Fused BVH to Ray Trace Level of Detail Meshes

Paritosh Kulkarni
Advanced Micro Devices, Inc.
Canada

Sho Ikeda
Advanced Micro Devices,Inc.
Japan

Takahiro Harada
Advanced Micro Devices,Inc.
USA

ABSTRACT

The paper presents different strategies for fusing Bounding Volume Hierarchies (BVHs) of varying level of detail (LODs). Our method makes it possible to traverse BVHs of varying LODs in a cache-friendly way for path tracing. Considering BVH of the finest level (we call it the Base LOD BVH), the method defines a strategy to fuse subtrees from BVH of other LODs to the Base LOD BVH. Our method proposes a heuristic to find such subtrees in non-Base LOD BVHs and an algorithm to find fusion or insertion points for these subtrees in the Base LOD BVH without much increasing the overall the volume of the Base LOD BVH and the number of nodes traversed as a result of this fusing.

KEYWORDS

ray tracing, global illumination, Bounded Volume Hierarchies, Level Of Detail

1 INTRODUCTION

Monte Carlo ray tracing is a simple and robust rendering algorithm that has been studied for years [Kajiya 1986]. Ray casting is the core component of ray tracing, which scales better in terms of the number of primitives in the scene than rasterization which has linear complexity. However, even with the logarithmic complexity in ray tracing, the computational cost is still high for real-time applications. Therefore, developers have been attempting to minimize the number of rays to be cast, and rely heavily on denoising in video games [Akenine-Moller et al. 2019].

The level of detail (LOD) is one of the optimizations tried to reduce the cost of ray casting. Current approaches like [Lee et al. 2019] extend the current ray tracing pipeline to enable the selection of LOD. It assumes Bounding Volume Hierarchy (BVH) for each LOD is present in memory so the memory overhead is large, and when we to switch to a different LOD, we will start traversing an entirely different tree causing a cache performance hit. Another LOD approach [Ikeda et al. 2022] is to generate proxy geometry using existing BVH to simulate LOD. This approach does not need all LOD BVHs present in memory so the memory overhead is very low, ray casting performance improvements are good but it suffers issues like darkening bias. Our method takes user-generated LOD meshes as input and generates a single fused LOD BVH which results in memory compact BVH representation and cache-friendly access when switching LOD during ray casting.

The paper describes a heuristic to find a cut in non-Base LOD BVHs which we call the Mesh Cut, an algorithm to fuse subtrees from a cut in the Base LOD BVH, select proper depth in the Base LOD BVH for fusion, and handle possible collisions during fusion.

email: { Paritosh.Kulkarni, Sho.Ikeda, Takahiro.Harada }@amd.com
Advanced Micro Devices, Inc. Technical Report No. 22-10-b79d, October 4, 2022.

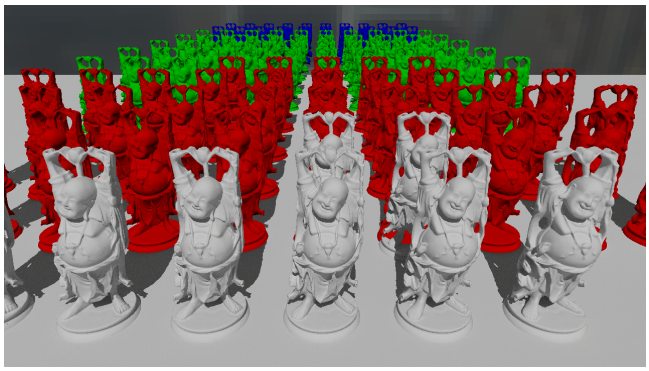


Figure 1: Buddha model rendered with distance based LOD selection as described in [Lee et al. 2019] for shadow and diffuse rays.

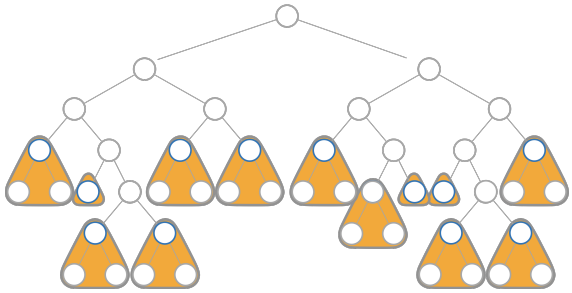


Figure 2: A mesh cut formed by subtrees from LOD 1 BVH with a specified threshold.

2 FUSED BVH FOR LOD MESHES

Instead of storing a BVH for each LOD in memory, we propose to store a single BVH with additional information. We start with the Base LOD BVH and propose a generalized fusion method to find a cut of subtrees from non-Base LOD BVHs and insert them in the Base LOD BVH. We will now describe the method assuming we have the two LODs: 0 and 1.

2.1 Finding a Mesh Cut

We start by finding subtrees forming a cut in LOD 1 BVH at decent depth. For this purpose, we introduce a threshold T as a number of child nodes. We start traversing LOD 1 BVH until we reach a node with the number of child nodes less than or equal to T. We add such a node to a mesh cut and continue traversal till we have visited all the nodes as shown in Fig 2. If we want a finer mesh cut, we can set T=1. This T will result in a mesh cut with all the triangle primitives.

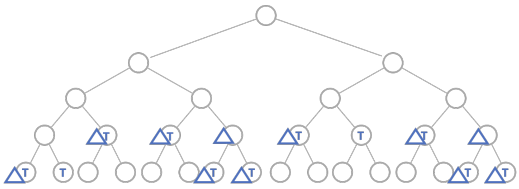


Figure 3: An illustration of the fusion of subtrees from a single LOD to the Base LOD BVH. Note that the inserted subtrees in blue do not form a cut of the tree. The terminal nodes with the label "T" in this figure form a cut.

2.2 Fusing Mesh Cut in the Base LOD BVH

As briefly described already, the method starts with finding a mesh cut in a non-Base LOD BVH. Our method does not put any assumption about the BVH building method so any method can be used. We go through all nodes in a mesh cut and it is inserted at an appropriate internal node in the Base LOD BVH by traversing the Base LOD BVH from the root node. At each node in the Base LOD BVH, if the node overlaps a node in a mesh cut we move down the Base LOD BVH. We also check the termination criteria at each node for which we use the surface area of an AABB of the node. We terminate at the first Base LOD BVH node for which a node in the mesh cut when merged is increasing the surface area of the node in the Base LOD BVH.

We only allow inserting a single node in a mesh cut to a single node in Base LOD BVH. This simplifies the data structure and the traversal logic. But, the logic can report the same node from the Base LOD BVH as the insertion point for multiple nodes in the mesh cut. This results in a conflict of insertion. When a conflict happens, we descend the conflicted node in the Base LOD BVH choosing the child that has the maximum overlapped surface area with a node in mesh cut and report it as the insertion point.

After all nodes in a mesh cut are inserted, we set a flag for a terminator of a LOD. This is necessary to avoid unnecessary traversal as shown in Fig 3. Think of a case where we traverse the Base LOD BVH to find an intersection for LOD 1 BVH. Then after testing a LOD 1 fused node, we do need to descend the Base LOD BVH if there is another node containing a LOD 1 fused node in the descendants. However, if there is no node containing a LOD 1 fused node in the descendants, we do not need to descend the Base LOD BVH. Therefore, we tag the very last fused node with a LOD 1 as a terminator. This fused node can be found by a bottom-up traversal of the tree.

However, there is no guarantee that nodes with LOD 1 primitives form a cut of the BVH as shown in Fig 3. as the fused node insertion is arbitrary. When there is a part of the tree where no fused node exists for LOD 1, we traverse the Base LOD BVH to the leaf node which is not necessary if we are only interested in intersecting against LOD 1 primitives. Therefore, we also need to insert a terminator flag to a node in the Base LOD BVH where we can say there is no need to descend the tree anymore. The node we need to flag can be found by another traversal of the Base LOD BVH. The condition is if a node does not have a descendant with a terminal flag and the sibling of the node has a descendant with a terminal flag, the node needs to be flagged as a terminal.

The last thing we need to do is refit the AABBs in the Base LOD BVH. The AABBs computed for the Base LOD BVH do not always enclose the subtree fused. Thus, we need to refit the AABBs by a bottom-up traversal of the Base LOD BVH. When we encounter a node having a fused LOD 1 subtree, the AABB of the node is adjusted to enclose the fused subtree.

2.3 Optimal Fusing Based on Node Cost

After a mesh cut is formed we will be fusing these subtrees from the mesh cut in the Base LOD BVH. However, the order of the root nodes of the subtree in the mesh cut is significant. Let's assume the root nodes of the subtrees in a mesh cut are present in descending order of efficiency cost such that the first node is the most efficient node. Efficiency cost deals with many scenarios and tries to find the surface area-wise compact node. So when we fuse these subtrees from the mesh cut, the subtree with the most efficient root node will be fused first. The inefficient root nodes will be fused later on so they will be fused at a depth higher than previously fused efficient root nodes. After every subtree is fused, we will have to adjust the volume of the Base LOD BVH and fusing subtrees with the inefficient root node at higher depth will increase the volume of the Base LOD BVH a lot. This results in visiting more nodes in the traversal of the Base LOD BVH.

To avoid this problem we define a cost model in section 2.4 to calculate the inefficiency cost of a node. While forming mesh cut we also calculate the inefficiency cost of a node being added to a cut and then sort the cut based on this cost. We observed that if we use ascending order of efficiency cost for insertion the number of nodes visited per pixel increased by 16% while descending order increased it by 8% as compared to the baseline.

2.4 Cost Model for Selecting Nodes in Mesh Cut

In order to find the optimal mesh cut in LOD 1 BVH we should first choose the nodes with lowest cost overhead (surface area increase) in the tree. To achieve this, we need a node inefficiency measure.

The first inefficiency measure M_{SUM} for a node corresponds to a component of the cost model used by [Lauterbach et al. 2006] and is evaluated as:

$$M_{SUM}(N) = \frac{SA(N)}{1/|\text{ children of } N\ | \cdot \sum_{X \in \text{ children of } N} SA(X)} \quad (1)$$

This measure estimates the relative increase of the surface area of the node with respect to average surface areas of the children. Thus, if there is a lot of empty space inside the node, this measure will be large.

The second inefficiency measure for a node used by [Bittner et al. 2013] is evaluated as:

$$M_{MIN}(N) = \frac{SA(N)}{\text{Min}_{X \in \text{ children of } N} SA(X)} \quad (2)$$

This measure aims to handle the situation when the node contains child nodes of significantly different sizes (e.g. one large node representing the whole terrain and a small node representing a particular object on the terrain). Then the first measure equation (1) defined above might not identify such a node as problematic as it takes the average surface area of the children which in this example

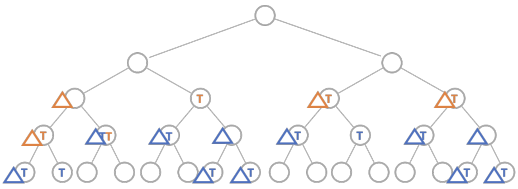


Figure 4: An illustration of multiple LODs fused in the Base LOD BVH.

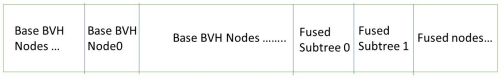


Figure 5: Memory layout of fused BVH. The subtrees from the LOD 1 BVH fused in with the Base BVH.

will still be large. On the contrary, the equation (2) measure will detect such a situation.

The third measure is the surface area of the node itself. This will help us determine the contribution by the node itself

$$M_{AREA}(N) = SA(N) \tag{3}$$

As observed by [Bittner et al. 2013] we combine the above three measures to obtain the final inefficiency measure for a node.

$$M_{COMB}(N) = M_{SUM}(N) \cdot M_{MIN}(N) \cdot M_{AREA}(N) \tag{4}$$

2.5 Building Fused BVH with Many LOD Meshes

Inserting more LODs is an extension of the method described in section 2.2. Instead of simply inserting a single level, we loop through all the levels and keep inserting subtrees from a mesh cut to the Base LOD BVH. After that, we set terminal flags. Note that terminal flags need to be set for each level. Fig 4 shows an example of where we store 3 levels, 2 levels are inserted into the Base LOD BVH. Note that there is a node that has terminator flags for both levels.

We added single 32-bit data for each node in the Base LOD BVH to store this information. More specifically, our implementation stores up to 6 levels for an instance. We allocated 3, 5, and 24 bits to store level, terminal flag (1 bit per level), and root node index of the subtree from a mesh cut.

2.6 Traversal of Fused BVH

The traversal of a fused BVH is similar to a traversal of a regular BVH but with an additional conditional execution at each node in the Base LOD BVH. When we visit a node, we check the additional data as described in section 2.5 and perform an intersection if the node has the level we are interested in. From this point on we will continue traversing the LOD subtree that is fused at this node. After that, we check the terminal flag of the level in the node. If the node is a terminal of the level, we skip the traversal of its children. Otherwise, we execute the traversal as usual.

If we want to traverse for LOD 1, we will always start from the Base LOD BVH. The fused LOD 1 subtrees are stored with the Base LOD BVH in continuous memory as you can see from Fig 5. This will result in cache-friendly access patterns. As we find the subtrees to fuse from LOD 1 BVH at a certain decent depth hence, we are avoiding traversal of all the nodes above that depth in LOD 1 BVH and the subtrees are compact in memory. In absence of fusing the traversal of the LOD, BVHs are random as they are not packed in continuous memory.

3 CONCLUSIONS AND FUTURE WORK

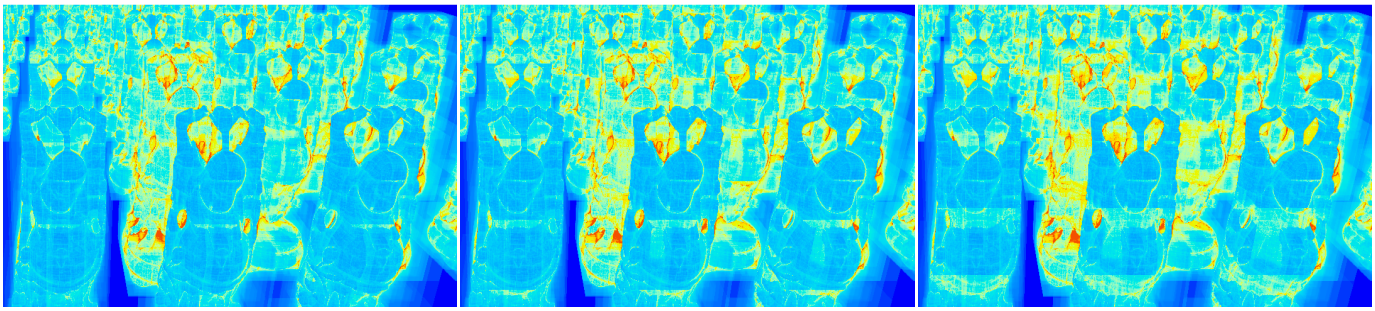
We proposed a generalized framework for fusing the LODs which will result in a memory compact single BVH. The reduction in the memory overhead as a result of fusing outweighs the additional cost added by an overall increase in the volume of the Base BVH. However, We need better strategies for a case where many collisions occur and we reach the leaf node of the Base LOD BVH without finding the fusing point. Also, we would like to investigate the efficient GPU implementation for fused BVH. The fused BVH is definitely appealing for the GPU where reduction in the memory overhead is a big win. Lastly, we would like to investigate different BVH traversal logic for fused BVH. One of the directions is to sort rays with the same LOD in batches and do ray stream traversal.

4 RESULTS

The process of fusing LOD BVH does increase the overall volume of the Base LOD BVH but this increase is very well compensated by the reduction in the memory overhead. To quantify the memory overhead reduction we used Buddha, Dragon, and Erato models. The result of the fusing is the BVH with a very compact memory layout. The LODs other than the Base BVH are reduced in memory as seen from Table 1. We achieve 60.73%, 60.74%, and 60.73% reduction in the memory overhead for Buddha, Dragon, and Erato models, respectively. The resulting reduction in the memory overhead will eventually reduce the memory traffic leading to better performance in ray casting.

To quantify the number of nodes visited per pixel we prepared a scene with 100 instances of the Buddha model, 4 LODs fused in the Base LOD BVH with different values of T and compare it with [Lee et al. 2019] method (we consider it baseline) using our in-house path tracer. As we fuse more LODs or use smaller values for T the number of subtrees to be fused increases resulting in more collisions and causing the subtrees to be fused at higher depth. This will increase the overall volume of the Base BVH; hence, the number of nodes visited per pixel is increased from 9% to 13% as we can see in Fig 6a, 6b, 6c and Table 2. We observed with $T = 256$ we got an optimal trade-off between memory compaction and an increase in the number of nodes visited.

To quantify the effect of the cost model described in 2.4 on the number of nodes visited per pixel we used the buddha scene with 100 instances, 4 LODs fused in the Base LOD BVH with T=256, 1280x720 resolution, 64 samples per pixel, and 10 recursion depths as parameters. We observed that for the baseline method on average 1876 nodes/pixel were visited, while with fused BVH we visited 2040 nodes/pixel. If we do not use the optimal fusing strategy explained in the supplementary document, the node visited per pixel count goes to 2176 nodes/pixel. So, if we use ascending order of efficiency cost for insertion the number of nodes visited per pixel increased by 16% while descending order increased it by 8% as compared to the baseline.



(a) LOD0 + LOD1 (b) LOD0 + LOD1 + LOD2 (c) LOD0 + LOD1 + LOD2 + LOD3

Figure 6: Zoomed view of Fig 1. Shows a heat map of the number of nodes visited per pixel during ray cast for primary rays with the fused BVH. We set 1280 x 720 resolution and T=256. As we can see with the fused BVH the number of nodes visited are increased as we add LODs.

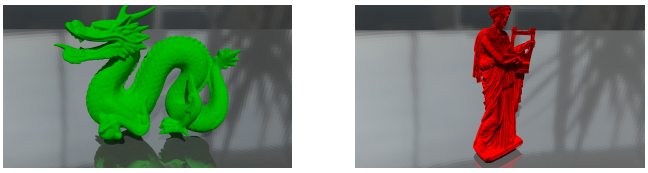


Figure 7: Rendered scene with Dragon and Erato Models.

LOD	Buddha Base-line(kb)	Buddha Fused T=256	Dragon Base-line	Dragon Fused T=256	Erato Base-line	Erato Fused T=256
1	11135.29	6927.6	25651.13	15957.82	24815.80	15439.39
2	1391.68	865.74	10260.41	6383.61	10031.80	6240.60
3	389.44	242.27	2565.05	1595.64	2639.80	1642.43

Table 1: LODs and size of the BVHs in Kb for the Baseline vs the Fused BVH. Buddha model LOD 0, 1, 2, 3 with 200K, 80K, 10K, 3K primitives respectively. Dragon and Erato models LOD 0, 1, 2, 3 with 400K, 200K, 80K, 10K primitives respectively. LOD 0 BVH or the Base BVH is not affected by fusion hence its sizes are not provided.

number of nodes visited per pixel nodes/pixel				
LOD levels	Baseline	Fused BVH T=128	Fused BVH T=256	Fused BVH T=512
0+1	480.18	584.82	559.05	552.13
0+1+2	479.96	603.02	570.16	559.05
0+1+2+3	479.89	617.90	579.75	560.99

Table 2: Measurement of nodes/pixel visited with multiple LODs with Buddha scene.

REFERENCES

Tomas Akenine-Moller, Eric Haines, and Naty Hoffman. 2019. Real-time rendering. AK Peters/crc Press.

Jiří Bittner, Michal Hapala, and Vlastimil Havran. 2013. Fast insertion-based optimization of bounding volume hierarchies. In Computer Graphics Forum, Vol. 32. Wiley Online Library, 85–100.

Sho Ikeda, Paritosh Kulkarni, and Takahiro Harada. 2022. Multi-Resolution Geometric Representation using Bounding Volume Hierarchy for Ray Tracing. GPUOpen technical report 22-02-f322 (2022).

James T. Kajiya. 1986. The rendering equation. In Computer Graphics. 143–150.

Christian Lauterbach, Sung-Eui Yoon, Dinesh Manocha, and David Tuft. 2006. RT-DEFORM: Interactive ray tracing of dynamic scenes using BVHs. In 2006 IEEE Symposium on Interactive Ray Tracing. IEEE, 39–46.

Won-Jong Lee, Gabor Liktor, and Karthik Vaidyanathan. 2019. Flexible Ray Traversal with an Extended Programming Model. In SIGGRAPH Asia 2019 Technical Briefs. 17–20.