Stackless Multi-BVH Traversal for CPU, MIC and GPU Ray Tracing

Attila T. Áfra[1,2], László Szirmay-Kalos[1]
[1]Budapest University of Technology and Economics (Hungary), [2]Babeş-Bolyai University (Romania)

Introduction

Using a stack for ray traversal is typically the most straightforward and efficient approach. However, if many rays are traced in parallel, the storage and bandwidth costs of maintaining a full stack for each ray can be very high. Notable examples are dynamic ray scheduling algorithms that improve memory access coherence for random ray distributions [1]. For such ray tracing methods, stackless traversal is better suited.

We propose a new efficient stackless ray traversal algorithm for multi bounding volume hierarchies (MBVHs) that supports distance-based ordered traversal without restarts. Two variations of the algorithm are presented: one variation for 4-way branching MBVHs (**MBVH4**) and one for binary BVHs having two child boxes per node (**MBVH2**), which outperforms similar prior methods [6, 3].

Overview

Our goal is to traverse the same sequence of *N*-way tree nodes as a stack-based algorithm with a distance-based order heuristic. We replace the stack pop with **backtracking** in the tree from the current node. The purpose of this operation is to find the next unprocessed node. This is a node whose bounding box was hit by the ray while processing the parent, but which has not been traversed yet. It is a sibling of either the current node or one of its ancestors. We add parent and sibling pointers to each node for efficient traversal.

The backtracking is guided by a bitmask that encodes which part of the tree needs to be traversed. It stores $N-1$ bits for each visited level (except the first), and is updated similarly to a stack, using bitwise push and pop operations. We call this special bitmask a **bitstack**. The per-level values in the bitstack are **skip codes**. These indicate which siblings of the most recently visited node on the respective level have already been processed and thus must be skipped.

MBVH2 Traversal

We use two state variables: a pointer to the current node and the bitstack, a 32- or 64-bit integer. For binary trees, the skip codes pushed onto the bitstack are **1-bit flags**:

0: Skip the sibling of the current node; go to the parent.

1: Traverse the sibling of the current node.

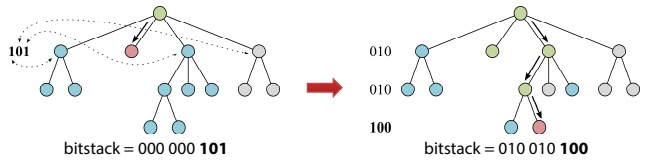
The top of the bitstack is implicitly the least significant bit. This means that when pushing or popping an item, all the items in the stack must be shifted by one position. The initial value of the bitstack is 0, which is equivalent to an empty stack. The advantage of this representation is that the traversal can be terminated earlier than returning to the root node, avoiding unnecessary backtracking steps.

MBVH4 Traversal

To traverse 4-way trees, we extend the skip codes to **3 bits**, where each bit corresponds to a sibling of the respective node. These bits have the same semantics as 1-bit skip codes: a 0 bit means that the sibling must be skipped, whereas a 1 bit means that it must be traversed. The siblings of a particular node are indexed **circularly** starting from the next node in the sibling group. Nodes that have less than 4 children are padded with invalid or empty node references, thus every node has exactly 3 siblings.

Bitstack

The following figure depicts two MBVH4 traversal steps where backtracking is triggered. Blue nodes represent unprocessed nodes, green nodes have been already processed, gray nodes have been culled, and the red node is the current node. The skip codes are shown on the corresponding tree levels. The top of the bitstack is highlighted in bold.



MBVH4 Traversal Pseudocode

```
ShootRay(ray)
 1  node ← Root
 2  bitstack ← 0
 3  loop
 4    if IsInner(node) then
 5      intersect ray with children
 6      mask ← bitmask of child hits
 7      if any child was hit then
 8        bitstack ← bitstack ≪ 3
 9        if one child was hit then
10          node ← the child that was hit
11        else
12          nearPos ← index of the nearest child
13          node ← Child(node, nearPos)
14          skipCode ← SkipCode(mask, nearPos)
15          bitstack ← bitstack ∨ skipCode
16        continue
17    else
18      intersect ray with primitives
19      shorten ray if closer intersection found
20    // Backtrack
21    while (skipCode ← bitstack ∧ 7) = 0 do
22      if bitstack = 0 then
23        return // Terminate
24      node ← Parent(node)
25      bitstack ← bitstack ≫ 3
26    siblingPos ← BitScan(skipCode)
27    node ← Sibling(node, siblingPos)
28    bitstack ← bitstack ⊕ SkipCodeNext(skipCode)

SkipCode(mask, pos)
29  return ((mask ≫ (pos+1)) ∨ (mask ≪ (3−pos))) ∧ 7

SkipCodeNext(code)
30  return code ⊕ (code ≫ (BitScan(code)+1))
```

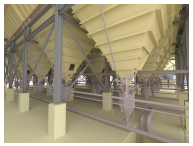
Test Scenes


Conference (282K tris)


Crytek Sponza (262K tris)


Fairy (174K tris)


Hairball (2.9M tris)

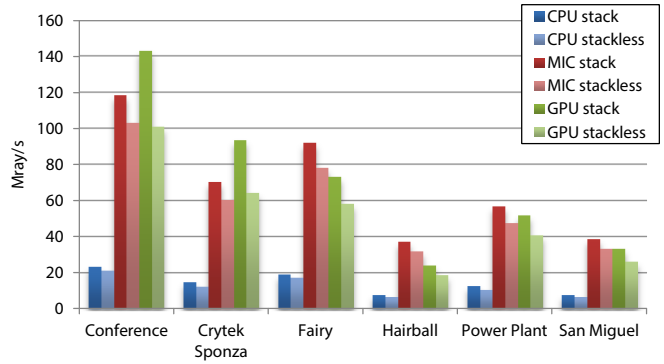

Power Plant (12.7M tris)


San Miguel (10.5M tris)

Results

We evaluated the performance of our traversal algorithms using a simple **8-bounce diffuse** path tracer on CPU (**Intel Core i7-3770**), MIC (**Intel Xeon Phi SE10P**), and GPU (**NVIDIA Tesla K20c**) architectures. Our stackless algorithms, similarly to previous methods, are somewhat slower than the reference stack-based ones [5, 4, 2] when used for ordinary ray tracing; however, they maintain about 22–51× smaller traversal states (**12 B/ray** for MBVH2 and **20 B/ray** for MBVH4). For special rendering methods that trace many rays in parallel, low memory footprint is essential and could lead to a much higher overall performance.

For our test scenes, stackless traversal is slower by 9–17% on the CPU, 13–16% on the MIC, and 20–31% on the GPU. This is caused by the more complex traversal logic, more irregular memory accesses, and on the CPU and MIC the slightly higher number of box and triangle intersections.



Acknowledgements

This work was supported by POSDRU/107/1.5/S/76841 and by OTKA K-104476 (Hungary).

We would like to thank Paul Navrátil and the Texas Advanced Computing Center at The University of Texas at Austin for providing us access to the Stampede supercomputer. We gratefully acknowledge the support of NVIDIA with the donation of the Tesla K20c GPU.

References

[1] AILA T., KARRAS T.: Architecture considerations for tracing incoherent rays. In *Proceedings of HPG '10* (2010), pp. 113–122.

[2] AILA T., LAINE S., KARRAS T.: *Understanding the Efficiency of Ray Traversal on GPUs – Kepler and Fermi Addendum*. Tech. Rep. NVR-2012-02, NVIDIA, June 2012.

[3] BARRINGER R., AKENINE-MÖLLER T.: Dynamic stackless binary tree traversal. *Journal of Computer Graphics Techniques (JCGT) 2*, 2 (March 2013), 38–49.

[4] BENTHIN C., WALD I., WOOP S., ERNST M., MARK W.: Combining single and packet-ray tracing for arbitrary ray distributions on the Intel MIC architecture. *IEEE TVCG 18*, 9 (September 2012), 1438–1448.

[5] ERNST M.: Embree: Photo-realistic ray tracing kernels. *ACM SIGGRAPH 2011 Exhibitor Tech Talk* (2011).

[6] HAPALA M., DAVIDOVIČ T., WALD I., HAVRAN V., SLUSALLEK P.: Efficient stack-less BVH traversal for ray tracing. In *Proceedings of SCCG '11* (2011), pp. 7–12.