Envoy:
An High-Performance Bounding Volume Hierarchies System on Building and Intersecting

Krr

Contents

1 Introduction

Envoy is an work-in-progress and experimental Bounding Volume Hierarchies (BVH) backend for renderers, with relatively heavy engineering effort.

As an introduction, BVH is motivated by the property that, if a ray do not intersects with some triangles' bounding volume, exact ray-triangle intersection can be mitigated. By dichotomy on space, we can construct a hierarchy of bounding volumes, where bounding volumes contain other sub-volume recursively. This hierarchy serves as a tree like *accelerating data-structure*, which is quite common in Computer Graphics. For data-structure like this, the whole algorithm is separated into two phases, *construction* phase and *intersecting* phase. Specific information of BVH will be introduced later.

In real scenarios, to build an offline renderer, the *construction* phase can be slow. We can even endure an relatively long *tree optimization* process, e.g., tree balancing, triangle split or rearrange. But the *intersecting* phase must be fast. Or to say, the **tree traversal** and **ray-triangle intersection** must be efficient. However, in real-time renderer, between frames, we reconstruct our trees because objects can be moving. In modern GPU-pipeline, this operation should be executed purely on GPU for performance and coherency. Ray-triangle intersection can be trivially ported to GPU, but *tree construction* does not. See table 1 for more information.

	Construction	**Intersection**
Offline Renderer	Tree quality, Reasonable performance, On CPU	High-performance, Inherently parallel
Real-time Renderer	Can sacrifice tree quality, but parallel on GPU	Reasonable performance on CPU and GPU, Inherently parallel

Table 1: Requirements for BVH on different scenarios.

Our target (currently in this Parallel Computing Final Project) is to implement Karras's paper [5] with SIMD ray-triangle intersection in *Plucker Coordinates* on CPU, which trained me to build parallel math libraries in the renderer, I'll discuss about this later.

> **Temporary Note**: The project is already WIP in its GitHub repository. Because I had COVID-19 just in final-exam week, time constraints prevent me from fulfilling all my initial commitments. This project will be continued.

2 Background

In this section, I'll briefly introduce BVH's implementation detail, existing works and further improvement in this project.

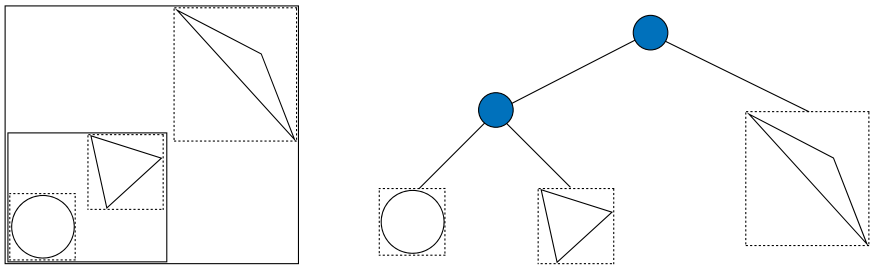
Let's demonstrate this process with fig. 1. This outdoor scene contains approximately 3.1 billion triangles. We hope to proceed as little triangles as possible. In traditional rasterization-based renderer, only the triangles within the *view frustum* are needed to render a complete scene. So there is a *culling* process, on both CPU and GPU in modern renderer pipeline. However, in our case, when the interface is

$$\textsc{Intersect} : \underbrace{\mathbb{R}^3 \to \mathbb{R}^3}_{\text{ray}} \to \underbrace{\textit{intersectable}}_{\triangle/\bigcirc/\square/\cdots} \to \textit{intersection}$$

Where *intersection* is a tuple that might contain information like normal, uv in $\mathbb{R}^2$ and position, etc. The easiest way is to traverse all triangles, since *Triangle* also provides this interface.

However, a desired way is to traverse on a structure like fig. 2, where objects(primitives) are arranged in a tree-like way. The only requirement of

Figure 1: Complex realistic outdoor landscape scene from Physically Based Rendering book.



(a) Spatial split. (b) Tree structure.

Figure 2: A demonstration of BVH. Figure adopted from PBRT.

these objects is that they implements the Intersect function, i.e., they are intersectable. So a BVH itself can also be embed into another BVHs.

The BVH traversal process is demonstrated in algorithm 1, note that this process should be extensively optimized.

Some questions are left here. How is this hierarchy constructed, and how are the triangles intersected, i.e., what's the implementation of Intersect. These questions will be answered in the following two sections.

3 Parallel Details

3.1 SIMD Math Library

As I mentioned before, if we implement the ray-triangle intersection process in CUDA, with appropriate memory layout, i.e., SoA, SIMD is automatically enabled. This indicates that, we don't have to rewrite code for SIMD, **the original code is expressive enough**.

To accomplish this, we have multiple ways. One is `ispc` compiler [10], which

Algorithm 1 BVH Tree traversal.

```
function BvhTreeTraversal(root, ray)
    return empty if root is empty
    □ ← getBoundingBox(root)
    △ ← getTriangles(root)                    ▷ read properties from node
    if Intersect(ray, □) then
        if root is leaf then
            return Intersect(ray, △)
        else
            return BvhTreeTraversal(left(root), ray)
                 | BvhTreeTraversal(right(root), ray)
                                  ▷ merge the results from left right sub-tree
        end if
    end if
end function
```

provide a C-like language for SPMD programming and structure design. The reason that I did not use `ispc` is that its intrusive. A shader-like language will break the abstraction, and is more suitable for writing execution kernels, handling tasks in a pipeline. CUDA is great, but I'm writing CPU code.

Anyway, this is a Math Library problem, and can be solved using new compilers or powerful math library. Another way is to imitate `enoki`, which is the math backend of Mitsuba Renderer 2 [9]. We currently neglect its autodiff feature (which also relies on math library), focusing on its auto-vectorization design.

Mitsuba Renderer 2's implementation relies on C++ template, adding the primitive type, e.g., *Float* as a template parameter. In theory, method call to *Array* should perform the same as to its items. Suppose we define dot product function `dot` on two `vec3`, i.e., `dot(a: vec3, b: vec3)`. Then for dot product on two `Array<vec3, 8>`, the original dot product should be executed element-wise, which is a transformation from

$$\text{dot} : vec3\langle\mathbb{R}\rangle \to vec3\langle\mathbb{R}\rangle \to \mathbb{R}$$

to

$$\text{dot} : \forall n \in \mathbb{N}, array\langle vec3\langle\mathbb{R}\rangle, n\rangle \to array\langle vec3\langle\mathbb{R}\rangle, n\rangle \to array\langle\mathbb{R}, n\rangle$$

where $vec3\langle\mathbb{R}\rangle$ should be rearrange into SoA in $array$.

This imposes a harsh requirement, every component in function parameters should be able to convert into SoA form with a template parameter for efficient calculation, which prohibits us from using external library in core part of the project that doesn't support this operation. However in later parts, this abstraction can be preserved by not using SoA. In our case, we are not building a complete renderer from scratch, but a self-contained math library is needed. Embree [11] has an implementation of this type of math library.

In our case, we used `xtensor-stack/xsimd` as our SIMD backend (although I think Enoki is more well-designed). In renderer, all the calls should not

be in member-function-call form, which affects the compatibility. When we want to migrate from `glm` to `Eigen`, this is convenient (however, we have to implement all these math functions with some are already provided by `xsimd`). With all these complex preparations, I implement the ray-triangle intersection in `src/intersector.h`.

There is more to discuss about in SIMD. Previously we assume that only triangles can be packed in a SIMD fashion, but so do rays. We'll not discuss about this here.

3.2 Parallelism in the Construction of BVH

This part is based on Karras's paper [5]. Which core lies in converting a sorted array into a tree in $O(n \log n)$ and $O(\log n)$ time. Methods based on converting BVHs into sorted arrays and then build tree structures are called *linear BVHs*[7] (LBVH). But first, let me answer the question, how is the tree built with pseudo-code here algorithm 2.

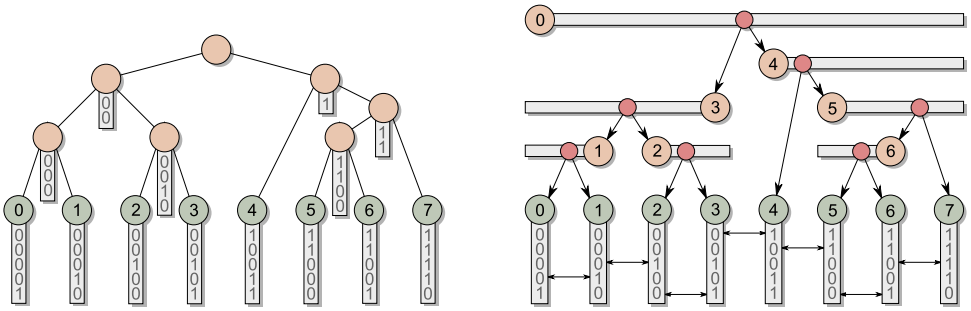
Algorithm 2 BVH Tree build.

function BvhTreeBuild(*primitives*, *depth*)
▷ *primitives* can be represented in `std::span`
return empty if *primitives* is empty
□ ← getBoundingBox(*primitives*)
node ← MakeNode(*null*, □)
if Size(*primitives*) ≤ 1 **then**
return MakeNode(*primitives*, □)
else
mid ← SelectMidHeuristic(*primitives*, *depth*)
▷ This function also partition the *primitives*
left(*node*)
← BvhTreeBuild(*primitives*[0 → *mid*], *depth* + 1)
right(*node*)
← BvhTreeBuild(*primitives*[*mid* + 1 → *n*], *depth* + 1)
end if
end function

We want to parallelize this process in PRAM model, with or without *linear BVHs*. Notice that this process is similar to QuickSort, where each recursion involves an expensive heuristic function selecting *pivot*.

The original LBVH paper achieved this by constructing two working queues on GPU, maintained by two separate kernel execution phases: *split kernel* and *compaction kernel*. This way enables the use complex heuristic, e.g., SAH heuristic, which balances the two sub-nodes by maximizing a parameter related to their surface areas. However in our fast-construction approach, this is not feasible since the tree structure is pre-determined.

This fast-construction approach exploits the property of *binary radix tree*, which accepts an ordered array and produce a balanced binary tree. A crucial



(a) Binary Radix Tree. (b) Tree structure with range.

Figure 3: A demonstration of binary radix tree. Figure adopted from the original paper[5].

property of which is its construction can be fully parallelized. We give a brief introduction to *binary radix tree* here.

The tree structure is demonstrated in fig. 3. By viewing all values on the leaves in their binary form, we define $\delta(i, j)$ on this array which is the **longest common prefix** between keys k_i and k_j. This implies that $\forall i', j' \in [i, j]$, $\delta(i', j') \geq \delta(i, j)$. Nodes are separated into $n-1$ *internal nodes* I and n *leaves* L, I, L are sets.

Let's consider a node I_j in I to have three properties, a range $[i, j]$, a split position γ and a value $\delta(i, j)$. The two children must have ranges that are $[i, \gamma]$, $[\gamma + 1, j]$. Then, how to choose the split position $\gamma \in [i, j-1]$? We choose the position by the first bit following $\delta(i, j)$. For $k_\gamma, k_{\gamma+1}$ this bit are $0, 1$ respectively. Some properties appear,

1. The index of internal node i is some other's split position γ or $\gamma + 1$.
2. Each internal node can independently decides their direction of coverage because of $\delta(\gamma, \gamma + 1)$ (see fig. 3).
3. Each internal node can independently decides their coverage because they cannot exceed their parents' $\delta(i, j)$.
4. Because $\delta(i, k)$ is monotonically decreasing on k, we can dichotomy k to get γ.
5. Till now, all properties of a node can be found in parallel with $O(\log n)$ time and work.

For PRAM analysis, its trivial that construction takes $O(n \log n)$ work and $O(\log n)$ time.

The only thing last is how to construct the sorted array. Using space-filling curve, as you can see from fig. 4, any partition on the continuous curve will result in two almost non-overlap spaces. Then we assign a morton-code to each triangles (primitives) and sort them using *radix sort*, producing the input that can be feed to *binary radix tree*.

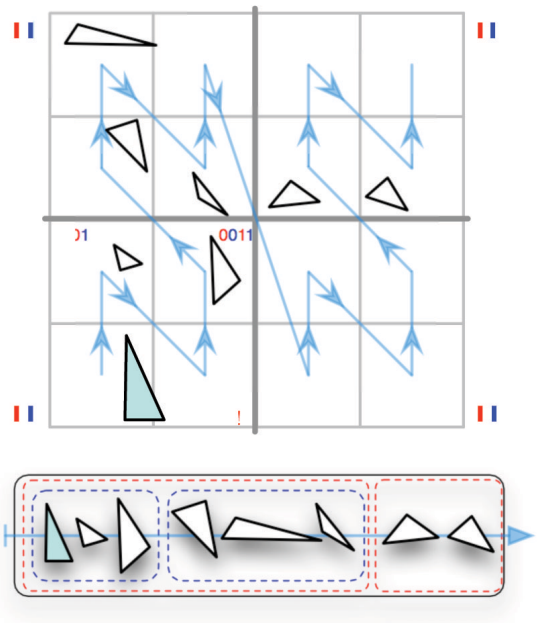


Figure 4: Morton code for LBVH.

3.3 Pre-partition

As we've mentioned before, the tree structure is pre-determined by keys k_j, leaving us no area to use heuristics. And unlike the traditional algorithm 2, its hard for us to perform early termination in tree construction, i.e., terminate when *depth* is too large or when Size(*primitives*) is small enough. This motivates us to pre-partition the triangles using numerous of techniques, grouping them into *primitives* and implement Intersect on them.

Actually, this idea comes from *streaming BVH* [2], which is to load triangles on-the-fly when intersecting. Triangles are better packed into packs which have a size of a physical page, so that we can maintain these pages with a LRU cache. Further ideas will be discussed in later section. However, leaving page-sized triangles on the BVH leaves will result in poor performance since the granularity is too large. BVHs work fine when the number of elements on their leaves is low. Results will yet be presented later.

There are mainly three ways to pack triangles,

1. Using yet another BVH to partition.
2. Using morton-code to partition.
3. Using *graph-partition* (METIS library) on the dual graph of triangle mesh.

I'll test them later.

4 Result

The code is published in `kriiae/envoy`. It is implemented in C++ and uses `xmake-io/xmake` as my build system to handle all the dependencies. Here is my testing platform,

Name	Value
Kernel	6.0.5-x64v2-xanmod1-1
CPU	AMD Ryzen 9 5900X (24) @ 3.700GHz
Core Frequency	3.700 GHz
Hyper-threading	ON
Memory	3200 MHz, 32G
Compiler	gcc (GCC) 12.2.0
Compile Options	`-std=c++20 -O3 -g`

As an introduction, we test the performance of the parallel building process with different number of cores, with different models. The test is done with `taskset` since TBB do not conveniently support specifying the number of threads. See the result at table table 2. Note that the performance index is MOPs, based on the number of primitives it is processing.

t	dragon	sphere	dambreak0	bun_zipper
0	2.73	3.70	3.26	4.21
1	3.81	4.94	4.51	5.78
2	4.21	4.85	5.15	6.60
3	4.35	4.96	5.70	7.07
4	4.52	5.53	5.82	7.50
5	4.52	6.10	5.97	7.53
6	4.49	4.91	5.92	6.40
7	4.87	5.38	5.84	6.13
8	4.86	5.37	5.68	5.91
9	4.83	5.52	5.96	7.16
10	4.85	5.53	4.93	6.99
11	4.92	5.18	5.77	6.80

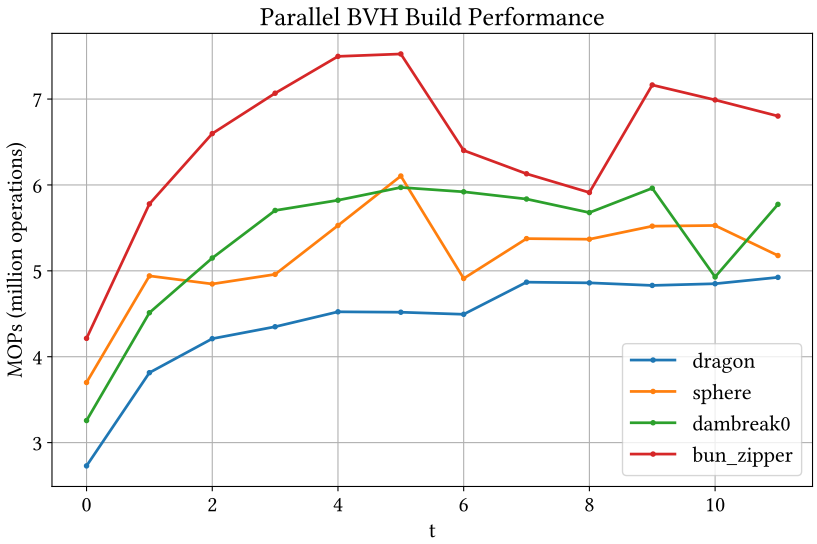
Table 2: The build performance on different meshes

Note that the algorithm is not that scalable, maybe its because of the *sort* algorithm I used and the number of threads of TBB cannot be readily controlled by `taskset`. Although I also implemented it with OpenMP backend, the *parallel_sort* that I used is based on TBB. I also give the plot here in section 4.

Here's the performance on four threads comparing to the original BvhBuildTree version. See table 3. The performance boost is considerable.

There is one more experiment that worth doing here, the *intersection performance*. I've mentioned that using this method, we lack the ability to manually control the tree structure.

First, it worth mentioning here that the tree traversal optimization is hard.



	dragon	sphere	dambreak0	bun_zipper
serial	0.90	1.24	1.10	1.13
radix	4.62	5.83	5.98	7.51
relative	**5.16x**	**4.70x**	**5.41x**	**6.64x**

Table 3: The performance of serial BVH build and our implementation.

Although there is some methods like [1] that enables fast tree-traversal, I do not have enough time to implement. So I choose the similar naive tree-traversal presented in algorithm 1 on a thread-local stack. One great metrics to measure tree quality is the number of triangles intersected. Implementation details are presented in `src/stats.cpp` for metrics, which utilized Boost.Accumulators.

Let's first test the intersection performance on pure triangle intersections. The test result is simple, but due to the limit of `xtensor-stack/xsimd`, which do not even provide a scalar backend as a fallback for now (9.0.1), which, I previously thought, will be presented in every well-designed SIMD library (It do provide a *generic* backend, but it cannot compile, because it is not fully implemented). So I can only compare the performance on SSE and AVX, which have a width corresponding to 4 and 8. For AVX, the intersection performance is approximately 704.28×10^6 triangles per second. While SSE can achieve 431.70×10^6 triangles per second. (on a single core).

Then let's demonstrate the intersection time per traversal in these two tree building mechanisms. On a basic tree with SAH, 10^6 ray queries finished in 435ms, resulting in approximately 18 ray-triangle intersections per query. For radix BVH, queries finished in 516ms, the overhead is less than 20%, while it takes approximately 32 ray-triangle intersections per query. Currently, the two different methods uses BVH as their *pre-partition* method. We add the result of using Z-Order Curve to pre-partition triangles in table 4.

build method	**intersections per ray**	**time**
BVH+SAH BVH	18	435ms
BVH+RADIX BVH	32	516ms
Z-Curve+SAH BVH	113	664ms
Z-Curve+RADIX BVH	128	864ms

Table 4: Intersection performance with different building mechanisms (on `sphere.ply`, with 10^6 ray queries).

> Although I did not present the result comparing to Embree [11] here, Embree's performance is no feasible for solely a course project to achieve. I'll implement those efficient algorithms later.

5 Further work (on engineering)

Although this course project finishes here, there is a lot to progress. For example, following this work [5], there is [6, 12] that enables much more efficient BVH to build with triangle split, and [1] that boosts the performance of BVH traversal.

There are also some existing efficient BVH implementations on GitHub, for example, `brandonpelfrey/Fast-BVH`, `madmann91/bvh`. These are much more efficient than my implementation. Before I move on to more advanced architectures, these optimizations must be done in advance.

I mentioned a prospect in my previous proposal. Three parts are crucial and general in a complete renderer, whether online or offline. Let's consider that we are to render scenes that cannot be stored inside of a PC's RAM [2]. They can have so many lights that standard sampling techniques (UniformSampleOneLight or UniformSampleAllLights) cannot handle. They can have scene geometries of more than 100 GB. **We would want to render in real-time in a scene like this**. These scenes are actually common in games or movies, as mentioned in [2]. Three Treasures of Modern Renderers are *efficient multi-platform and heterogeneous math library (or compiler)*[9, 4], *general importance sampler*[8], *streaming BVHs with LoD*[2]. Using these techniques, the importance sampler can handle many lights, effectively selecting the lights that contribute to this scene. With an appropriate math library (or compiler) design, in extreme cases, a renderer written in Taichi [3] can achieve heterogeneous computing for free. Streaming BVHs also requires no modification to the original renderer. LoD is necessary for large scenes. Consider a scenario where we can see the whole city. We don't expect to load the whole scene into memory, taking up a lot of memory bandwidth. These are what I'm going to learn, research and implement in the future.

References

[1] Nikolaus Binder and Alexander Keller. "Efficient Stackless Hierarchy Traversal on GPUs with Backtracking in Constant Time". In: *Proceedings of High Performance Graphics*. HPG '16. Dublin, Ireland: Eurographics Association, 2016, pp. 41–50. ISBN: 9783038680086.

[2] Brent Burley et al. "The Design and Evolution of Disney's Hyperion Renderer". In: *ACM Transactions on Graphics* 37 (July 2018), pp. 1–22. DOI: 10.1145/3182159.

[3] Yuanming Hu et al. "Taichi: a language for high-performance computation on spatially sparse data structures". In: *ACM Transactions on Graphics (TOG)* 38.6 (2019), p. 201.

[4] Wenzel Jakob et al. "Dr.Jit: A Just-In-Time Compiler for Differentiable Rendering". In: *Transactions on Graphics (Proceedings of SIGGRAPH)* 41.4 (July 2022). DOI: 10.1145/3528223.3530099.

[5] Tero Karras. "Maximizing Parallelism in the Construction of BVHs, Octrees, and k-d Trees". In: June 2012, pp. 33–37. DOI: 10.2312/EGGH/HPG12/033-037.

[6] Tero Karras and Timo Aila. "Fast parallel construction of high-quality bounding volume hierarchies". In: *High Performance Graphics*. 2013.

[7] Christian Lauterbach et al. "Fast BVH construction on gpus". In: *Computer Graphics Forum* 28 (Apr. 2009), pp. 375–384. DOI: 10.1111/j.1467-8659.2009.01377.x.

[8] Daqi Lin et al. "Generalized Resampled Importance Sampling: Foundations of ReSTIR". In: *ACM Trans. Graph.* 41.4 (July 2022). ISSN: 0730-0301. DOI: 10.1145/3528223.3530158. URL: https://doi.org/10.1145/3528223.3530158.

[9] Merlin Nimier-David et al. "Mitsuba 2: A Retargetable Forward and Inverse Renderer". In: *Transactions on Graphics (Proceedings of SIGGRAPH Asia)* 38.6 (Dec. 2019). DOI: 10.1145/3355089.3356498.

[10] Matt Pharr and William R Mark. "ispc: A SPMD compiler for high-performance CPU programming". In: *2012 Innovative Parallel Computing (InPar)*. IEEE. 2012, pp. 1–13.

[11] Ingo Wald et al. "Embree: a kernel framework for efficient CPU ray tracing". In: *ACM Transactions on Graphics (TOG)* 33.4 (2014), pp. 1–8.

[12] Henri Ylitie, Tero Karras, and Samuli Laine. "Efficient Incoherent Ray Traversal on GPUs through Compressed Wide BVHs". In: *Proceedings of High Performance Graphics*. HPG '17. Los Angeles, California: Association for Computing Machinery, 2017. ISBN: 9781450351010. DOI: 10.1145/3105762.3105773. URL: https://doi.org/10.1145/3105762.3105773.