Computer Graphics International 2011

Stack-Less LBVH Traversal for Real-Time Ray Tracing

Sergio Murguía · Arturo García · Francisco Ávila · Leo Reyes

June 2011

Abstract This paper introduces a new approach for traversing and constructing a Linear Bounding Volume Hierarchy (LBVH) in CPU-based ray tracing. It reduces memory accesses and preserves traversal performance. A stack-less and pointer-less binary heap is built using a middle split on each iteration. While traversing, the next node to be visited is easily computed using simple arithmetic operations. Its construction time is faster than the current Grid and BVH implementations, allowing full scene rebuilds with the ability of handling dynamic scenes. Furthermore, traversal performance is competitive with the current BVH SAH-based implementation.

The space partitioning follows an octree scheme. After each partition, one side is assigned to a left node while the other side to a right node, building a binary tree. The partition method follows a middle split approach, which builds a heap. The primary advantage of this approach is that it is fairly easy to compute the left and right child of a parent node. The left child node is given by $2n$, and the right child is given by $2n+1$ where n is the offset of the current parent node on the array.

Keywords ray tracing · acceleration structure · BVH · LBVH · octree · real time · heap · interactive

CR Subject Classification I.3.6 · I.3.7

1 Introduction

This paper discusses a stack-less and pointer-less data structure, which improves memory handling and allows an easy traversal. The advantages of this approach are: less memory accesses and less memory required due to the stack-less scheme. Data is allocated in the same block of memory using a heap-based pointer-less structure. This scheme enables the computation of the next node to traverse through simple register operations only, which are considerably faster than common data operations. This also has the effect of improving memory access times. It is well-known that the effectiveness of each acceleration technique depends on the scene, application, and implementation. The current framework is far from being completely optimized. However, all our acceleration structures are equally affected. This paper is focused on the acceleration structure algorithm efficiency. Packet traversal is not being used, nor are SIMD instructions or an efficient multi-core parallelism.

1.1 Contributions

This paper focuses on two contributions:

1. An algorithm that builds a stack-less and pointer-less binary heap as a LBVH in $O(N \log N)$, the asymptotic lower bound for building BVHs.
2. Four simple operations that compute the left node, the right node, the parent node, and the next node to traverse. This technique shuns the stack in our implementation.

2 Stack-Less LBVH Implementation

As usually done in classic bounding volume hierarchies, six 32-bit floats are stored representing a full axis-aligned bounding box (AABB) per node. Each leaf node stores a 32-bit integer representing the number of primitives inside the node and another 32-bit integer for a primitive offset. In the case of an internal node, the number of primitives is stored as a

Sergio Ricardo Murguía Santana E-mail: sergio.r.murguia.santana@intel.com · Arturo José García García E-mail: arturo.garcia@intel.com · Francisco Ávila Beltrán E-mail: francisco.avila.beltran@intel.com · Leo Hendrik Reyes Lozano E-mail: leo.h.reyes.lozano@intel.com ·

negative number to differentiate from leaf nodes. Node size is 32 bytes which allows to allocate two nodes per cache line in x86 architectures.

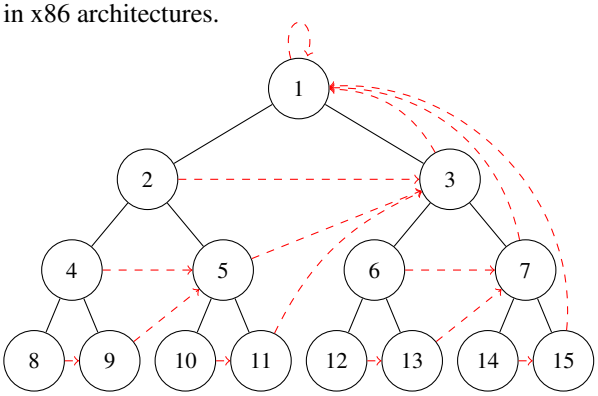

Fig. 1: Tree of depth 4, dashed lines represent next node to visit on DFS backtracking.

The LBVH construction is pointer-less and possesses the following properties:

- The Root node is at position $i = 1$.
- The Left child of a given node n is at position $2n$.
- The Right child of a given node n is at position $2n+1$.

This representation tree assumes that the number of nodes will be constant given depth D and will not be affected by the scene complexity. A tree with depth D requires an array of $2^D - 1$ elements. Some relationships among node indexes can be deduced given this heap behavior. For example:

- Parent of node n is at position $i = \left\lfloor \frac{n}{2} \right\rfloor$.
- The first *right* ancestor of node n is found removing all the least significant bits that are 0 in the binary notation of n (Check *bsf* function [1] and *firstbitlow* function [2].

It is also possible to compute the next node without the *bfs* or *firstbitlow* functions. Assuming that the least significant bit of n (that is 0) is at position p, $next(n)$ can be found as $\frac{n+1}{2^p}$. In order to compute 2^p, it can be observed that the difference between n and $n+1$ is that all the bits from 0 to p are reversed, 0's are changed to 1's and 1's to 0's, thus, using an XOR between n and $n+1$ will generate a number in which bits 0 to p are all set to 1 and the rest to 0 which is $2^{p+1} - 1$. This number could be incremented by 1 and then divided by 2 to get 2^p or alternatively, using the AND operation with $n+1$ could lead to the same result.

$$next(n) = \frac{n+1}{((n+1) \oplus n) \,\&\, (n+1)}$$

2.1 Construction

The construction algorithm consists on sorting primitives according to a tree structure. Primitives are recursively partitioned either left or right according to an specific function (called *Decision*. Internal nodes contains information of its AABB (Axis Aligned Bounding Box), and leaf nodes of its first primitive offset and a primitive counter. The recursion stops when a leaf has been reached (the node level is D, the depth of the tree), or when there is only one primitive on the node. Pointers to children nodes are not necessary since they can be computed directly avoiding costly memory accesses.

The algorithm 1 describes the recursive construction of the LBVH.

Algorithm 1 LBVH Tree Construction

Input: *Prim* an array of primitives, n current node index, *PCount* and $POffset$ the number of primitives on n and the position of them in *Prim*
Output: An array of 2^D nodes, N, which will represent the tree
Output: $Prim[POffset]$ *to* $Prim[POffset + PCount - 1]$ will be ordered according to tree traversal
if $Pcount \leq 1$ or $n \geq 2^{D-1}$ **then**
  $N[n].BoundingBox \leftarrow$ bounding box of all primitives in $POffset$ to $POffset + PCount - 1$
  $N[n].Count \leftarrow PCount$
  $N[n].Position \leftarrow POffset$
else
  for $P \in \{POffset, ..., POffset + PCount - 1\}$ **do**
    Decide if $Prim[P]$ goes to left or right child
  end for
  Swap primitives in *Prim* as needed
  Apply recursively this algorithm on the left and right children
  $N[n].BoundingBox \leftarrow$ smallest bounding box that covers $N[2n].BoundingBox \cup N[2n+1].BoundingBox$
  $N[n].Count \leftarrow PCount$
  $N[n].Position \leftarrow POffset$
end if

This algorithm can be applied to other structures such as kd-trees, BSPs or BIHs. This is possible by changing the partitioning scheme. On algorithm 1, this is represented by the line starting with *Decide* inside the *for* loop.

2.1.1 Morton Code Split

This algorithm is equivalent to a radix-2 sort using Morton codes [3] as the keys on a most-significant-bit basis. Primitives are recursively partitioned either left or right according to its Morton code. The centroid of the primitive is taken as the primitive representative point during Morton code calculation. Primitives are grouped into buckets based on their Morton codes and a BVH can be constructed using this data. This procedure is recursively applied to each primitive by checking if the current k bit is either 0 or 1. Since our LBVH construction is equivalent to radix-sort, the theoretically lower bound of the algorithm is $O(nlogn)$.

One improvement of this algorithm consists on skipping bits on the Morton code whenever all the primitives on a node contains the same information on this bit, reducing the amount of memory needed to store empty ramifications of the tree. The morton code scheme can be changed by a

median split scheme or cutting on the longest axis such as BIH [4]. This can be implemented changing the "Decide" instruction in algorithm 1.

2.2 Traversal

The algorithm is similar to a classic BVH traversal. If the intersected node is a leaf, all primitives within are tested for an intersection with R. If the intersected node is an internal node, the left child is visited. Otherwise, the function $next(node)$ substitutes the commonly used stack by computing the next node to visit. If R does not intersect the bounding box, the next node offset must be computed. This loop stops when the next node to visit is the root. It is important to mention that even invalid nodes are never visited; they are stored at its corresponding position to satisfy the previously discussed formulas.

Tree traversal is the same regardless of which tree construction is utilized. It only must satisfy three conditions: the root node is at position 1, the left child of node n is at position $2n$, and right child is at $2n+1$. Algorithm 2 explains in detail the tree traversal.

Algorithm 2 Tree traversal for ray intersection

Input: R a ray, N an array of nodes that represents the tree
Output: ID will contain primitive ID in case the ray hits one, or -1 in case it does not hit any primitive
Output: t will have the distance between the ray origin and the intersection point

```
n ← 1
ID ← −1
t ← ∞
repeat
  if R hits N[n].BoundingBox then
    for P primitive in N[n] do
      if R hits P at intersection point X then
        t′ ← distance(R.Origin, X)
        if t′ ≥ 0 and t′ < t then
          ID ← primitive P ID
          t ← t′
        end if
      end if
    end for
    if N[n] is an internal node then
      n ← leftchild(n)
    else
      n ← next(n)
    end if
  else
    n ← next(n)
  end if
until n = 1
```

3 Results

The tests were executed in our current ray tracing framework on a 3.19GHz Intel Core i7 CPU (four cores + hyper-threading = eight threads) with 6Gb of DDR3. It was compiled as a 32-bit application. Although our implementation is not properly optimized following current state-of-the-art methods such as SIMD instructions or packet traversal, it is possible to implement them in our current LBVH. Those upgrades would especially help traversal performance and equally impact on the acceleration structures in our framework. Eight cameras with different positions and different directions were set-up to measure the frame rate of the five scenes analyzed in table 1. The camera positions for the Stanford Bunny, the Stanford Dragon and the Stanford Buddha were set outside the models. The camera positions for the Conference Room and Crytek Sponza were set inside the models.

The results for our LBVH implementation are listed in table 1. Neither its construction time nor its frame rate is the fastest of the current state-of-the-art algorithms. However, our LBVH construction is a CPU single-core version of the original GPU multi-core version. If we scale those results to 8 threads the construction time might speed-up at least by a factor of $\times 4$ (approximately). According to the original GPU-LBVH paper [3], the conference room is built in 19ms while our LBVH builds the same model in 59ms in a single core which is just 3 times slower than the GPU multi-core version.

Furthermore, our current approach introduces less overhead than other stack-less structures such as the ropes scheme for kd-trees [5]. The size of the kd-tree with ropes for the Stanford Bunny is 23Mb and ours is 8Mb. Its Conference Room occupies 85Mb in memory and ours 32mb. Our LBVH possesses low memory footprint as a stack-less acceleration structure. However, a stack-based BVH utilizes less memory in almost all cases because it does not have empty nodes.

The figure 2 shows the average ray intersection tests against "boxes" (AAAB, intervals or voxels) and primitives on the Happy Buddha model. We implemented non-optimized versions of the following acceleration structures: grid, BIH and kd-tree. Since the BIH does not store boxes, the bounding box tests are calculated for each internal node visited during traversal. Fewer intersections are better. The number of intersections in the LBVH is lower than grids and similar to BVH and BIH, but far from kd-trees. The kd-tree is the most efficient structure during traversal performing few tests per ray.

4 Conclusions and Future Work

We introduced a new indexing scheme with faster construction and traversal for an LBVH implementation in CPU. Simple arithmetic operations substitute the stack in our traversal algorithm which might help GPU architectures; albeit GPU-based traversal could be highly impacted due to memory holes given its current architecture [6]. The BVH

Model	Tris	BVH	LBVH
Stanford Bunny	69K	148ms 8.8fps 4.24mb	19ms 7.2fps 8mb (D=18)
Crytek Sponza	280K	648ms 0.3fps 17.04mb	69ms 0.14fps 32mb (D=20)
Conference Room	282K	693ms 0.5fps 17.26mb	61ms 0.3fps 32mb (D=20)
Stanford Dragon	870K	2153ms 6.1fps 53.19mb	200ms 3.5fps 64mb (D=21)
Stanford Buddha	1.1M	2696ms 8.0fps 66.39mb	236ms 4.2fps 64mb (D=21)

Table 1: Construction timings and structure quality: The first row shows timings for complete hierarchy construction on a single CPU core. The second row shows the average frames per second on different camera positions and directions. The third row shows the memory footprint for each model. The SAH BVH uses a bucket approach instead of full SAH. The resolution is set at 1024^2.

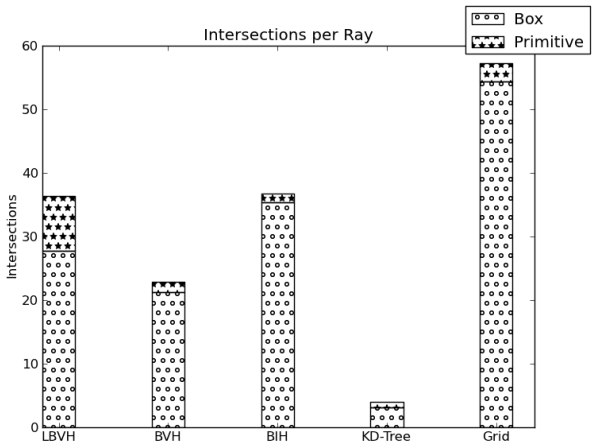

Fig. 2: This graph shows the average of ray intersection tests on Happy Buddha model with a camera position at (0,0,-1.5). Fewer intersections are better.

frame rate on GPU should be higher than LBVH frame rate on GPU though memory optimizations are currently being analyzed.

The LBVH construction is fast enough to handle dynamic scenes. It would be interesting to test models such as Fairy model or exploding Bunny-Dragon model with full rebuilds on each frame.

A deeper tree helps traversal performance (for complex scenes); but it impacts significantly on tree construction time. Since it obeys an object-splitting scheme, a high primitive density is a major challenge to overcome in spatial-scheme structures. The depth of the LBVH affects construction time, traversal performance and memory footprint. A higher depth would increase construction time and boost traversal performance on complex scenes (such as Happy Buddha and Stanford Dragon). However, memory footprint will increase exponentially.

The parallelization of our algorithm is possible since the original LBVH was proposed for GPU [3]: the morton codes of each primitive can be computed independently; the tree construction is based on a radix sort algorithm (which is also parallelizable using approaches such as `tbb::parallel_sort` or GPU sorts [7]). Contrary to the original LBVH, which requires an initial sort to build the tree, our algorithm allows to build the tree at the same time the primitives are being sorted.

Although the construction time of our LBVH could be the fastest known construction algorithm using a multi-thread approach, the traversal is not as efficient as desired. The number of ray-box and ray-triangle intersection tests is far from optimal. This is due to an inherent issue of our indexing scheme. The arithmetic operations allow the computation of the next-node-to-visit considering that the left node will always be visited first even if the direction of the ray suggests that the right node should be visited before the left node. We are working in this issue since this should boost traversal performance.

We would like to implement a breadth-first traversal algorithm allowing improved cache accesses. The construction algorithm for this new traversal is similar to algorithm 1. The advantages of this new scheme is that there will not be memory holes and the memory access will be always in one direction, improving cache usage. The main disadvantage of this new approach is the usage of a counter or a stack.

Acknowledgements The authors would like to thank Elzbieta Sanjuan for his contribution on the formulation of the arithmetic operations and to Stanford University 3D repository for Stanford Bunny, Stanford Dragon and Happy Buddha models. The Sponza model is a modified version by Crytek.

References

1. Intel. Intel architecture software developer's manual. ftp://download.intel.com/design/pentiumii/manuals/24319102.pdf
2. Microsoft. Intrinsic functions (directx hlsl). http://msdn.microsoft.com/en-us/library/ff471376(v=VS.85).aspx
3. C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, D. Manocha, Comput. Graph. Forum **28**(2), 375 (2009). URL http://dblp.uni-trier.de/db/journals/cgf/cgf28.html
4. C. Wchter, A. Keller, in *IN RENDERING TECHNIQUES 2006 PROCEEDINGS OF THE 17TH EUROGRAPHICS SYMPOSIUM ON RENDERING* (2006), pp. 139–149
5. S. Popov, J. Günther, H.P. Seidel, P. Slusallek, Computer Graphics Forum **26**(3), 415 (2007). (Proceedings of Eurographics)
6. E. Lindholm, J. Nickolls, S. Oberman, J. Montrym, IEEE Micro **28**, 39 (2008). DOI 10.1109/MM.2008.31. URL http://portal.acm.org/citation.cfm?id=1373105.1373197
7. N. Satish, M. Harris, M. Garland, Parallel and Distributed Processing Symposium, International **0**, 1 (2009). DOI http://doi.ieeecomputersociety.org/10.1109/IPDPS.2009.5161005