Dynamic Stackless Binary Tree Traversal

Rasmus Barringer
Lund University

Tomas Akenine-Möller
Lund University and Intel Corporation

Figure 1. CHESS scene rendered using our stackless binary tree traversal algorithm.

Abstract

A fundamental part of many computer algorithms involves traversing a binary tree. One notable example is traversing a space-partitioning acceleration structure when computing ray-traced images. Traditionally, the traversal requires a stack to be temporarily stored for each ray, which results in both additional storage and memory-bandwidth usage. We present a novel algorithm for traversing a binary tree that does not require a stack and, unlike previous approaches, works with dynamic descent direction without restarting. Our algorithm will visit exactly the same sequence of nodes as a stack-based counterpart with extremely low computational overhead. No additional memory accesses are made for implicit binary trees. For sparse trees, parent links are used to backtrack the shortest path. We evaluate our algorithm using a ray tracer with a bounding volume hierarchy for which source code is supplied.

1. Introduction

Traversing a binary tree is a fundamental operation for many computer algorithms. One notable example, related to computer graphics, is traversing a space-partitioning acceleration structure when performing ray tracing [Whitted 1980]. The traversal

usually requires a stack, that contains nodes that are still to be processed, to be temporarily stored. However, in some cases, a stack is prohibitively expensive to maintain or access [Laine 2010], e.g., for highly parallel architectures with many active traversal states. There may also be situations where the traversal state is suspended and resumed [Hapala et al. 2011], in which case storing or transferring the full stack is expensive. For these reasons, stackless algorithms have been explored. Hughes and Lim [2009] demonstrate stackless traversal for dense implicit kd-trees. Their approach requires a k-by-3 matrix to be stored in constant memory, where k is the depth of the tree. When traversal ascends in the tree, additional heuristics are needed to know which child to continue traversing, requiring additional knowledge of the data structure. Another approach that is not restricted to kd-trees involves using a short stack and encoding a restart trail in a bit mask [Laine 2010]. When the short stack is insufficient, traversal restarts from the root node and descends along the stored restart trail. Hapala et al. [2011] use parent pointers to achieve stackless traversal by backtracking. Their algorithm needs to determine traversal order among two siblings again when ascending in the tree. This is prohibitively expensive for anything but a simple ordering heuristic. As a result, the traversal order in their bounding volume hierarchy is based solely on ray direction. Computing the actual distance to the siblings' bounding boxes, and sorting them based on distance, would require re-intersecting both nodes to determine the traversal order.

In this paper, we introduce low-overhead stackless traversal algorithms for binary trees that, unlike previous approaches, support dynamic descent direction without restarting. In particular, we introduce two algorithm variations for implicit binary trees, as well as one variation for sparse trees. Our algorithms will visit exactly the same sequence of nodes as a stack-based counterpart with extremely low computational overhead. No additional memory accesses are made for implicit binary trees. For sparse trees, parent links are used to backtrack the shortest path.

2. Implicit Traversal Algorithm

For implicit binary trees, we store the nodes in each level sequentially in memory, i.e., the root node at location 0 and its left and right children at location 1 and 2, respectively. In general, the nodes at a depth, d, are enumerated as $\{2^d - 1, \ldots, 2^{d+1} - 2\}$, where $d = 0$ indicates the root level. As such, the relationships between different nodes is explicitly known. Given the address of a node, it is possible to calculate the address of the parent, children, and sibling.

The trees for bounding-volume hierarchies, for example, are generally not perfectly balanced. In those cases, we simply leave gaps in the memory layout for unused nodes. Since they will never be accessed, it is enough to allocate the address space for them. CPUs can utilize virtual memory to reserve the entire address space for the tree

but only map pages that actually contain nodes. This reduces the memory overhead associated with filling out the gaps. In Section 3, it is shown that the algorithm can be extended to sparse binary trees with parent pointers.

We start by describing a simplified traversal algorithm that assumes that the left child in a tree is always traversed first, i.e., a typical depth-first traversal order. Then, we extend this algorithm to support tree traversal in any order.

2.1. Left-First Traversal

Knowing that we always choose the left child during traversal, we can keep track of the traversal state using two integers. We need one bit for each level of the tree and, thus, 32 bits is enough for reasonably balanced trees. However, for the purpose of our algorithm, integers of any size can be used. The algorithm is shown in Algorithm 1; the first integer, $levelStart$, stores 2^d, where d is the current depth of the traversal. This variable is used to calculate the address of the first node in the current level, which is given by $levelStart - 1$. The second integer, $levelIndex$, stores the current node relative to the first node of the current depth level. The index of a node is thus given by $levelStart + levelIndex - 1$ (see Figure 2). When the algorithm descends to the left child of a node, we recalculate the two integers to point to the left child in the next depth level. This is accomplished using simple shift operations.

If we decide to skip a node, or have found and processed a leaf node, we either need to traverse to the right sibling, or ascend upward in the tree. This is where we would normally need a stack containing the next node to process. However, since we always traverse to the left child first, we simply need to ascend in the tree when the right child of the parent node has been processed. This criteria is equivalent to

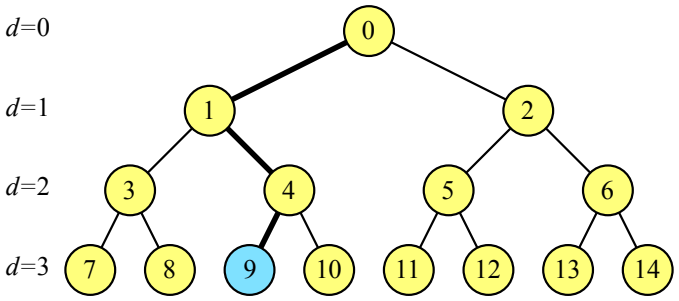

Figure 2. An example of a complete binary tree with the depths, d, of each level shown at the left. The node number is shown inside each node circle. For the blue node at the bottom, we have $levelStart = 2^d = 2^3$ and $levelIndex = 2$, which makes it possible to compute the node number as $2^3 + 2 - 1 = 9$. For the blue node, we also have $swapMask = 010$ (binary), which indicates the path from the root down to the node, where 0 is a descent to the left child, and 1 is a descent to the right child. The $swapMask$ is used in Algorithm 2.

Algorithm 1 (Left-first implicit traversal.)

```
levelStart ← 1
levelIndex ← 0
repeat
  node ← levelStart + levelIndex − 1
  if node is leaf then
    process leaf
  else
    test node
    if accepted then
      levelStart ← levelStart ≪ 1
      levelIndex ← levelIndex ≪ 1
      continue
    end if
  end if
  levelIndex ← levelIndex + 1
  up ← ctz(levelIndex)
  levelStart ← levelStart ≫ up
  levelIndex ← levelIndex ≫ up
until levelStart ≤ 1
```

the criteria that *levelIndex* have the same number of trailing zeros as the number of levels to ascend in the tree. This approach is similar to that used by Knoll et al. [2009] for finding the leftmost root of implicit functions. The main difference is how they used intervals and a loop with floating-point division to achieve the stackless traversal, while our approach can be implemented using a single count-trailing-zeros instruction (CTZ), which is common in many architectures. The integers are then adjusted by this amount using right shifts.

2.2. Generalized Traversal

When traversing a binary tree, it is often beneficial to start with a certain child based on a dynamic heuristic. For example, for efficient visibility computations using ray tracing, it is important to traverse to the child node whose content is most likely to shorten the ray, making traversal to the sibling unnecessary. A common heuristic is to start with the closest bounding volume. The traversal technique in Section 2.1, however, always traverses to the left child first, and if used in a ray tracer, performance would suffer substantially since rays would not benefit from early occlusion.

One way to enable arbitrary descent order (left or right child) is to actually traverse exactly as in Section 2.1 with the exception that the left and right children are swapped when traversal to the right child is preferred. It is, of course, not feasible to actually swap the memory of the nodes in the tree. Instead, we define a function $f_n(x)$ that

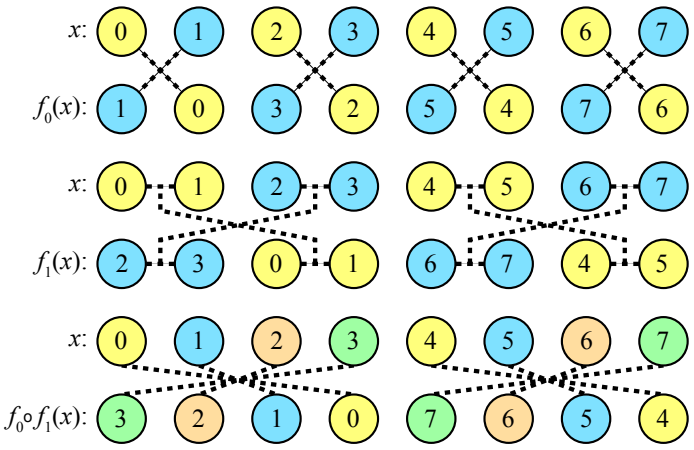

Figure 3. The result of applying various combinations of $f_n(x)$ to a list of indices, x. Top: f_0; Middle: f_1; Bottom: $f_0 \circ f_1$.

swaps the indices of each node at a certain level in the tree:

$$f_n(x) = x + 2^n - 2(x \wedge 2^n),$$

where $\wedge$ represents bitwise AND. If the previous level took a right turn, the relative index would not be *levelIndex*, but rather $f_0(levelIndex)$. Some examples of applying this function to a list of indices is shown in Figure 3.

Since any level can take a right turn, we need to be able to apply composite functions to describe an arbitrary path through the tree:

$$(f_i \circ f_j \circ \ldots)(x),$$

where $i, j, \ldots$ are the levels where the traversal took a right turn and $(g \circ f)(x)$ is the same as $g(f(x))$.

From Figure 3, one can see that the result is independent of the order of composition, i.e., $f_i \circ f_j(x) = f_j \circ f_i(x)$, which suggests that the entire composite function can be expressed in a very simple manner. Let d denote the current depth of the traversal. If we create a bitmask, called *swapMask*, where each bit i is set if level $d - i - 1$ took a right turn, then the entire composition becomes:

$$f_{\text{comp}}(x) = x + swapMask - 2(x \wedge swapMask). \tag{1}$$

Given $f_{\text{comp}}(x)$, we are now ready to describe the generalized traversal algorithm. We need one more integer to store *swapMask*. Besides some simple bookkeeping for *swapMask*, all we need to do is apply Equation (1) to *levelIndex*. The generalized traversal technique is shown in Algorithm 2.

In this algorithm, we essentially compute the dynamic descent *levelIndex* from a left-first *levelIndex* at each iteration. It is also possible to incrementally update the

Algorithm 2 (Generalized implicit traversal.)

```
levelStart ← 1
levelIndex ← 0
swapMask ← 0
repeat
  node ← levelStart + levelIndex − 1
       + swapMask − 2(levelIndex ∧ swapMask)
  if node is leaf then
    process leaf
  else
    test children of node
    if any accepted then
      levelStart ← levelStart ≪ 1
      levelIndex ← levelIndex ≪ 1
      swapMask ← swapMask ≪ 1
      if right child first then
        swapMask ← swapMask ∪ 1 {bitwise OR}
      end if
      if rejected one child then
        levelIndex ← levelIndex + 1
        swapMask ← swapMask ⊕ 1 {bitwise XOR}
      end if
      continue
    end if
  end if
  levelIndex ← levelIndex + 1
  up ← ctz(levelIndex)
  levelStart ← levelStart ≫ up
  levelIndex ← levelIndex ≫ up
  swapMask ← swapMask ≫ up
until levelStart ≤ 1
```

dynamic *levelIndex*. Denote the dynamic *levelIndex* as $levelIndex_{dynamic}$. It is obvious that any dynamic descent can incrementally update $levelIndex_{dynamic}$ by assigning it the child traversed. We also observe that any ascent in the tree is independent of descent order; the parent at a given level is the same whether we traversed the left or the right child. The problem becomes which child to continue traversing after ascent. This is easy to answer since $levelIndex_{dynamic}$ indicates which child has already been traversed. We simply need to switch to the sibling of $levelIndex_{dynamic}$ after ascending in the tree. These observations are summarized in Algorithm 3. The main difference compared to Algorithm 2 is that the swap only occurs at the current level after ascending in the tree.

Even though Algorithm 2 indicates separate variables for *levelStart* and *levelIndex*, it is possible to combine them into one. The start of the current level,

Algorithm 3 (Optimized generalized implicit traversal.)

```
levelStart ← 1
levelIndex ← 0
levelIndex_dynamic ← 0
repeat
  node ← levelStart + levelIndex_dynamic − 1
  if node is leaf then
    process leaf
  else
    test children of node
    if any accepted then
      levelStart ← levelStart ≪ 1
      levelIndex ← levelIndex ≪ 1
      levelIndex_dynamic ← levelIndex_dynamic ≪ 1
      if right child first then
        levelIndex_dynamic ← levelIndex_dynamic + 1
      end if
      if rejected one child then
        levelIndex ← levelIndex + 1
      end if
      continue
    end if
  end if
  levelIndex ← levelIndex + 1
  up ← ctz(levelIndex)
  levelStart ← levelStart ≫ up
  levelIndex ← levelIndex ≫ up
  levelIndex_dynamic ← levelIndex_dynamic ≫ up
  levelIndex_dynamic ← levelIndex_dynamic + 1 − 2(levelIndex_dynamic ∧ 1)
until levelStart ≤ 1
```

levelStart, is always represented by a single set bit that is higher than any set bit in *levelIndex*. By introducing $index = levelStart + levelIndex$, we can replace all instances of the variables with *index*. This works because all operations on *levelIndex* are unaffected by the high bit from *levelStart*. This optimization reduces the number of state variables to two, and avoids one addition and two redundant shift operations. The same optimization can be introduced in Algorithm 3 by setting $index = levelStart + levelIndex_{dynamic}$.

3. Sparse Traversal Algorithm

Sometimes the restrictions of an implicit binary tree cannot be met, for example, when the tree is badly balanced or when the implicit memory layout cannot be used.

Algorithm 4 (Sparse traversal.)

```
levelIndex ← 0
node ← root
repeat
  if node is leaf then
    process leaf
  else
    test children of node
    if any accepted then
      levelIndex ← levelIndex ≪ 1
      node ← left or right child
      if rejected one child then
        levelIndex ← levelIndex + 1
      end if
      continue
    end if
  end if
  levelIndex ← levelIndex + 1
  while levelIndex ∧ 1 = 0 do
    node ← parent(node)
    levelIndex = levelIndex ≫ 1
  end while
  node ← sibling(node)
until node = root
```

In those cases, a stackless algorithm can still be used [Laine 2010; Hapala et al. 2011]. It turns out that Algorithm 3 is actually well-suited for sparse trees as well, given that there is a method to ascend in the tree. Assuming the existence of parent pointers that allows us to backtrack [Hapala et al. 2011], we replace both $levelStart$ and $levelIndex_{dynamic}$ with a single node pointer. The node pointer is updated analogously to $levelIndex_{dynamic}$. When descending in the tree, the node pointer simply follows the appropriate child link. When ascending in the tree, instead of jumping to the appropriate parent using a shift instruction, we backtrack using the parent pointers until the destination level is reached. The swap function at the end of the loop is replaced by an analogous function that determines the sibling of a node. It can either be implemented by calculating the sibling directly, or, by taking a round trip to the parent, depending on the memory layout of the tree. The algorithm for sparse trees is given in Algorithm 4.

4. Results

Our three algorithms, referred to as IMPLICIT-A (Algorithm 2), IMPLICIT-B (Algorithm 3) and SPARSE (Algorithm 4), were implemented in a simple single-threaded

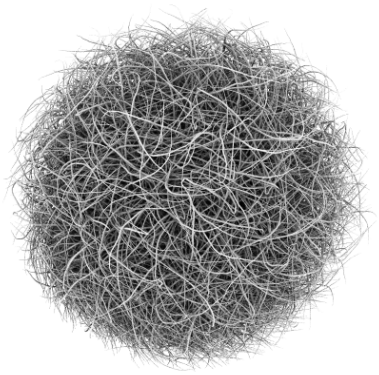

Figure 4. HAIRBALL scene with 2.8 million triangles.

	CHESS	HAIRBALL
STACK	24.6 s	149 s
SPARSE	26.5 s	157 s
IMPLICIT-A	27.5 s	164 s
IMPLICIT-B	26.2 s	160 s
HAPALA	35.8 s	189 s
LAINE-0	44.7 s	368 s
LAINE-1	31.6 s	232 s
LAINE-2	28.1 s	191 s
LAINE-4	26.4 s	166 s
LAINE-8	26.1 s	155 s

Table 1. Table showing the time required for each algorithm to render the test scenes.

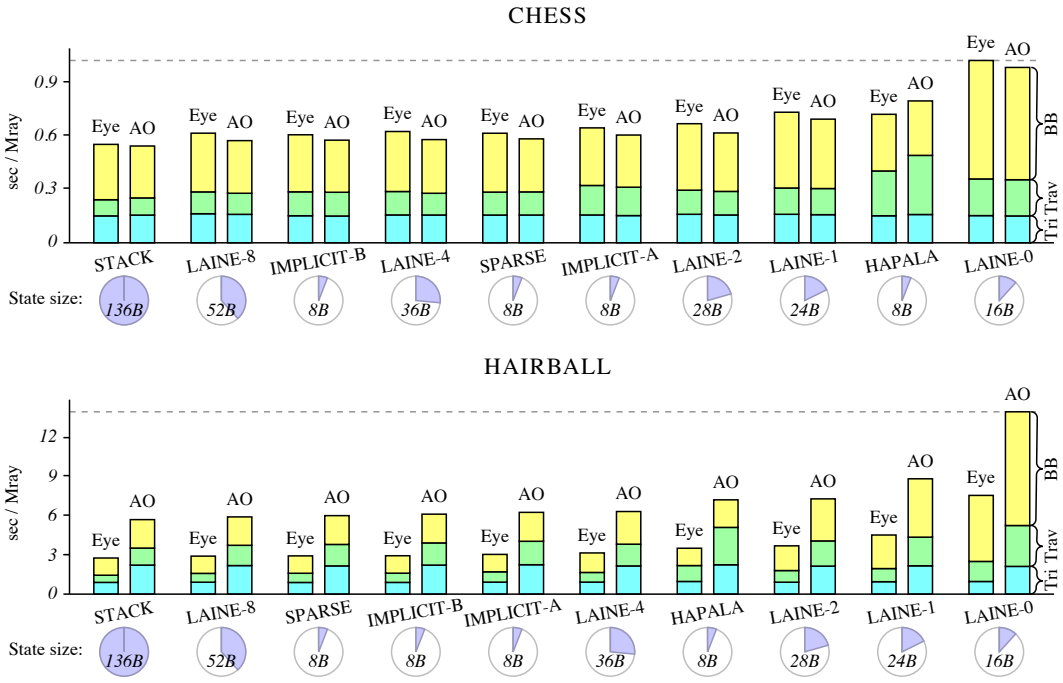

Figure 5. Bar charts showing detailed performance measurements from CHESS (top) and HAIRBALL (bottom). Each algorithm shows the performance of eye and AO rays. Additionally, a time break-down into triangle intersection, traversal, and AABB intersection is illustrated within each bar. Below each algorithm, the size of its traversal state is shown. The state size is based on the number of state variables needed (4 bytes each). A non-zero stack size adds an extra state variable for the stack head and 4 bytes for each entry. The algorithms are ordered according to increasing rendering time. Note that the ordering is biased toward AO ray performance since they represent the majority of rays traced.

CPU ray tracer written in C++. All tests were performed on an Intel Core i7 processor clocked at 2.66 GHz and with 8 GB of 1067 MHz DDR3 RAM. The target compiler was Apple Darwin LLVM GCC 4.2.1. For reference, a conventional stack-based traversal, STACK, was included. For comparison to other stackless approaches, we implemented the algorithms by Laine [2010] and Hapala et al. [2011], referred to as LAINE and HAPALA respectively. For LAINE, we investigate different short stack sizes and denote them LAINE-x, where x indicates the size of the short stack. The ray tracer uses a bounding volume hierarchy (BVH), containing axis-aligned bounding boxes (AABBs), optimized using surface area heuristic (SAH) for the first 8 depth levels, and median split along the longest axis of separation for the rest in order to get a reasonably balanced tree. The different traversal algorithms use exactly the same BVH. The implicit traversal algorithms, IMPLICIT-A/B, will, however, move all nodes to their predetermined memory location, including leaving gaps for unused nodes. Note that while IMPLICIT-A/B and SPARSE will visit exactly the same sequence of nodes as STACK, HAPALA will not, due to its need for a simple traversal-order heuristic. While the other algorithms will start with the closest child node, HAPALA will base its order on the ray direction along the split axis of a node. Although LAINE-x uses the same traversal order as STACK, it performs restarts from the root node. In the test setup, we used the CHESS scene shown in Figure 1 rendered at a resolution of 800×300 pixels and the HAIRBALL scene shown in Figure 4 rendered at a resolution of 512×512 pixels. Both scenes use 16 eye rays and 256 ambient occlusion (AO) rays per pixel. The times required to render the scenes for the different algorithms are shown in Table 1. A more detailed performance break-down is shown in Figure 5.[1]

It is clear that our stackless algorithms are very competitive in terms of performance while maintaining a small traversal state. This is true for both the simpler CHESS scene, containing 64 k triangles, as well as the more complex HAIRBALL scene containing 2.8 M triangles. For the CHESS scene, IMPLICIT-B performs best of our algorithms and is only outperformed by a full stack (STACK) or a longer short stack (LAINE-8), while using a fraction of the memory for its traversal state. In HAIRBALL, SPARSE is faster than IMPLICIT-B, even though it backtracks using parent pointers. One explanation for this is the padding of the implicit BVH; the extra pages creates more cache misses in the translation lookaside buffer. IMPLICIT-B is consistently faster than IMPLICIT-A due to a lower traversal cost. LAINE-x gets worst performance for $x = 0$ in both test scenes due to frequent restarts. As the short stack increases in size, the performance approaches STACK at the cost of a larger traversal state. From Figure 5, it is clear that even LAINE-0 has a larger traversal state than the other stack-

[1] The intersection and traversal ratios have been measured by prohibiting the compiler from inlining the intersection tests, and sampling using a profiler (Instruments 4.5 Time Profiler). The ratios are thus based on a slightly different workload.

less algorithms. The performance of HAPALA is at the lower end of the performance spectrum in our tests.

5. Conclusion and Future Work

We have presented stackless traversal algorithms for both implicit binary trees and sparse binary trees with parent pointers. These algorithms use efficient bit manipulation and have low computational overhead. At the same time, they support dynamic descent direction without having to re-evaluate sibling order or restarting. Our algorithm for sparse trees with parent pointers has been shown to perform well while requiring only a minimal traversal state of two variables. Of our two algorithms for implicit trees, Algorithm 3 is performs best in our tests and should generally be preferred. A possible exception would be if the direction bit mask maintained by Algorithm 2 is useful for other purposes.

In the future, we would like to investigate more uses for implicit binary trees, e.g., solving mathematical optimization problems where each node can be seen as an interval over an objective function. As a concrete example, our implicit traversal algorithm can be used to find the closest point on a curve evaluated using interval arithmetic. Another promising avenue for future work is investigating how stackless traversal algorithms can be employed by specialized ray-tracing hardware. They can potentially be very useful for dynamic ray reordering during traversal in order to increase memory locality.

Supplemental Materials

The C++ source for the ray tracer used in Section 4, as well as for the implemented algorithms, is available under MIT license. The given algorithms are implemented in `[name]accelerator.cpp`. The main program, found in `main.cpp`, will parse the first command-line argument for the algorithm to use, load the provided mesh `battlefield.obj`, build a bounding volume hierarchy, ray trace an image with ambient occlusion shading, and save the result to `image.ppm`. The build system is comprised of a simple make file suitable for Mac OS X or Linux with `make` and `g++` installed. It is also possible to compile the code on Windows using Visual Studio by creating a empty command-line C++ project and adding all source files to it.

Acknowledgements

The authors thank the anonymous reviewers for their valuable comments and suggestions. The HAIRBALL model is courtesy of Samuli Laine. Tomas Akenine-Möller is a *Royal Swedish Academy of Sciences Research Fellow* supported by a grant from the Knut and Alice Wallenberg Foundation.

References

HAPALA, M., DAVIDOVIC, T., WALD, I., HAVRAN, V., AND SLUSALLEK, P. 2011. Efficient Stack-less BVH Traversal for Ray Tracing. In *Proceedings 27th Spring Conference on Computer Graphics (SCCG) 2011*, ACM, New York, NY, 29–34. 39, 45, 47

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*, 6, 1555–1562. 39

KNOLL, A., HIJAZI, Y., KENSLER, A., SCHOTT, M., HANSEN, C., AND HAGEN, H. 2009. Fast Ray Tracing of Arbitrary Implicit Surfaces with Interval and Affine Arithmetic. *Computer Graphics Forum, 28*, 1, 26–40. 41

LAINE, S. 2010. Restart Trail for Stackless BVH Traversal. In *High Performance Graphics 2010*, Eurographics Associaton, Aire-la-Ville, Switzerland, 107–111. 39, 45, 47

WHITTED, T. 1980. An Improved Illumination Model for Shaded Display. *Communications of the ACM, 23*, 6, 343–349. 38

Author Contact Information

Rasmus Barringer
Lund University
Ole Römers väg 3
223 63 Lund, Sweden
rasmus@cs.lth.se

Tomas Akenine-Möller
Lund University and Intel Corporation
Ole Römers väg 3
223 63 Lund, Sweden
tam@cs.lth.se

Barringer and Akenine-Möller, Dynamic Stackless Binary Tree Traversal, *Journal of Computer Graphics Techniques (JCGT)*, vol. 2, no. 1, 38–49, 2013
http://jcgt.org/published/0002/01/03/

Received:	2012-10-10		
Recommended:	2013-02-12	Corresponding Editor:	Jaakko Lehtinen
Published:	2013-03-18	Editor-in-Chief:	Morgan McGuire