	3.5
	interval	58.8	8.2	9.0	0.058
	omni	58.8	10.2	9.0	3.9

Table 3: Traversal coherence (c_t) and traversal packet coherence (p_t) for different traversal algorithms with 16 rays per bundle. The recursion depth is one.

When using **none** packet tracing, we can only trace one direction/sign configuration at a time, which force us to trace each packet several times, but with different elements masked out. *Omni-directional Ray Tracing* [18] borrows some ideas from how Bounding Volume Hierarchies can be traversed, which allows packets with different direction/sign configurations to be traced as one packet, without masking. The cost is a somewhat slower inner traversal loop and reversed traversal order for some rays, i.e. some rays will not visit nodes in *distance from origin* order.

In Table 3, we see how omni-directional ray tracing gives a slight increase in packet coherence, but we did not succeed in making an implementation that is faster (in terms of frame rates) than our **none**-tracer.

6 Discussion and Future Work

Although there is clearly coherence among secondary rays in the scenes to exploit, it is hard to design a heuristic that is both simple, fast and significantly better than the standard approach using masking. More elaborate ray sorting heuristics can be suggested, but the performance bound set by tracing the secondary rays by masking techniques is hard to beat. In our implementation, the results obtained do not justify a rewriting of the ray tracing kernel (from a frame rate perspective), but are still interesting for future ray tracing research.

For future work, we would like to implement our system in hardware, as we believe these concepts might be more important there, and the overhead of the ray caching system might be more acceptable. In addition, a CUDA [14] implementation might prove to be an interesting avenue of future research.

We would also like to extend the system with a shader cache, that analogously sorts the shader calls for increased shading coherence and simplified batch shading.

References

[1] J. Arvo and D. Kirk. Fast Ray Tracing by Ray Classification. *Computer Graphics (Proceedings of SIGGRAPH '87)*, 21(4):55–64, 1987.

[2] J. Arvo and D. Kirk. *An Introduction To Ray Tracing - A survey of Ray Tracing Acceleration Techniques*. Academic Press, 1989.

[3] C. Benthin. Realtime Ray Tracing on current CPU Architectures. *PhD thesis, Saarland University*, 2006.

[4] J. Bikker. Arauna Ray Tracer. Website/forum "http://ompf.org/forum/viewtopic.php?p=2072#2072", 2007.

[5] S. Boulos, D. Edwards, J. D. Lacewell, J. Kniss, J. Kautz, P. Shirley, and I. Wald. Interactive Distribution Ray Tracing. Technical Report UUSCI-2006-022, SCI Institute, University of Utah, 2006.

[6] S. Boulos, D. Edwards, J. D. Lacewell, J. Kniss, J. Kautz, P. Shirley, and I. Wald. Packet-based Whitted and Distribution Ray Tracing. In *Graphics Interface*, pages 177–184, May 2007.

[7] N. A. Carr, J. D. Hall, and J. C. Hart. The Ray Engine. In *Graphics Hardware*, pages 37–46, 2002.

[8] T. Hachisuka. *GPU GEMS 2 - High Quality Global Illumination Rendering Using Rasterization*. Addison-Wesley, 2005.

[9] E. A. Haines and D. P. Greenberg. The Light Buffer: A Ray Tracer Shadow Testing Accelerator. *IEEE Computer Graphics and Applications,*, 6(9):6–16, 1986.

[10] V. Havran, J. Bittner, R. Herzog, and H.-P. Seidel. Ray Maps for Global Illumination. In *Eurographics Symposium on Rendering*, pages 43–54, 2005.

[11] V. Havran, R. Herzog, and H.-P. Seidel. Fast Final Gathering via Reverse Photon Mapping. *Computer Graphics Forum (Proceedings of Eurographics 2005)*, 24(3):323–333, 2005.

[12] K. Nakamaru and Y. Ohno. Breath-First Ray Tracing Utilizing Uniform Spatial Subdivision. *IEEE Transactions on Visualization and Computer Graphics*, 3(4):316–328, 1997.

[13] P. A. Navrátil, D. S. Fussell, and C. Lin. Dynamic ray scheduling for improved system performance. Technical Report TR-07-19, The University of Texas at Austin, April 12 2007.

[14] NVIDIA. CUDA, Compute Unified Device Architecture, Programming Guide. Technical report, Version 1.0, 2007.

[15] M. Ohta and M. Maekawa. Ray Coherence Theorem and Constant Time Ray Tracing Algorithm. In *Computer Graphics International*, pages 303–314, 1987.

[16] M. Pharr, C. Kolb, R. Gerbstein, and P. Hanrahan. Rendering Complex Scenes with Memory-Coherent Ray Tracing. In *Computer Graphics (Proceedings of SIGGRAPH '97)*, pages 101–108, 1997.

[17] A. Reshetov. Omnidirectional Ray Tracing Traversal Algorithm for kd-trees. In *IEEE Symposium on Interactive Ray Tracing*, pages 57–60, 2006.

[18] A. Reshetov, A. Soupikov, and J. Hurley. Multi-Level Ray Tracing Algorithm. *ACM Transaction on Graphics*, 24(3):1176–1185, 2005.

[19] J. Steinhurst, G. Coombe, and A. Lastra. Reordering for Cache Conscious Photon Mapping. In *Graphics Interface*, pages 97–104, 2005.

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

A Scene details

Dragon Four instances of the Stanford Dragon in a Cornell box. 61k triangles. Source: Stanford 3D Scanning Repository.

Fairy Scene with a *Teapot in a stadium* problem featuring one fairy and one dragonfly, both highly tessellated, in an a forest scene. 173k triangles. Source: Utah 3D Animation Repository, Ingo Wald.

Newtree This scene is similar to the Tree scene, but has fewer leaves and more branches with complex shadowing. 56k triangles.

Oldtree The camera looks at a reflective wall with complex geometry. Secondary rays are reflected into a tree with near random distribution of leaves. 410k triangles.

Sponza Standard architecture scene. Features many large polygons and some curved elements. 42k triangles. Source: Marko Dabrovic.

		recursion level 1				recursion level 2				recursion level 3			
		t		i		t		i		t		i	
scene	heuristic	c_t	p_t	c_i	p_i	c_t	p_t	c_i	p_i	c_t	p_t	c_i	p_i
dragon	none	18.1	5.2	20.2	3.2	16.3	4.0	17.7	2.8	14.1	3.3	14.8	2.5
	dir		5.6		3.2		4.3		2.8		3.5		2.5
	mdir(96)		5.2		3.1		4.2		2.8		3.5		2.5
	fastpos		4.9		3.1		3.6		2.6		2.9		2.3
	pos(1.0)		5.0		3.1		3.9		2.8		3.3		2.5
	opos		5.9		3.4		4.9		3.0		4.2		2.7
fairy	none	39.1	7.9	16.8	4.4	45.3	6.8	26.3	4.6	49.1	6.2	33.8	4.8
	dir		8.1		4.3		6.4		4.4		5.8		4.3
	mdir(96)		7.9		4.4		6.5		4.5		6.0		4.6
	fastpos		7.0		4.0		5.5		3.9		4.9		3.9
	pos(1.0)		7.2		4.1		5.7		4.1		5.2		4.0
	opos		8.3		4.4		7.7		4.9		7.4		5.2
newtree	none	33.0	6.8	23.2	4.0	21.4	4.3	16.5	2.9	16.3	3.1	12.6	2.3
	dir		7.0		4.0		4.7		2.9		3.4		2.3
	mdir(96)		6.5		3.8		4.7		3.0		3.4		2.4
	fastpos		6.5		3.9		4.1		2.7		3.0		2.2
	pos(1.0)		6.8		4.0		4.2		2.8		3.0		2.2
	opos		7.4		4.2		5.4		3.3		4.2		2.7
oldtree	none	9.0	3.5	6.2	2.1	7.4	2.4	5.1	1.7	6.6	2.0	4.4	1.5
	dir		3.7		2.1		2.6		1.7		2.1		1.5
	mdir(96)		3.7		2.1		2.6		1.7		2.1		1.5
	fastpos		3.3		2.0		2.3		1.6		1.8		1.4
	pos(1.0)		3.4		2.0		2.4		1.6		2.0		1.5
	opos		4.0		2.2		2.9		1.8		2.4		1.6
sponza	none	36.6	6.7	16.8	3.3	30.5	5.2	14.2	2.8	26.0	4.2	12.1	2.4
	dir		7.3		3.3		5.6		2.7		4.4		2.4
	mdir(96)		7.5		3.3		5.9		2.8		4.7		2.5
	fastpos		6.6		3.3		4.3		2.5		3.5		2.2
	pos(1.0)		6.5		3.2		5.0		2.7		4.0		2.4
	opos		7.8		3.4		6.7		3.0		5.7		2.7

Table 4: Coherence measurements for secondary rays with different reordering heuristics at recursion level 1,2 and 3. All heuristics use a packet size of 16 rays. For each heuristic, traversal coherence (c_t), traversal packet coherence (p_t), intersection coherence (c_i) and intersection packet coherence (p_i) are given. Better sorting techniques gives slightly increased packet coherence for all scenes, more notably in the traversal measures than in the intersection measures.