Efficient Stack-less BVH Traversal for Ray Tracing

Michal Hapala[1] Tomáš Davidovič[2] Ingo Wald[3] Vlastimil Havran[1] Philipp Slusallek[2]

[1]Czech Technical University in Prague, Faculty of Electrical Engineering
[2]Saarland University and DFKI GmbH
[3]Intel Corp.

Figure 1: The three test scenes used for evaluating our stack-less BVH traversal algorithm, all rendered with path tracing: `Conference Room` (289.9k triangles), `Fairy Forest` (174.1k triangles), and `Sibenik Cathedral` (80.5k triangles).

Abstract

We propose a new, completely iterative traversal algorithm for ray tracing bounding volume hierarchies that is based on storing a parent pointer with each node, and on using simple state logic to infer which node to traverse next. Though our traversal algorithm does *re-visit* internal nodes, it *intersects* each visited node only once, and in general performs exactly the same ray-box tests and ray-primitive intersection tests—and in exactly the same order—as a traditional stack-based variant. The proposed algorithm can be used for computer architectures that need to minimize the use of local memory for processing rays or those that need to minimize the data transport such as distributed multi-CPU architectures.

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

Keywords: ray tracing, bounding volume hierarchy, stack-based and stack-less traversal algorithm.

1 Introduction

Traversing a ray through a hierarchical data structure such as a bounding volume hierarchy (BVH) is usually carried out in a recursive manner by maintaining a stack. Having to maintain a full stack per ray can lead to problems, however, in particular on highly parallel architectures that process many rays in parallel (thus, needing a lot of memory to store all those stacks), or in situations where one needs to move (or suspend/resume) the ray's state in mid-traversal. This has recently prompted several authors to investigate stack-less traversal algorithms that, however, so far either have to perform infrequent restarts of the traversal from the root and/or traverse and intersect more nodes than their stack-based counterparts.

In this paper, we propose a new, completely iterative BVH traversal algorithm that is based on two key ideas: First, we store a parent pointer with each node, which enables us to "go back upwards" in the tree without having to maintain a stack. Second, we use a simple deterministic automaton algorithm with three states to encode the traversal logic that determines which node to traversed next. This is inferred based on one of three traversal states and computes a new state for the next traversal step. The proposed algorithm does *re-visit* internal nodes, but intersects each node only once, and performs exactly the same ray-box intersection tests—and in exactly the same order—as a traditional stack-based traversal algorithm with an axis-based traversal order.

2 Previous work

Approaches to stack-less traversal algorithms fall into three categories: those that perform some sort of restart of the traversal, those that use some sort of links between different nodes, and those that exploit the regularity of the data structure to compute the next node implicitly. Algorithms in the last category only work for certain special cases (like volume data organized in implicit kd-trees [Hughes and Lim 2009]) and will not be considered in this paper.

Foley and Sugerman [2005] have proposed two variants of restart algorithms in the context of kd-trees: *kd-restart*, and *kd-backtrack*. Kd-restart tracks a point along the ray that marks the end-point of the already-traversed ray segment. In each iteration, restart traverses this point all the way from the root to its containing leaf, and intersects those triangles. After processing this leaf it then advances this point to right behind that leaf, and "re-starts" the next iteration from the root. This algorithm corresponds to the original ray traversal algorithm for kd-trees by Kaplan [1985]. To avoid having to do a full re-start after every leaf, kd-backtrack adds bounding boxes and

parent-pointers to each node; the traversal algorithm still advances the current traversal end-point, but, rather than starting all the way from the root, finds the next node by starting at the leaf, and going up to the first node that contains this end-point. This approach was expanded to kd-push-down by Horn et al. [2007], who stores information about the depth-wise lowest node that completely contains the valid intersection interval. Instead of the root node, this node is then used when a traversal is restarted.

These algorithms are not directly applicable to BVHs because BVH nodes can overlap, meaning that each "end-point" could be in multiple leaf nodes. Laine [2010] explains this, and offers an alternative approach to a BVH restart algorithm by using a 32- or 64-bit variable to track which levels of the tree do not need to be traversed any more. Every time a restart occurs the next node is found using this "trail". This in principle works the same as the shortening of the ray, except that the information is saved in a different way. Laine also provides pointers for an efficient implementation that handles the using and updating of the trail with simple bit-wise operations.

The second category of stack-less traversal algorithms utilizes additional information regarding the internal structure of the tree. MacDonald and Booth [1990] (and later, Havran et al. [1998]) have investigates *neighbor-links* (or "ropes") that store, for each leaf node in a kd-tree, a pointer to the subtree that is spatially adjacent to that side (this again works only for kd-trees, in which nodes do not overlap). Whenever the traversal algorithm leaves a leaf node, it determines which of the six sides the ray leaves that node, and follows the respective link. Following neighbor-links means no stack is ever needed; however, adding all those pointers implies a significant memory overhead. The method was also used for GPU based algorithm by Popov et al. [2007].

For BVHs, Smits [1998] proposed an approach in which each node contained a so-called "skip node pointer" that specified which node to traverse next if the ray missed the current node. This approach was later used for a GPU implementation by Torres et al. [2009]. While elegant, this method imposes the same traversal order for all rays, leading to some rays traversing the hierarchy "back-to-front" (which in turn can lead to a significant increase in box and primitive tests). This can be avoided by having each node store a different skip pointer for different ray orientations [Boulos and Haines 2006], but the amount of additional storage required for this traversal algorithm makes this approach impractical.

Our approach basically follows a link-based traversal algorithm, but guarantees the same traversal order as a stack-based variant. It requires only one additional pointer (the parent pointer) per node, and, for commonly used node layouts as the one used by Aila et al., this pointer can squeezed into the existing node layout without increasing total memory consumption.

3 Algorithm outline

Before deriving the logic of a new traversal algorithm, we first specify some assumptions we need for our algorithm. In particular, we assume that:

- we are using a binary BVH, in which all primitives are stored in leaf nodes, and in which each inner node n has exactly two children $c_{LEFT}(n)$ and $c_{RIGHT}(n)$ (also called "siblings"),
- for each node n there is an efficient way to determine its parent $parent(n)$ and sibling $sibling(n)$,
- for each inner node n there is a unique traversal order $(nearChild(n), farChild(n))$ in which its children are traversed. This order varies from ray to ray, but for any given ray may not change during traversal (see Figure 2).

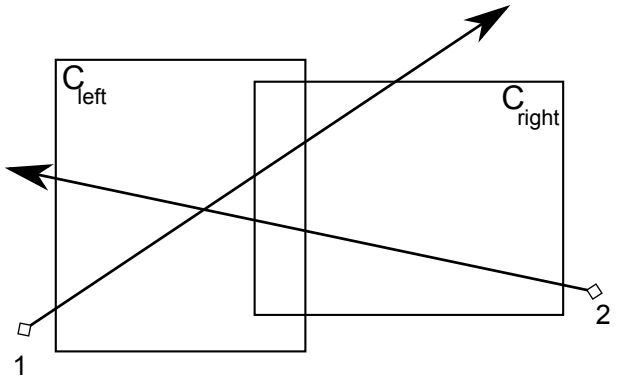

Figure 2: Traversal order example. Ray 1 will first traverse c_{LEFT} and then c_{RIGHT} node, c_{LEFT} for him is *nearChild* and c_{RIGHT} is $farChild$. Ray 2 will traverse the children in the opposite order.

Since we do not want to assume an implicit hierarchy, the only way we can determine a node's parent is to store an explicit parent pointer for this node. This can be done either by storing a separate array of parent pointers, or by squeezing this parent pointer into unused parts of an existing BVH node layout.

For the traversal order there are various different alternatives. One often-used option is to store, for each node, the coordinate axis along which the builder split the parent node, and to use the ray's direction sign in this dimension to determine the two nodes' traversal order. If the split axis information is not available from the builder, one can also use the dimension in which the two child nodes' centroids are widest apart. This separation dimension can be computed on the fly, or stored with each node. In our approach, we use the maximum separation axis, and store this with each node.

Another alternative to determine two sibling's traversal order is to compute the actual distance to the siblings' bounding boxes, and sort them based on distance. This, however, would require to re-intersect both nodes every time we want to determine two siblings' traversal order.

3.1 State logic

Rather than using a stack, our traversal algorithm uses a simple state machine to infer which node to traverse next. To better understand this approach, let us consider a single "parent-plus-two-siblings" configuration. Without loss of generality, let's assume that a ray regards that $c_{LEFT}(n)$ is *nearChild* and $c_{RIGHT}(n)$ is $farChild$.

First, let us iterate exactly how any recursive traversal algorithm works in general: After having successfully intersected the parent node, traversal first goes to *nearChild*, and does a ray-box test for this node. If this node is missed, traversal immediately proceeds to $farChild$; if not, the node is "processed", either by intersecting its primitives (in case it is a leaf), or by recursively entering this node's subtree (in case it is an inner node). Once *nearChild* is fully processed, traversal resumes with $farChild$ in the same way as if the node had been missed. For $farChild$, exactly the same sequence of events takes place (test the node, and either skip or process it) except that the next node after $farChild$ is *parent*.

From this we can observe that there are only three ways ("states") of how any given node can be reached during recursive traversal: from its parent (on the way down, when entering *parent*'s subtree); from its sibling (when going from *nearChild* to $farChild$); or from

one its own children (after having traversed its own subtree). Let us call these cases *fromParent*, *fromSibling*, and *fromChild*. Now, we can formulate the above traversal logic depending on exactly these three states (see Figure 3).

In the *fromChild* case the *current* node was already tested when going down, and does not have to be re-tested. The *next* node to traverse is either *current*'s sibling $farChild$ (if *current* is *nearChild*), or its *parent* (if *current* was $farChild$).

In the *fromSibling* case, we know that we are entering $farChild$ (it cannot be reached in any other way), and that we are traversing this node for the first time (i.e. a box test has to be done). If the node is missed, we back-track to its *parent*; otherwise, the *current* node has to be processed: if it is a leaf node, we intersect its primitives against the ray, and proceed to *parent*. Otherwise (i.e. if the node was hit but is not a leaf), we enter *current*'s subtree by performing a *fromParent* step to *current*'s first child.

Finally, in the *fromParent* case, we know that we are entering *nearChild* and we do exactly the same as in the previous case, except that every time we would have gone to *parent* we go to $farChild$ child.

The corresponding pseudo-code is in Listing 1. In that code, every line with a state change includes a commentary associating it with an image in Figure 3.

```
void traverse(ray, node)
 int current=nearChild(root);
 char state=fromParent; // we start by going down
 while(true) {
   switch(state) {
    case fromChild:
    if (current==root) return; // finished
    if (current==nearChild(parent(current))) {
      current=sibling(current); state=fromSibling; //(1a)
    }
    else {
      current=parent(current); state=fromChild; //(1b)
    }
    break;
    case fromSibling:
    if (boxtest(ray, current)==MISSED) {
      current=parent(current); state=fromChild; //(2a)
    }
    else if (isLeaf(current)) {
      // ray-primitive intersections
      processLeaf(ray, current);
      current=parent(current); state=fromChild; //(2b)
    }
    else {
      current=nearChild(current); state=fromParent; //(2a)
    }
    break;
    case fromParent:
    if (boxtest(ray, current)==MISSED) {
      current=sibling(current); state=fromSibling; //(3a)
    }
    else if (isLeaf(current)) {
      // ray-primitive intersections
      processLeaf(current);
      current=sibling(current); state=fromSibling; //(3b)
    }
    else {
      current=nearChild(current); state=fromParent; //(3a)
    }
    break;
  }
 }
```

Algorithm 1: Basic state-based traversal. Also see Figure 3.

It is relatively straightforward to show that the proposed traversal algorithm is correct by considering all the cases that can occur when visiting a node, either an interior node or leaf. We have to consider the state in which we process a node to make the proof on correctness complete.

3.2 Comparison to Stack-Based Traversal

Compared to a stack-based traversal algorithm with the same axis-based traversal order heuristic the above code performs *exactly* the same box tests and triangle tests as the stack-based one (except that it never tests the root node), and also performs those in exactly the same order. Statistically, the biggest difference is that some inner nodes are "accessed" (i.e. read from memory) twice—once on the way down, and once on the way up, and that the traversal order heuristic (*nearChild/farChild*) may be executed twice. As long as this heuristic is cheap, however, the latter is not an issue, and apparently roughly as expensive as performing stack operations instead (though in a clever implementation a stack-pop *can* skip multiple levels at once). Reading some nodes twice, however, does increase bandwidth, in particular when caches are so small that this node is not found in cache.

4 CUDA Implementation

While the previous section's naïve state machine code can be taken almost literally on a traditional CPU, implementing the traversal algorithm in a CUDA or OpenCL requires some changes. If enough rays in warp are in different states of their traversal, the warp will eventually execute all three traversal cases in each iteration. For a more efficient implementation we realize that many of the traversal cases actually perform very similar work (e.g. *fromParent* and *fromSibling* differ only in which node to traverse next). By reordering the code such that different cases' operations are performed in the same basic block we can essentially "share" these operations among threads that are nominally in different states.

We have integrated this algorithm in the freely available CUDA ray tracer by Aila et al. [Aila and Laine 2009; Karras et al. 2009].

```
...
near = nearChild(current);
far  = farChild(current);
// already returned from far child - traverse up
if (last == far) {
 last = current; current = parent(current);
 continue;
}
// if coming from parent, try near child, else far child
tryChild = (last == parent(current))?near:far;
if (boxtest(ray, current) {
 // if box was hit, descend
 last = current; current = tryChild;
}
else {
 // if missed
 if (tryChild == near) {
  // next is far
  last = near;
 }
 else {
  // go up instead
  last = current; current = parent(current);
 }
}
```

Algorithm 2: CUDA state-based traversal.

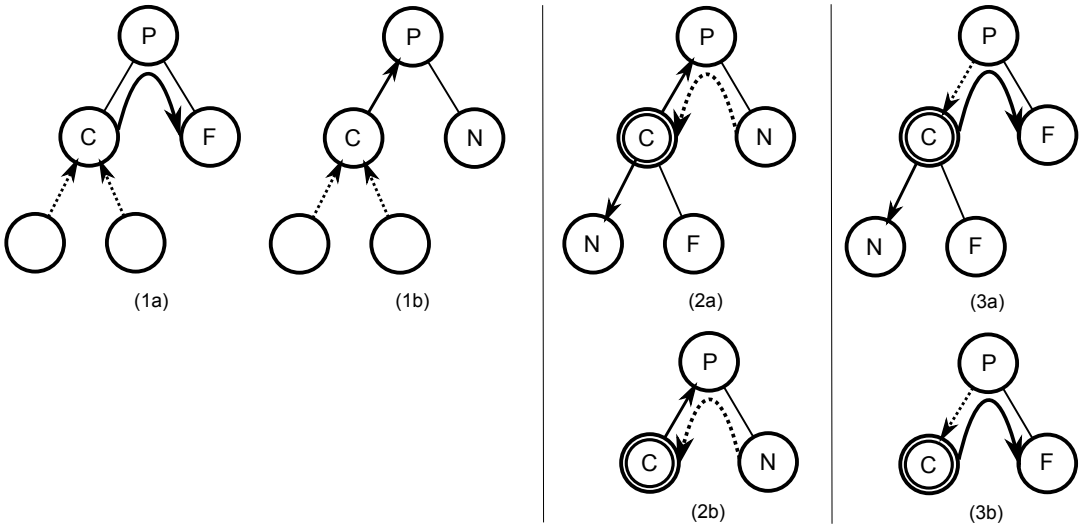

Figure 3: Traversal states: (1a) and (1b) *fromChild*, (2a) and (2b) *fromSibling*, and (3a) and (3b) *fromParent*. Legend: C is the current node, P is parent of C, N and F signify near and far nodes with regard to the orientation of the current ray and the parent of N or F. Dotted lines show the traversal we have taken into the current node whereas thick lines show next traversal step (either one or two possible nodes). The cases (2b) and (3b) are the cases where current node is a leaf. Doubled rings signify where a ray-box test is needed to decide where to traverse next.

For alignment reasons this implementation used a node layout with two un-used 32-bit data words, which we can use to store parent pointer and separation axis without increasing the memory footprint. We implemented two separate CUDA kernels, one optimized for Tesla [NVIDIA 2007], one for Fermi [NVIDIA 2009]. There were no major differences concerning those two implementations, except for the ones already present in the original Aila code (i.e. the Tesla version uses persistent threads, and stores BVH nodes in texture cache).

A (slightly simplified) version of our CUDA implementation's traversal code is given in Listing 2.

5 Results

To evaluate our algorithm we have implemented it on two different architectures—NVidia Tesla (GT260), and NVidia Fermi (GT470)—using CUDA Toolkit 3.2 [NVIDIA 2011]. The reference implementation is from [Karras et al. 2009].

For our measurements we use three test scenes (Conference Room, Fairy Forest, and Sibenik Cathedral, see Figure 1); each scene is rendered at 512×512 pixels, using a Monte Carlo path tracer [Kajiya 1986] performing a fixed number of bounces (no Russian Roulette termination is used). At each bounce, if the path did not leave the scene we shoot one visibility ray to the light source. For the sake of simplicity, we only report results for for two different path length settings (4 and 8); other path length settings produce comparable results. Each frame shoots one path per pixel, with convergence happening through accumulation of successive frames.

In Table 1 we report, for both 4 and 8 bounces, some architecture independent statistics for the reference stack-based algorithm: the number of ray-triangle intersections per ray N_{IT}, the number of traversal steps per ray N_{TS}, and the number of visited leaves per ray N_L. In the reference algorithm, each traversal step performs exactly two box tests (stack operations are not included).

In Table 2 we report the differences between the stack-based algorithm and the proposed stack-less algorithm. Since the reference algorithm actually uses a slightly different traversal order (rather than using the separation axis, Aila [2009] computes the intersection with bounding boxes of both children when visiting interior nodes, and picks the closer of both) the number of box tests is different by a small amount. In this metric our algorithm is, in fact, even slightly better than the reference algorithm.

The stack-less algorithm does, however, visit more than twice the number of nodes than the stack-based algorithm does. To some degree, this is due to a different way of counting traversal steps (we process individual nodes, while for inner nodes the reference algorithm processes two children at once); but at least partially, it is because we visit some nodes twice. These re-visits are cheaper than the first visit, but nevertheless require more memory access as well as more evaluations of the traversal logic, neither of which comes for free. Because of these increased node visits, the stack-based reference algorithm is roughly 30% faster for our path tracing application, independent of the maximum path length used (though we only report numbers for 4 and 8 we tested other path lengths as well).

Discussion

Stack-based approaches are natural for commodity CPU based systems where caches are large, and where the architecture is optimized for high spatial and temporal locality of data access; and for architectures and applications where a stack can be realized efficiently the stack-based traversal still performs best.

For scenarios where storing a complete stack per ray is a problem, however, the stack-less variant provides an interesting option: The stack-less algorithm requires only the description of the ray (ray origin and direction of 3 floats each), the distance along the ray to the nearest object so far (1 float), the pointer to the currently traversed node, and the traversal state (2 bits). This is an interesting feature for architectures where the number of rays is high (and where keeping the stack along with the ray is expensive), such as special hardware architectures for ray tracing [Woop et al. 2005; Caustic Graphics, Inc. 2009]. Another interesting option for stack-less algorithms are distributed memory architectures with many CPUs, where the

Scene	GPU architecture	trav.alg	Path Tracing 4				Path Tracing 8			
			N_{IT} [-]	N_{TS} [-]	N_{IB} [-]	N_L [-]	N_{IT} [-]	N_{TS} [-]	N_{IB} [-]	N_L [-]
Conference Room	Tesla/Fermi	stack based	7.01	21.16	42.33	2.52	6.78	21.10	42.20	2.45
Fairy Forest	Tesla/Fermi	stack based	8.41	25.39	50.78	1.84	8.62	26.82	53.64	1.92
Sibenik Cathedral	Tesla/Fermi	stack based	5.24	26.69	53.38	1.63	5.53	26.82	53.64	1.69

Table 1: Platform independent statistics for the reference stack-based GPU traversal algorithm.

Scene	GPU arch.	trav.alg	Path Tracing 4				Path Tracing 8			
			ratio active threads[%]	ΔN_{TS} [%]	ΔN_{IB} [%]	$\Delta Perf$ [%]	*ratio active threads*[%]	ΔN_{TS} [%]	ΔN_{IB} [%]	$\Delta Perf$ [%]
Conference Room	Tesla	stack based	99.01	0	0	0	97.80	0	0	0
Conference Room	Tesla	stack-less	99.01	+134.8	-5.39	-28.53	97.80	+134.8	-5.36	-28.94
Conference Room	Fermi	stack based	99.01	0	0	0	97.80	0	0	0
Conference Room	Fermi	stack-less	99.01	+134.8	-5.39	-30.3	97.80	+134.8	-5.36	-28.97
Fairy Forest	Tesla	stack based	46.87	0	0	0	28.32	0	0	0
Fairy Forest	Tesla	stack-less	46.87	+138.61	-3.54	-31.24	28.32	+135.67	-3.65	-29.01
Fairy Forest	Fermi	stack based	46.87	0	0	0	28.32	0	0	0
Fairy Forest	Fermi	stack-less	46.87	+138.61	-3.54	-35.7	28.32	+135.67	-3.65	-35.58
Sibenik Cathedral	Tesla	stack based	97.67	0	0	0	96.30	0	0	0
Sibenik Cathedral	Tesla	stack-less	97.67	+132.94	-5.12	-31.96	96.30	+133.32	-5.08	-33.62
Sibenik Cathedral	Fermi	stack based	97.67	0	0	0	96.30	0	0	0
Sibenik Cathedral	Fermi	stack-less	97.67	+132.94	-5.12	-27.99	96.30	+133.32	-5.08	-29.06

Table 2: Performance results for stack-based and stack-less GPU traversal algorithm, respectively, for both Tesla and Fermi.

scene data is distributed over the whole system, and where the rays are passed from one CPU to another during traversal. Such a system was suggested for example by Kato et al. [2002].

Finally, having only a small amount of state per ray is interesting for algorithm that work by *re-ordering* rays, which usually requires temporarily saving a ray's state, and re-storing it at a later time. Such algorithms have recently been proposed by a variety of authors [Navratil et al. 2007; Gribble and Ramani 2008; Moon et al. 2010], and would be particularly interesting for special hardware solutions [Ramani et al. 2009].

6 Conclusion

We have presented a traversal algorithm for BVH that does not need a stack and hence minimizes the memory needed for a ray. It is based on a three-state logic and keeping the pointer to the parent for all nodes. The proposed algorithm can be used efficiently in approaches where we process many rays in parallel. In these cases we need to minimize the book-keeping data for individual rays either locally or for data transfer among processing units.

The recently published BVH stack-less algorithm by Laine [2010] traverses approximately the same number of additional nodes, but also does ray-box intersection for every of these visited nodes. The proposed algorithm, on the other hand, does only the necessary minimum of ray-box intersections, as would a stack-based algorithm do.

We have shown the results when the traversal algorithm implemented in CUDA for Tesla and Fermi architecture as the most commonly accessible highly parallel architectures. We show that for the contemporary GPU architectures the traversal algorithm with local stack is more efficient than stack-less algorithm that needs twice as many traversal steps. Although employing a stack demands frequent access to memory, modern GPUs can run thousands of threads at once and effectively hide memory latencies.

There are however architectures or applications where having minimal memory per ray is paramount. These are e.g. special hardware units, memory distributed CPU/GPU architectures designed for tracing rays, where the scene is distributed among different processing units or ray-reordering traversal schemes.

In future work, we would like to test the proposed algorithm on highly parallel CPU based architecture with distributed memory and for the schemes that use ray-reordering to optimize for the performance of ray tracing.

7 Acknowledgment

We want to thank the authors of the scenes we have used in our work: *Fairy Forest* is from the Utah Animation Repository (`http://www.sci.utah.edu/~wald/animrep/`), *Conference Room* is from Greg Ward's Radiance rendering package (`http://radsite.lbl.gov/radiance/`); and *Sibenik Cathedral* has been modeled by Marko Dabrovic (`http://hdri.cgtechniques.com/~sibenik2/`). We would also like to thank Tero Karras, Timo Aila and Samuli Laine for releasing their CUDA ray tracer source codes into the public domain.

This work has been partially supported by the Ministry of Education, Youth and Sports of the Czech Republic under research programs MSM 6840770014, LC-06008 (Center for Computer Graphics) and MEB-060906 (Kontakt OE/CZ), the Grant Agency of the Czech Republic under research program P202/11/1883, the Grant

Agency of the Czech Technical University in Prague, grant No. SGS10/289/OHK3/3T/13, German Research Foundation (Excellence Cluster 'Multimodal Computing and Interaction') and Intel Visual Computing Institute.

References

AILA, T., AND LAINE, S. 2009. Understanding the Efficiency of Ray Traversal on GPUs. In *Proc. High-Performance Graphics 2009*, 145–149.

BOULOS, S., AND HAINES, E. 2006. Notes on Efficient Ray Tracing. *Ray Tracing News 19*.

CAUSTIC GRAPHICS, INC. 2009. CausticRT platform. http://www.caustic.com/.

FOLEY, T., AND SUGERMAN, J. 2005. KD-tree acceleration structures for a GPU raytracer. In *Proceedings of Graphics Hardware*, 15–22.

GRIBBLE, C., AND RAMANI, K. 2008. Coherent ray tracing via stream filtering. In *Interactive Ray Tracing, 2008. RT 2008. IEEE Symposium on*, 59 –66.

HAVRAN, V., BITTNER, J., AND ŽÁRA, J. 1998. Ray Tracing with Rope Trees. In *Proceedings of SCCG'98 (Spring Conference on Computer Graphics)*, 130–139.

HORN, D. R., SUGERMAN, J., HOUSTON, M., AND HANRAHAN, P. 2007. Interactive k-d tree GPU raytracing. In *SI3D*, 167–174.

HUGHES, D. M., AND LIM, I. S. 2009. Kd-jump: a path-preserving stackless traversal for faster isosurface raytracing on gpus. *IEEE Transactions on Visualization and Computer Graphics 15* (November), 1555–1562.

KAJIYA, J. T. 1986. The rendering equation. In *Computer Graphics*, 143–150.

KAPLAN, M. 1985. Space-Tracing: A Constant Time Ray-Tracer. In *SIGGRAPH '85 State of the Art in Image Synthesis seminar notes*, 149–158.

KARRAS, T., AILA, T., AND LAINE, S., 2009. Understanding the Efficiency of Ray Traversal on GPUs; Google Code. [online] `http://code.google.com/p/understanding-the-efficiency-of-ray-traversal-on-gpus/`.

KATO, T., AND SAITO, J. 2002. "kilauea": parallel global illumination renderer. In *Proceedings of the Fourth Eurographics Workshop on Parallel Graphics and Visualization*, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, EGPGV '02, 7–16.

LAINE, S. 2010. Restart trail for stackless bvh traversal. In *Proceedings of the Conference on High Performance Graphics*, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, HPG '10, 107–111.

MACDONALD, J. D., AND BOOTH, K. S. 1990. Heuristics for Ray Tracing Using Space Subdivision. *Visual Computer 6*, 153–65.

MOON, B., BYUN, Y., KIM, T.-J., CLAUDIO, P., KIM, H.-S., BAN, Y.-J., NAM, S. W., AND YOON, S.-E. 2010. Cache-oblivious ray reordering. *ACM Trans. Graph. 29*, 3, 1–10.

NAVRATIL, P. A., FUSSELL, D. S., LIN, C., AND MARK, W. R. 2007. Dynamic ray scheduling to improve ray coherence and bandwidth utilization. In *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*, IEEE Computer Society, Washington, DC, USA, 95–104.

NVIDIA, C., 2007. Tesla technical brief. [online] `http://www.nvidia.com/docs/IO/43395/tesla_technical_brief.pdf`.

NVIDIA, C., 2009. Whitepaper nvidia, next generation cuda compute architecture: Fermi. [online] `http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf`.

NVIDIA, C., 2011. NVIDIA CUDA Compute Unified Device Architecture - Programming Guide, Jan. Version 3.2, [online] `http://developer.nvidia.com/object/cuda_3_2_downloads.html`.

POPOV, S., GÜNTHER, J., SEIDEL, H.-P., AND SLUSALLEK, P. 2007. Stackless kd-tree traversal for high performance GPU ray tracing. *Computer Graphics Forum 26*, 3 (Sept.), 415–424. (Proceedings of Eurographics).

RAMANI, K., GRIBBLE, C. P., AND DAVIS, A. 2009. Streamray: a stream filtering architecture for coherent ray tracing. *SIGPLAN Not. 44* (March), 325–336.

SMITS, B. 1998. Efficiency issues for ray tracing. *J. Graph. Tools 3* (February), 1–14.

TORRES, R., MARTIN, P. J., AND GAVILANES, A. 2009. Ray Casting using a Roped BVH with CUDA. In *25th Spring Conference on Computer Graphics (SCCG 2009)*, 107–114.

WOOP, S., SCHMITTLER, J., AND SLUSALLEK, P. 2005. Rpu: a programmable ray processing unit for realtime ray tracing. In *ACM SIGGRAPH 2005 Papers*, ACM, New York, NY, USA, SIGGRAPH '05, 434–444.