Comparison of BVH construction methods

KEDADRY Yannis

***Abstract*—With advancements in GPU hardware, Ray Tracing (RT) has become viable for real-time applications like video games. RT simulates light behavior by casting millions of rays in a 3D scene, but ray-object intersection tests remain a major bottleneck. To address this, acceleration structures like Bounding Volume Hierarchies (BVH) are essential. BVH construction methods generally fall into two categories: top-down and bottom-up. In this report, we compare a top-down approach based on Surface Area Heuristics (SAH) from Ganestam and Doggett (2016) [GD16] with a bottom-up method using locally-ordered agglomerative clustering from Meister and Bittner (2017) [MB18]. We analyze their trade-offs in terms of efficiency, parallelizability, and implementation complexity.**

I. Introduction

Ray Tracing is a rendering technique that simulates the physical behavior of light by tracing the path of light rays as they interact with objects in a virtual 3D scene. It is considered the most accurate and photorealistic rendering method available, producing high-quality images with realistic reflections, refractions, shadows, and global illumination effects. However, the computational complexity of ray tracing has historically made it impractical for real-time rendering applications.

A. Related Work

Despite its accuracy, RT has historically been too computationally expensive for real-time applications. However, recent advancements in GPU hardware and optimized acceleration structures have significantly improved RT performance. One of the key data structures that has enabled these performance gains is the bounding volume hierarchy (BVH).

BVH are hierarchical spatial data structures that enable efficient ray-scene intersection tests by skipping large portions of geometry. Various BVH construction methods exist, broadly categorized into top-down and bottom-up approaches. Top-down methods recursively split geometry, while bottom-up techniques iteratively merge primitives. Hybrid approaches also exist, balancing efficiency and quality. The choice of BVH construction algorithm can have a significant impact on the quality and performance of the resulting acceleration structure.

Other acceleration structures, such as kd-trees, octrees, and binary space partitioning (BSP) trees, have also been used in ray tracing. BVH structures are generally considered more efficient for ray tracing, as they do not attempt to map all points in the 3D space, but instead only consider the specific geometry that needs to be ray traced.

B. Contributions

This report presents a comprehensive analysis of 2017's the state-of-the-art in bounding volume hierarchy (BVH) construction algorithms for real-time ray tracing. Specifically, we focus on comparing top-down and bottom-up approaches to BVH construction.

For the top-down approach, we implemented the algorithm proposed by Ganestam and Doggett in 2016 [GD16], which utilizes surface area heuristics to recursively split the scene geometry into smaller bounding volumes. In contrast, for the bottom-up approach, we implemented the algorithm introduced by Meister and Bittner in 2017 [MB18], which builds the BVH by merging individual primitive bounding volumes using locally-ordered agglomerative clustering.

Through our comparative analysis, we aim to provide insights into the respective strengths and trade-offs of these two BVH construction paradigms. Specifically, our contributions are as follows:

- We present implementations of baseline top-down and bottom-up BVH construction algorithms, providing a solid foundation for the comparative analysis.
- We implement the state-of-the-art algorithmic techniques proposed in the 2016 and 2017 papers [GD16; MB18], allowing us to evaluate the latest advancements in both top-down and bottom-up BVH construction.
- We conduct a comprehensive comparison of the top-down and bottom-up approaches in terms of construction time, BVH quality, parallelizability, and ease of implementation. This analysis sheds light on the practical advantages and disadvantages of each methodology, helping to explain the current trends in the field.

II. Method

In a ray tracing renderer, each ray determines the color of a pixel (if we omit subpixel anti-aliasing). The ray tracing pipeline can then be entirely handled in parallel. However, the computation of intersections between a ray and the objects in the scene remains a bottleneck. A popular acceleration technique consists of using tree-based acceleration structures such as Bounding Volume Hierarchy (BVH). In the following, we present the different construction methods we used for our comparisons.

A. Top-Down

There exists multiple way of building a BVH. In a Top-Down approach the root of the BVH is built first by encapsulating the entire scene. Each node is then recursively split in its two children. In our default implementation, a cut is performed along the half of the node's longest axis and the two children bounding volumes are built using the triangles in one or the other side of the cut.

The idea behind **[GD16]** is to used the Surface Area Heuristic (SAH) to guide the BVH construction. Usually, using

SAH to optimize a top-down BVH builder is done by at each level of recursion evaluating and minimizing the equation 1:

$$E_r = C_i A(n) + C_l \left(A(n_l)N(n_l) + A(n_r)N(n_r) \right) \quad (1)$$

The operator A represents the surface area of a node's bounding box and N represents the number of triangle in a node. The constants C_i and C_l represent the costs of traversing an internal node and a leaf node. n_l and n_r are the potential left and right child nodes. E_r is then compared to $E_t = C_t A(n)N(n)$ which represents the cost of terminating the recursion and using the current node n as a leaf node.

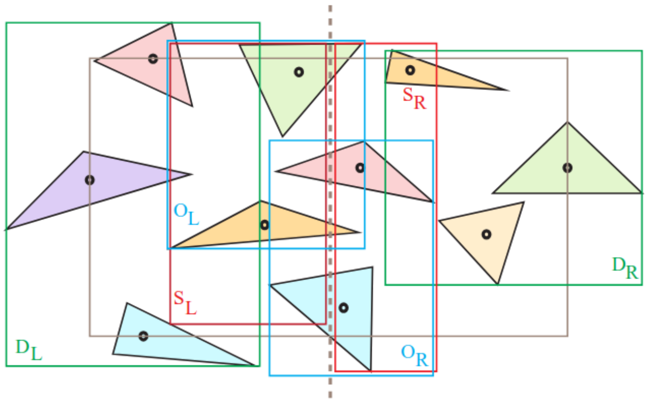


Fig. 1: The bounding boxes of the different sets of triangles used by our triangle split method. The brown box represents the mid-point bounds of all triangles and its spatial median is used to create the split plane. The blue bounds represent the overlap sets, the red bounds represent the split sets, and the green bounds represent the left and right disjoint sets

In our implementation, we construct a "complete" BVH where each leaf node contains exactly one triangle. Instead of the conventional top-down approach guided by the Surface Area Heuristic (SAH), we adopt the method proposed in **[GD16]**. At each recursive step, we determine the split plane using mid-point bounds. During partitioning, we classify triangles into six distinct sets, as illustrated in Figure 1. These sets include:

- **Disjoint sets**: D_L and D_R, which contain triangles that lie entirely on the left or right side of the split plane.
- **Overlap sets**: O_L and O_R, where triangles that intersect the split plane are placed based on their centroid position.
- **Split sets**: S_L and S_R, which store the left and right halves of subdivided triangles.

The triangles in each of the split sets are exactly the same as the triangles in the overlap sets: $O_L \cup O_R = S_L = S_R$. This constraints allows the same triangle in the scene to be in multiple leaves of the BVH.

To determine the best partitioning strategy, we compute the SAH cost using the overlap sets:

$$C_O = A(D_L \cup O_L)|D_L \cup O_L| + A(D_R \cup O_R)|D_R \cup O_R|$$

and compare it to the SAH cost using the split sets:

$$C_S = A(D_L \cup S_L)|D_L \cup S_L| + A(D_R \cup S_R)|D_R \cup S_R|$$

Partitioning continues with the set that yields the lower SAH cost.

B. Bottom-Up

Bottom-up BVH construction starts by assigning a bounding volume to each triangle in the scene. These bounding volumes are then iteratively merged into parent nodes based on specific criteria until a single root node remains. In our default implementation, merging is performed by combining neighboring nodes in the list. This approach provides reasonable results since triangles that are spatially close in the scene tend to be adjacent in the data representation.

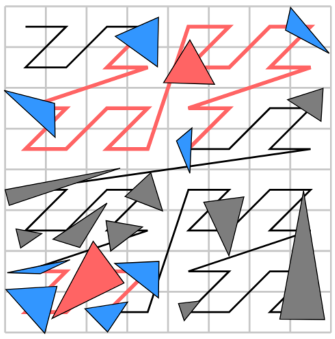


Fig. 2: Illustration of the nearest neighbor search $r = 2$. Two clusters (red triangles) search for their nearest neighbors (blue triangles) in the neighborhood (red curve). Notice how the algorithm adapts to the density of clusters in the neighborhood thanks to the Morton code sorting.

The PLOC algorithm parallelizes this bottom-up BVH construction. It begins by sorting all triangles using Morton codes, a space-filling curve that maps multidimensional data to a single dimension while preserving spatial locality. This step ensures that triangles that are close in the scene remain near each other in the sorted list, further improving the efficiency of the merging process.

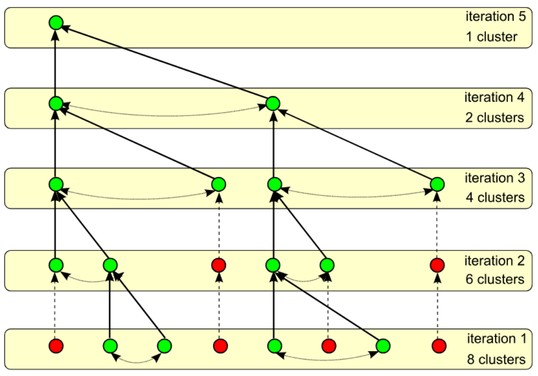


Fig. 3: Illustration of the proposed algorithm. In each iteration, we merge mutually corresponding nearest neighbors (green nodes connected by a dotted line). New clusters and not merged clusters (red nodes) enter the next iteration. The process is repeated until only one cluster remains.

```
1: $\mathcal{C}_{in} \leftarrow \mathcal{C}_{out} \leftarrow [C_0, C_1 \ldots, C_{n-1}]$
2: $\mathcal{N} \leftarrow \mathcal{P} \leftarrow [0, 1 \ldots, n-1]$
3: $c \leftarrow n$
4: while $c > 1$ do
5:     for $i \leftarrow 0$ to $c-1$ in parallel do
6:         /* NEAREST NEIGHBOR SEARCH */
7:         $d_{min} \leftarrow \infty$
8:         for $j \leftarrow \max(i-r, 0)$ to $\min(i+r, c-1)$ do
9:             if $i \neq j \wedge d_{min} > d(\mathcal{C}_{in}[i], \mathcal{C}_{in}[j])$ then
10:                $d_{min} \leftarrow d(\mathcal{C}_{in}[i], \mathcal{C}_{in}[j])$
11:                $\mathcal{N}[i] \leftarrow j$
12:            end if
13:        end for
14:        BARRIER()
15:        /* MERGING */
16:        if $\mathcal{N}[\mathcal{N}[i]] = i \wedge i < \mathcal{N}[i]$ then
17:            $\mathcal{C}_{in}[i] \leftarrow$CLUSTER($\mathcal{C}_{in}[i], \mathcal{C}_{in}[\mathcal{N}[i]]$)
18:            $\mathcal{C}_{in}[\mathcal{N}[i]] \leftarrow \clubsuit$
19:        end if
20:        BARRIER()
21:        /* COMPACTION */
22:        $\mathcal{P}[i] \leftarrow$ PREFIXSCAN($\mathcal{C}_{in}[i] \neq \clubsuit$)
23:        if $\mathcal{C}_{in}[i] \neq \clubsuit$ then
24:            $\mathcal{C}_{out}[\mathcal{P}[i]] \leftarrow \mathcal{C}_{in}[i]$
25:        end if
26:        BARRIER()
27:    end for
28:    $c \leftarrow \mathcal{P}[c-1]$
29:    if $\mathcal{C}_{in}[c-1] \neq \clubsuit$ then
30:        $c \leftarrow c+1$
31:    end if
32:    SWAP($\mathcal{C}_{in}, \mathcal{C}_{out}$)
33: end while
```

Fig. 4: Pseudocode of the main loop of PLOC. The input of the algorithm is a sequence of n clusters $C_0; C_1; ...; C_{n-1}$ sorted along the morton curve. Symbol $\clubsuit$ denotes an invalid cluster.

Once sorted, the algorithm enters an iterative loop that continues until the root node is constructed. The main loop consists of three key phases, as illustrated in Figure 4:

- **Nearest Neighbor Search (red phase)**: Each cluster searches for its nearest neighbor in parallel within a 1D interval of the sorted clusters. The search range is controlled by a parameter r and is clipped to prevent out-of-bounds memory access. Each cluster retains the closest neighbor found using a predefined distance function d. An example with $r = 2$ is depicted in Figure 2.
- **Merging Phase (green phase)**: In parallel, each cluster verifies if it is the nearest neighbor of its nearest neighbor. If this mutual condition is met, the two clusters are merged. To prevent conflicts, merging is handled by the thread assigned to the lower-indexed cluster. The first cluster is updated to represent the new merged cluster, while the second cluster is marked as invalid. Simultaneously, an interior node is created. To determine its index in the node buffer, a parallel prefix scan is performed following the approach from **[AM22]**. The final node index is obtained by adding the prefix scan result to the node counter.
- **Compaction Phase (blue phase)**: A global exclusive parallel prefix scan is executed to eliminate invalid clusters, ensuring continuity along the Morton curve. The prefix scan values determine the new positions of node indices, which are then written to the output buffer. It's important to notice that PLOC does not resort the clusters after merging, as the approximate ordering from the Morton curve is sufficient for nearest neighbor searches.

An illustration of several iterations of the algorithm is depicted in Figure 3

III. Results

In the following section we present our results and implementation details.

A. Implementation

We developed a GPU-based ray tracer using the Rust programming language **[HD10]** and the Vulkan API **[Gro16]**, interfaced through the Ash crate **[Dev17]**. The ray tracing logic is implemented in a compute shader using the Slang shading language **[Dev18]**. For this report, our material model consists of a simple ambient color field, and a single directional light is procedurally generated within the compute shader. However, extending the system to support additional material properties, more advanced BRDF models, and various light types would require minimal effort. The performance results presented in Table I were obtained on a system equipped with an Intel 11th Gen i5-1135G CPU and an Intel TigerLake-LP GT2 integrated GPU. Due to hardware limitations, we were unable to leverage Vulkan's dedicated ray tracing pipeline and instead implemented a custom solution, which still has room for optimization. Additionally, while our implementation utilizes the Slang shading language, we primarily focused on its basic features as a learning tool, leaving more advanced capabilities for future exploration.

B. Comparisons

Table I compares BVH construction methods across different scenes. As expected, rendering without acceleration structures becomes infeasible for complex scenes. For simple scenes, advanced BVH methods offer little advantage over basic approaches. However, as complexity increases, structured BVH construction becomes crucial. The CPU-parallelized PLOC algorithm, executed on four cores, achieves an approximate 3 times speedup over its sequential version. A GPU-based implementation could further improve performance, though parallel sorting remains a challenge.

Interestingly, the SAH-guided approach performs worse than expected. This can be attributed to the requirements of our custom ray tracing implementation. Our pipeline mandates that each BVH leaf contains exactly one triangle. However, SAH-guided construction is designed to terminate recursion early, balancing efficiency and partitioning quality. In the original paper, SAH-based methods typically stop at 512 or even

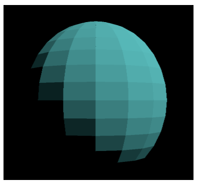
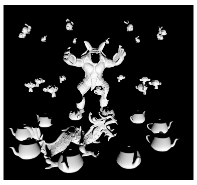
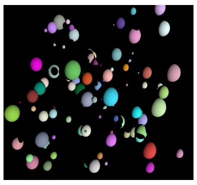
Scene	Triangles	BVH Construction Method	Time to Build (s)	Average FPS	Average Time Per Frame (ms)
	≈ 500	None	0	10	99.81
		Baseline Top-Down	**0.00285**	**60**	16.67
		Baseline Bottom-Up	0.00300	**60**	**16.65**
		SAH Guided Top-Down	0.00888	**60**	16.66
		PLOC	0.00486	**60**	16.66
		CPU Parallelized PLOC	0.02336	**60**	16.66
	$\approx 100,000$	None	0	X	X
		Baseline Top-Down	0.82288	54	18.41
		Baseline Bottom-Up	0.78625	58	17.25
		SAH Guided Top-Down	2.00633	56	17.70
		PLOC	1.27355	**60**	**16.67**
		CPU Parallelized PLOC	**0.61331**	**60**	**16.67**
	$\approx 1,000,000$	None	0	X	X
		Baseline Top-Down	10.74779	14	68.43
		Baseline Bottom-Up	10.70392	14	67.97
		SAH Guided Top-Down	26.89474	14	66.25
		PLOC	14.05147	**18**	**54.36**
		CPU Parallelized PLOC	**5.999515**	**18**	**54.36**
	$\approx 3,000,000$	None	0	X	X
		Baseline Top-Down	40.03281	11	85.4
		Baseline Bottom-Up	39.61821	20	51.0
		SAH Guided Top-Down	170.55518	20	50.68
		PLOC	45.59624	**24**	**40.39**
		CPU Parallelized PLOC	**19.17041**	**24**	**40.39**

TABLE I: Comparison of BVH Construction Methods (An X means *too slow to be measured*)

8192 triangles per leaf, whereas we must continue down to a single triangle per leaf. This constraint significantly increases construction time and can degrade BVH quality by introducing unnecessary complexity. Conversely, PLOC and other bottom-up approaches naturally construct BVH trees with single-triangle leaves, making them more suitable for our pipeline. Nevertheless, SAH-guided construction was significantly easier to implement than the PLOC algorithm.

From an implementation perspective, baseline and SAH-guided methods were simplest but less effective. PLOC outperformed them, and a GPU implementation could enable real-time BVH updates. Bottom-up methods remain state-of-the-art due to their parallel efficiency, as seen in PLOC++ **[Ben+22]** and HPLOC **[Ben+24]**. This is because they are inherently more parallelizable than top-down approaches. In a parallel execution model, it is more efficient to stop half of the active threads than to launch twice as many new threads at each recursion step.

IV. Conclusion

In this report, we explored different BVH construction techniques and evaluated their impact on rendering performance in a GPU-based ray tracer. Our findings highlight that while simpler BVH construction methods work well for small scenes, complex scenes benefit significantly from structured acceleration structures.

Among the methods tested, PLOC proved to be the most efficient, particularly when parallelized on the CPU. This suggests that a fully GPU-based implementation of PLOC could enable real-time BVH reconstruction. However, implementing efficient parallel sorting and reducing kernel launch overhead remains a challenge.

Additionally, our analysis showed that SAH-guided construction is less suited for our specific pipeline due to its reliance on early termination criteria, which conflicts with our requirement of single-triangle BVH leaves. Nonetheless, it was easier to implement compared to PLOC.

Future work includes optimizing our BVH construction pipeline further, particularly by fully utilizing GPU compute shaders for bottom-up methods. We began developing this GPU-based approach but were unable to complete it before submitting this report due to the complexity of parallel sorting. With a more efficient implementation, real-time BVH updates could become feasible, improving the flexibility and performance of our ray tracer.

References

[HD10] Graydon Hoare and The Rust Project Developers. *The Rust Programming Language*. Accessed: 2025-02-05. 2010. URL: https://www.rust-lang.org/.

[GD16] Per Ganestam and Michael Doggett. "SAH guided spatial split partitioning for fast BVH construction". In: *Computer Graphics Forum* 35 (May 2016), pp. 285–293. DOI: 10.1111/cgf.12831.

[Gro16] Khronos Group. *Vulkan API*. Accessed: 2025-02-05. 2016. URL: https://www.vulkan.org/.

[Dev17] The Ash Developers. *Ash: Vulkan bindings for Rust*. Accessed: 2025-02-05. 2017. URL: https://github.com/ash-rs/ash.

[Dev18] Slang Developers. *Slang Shading Language*. Accessed: 2025-02-05. 2018. URL: https://shader-slang.com/.

[MB18] Daniel Meister and Jiří Bittner. "Parallel Locally-Ordered Clustering for Bounding Volume Hierarchy Construction". In: *IEEE Transactions on Visualization and Computer Graphics* 24.3 (2018), pp. 1345–1353. DOI: 10.1109/TVCG.2017.2669983.

[Mei+21] Daniel Meister et al. "A Survey on Bounding Volume Hierarchies for Ray Tracing". In: *Computer Graphics Forum* 40 (2021). URL: https://api.semanticscholar.org/CorpusID:235337851.

[AM22] Andy Adinets and Duane Merrill. *Onesweep: A Faster Least Significant Digit Radix Sort for GPUs*. 2022. arXiv: 2206.01784 [cs.DC]. URL: https://arxiv.org/abs/2206.01784.

[Ben+22] Carsten Benthin et al. "PLOC++: Parallel Locally-Ordered Clustering for Bounding Volume Hierarchy Construction Revisited". In: 5.3 (July 2022). DOI: 10.1145/3543867. URL: https://doi.org/10.1145/3543867.

[BP23] Carsten Benthin and Christoph Peters. "Real-Time Ray Tracing of Micro-Poly Geometry with Hierarchical Level of Detail". In: *Computer Graphics Forum* 42 (2023). URL: https://api.semanticscholar.org/CorpusID:259290458.

[Ben+24] Carsten Benthin et al. "H-PLOC: Hierarchical Parallel Locally-Ordered Clustering for Bounding Volume Hierarchy Construction". In: *Proc. ACM Comput. Graph. Interact. Tech.* 7.3 (Aug. 2024). DOI: 10.1145/3675377. URL: https://doi.org/10.1145/3675377.