EG UK Computer Graphics & Visual Computing (2014)
Rita Borgo, Wen Tang (Editors)

Fast and Simple Agglomerative LBVH Construction

Ciprian Apetrei
University of Bucharest

Abstract

This paper continues the long-standing tradition of gradually improving the construction speed of spatial acceleration structures using sorted Morton codes. Previous work on this topic forms a clear sequence where each new paper sheds more light on the nature of the problem and improves the hierarchy generation phase in terms of performance, simplicity, parallelism and generality. Previous approaches constructed the tree by firstly generating the hierarchy and then calculating the bounding boxes of each node by using a bottom-up traversal. Continuing the work, we present an improvement by providing a bottom-up method that finds each node's parent while assigning bounding boxes, thus constructing the tree in linear time in a single kernel launch. Also, our method allows clustering the sorted points using an user-defined distance metric function.

Categories and Subject Descriptors (according to ACM CCS) : I.3.6 [Computer Graphics]: Methodology and Techniques—Graphics data structures and data types

1. Introduction

In the latest years, hierarchy construction methods have received more and more attention and the developing of the massive parallel computing allowed the construction of various types of trees in real-time even for millions of primitives. The need for real-time applications to process non-rigid models undergoing deformations or topological changes, fast broad-phase collision detection and particle interaction has given rise to an extensive literature on fast computation of BVHs optimized for these applications.

Bounding volume hierarchies (BVHs) are currently the most popular acceleration structures for GPU ray tracing because they are simple to construct, have low memory footprint and flexibility in adapting to temporal changes in scene geometry allowing refitting in animations. Over the last few years, ray tracing efficiency with BVHs has been tremendously improved due to novel and highly optimized traversal and constructions algorithms.

The majority of parallel BVH constructions start by sorting points along a space-filling curve and generating the tree in a top-down manner and then using a bottom-up algorithm for calculating the bounding boxes. In general, there seems to be a bias that favors top-down builders over other algorithms. Karras [Kar12] introduced a parallel method to generate a radix tree which processed each node independently and used it as a building block for other types of trees. Agglomerative or bottom-up clustering methods [WBK*08] and [GHF*13] aim to build high-quality BVHs and are typically implemented on the CPU using a greedy approach that selects at each step the best pair of clusters and combines them into a larger cluster.

In this paper we propose an improvement over the method proposed by [Kar12] by constructing the tree in parallel in a single bottom-up traversal and choosing the parent at each step while computing the bounding box. The resulted tree is partitioned the same, is simple to construct, faster and allows the use of a distance metric function by which to choose the parent.

2. Background

The majority of implementations of BVHs are constructed as described by Wald et al. [WBS07]: the primitive list is sorted based on the centroids of the AABBs. This ordered list is then split into two subsets and for each of them a bounding box is created and assigned. The process is then repeated recursively for each subset. The first parallel *linear* BVH construction method was introduced by Lauterbach et al. [LGS*09]. The method starts by assigning a Morton code to each primitive's barycenter or centroid, sorting them according to the Morton codes and then constructing the hierarchy by splitting where the highest bit differs between two Morton codes. A Morton code can be computed by mapping each coordinates to unit cube $[0,1]^3$ relative to the scene and interleaving each of the binary digits. The algorithm consists of several dependent processing steps, requires large temporary buffers, and is an order of magnitude slower than e.g. sorting the Morton codes. The algorithm was improved by Pantaleoni and Luebke [PL10] and Garanzha et al. [GPM11] which generated the hierarchy sequentially starting from the root, processing each level in parallel.

In 2012, Karras [Kar12] presented a fast method for constructing BVHs, k-d trees and octrees on the GPU that would be completely pointless on a single-core processor, but leads to substantial gains in a parallel setting. The algorithms generates all nodes of the tree simultaneously in a fully data-parallel fashion, requires no temporary storage, consists of a single fully parallelizable loop, and executes roughly two orders of magnitude faster than [LGS*09]. After construction, he used a parallel bottom-up reduction algorithm to calculate the bounding boxes. We aim to improve the method by using only the bottom-up algorithm.

Email: ciprian.apetrei@gmail.com

These methods are focused on tackling the problem of animated scenes by trading BVH quality for increased construction speed [AKL13]. The BVH quality achieved by these methods falls short of the gold standard, which makes them practical only when the expected number of rays per frame is small. To allow different quality vs. speed tradeoffs, these algorithms were combined with slower algorithms that produce higher quality trees resulting in hybrid methods or were used as a building block for other algorithms [BHH13, KT13] which take an existing low-quality BVH and modify it to match the quality of the best top-down methods. While we do not explicitly consider such methods in this paper, we believe that our approach can be combined with any appropriate high quality algorithm.

3. Binary Radix Tree Construction

A radix tree (also known as Patricia tree or radix trie or compact prefix tree) is a binary tree which is built based on the common prefixes of each key. In our case, keys are represented as bit strings. Each leaf node contains a single key and each internal node corresponds to the longest common prefix of all the keys it covers. Splitting a sequence of keys into two subsets summarizes to finding the highest differing bit. A naive construction algorithm would start partitioning the tree from the root by finding the first differing bit and then creating the child nodes and processing each child recursively.

Karras [Kar12] introduced a novel binary radix tree construction algorithm and used it as a building block for other types of trees. In order to generate the hierarchy in fully parallel fashion, the algorithm processes each node independently and performs one binary search to find the range of keys that each internal node covers and another binary search to find the split point for every internal node in the hierarchy. Then it uses a bottom-up traversal algorithm for bounding box calculation where each thread starts from a single leaf and advances toward the root and every node calculates its bounding box by looking at the bounding box of the children. To avoid duplicate work, each internal node has an atomic flag which prevents the first thread to enter and lets the second one through.

A binary radix tree is a compact tree in the sense every node either has two children or none. Therefore, a binary radix tree with n leaf nodes has $n-1$ internal nodes corresponding to a split point between two keys. In our approach we consider only ordered trees, where the keys are sorted and each internal node covers a linear range of keys.

Algorithm. In order to further optimize the construction algorithm, what we want is to do both hierarchy generation and bounding box calculation in a single kernel launch. To achieve this, we must construct the tree in a bottom-up fashion. What this means is that we start from each key and progressively group them together into larger and larger clusters until the root cluster containing all the keys is formed.

Similar to [Kar12], the basic idea is to utilize a specific node layout to establish a connection between the indices of internal nodes and the ranges of keys that they cover. In the case of BVHs, [Kar12] uses the connection to determine the children of each internal node in a separate processing step, followed by a custom bottom-up reduction algorithm to calculate the per-node AABBs.

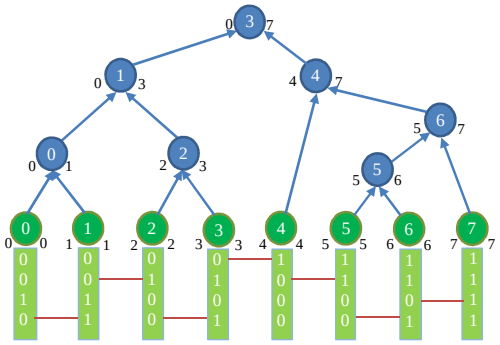

Figure 1: *Ordered binary radix tree. Leaf nodes are numbered from 0 to 7 and internal nodes from 0 to 6. Each leaf node contains a set of 4-bit keys in lexicographical order. The numbers in the left and right of each node represent the left and right ends of the linear range of keys covered by that respective node. The red lines between each key represent the index of the highest differing bit.*

Our approach combines the two steps by tracking the ranges of keys as a part of the bottom-up reduction and using them to deduce the index of the parent node at each step. Every leaf node covers a range of one key, and every internal node merges the ranges of its children.

We choose a node layout where each internal node i will split the hierarchy between keys i and $i+1$. The split point of an internal node is defined by the *last key* of its left child and the *first key* of its right child. What this implies is that the highest differing bit between the keys covered by an internal node i will always be between keys i and $i+1$. In contrast to the method proposed by [Kar12], where for each node he uses a binary search to find the highest differing bit, in our layout this isn't necessary as we already know it.

From a bottom-up construction point of view, for a given node that covers keys $[a,b]$, the node layout implies that indices $a-1$ and b will correspond to ancestors of the node and one of these ancestors will be its parent.

The general idea of our method is that we can analyze a split position of a given internal node to measure the dissimilarity between two subtrees. Being a bottom-up hierarchy construction, we start from each individual key and use the split point as an indicator for which node to choose as parent. To choose the parent, we define a function $\delta(i)$ as the index of the highest differing bit between the keys covered by node i. Because of the way a radix tree is defined, $\delta(x) > \delta(y)$ must be true if x is the ancestor of y. Thus, we can conclude that the index of the parent node is $a-1$ if $\delta(a-1) < \delta(b)$, and b otherwise.

Because we already know where the highest differing bit is for each internal node, the δ function basically represents a distance metric between two keys. Unlike the δ used by [Kar12], we are interested in the index of the highest differing bit and not the length of the common prefix. In practice, logical xor can be used instead of finding the index of the highest differing bit as we can compare the numbers. The higher the index of the differing bit, the larger the number.

Let us assume that the leaf nodes and internal nodes are stored in two separate arrays, L and I, the same way as proposed by [Kar12]. Each leaf node stores exactly

one key. The hierarchy construction starts from each leaf node and walks towards the root by finding the parent at each step. We process an internal node only after it has both its children set. To find the parent of each node we have to look at the nodes that split the hierarchy at the left and right ends of the keys covered by the respective node. We initially know that each leaf node L_i covers the range of keys [*i*,*i*]. As we described earlier, we look at the internal nodes with the index *i-1* and *i* and compare the values returned by the δ function. The one with the lowest value will be the parent because it splits the hierarchy between two more similar clusters (subtrees) than the other node. Because each parent merges the ranges of its children, the current node will pass to the parent the opposite range of the keys it covers. When we reach an internal node the algorithm works in the same way as we only need to know the range of keys it covers in order to find the parent.

In the figure, the only possible parent for L_0 is I_0. In the case of L_1, we compare δ(0) and δ(1) and find that δ(0) has the smaller value and that means the parent is I_0. After finding at which end of the range of keys is the parent, each node passes to their parent the opposite end. So, L_0 will pass 0 and L_1 will pass 1. Now that we have set the parent child relationship, the algorithm proceeds to process internal node I_0 where we know that it covers the keys [0,1].

GPU Implementation. We have used the parallel bottom-up reduction presented by Karras [Kar12] to implement the algorithm. The only addition is that instead of knowing the parent by generating the hierarchy beforehand, we determine it by using the range of keys it covers.

Because we can't detect if all the leaves covered by a given internal node are being processed by the same thread block, the disadvantage is that we can't reduce the number of global atomics by using the faster shared memory atomics.

During construction, our algorithm needs two integers per node to store the range of keys while [Kar12] needs only one, to store the parent of each node for the bottom-up traversal.

4. Overview

The algorithm for hierarchy construction can be summarized in 3 steps: (1) The first step consists of sorting the keys. (2) (Optional) We do a separate kernel launch in order to calculate the δ function for each internal node beforehand. We can choose what the δ function will compute depending on the application requirements. (3) We use a node layout where we associate each internal node to a split point between two keys. Then, we construct the hierarchy by starting from each leaf node. Because we know that leafs covers a single key, we can find the parent by choosing between the two ancestors at the left and right of the key it covers. We choose the parent according to the δ function that indicates a distance metric between the two keys where each internal node splits the hierarchy, the one with the lower value is the parent. Each parent then merges the ranges of its children and uses it in the same manner to advance towards the root.

By terminating the first thread that enters a node and letting the second one through, each node is processed by exactly one thread which leads to O(n) complexity.

```
1: def ChooseParent(Left, Right, currentNode)
2:     If ( Left = 0 or ( Right != n and δ(Right) < δ(Left-1) ) )
3:     then
4:         parent ←δRight
5:         InternalNodes_parent.childA ←currentNode
6:         RangeOfKeys_parent.left ←Left
7:     else
8:         parent ←Left - 1
9:         InternalNodes_parent.childB ←currentNode
10:        RangeOfKeys_parent.right ←Right
```

Figure 2: *Pseudocode for choosing the parent. Left and Right represent the range of keys covered by the current node. The root of the tree will be stored in the left child of the n-th internal node, where n represents the size of the key array.*

As presented by [Kar12] the radix tree construction algorithm can be used as a building block for other types of trees (e.g., BVH, k-d Tree, octree).

Resulted Tree. By knowing the range of each leaf node and implicitly two of its ancestors, the constructed tree is similar to a threaded binary tree. A binary tree is *threaded* by making all right child pointers that would normally be null point to the inorder successor of the node, and all left child pointers that would normally be null point to the inorder predecessor of the node.

Although we need the range of keys covered by each node only during construction, they could have some practical uses. For each node that covers a linear range of keys [*a*,*b*], we can use the range to find the index of first common ancestor between the current node and the nodes that contain the keys that are before *a* or after *b*. The right ancestors of each node are essentially equivalent to recording the steps of a depth first traversal and are similar with the escape links described by Toczek et al. [TDS10] which used them to traverse the tree without using a stack.

As the index of each internal node corresponds to a split between the keys covered by it, the layout of the nodes is preferable considering data locality.

Bounding Volume Hierarchy. The particularly interesting aspect about this algorithm is that it allows one to use an arbitrary distance metric for δ. This can be advantageous because the LBVH construction method can now be made independent of how we sort the points by choosing an appropriate metric. For example, for each internal node *i*, δ(*i*) could compute the surface area of the two keys where the node splits the hierarchy. We can then choose the parent in the same mode by comparing the values returned by the δ function. Alternatively, we can use the squared distance to compute the distance between two centroids. In these cases, it proved better to do a separate kernel launch where we compute the value of δ corresponding to each internal node.

5. Results

To explore the performance of our algorithms, we have implemented them in CUDA 6.0 along with the method proposed by Karras [Kar12] and benchmarked their performance on a NVIDIA GeForce 745M installed in a laptop with 2.60 GHz Intel Core i5 CPU running Windows 8.1.

Table 1 shows a breakdown of hierarchy construction time for a set of test scenes. We have used the thrust library to sort the keys for all methods and used 30-bit Morton codes. The method presented by Karras has two values, one representing the hierarchy construction and the other one the bounding box calculation. Our radix tree construction algorithm is implemented in a single kernel launch. The last column is a variant of our hierarchy building method which uses the squared distance between centroids as the δ function. We first used a kernel launch to precompute the squared distances and then another kernel launch to create the hierarchy. The difference in construction time between the method proposed by Karras (Previous) and ours gradually increases with the number of triangles. Both methods amount to less than 50% of the time required to sort the keys.

Discussion. Our algorithm improves upon all aspects of [Kar12]: it is strictly faster, produces an equivalent hierarchy, consists of only a few lines of code, and opens up new possibilities in terms of construction heuristics. The contribution of this paper is the ultimate simplification to the hierarchy generation step that essentially removes it altogether. Because the algorithm allows one to use an arbitrary distance metric for δ, it could be adapted for a number of different applications and this new freedom could be potentially used to improve the tree quality.

Scene	Sort	Previous	Radix tree (our)	Squared distance δ
Stanford Bunny (69K tris)	14.9	1.78 4.53	5.56	0.85 4.74
Armadillo (345K tris)	32.1	5.01 10.0	12.03	2.4 11.9
Skeleton Hand (654k tris)	77.8	14.1 28.3	32.5	6.54 31.8
Stanford Dragon (871K tris)	102	19.6 37.1	42.9	8.61 42.2
Happy Buddha (1087K tris)	125	23.2 46.8	53.4	10.7 52.7
Turbine Blade (1765K tris)	210	37.3 73.9	85.9	17.3 85.3

Table 1: *Construction time for Karras (Previous), our method for radix tree construction and the BVH construction method where we use the squared distance to compute the distance metric in milliseconds. The processing consists of sorting the Morton codes, building the hierarchy and bounding box calculation.*

References

[AKL13] Aila T., Karras T., Laine S. 2013. On Quality Metrics of Bounding Volume Hierarchies. In Proc. High-Performance Graphics 2013, 101-107. 2

[BHH13] Bittner H., Hapala M., Havran V. 2013. Fast Insertion-Based Optimization of Bounding Volume Hierarchies. In Computer Graphics Forum 32, 85–100. 2

[GHF*13] Gu Y., He Y., Fatahalian K., Blelloch G. 2013. Efficient BVH Construction via Approximate Agglomerative Clustering. In Proc. High-Performance Graphics 2013, 81-88.

[GPM11] Garanzha K., Panteleoni J., McAllister D. K. 2011. Simpler and faster HLBVH with work queues. In Proc. High Performance Graphics 2011, 59-64. 1

[Kar12] Karras T. 2012. Maximizing parallelism in the construction of BVHs, octrees, and k-d trees. InProc. High Performance Graphics 2012, 33-37. 1, 2, 3, 4

[KT13] Karras T., Aila T. 2013. Fast Parallel Construction of High-Quality Bounding Volume Hierarchies. In Proc. High Performance Graphics 2013, 89-100. 2

[LGS*09] Lauterbach C., Garland M., Sengupta S., Luebke D., Manocha D. 2009. Fast bvh construction on GPUs. In Computer Graphics Forum 28, 2 2009, 375-384. 1

[PL10] Pantaleoni J., Luebke D. 2010. HLBVH: Hierarchical LBVH construction for real-time ray tracing of dynamic geometry. In Proc. High Performance Graphics, 87–95. 1

[TDS10] Toczek T., Houzet D., Mancini S. 2010. Efficient Stackless Ray Traversal for Bounding Sphere Hierarchies with CUDA. International Conference on Computational science 2010, Elsevier, 1. 3

[WBK*08] Walter B., Bala K., Kulkarni M., Pingali K. 2008. Fast agglomerative clustering for rendering. In Proc. IEEE Symposium on Interactive Ray Tracing, 81–86. 1

[WBS07] Wald I., Boulos S., Shirley P. 2007. Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies. ACM Transactions on Graphics 26, 1. 1