Parallel Path Tracing using Incoherent Path-Atom Binning

David Coulthurst
University of Bristol,
Davec@cs.bris.ac.uk

Piotr Dubla
Warwick Digital Laboratory,
P.B.Dubla@warwick.ac.uk

Kurt Debattista
Warwick Digital Laboratory,
K.Debattista@warwick.ac.uk

Simon McIntosh-Smith
ClearSpeed Technology,
Simon@clearspeed.com

Alan Chalmers
Warwick Digital Laboratory,
A.G.Chalmers@warwick.ac.uk

Abstract

Current parallel graphics algorithms minimise memory access latency by tracing packets of coherent rays. This coherency, however, breaks down after several bounces, and is unsuited to acceleration techniques such as selective rendering. This paper presents an unbiased path tracing algorithm which is insensitive to the coherency of the rays traced, allowing it to run on diverse architectures including massively SIMD processors. Bins of path-atoms are created and processed to form a path tracing circular buffer. Latency is hidden by n-buffering the load/save operations between bins. We demonstrate our approach as an implementation on the massively parallel SIMD architecture, the ClearSpeed CSX600.

Keywords: path tracing, global illumination, parallel graphics

1 Introduction

The Rendering Equation as first presented in [Kajiya 1986] described all the illumination at any given point in a scene via an integral equation representing all the reflections in the scene. In the same paper the concept of path tracing was introduced, a stochastic method for solving the rendering equation. Non branching paths from the camera to light sources are traced through each intersection with the scene geometry, as compared to Whitted ray tracing [Whitted 1980] where at each intersection both specular and shadow rays are sampled. In [Cook et al. 1984] this is extended such that at each intersection, the entire hemisphere is sampled to gather an estimate. Path tracing requires many more samples per pixel than ray-tracing, but delivers an unbiased result. Bi-directional path tracing was independently developed by both [Lafortune and Willems 1993] and [Veach and Guibas 1994], after noting that not all significant light transport paths are easily found from starting at the camera.

Both path tracing and ray tracing require a lot of computational power to achieve a high-fidelity result. Traditional methods to reduce computational time by parallel processing have exploited coherence. However, such techniques limit the use of other acceleration techniques, such as selective rendering, and are not well suited to modern wide-width SIMD architectures.

In this paper we present a novel approach which is able to exploit the performance of modern wide-width SIMD processors and significantly reduces latency, not through coherence, but by data buffering. We reformulate the recursive nature of rendering into a set of basic atomic operations, which are inherently suited to parallelisation. The current way of formulating rendering does not map itself well to all architectures, and we show that if we break the computation into simple atomic instructions we can adapt and group such instructions according to the hardware requirments.

We demonstrate our approach on the ClearSpeed CSX architecture, a massively-parallel SIMD machine. We show the latency can be reduced with careful grouping of atomic instructions into bins. This allows us to reduce latency and improve perfromance without relying on the coherence commonly associated with fast ray tracing methods, enabling our method to be used for multiple bounce global illumination (GI) solutions and selective rendering.

2 Related Work

Previous parallel rendering work has tended to focus on ray tracing, and on the use of commodity CPUs or GPUs. In [Purcell 2004], significant speedup was achieved through adapting the ray tracing algorithm to a stream processor model. Purcell also showed in [Purcell et al. 2003] how the photon mapping algorithm could be adapted to run on comodity GPUs. However this is only an approximation of true GI methods.

An alternative approach is to design hardware specifically for tracing rays. In [Sven Woop and Slusallek 2005] a hardware ray-tracer was implemented. Ray-scene intersection using k-d trees, a programmable shading unit and the ability to handle the recursion necessary for ray tracing each have dedicated hardware on the chip. Specific hardware has the disadvantage of needing chip redesign if a new algorimthm emerges, such as the new acceleration structures being developed which are optimised for dynamic scenes [Wald et al.].

Fast ray-tracing has now become possible on commidity PCs. The concept of ray packets [Wald et al. 2001] has come to dominate attempts to harness the new SIMD instructions in modern CPUs. The original techniques of grouping rays into coherent packets to make use of the 4-SIMD CPU instructions has been extended to use much larger packets. These techniques form a conservative bound to the ray packets and use this to avoid unnecessary intersection tests and traversal steps. Dmitriev et al. [Dmitriev et al. 2004] first proposed this for triangle intersections, with Reshetov et al. [Reshetov et al. 2005] extending this to kd-tree traversal. This was then developed further to both grids [Wald et al. 2006] and BVHs [Wald et al. 2007a]. However, all these techniques fundamentally rely on using coherent ray packets.

Perceptually adaptive rendering techniques have achieved significant performance improvements for global illumination algorithms by exploiting knowledge of the human visual system [Myszkowski et al. 2001]. In particular, selective rendering, computes those areas of a scene to which a viewer is attending in high quality. The remainder of the scene is rendered at a much lower quality, and thus at a much lower computational cost, without the viewer being aware of this quality difference [Yee et al. 2001; Debattista 2006; Chalmers et al. 2006]. Adaptive techniques in general are not naturally coherent [Dubla et al. 2008] as the viewer may be attending to non-coherent regions throughout the scene.

3 Incoherent Path-Atom Binning Theory and Framework

Current parallel ray tracing techniques have become very focused on the concept of ray packets, as introduced in [Wald et al. 2001]. The original intent of ray packets was to minimise the latency of loading scene data before processing it. While this technique works well for narrow-width SIMD processors and primary rays, for large width SIMD global illumination it has severe drawbacks. Ray packets provide speed up where up to 8×8 and 16×16 packets are used, usually however this breaks down for larger packets, with a consequential loss of performance. Also, the coherence breaks down quickly, and the current work focuses on a single bounce at most [Wald et al. 2007b]. In [Boulos et al. 2007], the authors explored whether packetised ray tracing techniques could be extended to several bounces and non specular effects. The result shows that current packetisation methods can be applied, however once again it breaks down for higher SIMD widths and, in addition, does not provide an unbiased result.

Considering a processor architecture such as the CSX, the SIMD width is large, 96 wide for the CSX600 chip used for the experiments. This SIMD width consists of 96 "processing elements" or PEs. Each PE is similar to a VLIW processor, and it is the large number of PEs operating in parallel that gives the processor its computational power. With SIMD width this large, and likely to grow larger in the future, different ways for harnessing the increased computational power massively SIMD chips bring, need to be found. The original intent of ray packets is to minimise the latency of loading scene data before processing it. In our approach, we separate path generation from path tracing which allows a binning structure to be created that is inherently suited to parallelisation, by using several bins of path-atoms. This allows us to hide latency with n-buffering the load/save operations, while being insensitive to the coherence of the rays. The insensitivity to coherence means this approach is inherently more scalable as SIMD width increases.

3.1 Path tracing and Path Generation

Traditionally path-tracing is recursive with respect to calculating the final radiance value for each path. However, the generation of the path from the camera to a light source can be separated from calculating the actual radiance value. Further to this, the shading of each vertex can be separated from the calculation of the radiance value. Indeed, the calculation of the radiance value is only a very small part of the total calculation, shown to be less than 1% (see results) of our computation in our experiments. Assuming pertinent data of each vertex of the path is stored during calculation, the radiance value can be calculated at any point after the path is generated. The path generation process can be viewed then as a loop, rather than a recursive process, following the pseudo code below.

while $path \neq completed$ **do**
  intersect ray with scene
  generate shading details for intersection and new direction from brdf
  record new vertex in path
end while

Once the path has been generated, each vertex can be shaded in parallel. Finally the radiance value of the path is calculated using the normal path tracing recursive algorithm. However, as all the material contributions, BRDFs, etc are already known, this calculation is very small (less than 1%).

3.2 Binning Path-Atoms to form a Circular Buffer

The circular nature of path generation becomes significant when attempting to parallelise path tracing, due to the variable length of paths. If we are dealing with a recursive algorithm, we must recurse down to the level of the longest of the paths we are considering, not calculating for the ones that have a shorter path length. This has the obvious disadvantage of leaving the PE's of the shorter paths idle while the longer paths are computed. If the radiance value calculation is postponed until after the paths are generated, we have a better option. Each time the end of a path is reached on a PE, it simply loads a new ray from the begining of a new path and starts tracing.

This leads to a circular buffer model for generating the paths, whereby in each loop the rays are intersected with the scene. The intersection is subsequently processed and new direction calculated. The new vertex is recorded and then either the new direction formed to a ray, or a new ray added. This leads to three separate bins with operations to move between them, as shown in Figure 1. Each time a path is completed, the path is read out to another buffer to be processed and the ray corresponding to the next path is fed in. The algorithm loops around this buffer continuously, until all the paths required have been calculated.

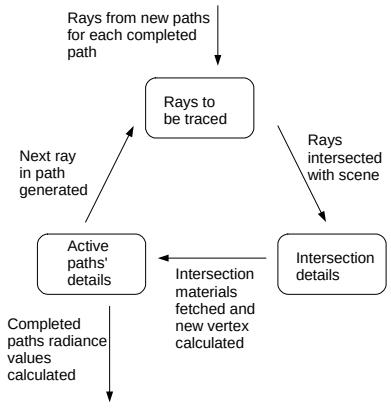

Figure 1: Circular Buffer

One elegant feature of this algorithm is the that of replacing a ray which completes half way through, but the rest of the bin does not. Instead of having to close the gap this leaves to keep the PE's all with data to process, a new ray can simply be dropped into its place and the corresponding path space in the path bin started. This removes the need for book keeping of which bin position corresponds to which path. Another advantage is that none of the path needs to be loaded to add a new vertex. Instead the number of vertices each path has is kept, and the new vertex is saved into the following position in the path bin.

3.3 N-Buffering to hide latency

The memory on the ClearSpeed card consists of two parts, mono memory and poly memory. Mono memory is a large block of shared memory, similar in size and use to the RAM in a PC used by a commodity CPU. The poly memory is a small piece of private memory on each PE, and can be viewed as similar to cache memory on a CPU. That is, data to be processed on the PEs must be loaded from mono memory to the poly memory of the PE it is to be processed on before it can be used. The path-atom bins are stored in their entirety in mono memory on the card. To process a SIMD width block of

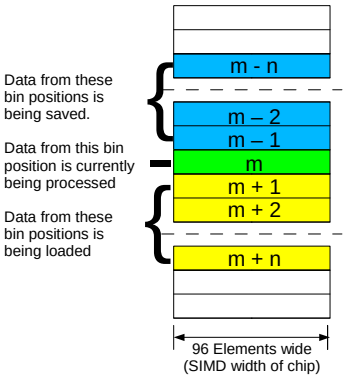

Figure 2: N-buffering allows the proceeding n results to be saved and subsequent n blocks of data to be loaded while the current bin position is processed

path-atoms, they must be loaded from mono to poly memory, processed, and the desired results saved back from poly to mono memory. In this way, the algorithm cycles through all of the path-atoms in each bin, loading-processing-saving in SIMD width chunks of the bin. First the ray bin is processed thus, then the intersection bin, then the path bin. The data to be loaded may depend on the result of a previous bins operation. For example the material needed for shading depends on the object intersected. A small piece of data is kept in poly memory between each bin to facilitate this. In the case of loading the material to be shaded, the intersected object's number would be kept in poly memory, and the address of the material to be loaded calculated from it when the next bin is processed.

The constant loading and saving of data during the processing of the bins introduces a large amount of latency, as the card has to wait for the I/O operations to complete. Part of the speedup achieved using ray packets is through having coherent rays that intersect the same objects as they are traversed. Thus less scene data needs to be loaded from RAM to the cache, as each ray can be intersected with the same set of objects that are already in the cache. Instead of using coherency as a way of minimising latency, the bin structure allows the loading and saving to be n-buffered to offset the latency. While bin position m is being processed, $m+1, m+2, \cdots, m+n$ are being loaded from memory. Similarly for saving, while bin position m is being processed, $m-1, m-2, \cdots, m-n$ are being saved to memory. For n-buffering to work, the bins must be at least $n \cdot (number of PEs)$ in length, so that a result being written to memory is not being read from memory. Figure 2 shows how it works in the general case. N-buffering can also be used to wrap loads/saves around between each type of bin. As the final n bin positions of a preceding bin are processed, the first n of the following bin are being loaded.

By using n-buffering, the latency introduced by the load and save operations necessary to process each block of path-atoms is offset, being carried out concurrently with the processing of earlier or later blocks respectively. As the latency is offset this way, there is no need for the rays being traced to be coherent. This gives the primary advantage of our method over ray-packets. The length of the paths, and where the paths start is entirely arbritrary. When using unbiased methods such as path tracing, where the rays do not keep coherence, this allows us to harness the power of wide width SIMD architectures. Further more, arbritraty path segments can be traced as coherence is no longer an issue. In the case of bi-directional path tracing, arbritrary and incoherent visibility tests are needed to check path validity, and path-atom binning fits in with this too. As well as this, methods such as selective rendering that use arbritrary numbers of rays per pixel, often very low, are unsuited to coherent packet tracing, as the benefits of selective rendering are based on tracing fewer rays and interpolating the results. In these cases again the method of path-atom binning is advantageous.

4 Implementation

Our implementation runs on the ClearSpeed CSX co-processor architecture. It is a multi-threaded array processor (MTAP), normally utilised for traditional High Performance Computing (HPC) topics such as scientific modelling. A MTAP co-processor shares some characteristics of a multi-core CPU, and some of a stream processor such as a GPU. It has a standard RISC control unit with instruction fetch, cache and I/O mechanisms. Additionally it has the main block of so called 'processing elements' or PEs. Each of these PEs consists of a register file, 6Kbytes of SRAM,a high speed I/O channel to adjacent PEs, an integer ALU and a 64-bit FPU. The 64-bit FPU, which implements full IEEE double precision, is responsible for the high throughput of 50 GFlops double precision. Branching in code is handled via an enable state, in a method similar to predicated instructions in some RISC CPUs. Figure 3 shows an overview of the architecture.

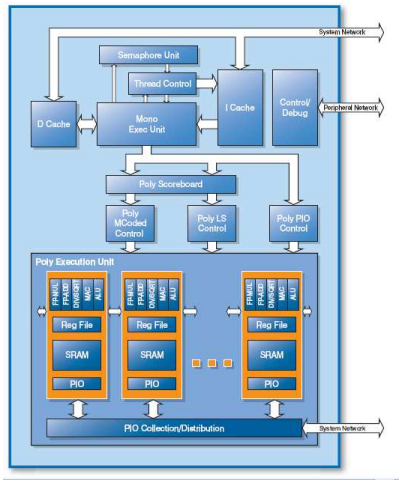

Figure 3: ClearSpeed CSX chip architecture overview

From a parallel graphics perspective the CSX architecture offers an interesting mix of the advantages of a stream procesor and multi-core CPU. The current GPU architecture for comparison is Nvidia's G80 achitecture. Although advertised as a 128 core mutli threaded scalar processor, in effect is a 32 width SIMD processor. This is because each "warp" of 32 threads is processed in SIMD. To achieve the high performance the hardware is capable of, all these threads must run the same instructions. So when considering high throughput applications such as parallel rendering, the G80 architecture should be considered a 32 width SIMD processor, not a 128 core scalar architecture. The CSX600 chip by comparison has a SIMD width of 96, which as described later is advantageous for the wide SIMD algoritms detailed in this paper. A second point is that the G80 architecture doesn't natively perform double precision, but supports it through multiple cycles of single precision, while the CSX architecture FPUs are double precision throughout.

From an algorithmic view, the subtle differences between a stream processor and MTAP are significant. The stream processor model consists of a host computer offloading very specific fine grained chunks of work to the processor, processing it and returning it to shared memory. The MTAP model is much closer to how a normal

CPU runs - the RISC control unit with a set of PEs model allows a complete program to be run on the board. The onboard memory is around that of a desktop PC - 1 to 4 GB, so, except for the most complex scenes, the entire scene description can be held on the card rather than on the host PC. Dedicated high speed buses between the PEs is similar to those found in multi-core chips, and allows algorithmic subtlety that isn't possible on a stream processor such as the G80. For example in the case where only n of the PEs succeed in a task. It is often the case that the spread of the succesful n is random, and only those succesful are to be stored into a bin, for further use. The high speed PE to PE buses can be used within algorithms to assign ascending numbers to each succesful PEs. In turn these ascending numbers can be used in addressing for saving results back to main memory. Alternatively these results can be retained in PE memory and while the PEs with unsuccesful task can load more data. This is in contrast to a stream processor where the results would have to all be unloaded, and the succesful ones reloaded along with fresh data to fill the gaps.

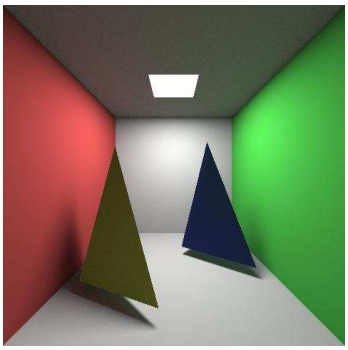

Figure 4: Cornell Box Test scene, rendered in 480×480 pixels, 1000 samples per pixel

4.1 Results

We used a simple Cornell Box scene (Figure 4) to test different aspects of the algorithm. The first thing we tested was the effect on rendering times of n-buffering the load and save operations. Table 1 shows our test results when using values of n from 0 to 3. 0-buffering simply means that each load or save operation was carried out when it occured in the code, with no attempt to offset it. To make comparison easier, the time recorded is the time taken for 1,000,000 path segments. Russian roulette was used to terminate paths. We ran our algorithm only tracing the path segments, tracing and direct lighting each path segment, and running the full rendering algorithm.

As can be seen in the results, as the value of n rises, the render time decreases. The gain in render time stops increasing once you start to buffer more than once (double buffering). This is due to the load/save operations acting on small pieces of data, around 10 double precision floats per PE. Our results show that computing the path segments with direct lighting forms 99% of the time taken, with only 1% of the computation taken up with recursively calculating the radiance values.

The second thing we tested was the effect of buffering different combinations of the load / save operations. To test whether it was either the load or save operations that slowed the algorithm more, we buffered only the loads, only the saves, both and neither. Table 2 shows our results, again showing the time taken to compute 1,000,000 path segments. As can be seen, buffering either the load or save operations gives a performance increase of around 10% compared to not buffering, whereas buffering both gives just over 20% performance increase.

Level of Buffering	Path Segments	Path Segment & Direct Lighting	Full Computation
0	7.63	16.84	16.97
1	5.94	13.17	13.28
2	5.94	13.21	13.36
3	5.96	13.46	13.61

Table 1: Render times in seconds of differing scenes when varying n in n-buffered computation

Type of Buffering	Path Segments	Path Segment & Direct Lighting	Full Computation
None	7.63	16.84	16.97
Load	6.64	14.81	14.96
Save	6.85	15.49	15.63
Both	5.94	13.17	13.28

Table 2: How number of samples per pixel affects rendering times in seconds

Finally we rendered the scene in Figure 4 to a size of 480×480 pixels, with 1000 samples per pixel. Using double buffering for both load and save operations, the scene took 3036.37 seconds, or 50.6 minutes. This gives an average of just over 150,000 incoherent path segments a second, unshaded, or 750,000 path segments a second with each path segments with a single shadow ray per path segment.

5 Conclusions and Future Work

We presented a new approach to the classic ray-tracing problem of memory access latency. Instead of grouping together packets of coherent rays, with all the limitations this implies, we reworked the problem to set aside the recursive calculation portion from the main computation. This allows us to bin the operation as a circular buffer. This has several important benifits. Firstly it makes it simple to keep the hardware with enough work so that it doesn't stand idle. Secondly it allows us to hide the latency using n-buffering. Thirdly, as latency is not coming from coherency, the type of rays processed is not important. Thus multiple bounces suffer no penalties, so unbiased techniques are possible. Also techniques that naturally result in incoherent ray processing can be used, such as selective rendering, and time-constrained rendering [Debattista 2006]. An average of 150,000 incoherent path segments a second, traced and shaded, was achieved for a simple Cornell box scene. The code used to generate these figures is not optimised, instead this is proof of concept work, to test our alternative approach to latency reduction.

Our method could be extended simply with the implementation of bi-directional path tracing [Lafortune and Willems 1993]. This could be achieved by separately tracing both a forward and light path though the scene. Indeed, both paths could be stored contiguously in memory, with the starting pint for the light and eye paths calculated at the same time. Then as soon as the light path is complete, the pointer to the path can be moved on to the start of the eye path, and processing continued. Once both paths are traced, each vertex from the forward path could be combined with every vertex in the light path. The required visibility test could be handled within the circular buffer framework.

Our method currently has not addressed the issue of acceleration structures, in terms of both building and traversing. Current work

favours the use of bounding volume hierarchies (BVH) [Wald et al. 2007a] with recent work on their use on parallel architectures [Ize et al. 2007]. Adapting this work to run on the ClearSpeed architecture is a clear next step. The current method already loads the object data to be intersected, similar to having a single leaf node. To extend to using an acceleration stucture, the intersection operation would have to be extended. A circular buffer similar in principle to the overall method presented here could be used, with an additional temporary bin would be needed. Each SIMD width block of rays and current leaf to be intersected with would be loaded and intersected. Those intersecting would have their details saved as in the current methods. The non-intersecting rays would be traversed to find the next leaf to be intersected with, and this result saved to the the temporary bin. Once all of the rays had been tested once, the temporary bin would have all the non-intersected rays and the next leaf for each ray to be tested against. The process could be repeated again, with the original bin being used to store the next round of un-intersected results. This would continue, the temporary and original bin swapping each time and with the number of non-intersecting rays falling until all have been intersected fully with the scene. The bins would only contain rays that still needed intersecting, meaning a full block of rays and objects can be loaded each time, keeping the processor fully working. As well as the rays and objects to be intersected, enough of the acceleration structure to traverse the ray to find the next leaf to intersect is necessary. Object hierarchies such as BVH may be the best suited, as the internal nodes of a sub-tree could be loaded (without the object data in the leaves). Clearly the next step is to research how large a sub-tree could be loaded, what if any impact this has on the algorithm (for example if only half a sub-tree can be loaded, how could the cases where this is insufficient be handled best).

References

BOULOS, S., EDWARDS, D., LACEWELL, J. D., KNISS, J., KAUTZ, J., WALD, I., AND SHIRLEY, P. 2007. Packet-based Whitted and Distribution Ray Tracing. In *Proceedings of Graphics Interface 2007*.

CHALMERS, A., DEBATTISTA, K., AND DOS SANTOS, L. 2006. Selective rendering: Computing only what you see. In *Graphite 2006*, ACM SIGGRAPH, 123–131.

COOK, R. L., PORTER, T., AND CARPENTER, L. 1984. Distributed ray tracing. *SIGGRAPH Comput. Graph. 18*, 3, 137–145.

DEBATTISTA, K., SUNDSTEDT, V., PEREIRA, F., AND CHALMERS, A. 2005. Selective parallel rendering for high-fidelity graphics. In *Proceedings of Theory and Practice of Computer Graphics 2005*, Eurographics Association, 59–66.

DEBATTISTA, K. 2006. Selective rendering for high-fidelity graphics. *PhD thesis, University of Bristol*.

DMITRIEV, K., HAVRAN, V., AND SEIDEL, H.-P. 2004. Faster ray tracing with simd shaft culling. Research Report MPI-I-2004-4-006, Max-Planck-Institut für Informatik, Saarbrücken, Germany, December.

DUBLA, P., CHALMERS, A., AND DEBATTISTA, K. 2008. An analysis of cache awareness for interactive selective rendering. In *Communications Papers proceedings*, WSCG, V. Skala, Ed.

IZE, T., WALD, I., AND PARKER, S. G. 2007. Asynchronous BVH Construction for Ray Tracing Dynamic Scenes on Parallel Multi-Core Architectures. In *Proceedings of the 2007 Eurographics Symposium on Parallel Graphics and Visualization*.

KAJIYA, J. T. 1986. The rendering equation. *SIGGRAPH Comput. Graph. 20*, 4, 143–150.

LAFORTUNE, E. P., AND WILLEMS, Y. D. 1993. Bi-directional Path Tracing. In *Proceedings of Third International Conference on Computational Graphics and Visualization Techniques (Compugraphics '93)*, H. P. Santo, Ed., 145–153.

MYSZKOWSKI, K., TAWARA, T., AKAMINE, H., AND SEIDEL, H.-P. 2001. Perception-guided global illumination solution for animation rendering. In *SIGGRAPH 2001, Computer Graphics Proceedings*, ACM Press / ACM SIGGRAPH, E. Fiume, Ed., 221–230.

OSULLIVAN, C., H. S. M. Y. M. R. 2004. Perceptually adaptive graphics. *Eurographics 2004, STAR*, 141–164.

PURCELL, T. J., DONNER, C., CAMMARANO, M., JENSEN, H. W., AND HANRAHAN, P. 2003. Photon mapping on programmable graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, Eurographics Association, 41–50.

PURCELL, T. J. 2004. *Ray tracing on a stream processor*. PhD thesis, Stanford, CA, USA. Adviser-Patrick M. Hanrahan.

RESHETOV, A., SOUPIKOV, A., AND HURLEY, J. 2005. Multi-level ray tracing algorithm. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, ACM, New York, NY, USA, 1176–1185.

SVEN WOOP, J. S., AND SLUSALLEK, P. 2005. Rpu: A programmable ray processing unit for realtime ray tracing. In *Proceedings of ACM SIGGRAPH 2005*.

VEACH, E., AND GUIBAS, L. J. 1994. Bidirectional Estimators for Light Transport. 147 – 161.

WALD, I., MARK, W. R., GÜNTHER, J., BOULOS, S., IZE, T., HUNT, W., PARKER, S. G., AND SHIRLEY, P. State of the Art in Ray Tracing Animated Scenes. In *Eurographics 2007 State of the Art Reports*.

WALD, I., BENTHIN, C., WAGNER, M., AND SLUSALLEK, P. 2001. Interactive rendering with coherent ray tracing. In *Computer Graphics Forum (Proceedings of EUROGRAPHICS 2001*, Blackwell Publishers, Oxford, A. Chalmers and T.-M. Rhyne, Eds., vol. 20, 153–164. available at http://graphics.cs.uni-sb.de/ wald/Publications.

WALD, I., IZE, T., KENSLER, A., KNOLL, A., AND PARKER, S. G. 2006. Ray Tracing Animated Scenes using Coherent Grid Traversal. *ACM Transactions on Graphics*, 485–493. (Proceedings of ACM SIGGRAPH 2006).

WALD, I., BOULOS, S., AND SHIRLEY, P. 2007. Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies. *ACM Transactions on Graphics 26*, 1.

WALD, I., GRIBBLE, C. P., BOULOS, S., AND KENSLER, A. 2007. SIMD Ray Stream Tracing - SIMD Ray Traversal with Generalized Ray Packets and On-the-fly Re-Ordering. Tech. Rep. UUSCI-2007-012.

WHITTED, T. 1980. An improved illumination model for shaded display. *Commun. ACM 23*, 6, 343–349.

YEE, H., PATTANAIK, S., AND GREENBERG, D. P. 2001. Spatiotemporal sensitivity and visual attention for efficient rendering of dynamic environments. *ACM Trans. Graph. 20*, 1, 39–65.