EG UK Computer Graphics & Visual Computing (2024)
A. Slingsby and D. Hunter (Editors)

EBBVH: A Novel Method for Constructing Bounding Volume Hierarchies

M. Houghton[1] and K. Spoerer[2]

[1] University of Nottingham, mh33188@gmail.com
[2] University of Nottingham, pszks@nottingham.ac.uk

Abstract

We present an attempt to improve upon the construction of the most prevalent acceleration structure that is used in ray traced rendering techniques, the Bounding Volume Hierarchy. Our improvement is a novel technique for BVH construction called 'Edge-Based Bounding Volume Hierarchy'. This algorithm uses a hybrid top-down & bottom-up approach to improve performance for raytracing in large scenes, by up to 10x in some scenes.

CCS Concepts
• ***Computing methodologies*** $\rightarrow$ ***Ray tracing;***

1. Introduction

The computational cost of ray tracing methods comes from the intersection tests between rays and primitives in a scene, in a naïve implementation each ray is tested against each primitive in a scene. One geometry intersection test has a time complexity of O(N) where N is the number of primitives in the scene, these primitives are typically triangles. In most other cases this time complexity would be quite good however in modern applications there can be millions of primitives in each scene. These millions of primitives may be tested several times per pixel, this means that the naïve O(N) approach won't be enough for a responsive real time application. This would even make offline rendering infeasible for even moderately complex scenes. To make ray tracing feasible even in an offline situation a spatial acceleration structure must be used. The most prevalent acceleration structure is the bounding volume hierarchy (BVH).

2. Previous Work

2.1. Top Down Approaches

Top-down construction methods for creating BVHs involve first creating a root node that contains all the primitives in the scene. Then recursively splitting this into disjoint groups until there are leaves that contain the desired number of primitives.

A distinguishing factor in these types of algorithms is deciding in which axis to split the node. Along with deciding where in that axis to split it. The simplest approach is to split the node spatially halfway along the longest object-space axis of the axis-aligned bounding box (AABB), using the triangles centroid to determine which child node they are placed in. Another is to use the surface area heuristic where each split point in each axis is evaluated and the best split is taken. Evaluating all possible split locations for the SAH construction is expensive so instead only a few possible split locations are considered [Wal07].

2.2. Bottom Up Approaches

Alternatively, to the top-down approaches there are bottom-up approaches such as Agglomerative Clustering [WBKP08]. These algorithms represent each primitive as a cluster and then forms them into larger clusters based on a cost function. This process is then repeated till only one root cluster remains. This process generally produces higher quality results but takes more time when compared to top-down approaches [MOB*21]. The Approximate Agglomerative Clustering [GHFB13] algorithm uses the space-filling Morton Curve to limit the nearest neighbour search area for Agglomerative Clustering. Parallel Locally Ordered Clustering [MB18] (PLOC) extends the combination of Morton Curve and Agglomerative Clustering to allow for bottom-up construction on the GPU. PLOC recognizes that if two clusters are the nearest neighbours of each other then they can be merged straight away, and in parallel. It then searches a predefined range up and down the sorted list of clusters to find the nearest neighbour cluster.

3. Proposed algorithm

The initial idea was sparked by a poster [AF23] from Imagination Technologies at the High-Performance Graphics conference. This poster describes a method to pair triangles within models to construct quads. We are using this posters definition of quads. Quads are two triangles that share an edge, they do not need to be coplanar.

DOI: 10.2312/cgvc.20241222

This is useful for real time ray tracing as most models are stored and transferred as triangles but, in GPU hardware, ray-quad intersection tests can be more efficient as we can share the computation for the shared edge. They require less memory, only 4 indices as opposed to 6 with a quad constructed from 2 triangles. It also reduces the number of primitives in a BVH which reduces both the memory and time to traverse it.

Our method of BVH construction EBBVH builds upon this method by using this pairing approach to progressively construct a BVH. The intuition behind this algorithm is that methods such as PLOC and LBVH [LGS*09] function by grouping together triangles that are nearby each other. They do this by exploiting the space-filling Morton Curve, assuming that if two primitives are close along the Morton Curve they are close together in regular 3D space. EBBVH uses the fact that if two triangles share an edge then we know for a fact that they are close by as they share this edge i.e. the triangles touch each other.

This idea is then applied iteratively where at the start, individual primitives are considered and paired if they share an edge, then these pairs are clustered together if the primitives inside them share an edge, these clusters are then considered, this is repeated until no more clusters can be clustered together i.e. none of the clusters share edges with each other.

This will leave us with several root nodes, these can then be thought of as the leaf nodes of another BVH so we then use another construction algorithm to combine these nodes into a single BVH. In our current implementation a recursive-split method using spatial median splits, detailed in [WK06], was used however this can be substituted for any approach.

In practice this means that EBBVH is a combination of a bottom-up phase and a top-down phase .

Algorithm 1 shows the pseudocode for our EBBVH construction method. The initializeEdges function creates an array of edges, for each triangle it creates 3 edges: v0-v1, v1-v2, v2-v0. InitializeClusters creates an initial cluster for each triangle as this is a bottom-up method.

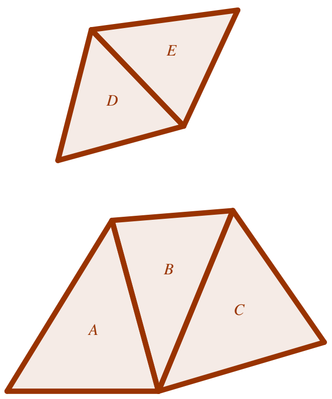
The edge list is sorted using indices[0] so that edges that are shared are next to each other, each triangle will add all 3 of its edges so shared edges appear twice. It is then sorted using the bounding box surface area (BBSA) of the edge so that the largest shared edges are paired first the reason for this is detailed in [AF23].

```
Input: Vertices, Indices, PAIR_CUTOFF
Output: Hierarchical Clusters
edges = InitializeEdges(Vertices, Indices);
sort edges using edge.indices[0];
stable sort edges using BBSA;
clusters = InitializeClusters(Vertices, Indices);
done_clusters = [];
HashMap<int, int> pairs;
roots = initializeRoots(clusters);
int i = clusters.size();
int numPaired = 0;
bool finished = false;
while not finished do
    finished = true;
    numPaired = 0;
    foreach pair of edges e1 and e2 in edges do
        if pairs contains e1.cluster or e2.cluster then
            continue;
        end
        if $e1.indices \neq e2.indices$ or
          $e1.cluster == e2.cluster$ then
            continue;
        end
        pairs[e1.cluster] = i;
        pairs[e2.cluster] = i;
        roots.insert(i);
        i++;
        done_clusters.insert(clusters[e1.cluster]);
        done_clusters.insert(clusters[e2.cluster]);
        roots.erase(e1.cluster);
        roots.erase(e2.cluster);
        clusters.insert(new Cluster(e1.cluster, e2.cluster,
          done_clusters.size() - 2));
        finished = false;
        numPaired++;
    end
    foreach edge in edges do
        if pairs[edge.cluster] exists then
            edge.cluster = pairs[edge.cluster];
        end
    end
    pairs.clear();
    if numPaired < PAIR_CUTOFF then
        break;
    end
end
FinishConstruction(roots);
```
Algorithm 1: ConstructEBBVH

3.1. Small Example

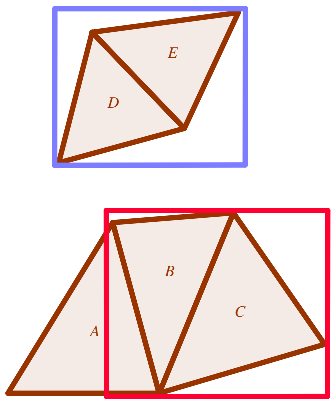
This is an illustrative example of the algorithm running on a small section of geometry. Bounding boxes have been expanded for the sake of visualization.

Figure 1: *Initial Geometry*



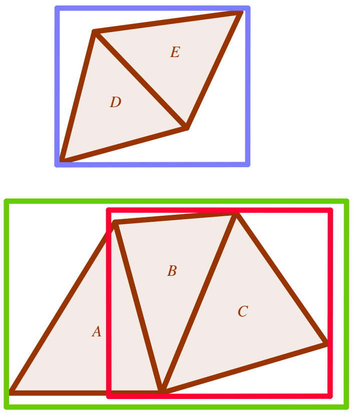
The first iteration of the bottom-up phase clusters triangles that share an edge.

Figure 2: *Iteration 1 of bottom-up phase*



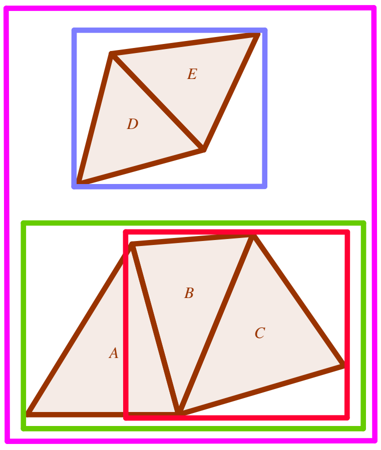
The next bottom-up iteration then combines clusters that share an edge.

Figure 3: *Iteration 2 of bottom-up phase*



The bottom-up phase is now completed leaving 2 root clusters. The top down phase then combines these. This leaves the final BVH.

Figure 4: *Result after top-down phase*



4. Methodology

We implemented the methods using C++. The same BVH traversal algorithm was utilised for each method in the comparison. The ray-tracing results where all obtained on a single CPU (Intel Core-i5 11300H @ 3.10GHz) core.

The 'Approximate Agglomerative Clustering' (AAC) algorithm was implemented from the pseudocode in [GHFB13]. The AAC BVH was constructed using a single thread on the CPU so construction time isn't representative of a multithreaded implementation. However the ray-tracing performance should still be representative.

The binned 'Surface Area Heuristic' (SAH) and recursive split used reference implementations by Jacco Bikker [Bikb] & [Bika], the binned SAH approach uses 8 bins for construction.

5. Results

Table 1: *Construction time for each algorithm, measured in milliseconds.*

	Recursive Split [Bikb]	AAC [GHFB13]	Binned SAH [Wal07]	EBBVH (ours)
Sponza	**48**	390	514	497
Bunny	**159**	191	251	312
Dragon	398	**351**	542	566
San-Miguel	**2496**	12161	24958	38127

Table 2: *Performance from camera, values in MRays/s.*

	Recursive Split [Bikb]	AAC [GHFB13]	Binned SAH [Wal07]	EBBVH (ours)
Sponza	0.00028	0.06	0.004	**0.5**
Bunny	**3.07**	0.26	2.88	0.89
Dragon	**3.47**	0.157	2.95	1.44
San-Miguel	0.002	0.002	0.013	**0.22**

Table 3: *Performance from random rays. Values in MRays/s.*

	Recursive Split [Bikb]	AAC [GHFB13]	Binned SAH [Wal07]	EBBVH (ours)
Sponza	0.0013	1.08	0.050	**3.26**
Bunny	5.02	0.98	**3.2**	2.05
Dragon	**5.33**	2.42	3.5	2.69
San-Miguel	7.06E-5	0.014	0.7	**0.95**

EBBVH currently has a construction time that is similar to a Binned SAH approach. It also has the worst average construction time out of all the algorithms that we have benchmarked. However, it has substantially better performance in large scenes such as Sponza and San-Miguel where it massively outperforms the other algorithms, by a factor of 2000 in the case of the recursive split algorithm. In such large scenes it also outperforms the binned-SAH approach by a factor of 20 in the San-Miguel test scene. EBBVH isn't well suited for construction of single objects such as Bunny and Dragon with it not performing at all as well as either binned SAH or even a simple recursive split approach.

More generally the type of scenes where EBBVH will perform the best are those with several disconnected parts of geometry. This is due to the bottom-up phase of the algorithm producing good quality leaf nodes. However if all of the geometry is connected for example in Bunny then the bottom-up phase will produce nodes closer to the root in the BVH which are less optimal.

The performance of EBBVH is also reasonably high in the random rays situation with it outperforming the other algorithms in Sponza and San-Miguel again. This means that it's performance advantage would not be lost with a more complex path-tracing algorithm that uses more discontinuous rays in its sampling.

The way that we have implemented EBBVH is somewhat lacking in terms of construction time optimizations with it using several hash maps which are slow in comparison to the other algorithms such as recursive split which only require array lookup operations. Along with this EBBVH also requires the construction of the edge-table which is somewhat costly. This cost might be better justified if the GPUs can gain a further performance benefit by pairing triangles into quads as seen in [AF23].

Figure 5: *Render of Sponza and San-Miguel using EBBVH.*

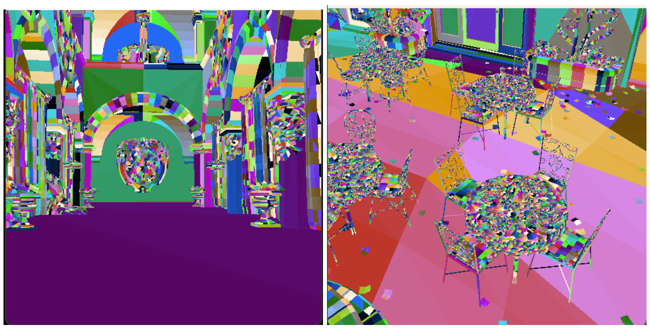
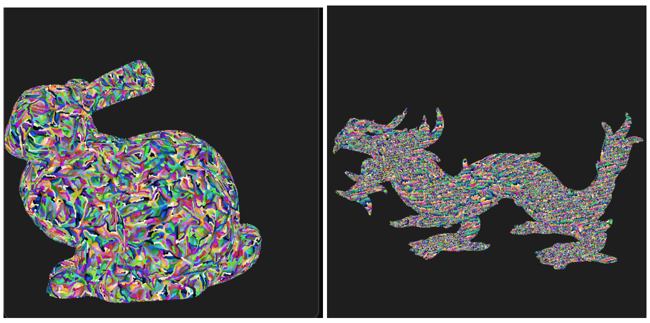


Figure 6: *Render of Bunny and Dragon using EBBVH.*



6. Conclusion & Further Work

An avenue for further optimization would be to create a GPU implementation of EBBVH, this could substantially reduce the construction time of both the edge table, associated sorting and the pairing phases. However, we're unsure if this is possible due to the nature of the algorithm. If a GPU implementation could be created this would almost certainly dramatically reduce construction times as seen with LBVH and AAC methods where GPU implementations require only milliseconds to run as seen in [LGS*09].

Further work could also entail investigating further exploiting the connected nature of the leaf nodes by rendering triangle strips, made of several connected quads, instead of triangles or quads. This could reduce memory usage further as each additional triangle in a strip only requires one additional vertex. This could even integrate into more complex traversal techniques such as those described in [WBB08].

Due to the large construction time and poor performance, we would not use EBBVH for situations where the BVH of single objects needs to be frequently reconstructed. Instead, it would be more suitable for a 3D rendering type workload where scenes are often disconnected and where the users are less sensitive to high construction times as the render times can be substantial. It may also be suited to a prototyping use case for games, again due to the user being less sensitive to construction time along with faster to construct methods not providing as good performance in disconnected scenes.

References

[AF23] AMAN A., FENNY S.: Fast triangle pairing for ray tracing. High Performance Graphics, 2023. URL: https://www.highperformancegraphics.org/posters23/Fast_Triangle_Pairing_for_Ray_Tracing.pdf. 1, 2, 4

[Bika] BIKKER J.: How to build a bvh part 3: quick builds. URL: https://jacco.ompf2.com/2022/04/21/how-to-build-a-bvh-part-3-quick-builds/. 4

[Bikb] BIKKER J.: How to build a bvh – part 1: Basics. URL: https://jacco.ompf2.com/2022/04/13/how-to-build-a-bvh-part-1-basics/. 4

[GHFB13] GU Y., HE Y., FATAHALIAN K., BLELLOCH G.: Efficient bvh construction via approximate agglomerative clustering. In *Proceedings of the 5th High-Performance Graphics Conference* (New York, NY, USA, 2013), HPG '13, Association for Computing Machinery, p. 81–88. URL: https://doi.org/10.1145/2492045.2492054, doi:10.1145/2492045.2492054. 1, 3, 4

[LGS*09] LAUTERBACH C., GARLAND M., SENGUPTA S., LUEBKE D., MANOCHA D.: Fast bvh construction on gpus. *Computer Graphics Forum 28*, 2 (2009), 375–384. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-8659.2009.01377.x, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1467-8659.2009.01377.x, doi:https://doi.org/10.1111/j.1467-8659.2009.01377.x. 2, 4

[MB18] MEISTER D., BITTNER J.: Parallel locally-ordered clustering for bounding volume hierarchy construction. *IEEE Transactions on Visualization and Computer Graphics 24*, 3 (2018), 1345–1353. doi:10.1109/TVCG.2017.2669983. 1

[MOB*21] MEISTER D., OGAKI S., BENTHIN C., DOYLE M. J., GUTHE M., BITTNER J.: A survey on bounding volume hierarchies for ray tracing. *Computer Graphics Forum 40*, 2 (2021), 683–712. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.142662, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.142662, doi:https://doi.org/10.1111/cgf.142662. 1

[Wal07] WALD I.: On fast construction of sah-based bounding volume hierarchies. In *2007 IEEE Symposium on Interactive Ray Tracing* (2007), pp. 33–40. doi:10.1109/RT.2007.4342588. 1, 4

[WBB08] WALD I., BENTHIN C., BOULOS S.: Getting rid of packets - efficient simd single-ray traversal using multi-branching bvhs -. In *2008 IEEE Symposium on Interactive Ray Tracing* (2008), pp. 49–57. doi:10.1109/RT.2008.4634620. 4

[WBKP08] WALTER B., BALA K., KULKARNI M., PINGALI K.: Fast agglomerative clustering for rendering. In *2008 IEEE Symposium on Interactive Ray Tracing* (2008), pp. 81–86. doi:10.1109/RT.2008.4634626. 1

[WK06] WÄCHTER C., KELLER A.: Instant Ray Tracing: The Bounding Interval Hierarchy. In *Symposium on Rendering* (2006), Akenine-Moeller T., Heidrich W., (Eds.), The Eurographics Association. doi:/10.2312/EGWR/EGSR06/139-149. 2