High Performance Graphics (2010)
M. Doggett, S. Laine, and W. Hunt (Editors)

Restart Trail for Stackless BVH Traversal

Samuli Laine

NVIDIA Research

Abstract

A ray cast algorithm utilizing a hierarchical acceleration structure needs to perform a tree traversal in the hierarchy. In its basic form, executing the traversal requires a stack that holds the nodes that are still to be processed. In some cases, such a stack can be prohibitively expensive to maintain or access, due to storage or memory bandwidth limitations. The stack can, however, be eliminated or replaced with a fixed-size buffer using so-called stackless or short stack algorithms. These require that the traversal can be restarted from root so that the already processed part of the tree is not entered again. For kd-tree ray casts, this is accomplished easily by ray shortening, but the approach does not extend to other kinds of hierarchies such as BVHs.
In this paper, we introduce restart trail, a simple algorithmic method that makes restarts possible regardless of the type of hierarchy by storing one bit of data per level. This enables stackless and short stack traversal for BVH ray casts, where using a full stack or constraining the traversal order have so far been the only options.

1. Introduction

Tree traversal lies at the heart of efficient geometric queries for large datasets. Arguably, the most important graphics-related query is the ray cast query, where the goal is to find the closest primitive hit by a ray. Two kinds of hierarchies are dominant: kd-trees and bounding volume hierarchies (BVHs). Generally, BVHs are easier to construct and work with, and there is a large body of recent research devoted to them. Among the strengths of BVHs is the ability to update the nodes bottom-up to allow deformations [WBS07], and a wide array of construction methods with varying focus on construction speed and tree quality [LGS*09, Wal07, SFD09]. Moreover, each primitive needs to be stored in a single leaf node only, yielding predictable memory usage, whereas kd-trees require multiple references for primitives that straddle multiple leaves.

A major advantage for kd-trees is the possibility to dispose of the usual traversal stack, as noted by Foley and Sugerman [FS05]. They presented two *stackless* kd-tree traversal algorithms, one that restarts the traversal at root when a stack pop would be required in the traditional algorithm, and one that backtracks the tree in lieu of stack pops. Based on the former variant, Horn et al. [HSHH07] developed the so-called *short stack* traversal algorithm, where a small number of topmost items of the stack are retained in order to avoid some of the expensive restarts. In practice, a very small short stack is sufficient—the authors report that with a three-item short stack, only 3% more nodes are traversed compared to the traditional full stack. This is a major improvement over stackless traversal, which roughly doubles the number of nodes processed due to restarts [FS05]. For the special case of implicit kd-trees [WFM*05], mostly useful for storing dense, volumetric data, restarts can be avoided altogether [HL09].

For general, i.e. non-implicit, kd-trees, both stackless and short-stack traversal algorithms rely on the ability to restart the traversal so that already processed parts of the tree are not entered again. A restart is necessary when stack cannot be popped, either due to lack of stack altogether in stackless traversal, or due to short stack being exhausted. In kd-tree ray cast, the restart is performed by shortening the ray from the start so that the new starting point is just beyond the end of the last processed node. Because the nodes in a kd-tree do not overlap, the traversal does not enter already processed parts of the tree after restart. In practice, the shortening is accomplished by setting the current traversal t to t_{max} of the current node. Care must be taken to make sure that equal t values after the restart do not cause re-entering already processed branches, leading to infinite loop. As a consequence, it becomes somewhat tricky to handle zero-thickness nodes

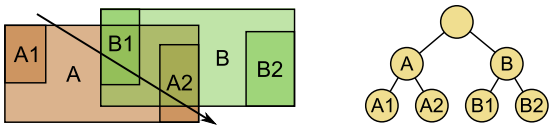


Figure 1: *Example of a case where ray shortening does not work for BVH restarts. The arrow represents the ray being cast. Assume that a stackless traversal has gone through nodes* A *and* A2*, and after that the stack cannot be popped to enter* B*. The restart should be performed so that the* A *branch is not entered, but the* B *branch is processed. To ensure that branch* A *is not entered again, the start of the ray would need to be after* A2*, but then* B1 *would be missed. Conversely, restarting with the ray origin at start of* B1 *would re-enter* A *and* A2*, causing an infinite loop. Short stack does not solve the general problem, because there may be a subtree between* A *and* A2 *that causes* B *to drop from the stack.*

that occur when all primitives in a subtree lie on an axis-aligned plane.

It is easy to see that the ray shortening method does not extend to BVH traversal. As Figure 1 illustrates, there is generally no way to shorten the ray so that a desired part of the hierarchy is excluded after restart. The BVH in the example is perhaps not entirely realistic, but allowing even a small amount of overlap between child nodes is enough to generate impossible situations.

If traversal order is fixed, the need for stack can be removed by storing a so-called *skip node* pointer in every node of the hierarchy [Smi98]. The skip node is the node to be processed next if the current node is not intersected, i.e. the subtree below the current node is to be skipped. The drawback of this approach is that ray direction cannot be taken into account when descending to the child nodes, so it is not possible to process the nearer node before the farther node. Boulos and Haines [BH06] discuss the extension of this technique by storing skip nodes for a number of traversal orders to allow some sensitivity to ray direction at the expense of increased per-node storage cost.

In this paper, we present *restart trail* that enables restarting hierarchy traversal in BVH ray casts. This allows stackless and short stack traversal techniques to be applied in BVH ray casts where using a full stack or constraining the traversal order have so far been the only options.

2. Restart trail

The restart trail method is based on explicitly storing which part of the hierarchy has been processed during the traversal using one bit per hierarchy level. A similar approach was used by Hughes and Lim [HL09] for traversing implicit kd-trees where pointers to ancestor nodes can be derived from current node address. In a general tree we cannot directly jump to the next unprocessed node, but instead we use the stored bits to guide the traversal to it when a restart occurs. We utilize the following properties of the ray cast operation in a BVH:

- Each node is visited at most once.
- There is an unambiguous order in which the children of each node are to be traversed, letting us label one of the children as the "near child" and the other "far child". This order does not change during the traversal.
- It is possible that some nodes become culled during the traversal, but it is not possible that the near child is culled while the far child is not.

The last property requires some clarification. When a primitive intersection is found, we cannot immediately terminate the traversal as in kd-trees, because there might be unvisited nodes that contain primitives that are closer than the one that was found. However, the ray can be shortened from the end so that nodes farther away than the closest found intersection are not visited. This optimization is crucial, because without it we could never terminate the ray before it exits the scene.

Therefore, at some point of the traversal we might want to visit both children of a node, but after processing the near child, we may find that it is not necessary to process the far child any more. This could be problematic for restarts, because the set of nodes to be traversed changes during the traversal. As we will see in a moment, it is possible to handle this situation as long as nodes can only be culled during the traversal, but never added.

Given these prerequisites, we define the semantics of the bit in the restart trail as follows:

- bit=0: Node has not been visited yet OR node has two children to be traversed and the subtree under near child has not been fully traversed yet.
- bit=1: Node has one child to be traversed OR node has two children to be traversed and the subtree under near child has already been fully traversed.

The pseudocode of BVH ray cast algorithm with restart trail is given in Figure 2. Lines 2 and 3 initialize the trail to all zeros and make *level* point to the first element in the trail. The first element $trail[0]$ acts as a sentinel and does not correspond to any hierarchy level, so $trail[1]$ holds the bit for the root node. Variable *level* therefore points to the bit corresponding to the parent of the current node (sentinel bit for root node). On line 4 we initialize variable *popLevel* that keeps track of level where the last stack pop would have taken us. The overall algorithm follows the conventional while-while [AL09] type scheduling for simplicity, but other loop structures could be implemented as well.

The branch in lines 9–17 handles the situation where both children of a node are intersected by the ray. The trail is consulted on line 12, where a set bit causes the traversal to skip the near child and proceed directly to the far child (line 13).

```
CAST-RAY(ray)
 1  node ← ROOT
 2  trail ← (0,0,...)
 3  level ← 0
 4  popLevel ← NONE
 5  while true
 6    while node is not leaf do
 7      intersect ray against children of node
 8      if both children were intersected then
 9        near ← child with smaller t_min along the ray
10        far ← child with greater t_min along the ray
11        level ← level + 1
12        if trail[level] = 1 then
13          node ← far
14        else
15          node ← near
16          PUSH(far)
17        end if
18      else if one child was intersected then
19        level ← level + 1
20        if level ≠ popLevel then
21          trail[level] ← 1
22          node ← the child that was intersected
23        else
24          POP()
25        end if
26      else
27        POP()
28      end if
29    end while
30    intersect ray against primitives in node
31    shorten ray at end if closer primitives found
32    POP()
33  end while

POP()
34  level ← largest i where i ≤ level and trail[i] = 0
35  trail[level] ← 1
36  trail[i] ← 0 for all i > level
37  if trail[0] = 1 then terminate traversal
38  popLevel ← level
39  if short stack is exhausted then
40    node ← ROOT
41    level ← 0
42  else
43    node ← pop short stack
44  end if
```

Figure 2: *Pseudocode of BVH ray cast with restart trail and short stack. See text for discussion.*

If the bit in the trail is not set, the traversal proceeds in a normal fashion to the near child, pushing the far child into short stack if one is available (lines 15–16). Note that we do not need to specifically know if a restart is taking place, because we ensure that the trail always contains zeros for unvisited nodes.

The single-child branch in lines 19–28 is somewhat more involved. On line 20 we test if this is the parent of the node where the previous pop operation would have taken us. In the common case it is not, so we set the corresponding bit in the trail (line 21) following the semantics defined above, and proceed to the child node. However, if *level* is equal to *popLevel*, we know that this is a node that originally had two children intersecting the ray, and the near child has been completely processed, as stack pop would have taken us to the far child. But the ray now intersects only one child, and the only explanation is that during the processing of the near child the ray has been shortened so that the far child is not intersected anymore. In this case, we must invoke POP (line 24) to avoid re-entering the near branch.

Finally, if no children were intersected, we execute POP (line 27). When we encounter a leaf node, the node intersection loop (lines 6–29) is terminated and we process the primitives in the leaf (line 30). If we find a primitive intersection that is closer than what we have already found, we shorten the ray to avoid traversing unnecessary nodes (line 31). After processing the primitives in the leaf node, we execute POP (line 32) and resume node traversal.

Subroutine POP updates the restart trail to reflect the situation where the current node has been completely processed. First, the level where the next unprocessed node resides is determined by finding the nearest zero bit in the trail above the current level (line 34). This zero bit corresponds to the node whose far child is the next node in the short stack. By letting *level* point to the zero bit, it is therefore in sync with the node popped from the short stack. After finding the zero bit, we must make the trail consistent with the fact that the branch under the near child pushed in the correponding node has been processed. This is done by changing the zero bit to one (line 35). We must also clear the remaining bits to indicate that all nodes in the far branch of the node are unvisited (line 36).

Line 37 checks if the sentinel bit was flipped, which indicates that the root node—i.e., the entire tree—has been fully processed, and if so, terminates the traversal. To prepare for restart, the level of the node whose near child was marked as fully processed is saved in *popLevel* (line 38). This ensures that even if we are forced to restart the traversal, we know when we have reached the node that the pop operation would have given us, and handle the case of culled far child appropriately. Finally, if short stack cannot be popped, the traversal is restarted (lines 40–41). If short stack pop succeeds (line 43), *trail* and *level* are in accordance with the popped node, and the traversal may continue from there.

Figure 3 illustrates an example traversal with restart trail. At each point in traversal, restart could be initiated, because the trail always leads to the next unprocessed node.

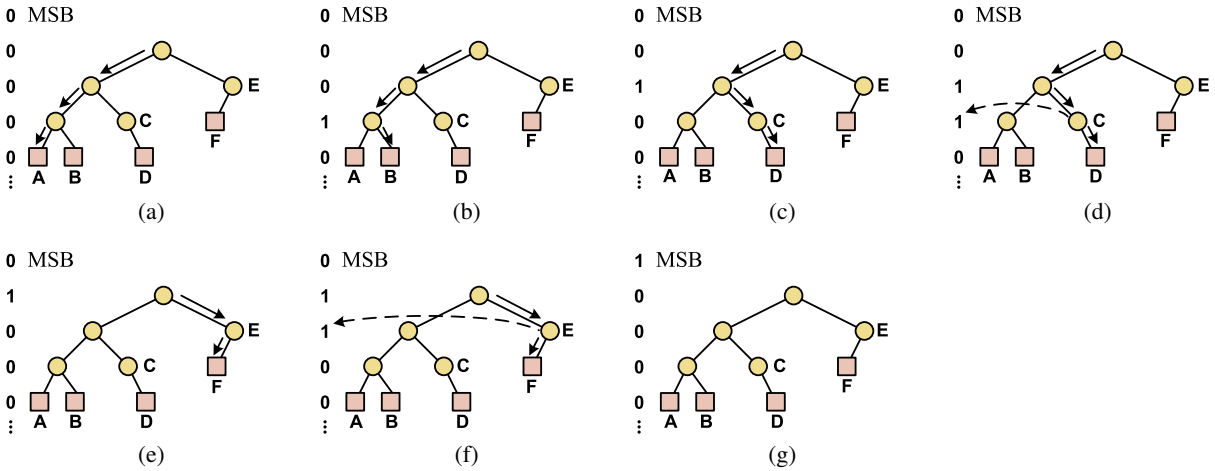


Figure 3: *An example of restart trail contents during traversal. The circles represent hierarchy nodes where traversal continues down, whereas squares represent nodes where* POP *is executed. These can be either leaf nodes or internal nodes where the ray does not intersect either child. The tree is ordered so that near child is always on the left. The arrows pointing down the tree indicate where the trail points after each traversal step. Trail contents are shown to the left of the tree, sentinel bit at the top. (a) After initialization trail leads to node A. (b) After updating trail in* POP *in node A, trail leads to node B. (c) After updating trail in* POP *in node B, trail leads to node D. (d) Traversing through node C sets the fourth bit in trail. (e) After updating trail in* POP *in node D, trail leads to node F. (f) Traversing through E sets the third bit in trail. (g) Updating the trail in* POP *in node F flips the sentinel bit, therefore terminating the traversal.*

2.1. Practical implementation

The bits in *trail* can be most conveniently stored in a single 32 or 64-bit register, and it turns out that the required manipulations can be performed very efficiently using bit operations available in current CPUs and GPUs. For maximum efficiency, variable *level* needs to be stored as a register with exactly one bit set corresponding to the bit being pointed to.

Let us denote these registers as `trailReg` and `levelReg`. *popLevel* can be encoded analogously to *level* and will not be elaborated on further. The bits in `trailReg` are ordered so that the most significant bit is used for the sentinel, and therefore `levelReg` should be initialized with bit configuration `10000...`

Restart trail related operations in the main routine CASTRAY (Figure 2) are trivial to realize with this representation. Incrementing *level* (lines 11 and 19) corresponds to shifting `levelReg` right by one bit, and testing if *trail*[*level*] is set (line 12) can be done by performing bitwise AND between `trailReg` and `levelReg`, and testing if the result is nonzero. Setting the bit in *trail* (line 21) is achieved by logical OR between `trailReg` and `levelReg`.

The updating of trail in POP is more interesting. To set the next highest zero bit in `trailReg` above or at `levelReg`, we can simply add `levelReg` to `trailReg`. This has the effect of clearing all set bits above and at the bit set in `levelReg` until the first zero bit, which is flipped to one. To clear the bits below `levelReg`, we can use a bitwise AND operation. Consequently, lines 34–36 can be implemented as

```
trailReg &= -levelReg;
trailReg += levelReg;
```

in C and variants such as CUDA. However, we also need to make `levelReg` point to the lowest set bit in `trailReg`. Unlike finding the highest set bit, which some processors offer as a native instruction, extracting the lowest set bit can be performed efficiently using standard bitwise operations as follows:

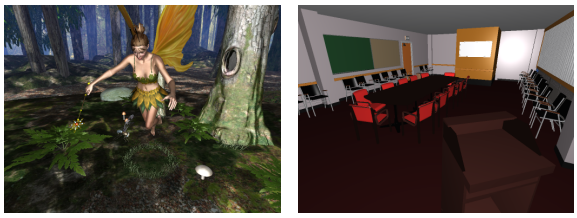
```
temp = trailReg >> 1;
levelReg = (((temp-1) ^ temp) + 1);
```

where `^` is the bitwise XOR operation.

3. Results and discussion

We tested how much the short stack traversal with restart trail affects the number of nodes processed in BVH ray casts using two common test scenes: Fairy Forest and Conference room (Figure 4). For both scenes, we constructed one BVH with the usual SAH heuristics and one with the SBVH algorithm of Stich et al. [SFD09].

In our tests, stackless traversal visited approximately 2.2–2.4 times as many nodes as ordinary traversal with full stack. This is in line with kd-tree results of Horn et al. [HSHH07]. Notably, a short stack with just one entry dropped the multiplier to 1.3–1.4. A three-entry short stack yielded 5–8%



Fairy Forest, 174K tris Conference room, 282K tris

Figure 4: *Test scenes used in the evaluation of the number of additional nodes visited due to BVH restarts. The images show one of the multiple viewpoints used for both scenes.*

extraneous node visits, compared to 3% for kd-trees as reported by Foley and Sugerman [FS05]. We hypothesize that the slight increase is caused by rays intersecting both children more frequently in BVHs than in kd-trees, causing more stack pushes and hence more restarts due to exhaustion of short stack.

Preliminary tests with the CUDA ray cast kernels of Aila and Laine [AL09] indicate that on G80 and GT200 NVIDIA hardware, the memory bandwidth benefits of using a short stack do not quite outweigh the additional costs caused by restarts, increased SIMD execution divergence, and additional instructions required by restart trail operations. However, the amount of memory traffic caused by a full stack is substantial, and effects such as cache thrashing may change the situation in the near future.

Avoiding the storage of a full stack could also be useful for a hardware hierarchy traversal unit. Restart trail and a small short stack can be stored locally in registers, but a full stack most probably cannot. The net effect on memory bandwidth is positive as long as the traffic caused by additional node fetches in restarts is less than the traffic caused by full stack. It should be noted that stack entries are generally small, making them comparably expensive to access in systems with wide memory buses such as GPUs.

Extending the method to larger-radix trees is left as future work. An interesting avenue for future research would be examining if restart trails could be used to avoid the use of full stack in other kinds of traversals such as closest-point and kNN searches that are involved in photon mapping and density estimation tasks.

Acknowledgements Many thanks to Timo Aila for discussions and help in developing the technique. Fairy Forest model courtesy of Ingo Wald, University of Utah. Conference room scene courtesy of Anat Grynberg and Greg Ward.

References

[AL09] AILA T., LAINE S.: Understanding the efficiency of ray traversal on GPUs. In *Proceedings of High-Performance Graphics 2009* (2009), pp. 145–149. 2, 5

[BH06] BOULOS S., HAINES E.: Notes on efficient ray tracing. *Ray Tracing News 19*, 1 (2006). 2

[FS05] FOLEY T., SUGERMAN J.: KD-tree acceleration structures for a GPU raytracer. In *Proceedings of Graphics Hardware 2005* (2005), pp. 15–22. 1, 5

[HL09] HUGHES D. M., LIM I. S.: Kd-jump: a path-preserving stackless traversal for faster isosurface raytracing on GPUs. *IEEE Transactions on Visualization and Computer Graphics 15* (2009), 1555–1562. 1, 2

[HSHH07] HORN D. R., SUGERMAN J., HOUSTON M., HANRAHAN P.: Interactive k-D tree GPU raytracing. In *I3D '07: Proceedings of the 2007 symposium on Interactive 3D graphics and games* (2007), pp. 167–174. 1, 4

[LGS*09] LAUTERBACH C., GARLAND M., SENGUPTA S., LUEBKE D., MANOCHA D.: Fast BVH construction on GPUs. *Computer Graphics Forum 28*, 2 (2009), 375–384. 1

[SFD09] STICH M., FRIEDRICH H., DIETRICH A.: Spatial splits in bounding volume hierarchies. In *Proceedings of High-Performance Graphics 2009* (2009). 1, 4

[Smi98] SMITS B.: Efficiency issues for ray tracing. *J. Graph. Tools 3*, 2 (1998), 1–14. 2

[Wal07] WALD I.: On fast construction of SAH-based bounding volume hierarchies. In *RT '07: Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing* (2007), pp. 33–40. 1

[WBS07] WALD I., BOULOS S., SHIRLEY P.: Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Transactions on Graphics 26*, 1 (2007). 1

[WFM*05] WALD I., FRIEDRICH H., MARMITT G., SLUSALLEK P., SEIDEL H.-P.: Faster isosurface ray tracing using implicit kd-trees. *IEEE Transactions on Visualization and Computer Graphics 11*, 5 (2005), 562–572. 1