

Received 5 October 2025, accepted 25 October 2025, date of publication 3 November 2025, date of current version 7 November 2025.

Digital Object Identifier 10.1109/ACCESS.2025.3627966

RESEARCH ARTICLE

Considerations for the Acceleration Structure of Sound Propagation on Mobile Devices: Kd-Trees Versus Multi-Bounding Volume Hierarchies

HYEON-KI LEE^{ID 1}, HYEJU KIM^{ID 1}, DONG-YUN KIM^{ID 1}, WOO-CHAN PARK^{ID 2}, AND JAE-HO NAH^{ID 1}

¹Department of Computer Science, Sangmyung University, Jongno-gu, Seoul 03016, Republic of Korea

²Department of Computer Engineering, Sejong University, Gwangjin-gu, Seoul 05006, Republic of Korea

Corresponding author: Jae-Ho Nah (jaeho.nah@smu.ac.kr)

This work was supported by the National Research Foundation of Korea (NRF) grant funded by Korean Government [Ministry of Science and ICT (MSIT)] under Grant RS-2025-00521436.

ABSTRACT Sound propagation algorithms can provide immersive auditory experiences for users in various domains, such as games, virtual/augmented reality, and other applications. Recently, there has been a growing trend to apply spatial sound effects not only on desktops but also on mobile devices with limited computational resources. In geometric-acoustic (GA) sound propagation utilizing ray-tracing methods for spatial audio effects, the choice of acceleration structure is an important factor that can either degrade or enhance performance. From this perspective, we seek to provide insights into the essential considerations for selecting acceleration structures for sound rendering on mobile devices. In this paper, we propose guidelines for mobile devices that address both how to select acceleration structures for sound rendering depending on the scene characteristics and how to optimize the selected acceleration structures. We used kd-trees and multi-bounding volume hierarchies (MBVHs), both of which are widely adopted acceleration structures in ray tracing. In addition, we applied optimization techniques that accelerate the traversal and intersection tests of these structures using SIMD intrinsics, and further improved the performance of kd-trees through on-the-fly pruning. According to our experiments, our optimization approach, when compared against the baseline kd-trees, not only achieved performance improvements of up to $1.35 \times$ and $1.44 \times$ for optimized kd-trees and MBVHs, respectively, with minimal increases in power consumption on a Google Pixel 8a, but also enabled analysis of the advantages and disadvantages of each acceleration structure in various test scenes. We expect that our research will serve as a valuable reference for future studies on sound propagation and the broader multimedia community.

INDEX TERMS Sound rendering, ray tracing, sound propagation, kd-tree, bounding volume hierarchy.

I. INTRODUCTION

In virtual and augmented reality (VR/AR), not only visual effects but also realistic sound play a vital role in delivering a fully immersive experience to users. For example, in VR games, sounds such as footsteps, gunfire, or explosions must be accurately propagated according to the player's position and orientation. Thus, simulating real-world and environmental sounds in VR/AR apps can amplify a sense of

The associate editor coordinating the review of this manuscript and approving it for publication was Jolanta Mizera-Pietraszko^{ID}.

realism and immersion. To achieve this goal, sound rendering technology has been actively studied. The term *sound rendering* refers to the process of generating sounds from a behavioral or physically-based simulation of component sound objects [1]. Because quite a few VR/AR contents are now played on mobile or embedded devices (e.g., standalone head-mounted displays (HMDs)), the computational costs of sound rendering are also important.

In sound rendering, sound propagation is modeled in acoustic space, utilizing the relationship between sound sources, listener locations, 3D models, and materials [2].

To offer users high-quality auditory experiences, sound propagation is one of the key stages in sound rendering.

Among various technologies for sound propagation, geometric-acoustics (GA) methods have gained great attention due to their effectiveness. GA methods trace the path of sound propagation geometrically between sound sources and listeners in the scene, and ray tracing [3] is one of the prominent GA methods. In ray tracing, the phenomenon of wave refraction, diffraction, and reflection can be approximated by tracing rays along the path of sound propagation. Because ray tracing [4] is a famous technique in the computer graphics as well, it has wide applicability to sound propagation by leveraging existing software or hardware-accelerated ray-tracing techniques [5]. Recently, Apple announced the audio ray-tracing technology used in Vision Pro for creating spatial audio [6].

The ray-tracing algorithm essentially requires a lot of ray-primitive intersections to find the closest hit or any hit points, so there has been extensive research on acceleration structures for ray tracing in computer graphics community. The choice and implementation of the acceleration structure can greatly affect ray-tracing performance. Representative acceleration structures are kd-trees and bounding volume hierarchies (BVHs) [7], and these two data structures have their respective pros and cons. If the primitive type is constrained to triangles, a proper implementation of the ray-triangle intersection test algorithm used in ray tracing is also important for achieving high performance. In other words, ray traversal on the acceleration structure and ray-triangle intersection tests are critical performance factors in real-time ray tracing [8], [9], [10]. Thus, we need to consider suitable methods that reduce computational overhead, such as algorithm optimization and enhancing parallelism, in response to these key factors.

In particular, these features are especially critical in mobile or embedded devices, which have significantly fewer computational resources than desktop systems. In such resource-constrained environments, power consumption must be carefully considered, as programs with high computational demands require not only substantial resources but also significant power [11]. For example, sound rendering involves repeated traversal and intersection tests, which are manageable on desktops but impose stricter limitations on mobile devices and may adversely affect battery life. However, research on acceleration structures for ray-tracing-based sound propagation in mobile environments remains limited.

Considering all of these challenges, in this paper, we propose guidelines for mobile devices that address both the selection of acceleration structures for sound rendering based on scene characteristics and the optimization of the chosen structures. We experimentally implement and analyze the impact of acceleration structures on sound propagation performance. Specifically, we examine two representative acceleration structures: kd-trees [12] and multi-bounding volume hierarchies (MBVHs) [13]. To enable

more efficient ray traversal and intersection tests on mobile CPU architectures, we apply optimization techniques using ARM NEON [14] single-instruction, multiple-data (SIMD) instructions. In addition, we incorporate commonly used optimization strategies specific to each acceleration structure to further enhance performance. According to our experiments with state-of-the-art sound rendering software for mobile and embedded devices [15], our optimization approach, when compared against the baseline kd-trees, achieved performance improvements of up to $1.35\times$ for optimized kd-trees and $1.44\times$ for MBVHs, with minimal increases in power consumption, on a Google Pixel 8a. Performance was also evaluated on two configurations: using big and middle cores (i.e., high- and mid-performance CPU cores in the ARM big.LITTLE architecture [16]—a concept proposed to reduce processor power by combining heterogeneous cores [17]), which delivered real-time frame rates; and using little cores only, which provided interactive frame rates with low power consumption.

II. RELATED WORK

A. ACCELERATION STRUCTURES

In recent decades, there has been an increase in research aimed at accelerating ray tracing to achieve real-time frame rates. An acceleration structure is a spatial data structure designed to organize scene geometry so that rays can skip non-intersecting regions, thereby reducing the number of ray-primitive intersection tests and avoiding unnecessary checks against all primitives. Among the various acceleration structures explored, kd-trees and BVHs have garnered significant attention. This section provides an overview of recent advancements in these two structures.

1) KD-TREES

Kd-trees are hierarchical data structures that recursively partition space using planes perpendicular to the coordinate axes [18]. This property makes them suitable for ray tracing, where early termination is crucial upon finding the closest hit point on a primitive in a leaf node. Typically, kd-trees are constructed using the surface area heuristic (SAH) [19], [20], which assumes that the probability of ray traversal through a node is proportional to its surface area. Shevtsov et al. [21] parallelized kd-tree construction by integrating object partitioning with binned [22] and exact SAH. Soupikov et al. enhanced the algorithm by introducing the on-the-fly pruning technique [12].

2) BOUNDING VOLUME HIERARCHIES (BVHS)

BVHs [23] are widely used for ray tracing as well. Unlike kd-trees, a BVH represents an object hierarchy; all objects are enveloped in bounding volumes, such as axis-aligned bounding boxes (AABBs) or object-oriented bounding boxes (OBBs), and they are grouped hierarchically to construct a tree structure. Modern ray-tracing engines, such as Intel Embree [10] and NVIDIA OptiX [8], are based on BVHs,

with continuous optimization efforts over the years. BVH construction techniques often involve SAH [24] for high tree quality or node refitting strategies [25], [26] to enhance update performance. Spatial splits during BVH construction have been suggested by Stich et al. to mitigate overlapping child nodes and improve traversal efficiency [27].

3) HYBRID DATA STRUCTURES

Hybrid approaches that combine features of kd-trees and BVHs have been explored. Kang et al. introduced gkDtree, a collection of kd-trees with multi-level hierarchical structures and parallelization techniques [28]. Utilizing a scene graph, gkDtree enables partial, concurrent updates of local subtrees. Bounding interval hierarchies (BIHs) store two parallel planes per node, facilitating node refitting similar to BVHs [29]. Dual-split trees also employ two planes per node but separate splitting and carving nodes, resulting in more compact, efficient representations [30].

4) SIMD OPTIMIZATION

To leverage parallel hardware resources for ray traversal, techniques called packet tracing have been initially developed. Packet tracing [31] involves tracing multiple rays together, but its efficiency diminishes when tracing incoherent rays within a packet. Data structures for single-ray SIMD traversal and intersections, such as multi-BVHs (MBVHs) [13], quad-BVHs [32], or wide-BVHs [33], have been proposed to address this issue. These approaches store multiple nodes and primitives as arrays of structures, respectively and utilize SIMD intrinsics for traversal and intersection tests, maintaining SIMD efficiency even with incoherent rays. The MBVH is also the base acceleration structure for hardware ray-tracing accelerators in some recent GPUs (e.g., AMD Radeon series [34]).

B. GEOMETRIC-ACOUSTICS (GA) METHODS

1) RAY-TRACING-BASED GA ALGORITHMS

Among numerous studies conducted on GA methods, we will briefly introduce ray-tracing-based approaches directly related to our study. Schissler and Manocha introduced GSOUND, a sound propagation software system [35]. This ray-tracing-based system supports early reflection and first-order edge diffraction at interactive rates. Subsequently, Schissler et al. proposed a novel hybrid convolution audio rendering algorithm capable of rendering tens of thousands of paths at interactive rates, combining backward ray tracing and sound propagation methods with source clustering [36]. This algorithm demonstrated immersive sound effects at high speeds in complex scenes with indoor or outdoor environments and dynamic content. Schissler et al. also proposed a dynamic geometric diffraction approach within a real-time GA framework using a comprehensive mesh preprocessing pipeline and complementary runtime algorithms to simulate diffraction [37]. Zhou et al. introduced an adaptive sound propagation method based on GA,

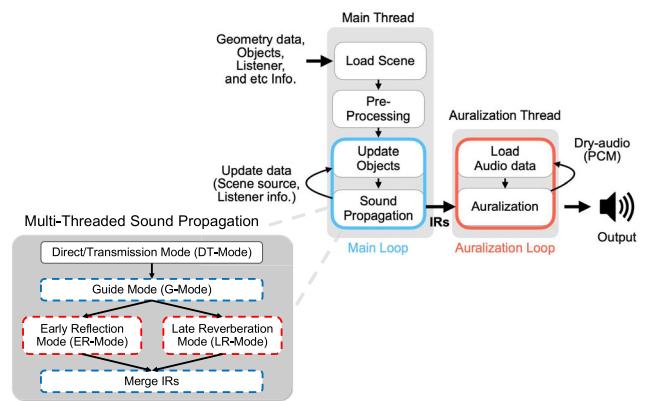


FIGURE 1. The figure shows the overall flowchart of the sound rendering pipeline, excerpted from [15]. The left flowchart illustrates the detailed procedure of the multi-threaded sound propagation method. In this detailed flowchart, the blocks outlined with blue and red dashed lines, respectively, indicate the modifications made to support multi-threading in the original single-threaded approach and the newly separated modes introduced for multi-threaded execution.

incorporating A-weighting variance to control the number of samples in sound propagation methods, reducing unnecessary samples [5]. Additionally, some recent sound propagation methods that utilize machine learning have been developed to speed up processing and improve accuracy [38]. However, they still require high computing resources (e.g., a high-end desktop GPU), which can be challenging for embedded systems.

2) MOBILE SW/HW APPROACHES

The GA methods based on ray tracing mentioned above have primarily been applied to PC-based platforms due to their high computational demands. For example, leveraging the high computational capabilities of desktop systems, Listen2Scene [39] was proposed as a material-aware learning-based approach for sound propagation in real 3D scenes, achieving significantly faster performance than interactive geometric sound propagation methods. However, implementing such methods on mobile or embedded environments with limited resources can pose challenges, particularly in achieving the real-time performance that we aim for when processed on CPUs. Kim et al. addressed this issue by proposing an adaptive depth control algorithm [40] and a multi-threaded sound propagation algorithm [15]. Later, Kim et al. proposed a hardware-friendly propagation-path calculation algorithm and a corresponding hardware architecture aimed at maximizing performance while minimizing required power consumption [41]. This architecture is a modified version of the previous kd-tree-based hardware architecture for ray-traced graphics [42], adapted for sound propagation.

III. SYSTEM OVERVIEW FOR EXPERIMENTS

A. BASELINE SOFTWARE

Our experiment builds upon the multi-threaded sound propagation algorithm introduced by Kim et al. [15], and

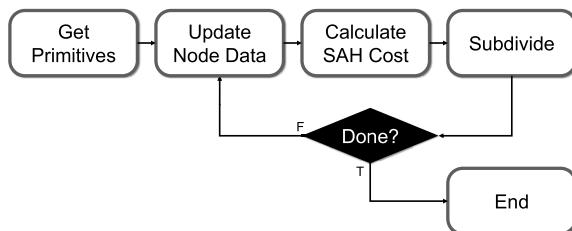


FIGURE 2. The SAH-based kd-tree build process (the preprocessing stage in Figure 1).

therefore, this subsection provides a concise overview of their algorithm implementation. As illustrated in Figure 1 (right), the algorithm first preprocesses the construction of a kd-tree for static objects after loading the scene. This kd-tree and Quilez's ray-triangle intersection test algorithm [43] are used during the sound propagation phase. Subsequently, within the main loop, object updates and sound propagation occur on each frame. The impulse responses (IRs) generated during sound propagation are then transmitted to the auralization loop. Utilizing these IRs, the auralization loop synthesizes the final signal by performing auralization on the dry audio in PCM format. Our analysis indicates that the cost of auralization is substantially lower than that of sound propagation (approximately $\frac{1}{20}$). Consequently, the overall performance is predominantly influenced by sound propagation within the main loop.

The primary contribution of Kim et al.'s algorithm [15] is the division of the main loop into two threads, as depicted in Figure 1 (left). This algorithm comprises the following steps: Initially, the kd-tree, incorporating dynamic objects in the scene, is rebuilt. Subsequently, various types of rays are shot and traced in multiple modes to compute impulse responses (IRs) for direct sound, diffraction, early reflection (ER), and late reverberation (LR) effects. The modes employed in this stage are delineated as follows. The initial mode executed is the direct/transmission mode (DT-mode), which calculates the propagation path of the direct sound reaching the listener earliest. Subsequently, in the guide mode (G-mode), multiple rays are emitted from the listener using spherical sampling, and by tracing them, combinations of intersected triangles are generated based on the depth of rays. In the ER-mode, these combinations validate the creation of valid paths, facilitating the computation of IRs for early reflection processing. Simultaneously, the LR-mode, operating on a distinct thread, shoots and traces rays from the sound source to the listener to retain valid paths. These valid paths are subsequently merged with the combinations of intersected triangles created in the G-mode to ultimately compute the IRs for late reverberation (LR). The IRs generated in these two threads are consolidated and outputted after the synchronization of LR- and ER-modes.

As mentioned earlier, accelerating the ray-tracing stage can lead to a reduction in the overall computation time of sound propagation because the threads of G-mode and LR-mode

execute backward and forward ray tracing independently. Therefore, our goal was to find an appropriate acceleration structure. To this end, we experimented with two different acceleration structures. The first experiment involved optimizing the kd-tree construction and partially vectorizing the ray-triangle intersection test algorithm using SIMD intrinsics. The second experiment applied the MBVH instead of the kd-tree and vectorized all traversal and intersection (T & I) steps using SIMD intrinsics. Each of these steps will be further elaborated in the subsequent subsections.

B. CASE 1: USING KD-TREES AS THE ACCELERATION STRUCTURE

In this subsection, we introduce the optimization techniques used with the kd-tree as the acceleration structure. The optimization approach elaborates the process of constructing the kd-tree with higher quality than the standard kd-tree and then improves the performance of intersection tests by leveraging SIMD intrinsics of the CPU.

First, let's discuss how the quality of a kd-tree can be optimized. In the baseline software [15], a kd-tree is constructed using the surface area heuristic (SAH) method, as depicted in Figure 2. This method initially loads primitive data, such as triangles in a scene, and selects the partitioning axis based on the distribution of this data. Then, the SAH cost is calculated for each split candidate, and the node is divided along the plane with the lowest SAH cost. This process is recursively applied to construct the hierarchical structure of a kd-tree.

The equation for calculating the SAH cost (C_{Node}) in kd-tree construction is represented by Eq. 1. In this equation, L and R denote the left and right child nodes, respectively, and P denotes the parent node. N_i represents the number of primitives contained in node i ($i \in \{L, R\}$), and $SA(i)$ denotes the surface area of the bounding box of node i .

$$C_{Node} = N_L \times \frac{SA(L)}{SA(P)} + N_R \times \frac{SA(R)}{SA(P)} \quad (1)$$

The number of split candidates used to calculate the SAH cost can be either the same as the number of planes of all primitives' AABBs in the current node on the current split axis (for exact SAH build) or the number of predefined bins (for binned SAH build). For the SAH cost calculation, Kim et al. [15] adopted a strategy similar to the H/W tree builder in RayCore [42]. They applied binned SAH [22] at the high level of the tree to achieve faster tree construction speed and accurately calculated the SAH cost at the low level of the tree to maintain high tree quality. The criterion for this switch is based on the number of primitives; when the number of primitives within a node is less than or equal to 512, the binned SAH is replaced with the exact SAH.

The SAH assumption during kd-tree construction generally performs well in actual ray tracing, but there are some exceptional cases. For instance, if the AABB of a primitive is found to be within a node, but the primitive itself does not actually belong to the space of this node (Figure 3),

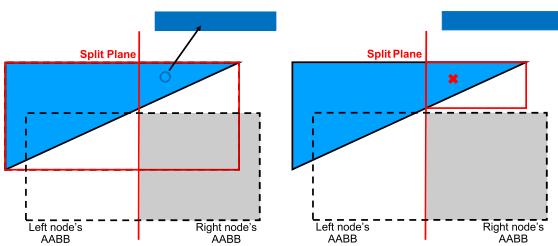


FIGURE 3. An example of on-the-fly pruning. The blue bar represents the primitive list of the right node for intersection tests. If only the split plane is considered during kd-tree construction, an issue can occur (left); the triangle is not actually included in the right node, but the AABB of the triangle is appended to the primitive list, thereby performing unnecessary ray-triangle intersection tests. When recalculating the AABB of a triangle tightly with respect to the split plane, the AABB of the right node no longer overlaps with the recalculated AABB of the triangle. Therefore, this triangle is no longer included in the primitive list belonging to the right node (right). This pruning prevents unnecessary intersection tests between this triangle and rays when visiting the right node.

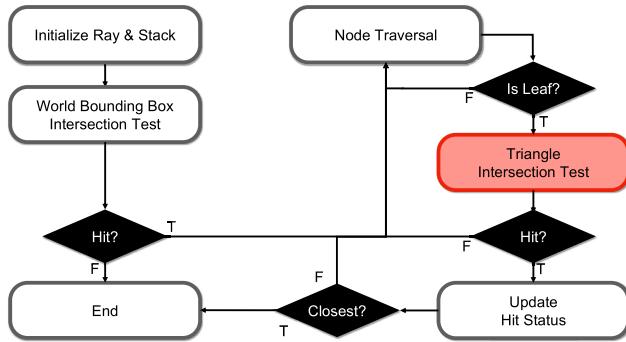


FIGURE 4. Flowchart of kd-tree traversal and our optimization part (highlighted in red).

intersection tests between rays and this primitive will always be determined to be false during traversal. Additionally, although less significant than the above issue, the switch policy from binned SAH to accurate SAH can impact the quality of the tree.

To address these issues, we enhance tree quality by utilizing the on-the-fly pruning [12] method. This method considers primitives that intersect the split plane during tree construction but are not fully contained in either child node. In this method, after clipping triangles for each left and right node separately and recalculating the AABBs for them, primitives that do not belong to either child node are excluded. Consequently, unnecessary ray-primitive intersection tests that would always yield false results can be effectively reduced (see Figure 3). Additionally, we apply the accurate SAH when more than 25% of primitives overlap both the left and right child nodes, as proposed in the on-the-fly pruning paper [12]. This adjustment is necessary because cases where many primitives overlap with both nodes indicate that the split plane calculated at the bin level is not optimal.

After completing the preprocessing steps, the sound propagation stage of the sound rendering pipeline, as depicted

LISTING 1. Triangle preprocessing step for ray-triangle intersections.

```

1 // ...
2 // Loading the triangle of mesh
3 Primitives[PrimNum].vtx0 =
4 vsetq_lane_f32(0, tri.v[0][2], tri.v[0][1], tri.
5   v[0][0]);
6 float32x4_t vtx1 =
7 vsetq_lane_f32(0, tri.v[1][2], tri.v[1][1], tri.
8   v[1][0]);
9 float32x4_t vtx2 =
10 vsetq_lane_f32(0, tri.v[2][2], tri.v[2][1], tri.
11   v[2][0]);
12
13 float32x4_t cVector =
14 vsubq_f32(vtx1, Primitive[PrimNum].vtx0);
15 float32x4_t bVector =
16 vsubq_f32(vtx2, Primitive[PrimNum].vtx0);
17 float32x4_t cCrossb =
18 simd_accelerate_cross_product(cVector, bVector);
19
20 Primitives[PrimNum].cVector = cVector;
21 Primitives[PrimNum].bVector = bVector;
22 Primitives[PrimNum].cCrossb = cCrossb;
23
24 // ...
  
```

in Figure 1, simulates the path of sound using kd-tree traversal. The process follows the structure shown in Figure 4. Initially, the ray checks for an intersection with the scene's bounding box, and only if it passes through, kd-tree traversal occurs. The search begins at the root node and progresses to leaf nodes. Upon reaching a leaf node, intersection tests are conducted between the triangles overlapping the leaf node and the current ray. Once the closest hit point is found, further traversal is terminated.

The kd-tree node traversal involves relatively minimal per-node operations compared to BVH node traversal, as it requires checking for ray intersections with a single plane for each node, involving only one addition and one multiplication operation. However, due to the limited number of operations that can be parallelized, it is not well-suited for SIMD acceleration. Packet tracing [31] can be used to improve this, but it is not effective for scenarios with low coherence between rays, such as in sound propagation.

Therefore, we introduce a novel SIMD optimization technique to accelerate the ray-triangle intersection test based on Quilez's algorithm [43]. First, while the original Quilez algorithm directly accesses vertices without preprocessing, our approach introduces a preprocessing technique that calculates the edge vectors connecting two vertices and stores them for each triangle in memory, as shown in Listing 1. This preprocessing reduces intersection costs by eliminating repetitive subtraction and cross-product operations. Although this approach increases memory usage, it is suitable for mobile or embedded systems where the number of triangles is relatively small, such as in sound rendering. Consequently, this preprocessing introduces an additional memory overhead of 64 bytes per triangle (four `float32x4_t` variables). This precomputation strategy is inspired by Wald's TriAccel data structure [31].

Second, ray-triangle intersection tests during the ray traversal stage are also SIMD-accelerated as outlined in Listing 2.

LISTING 2. SIMD-accelerated ray-triangle intersection algorithm.

```

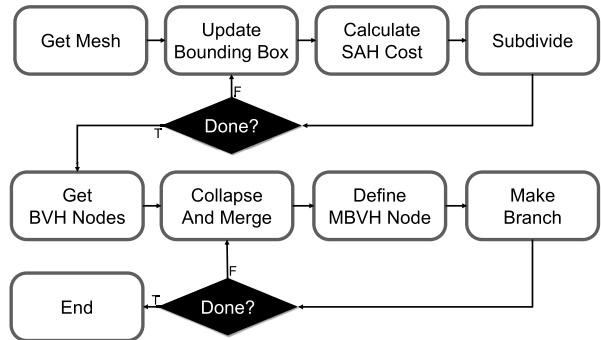
1 vec3 ray_tri_intersect(Ray ray, Primitive* Primitives, int PrimNum)
2 {
3     float32x4_t vtx0 = Primitives[PrimNum].vtx0;
4     float32x4_t cVector = Primitives[PrimNum].
5         cVector;
6     float32x4_t bVector = Primitives[PrimNum].
7         bVector;
8     float32x4_t cCrossb = Primitives[PrimNum].
9         cCrossb;
10
11    float32x4_t oVector = vsubq_f32(&vtx0, &ray.o3);
12    float32x4_t dCrosso =
13        simd_accelerate_cross_product(&ray.d3, &
14            oVector);
15
16    float32x4_t v0, v1, v2, v3, v4, v5;
17    v0 = vsetq_lane_f32(cVector.a, bVector.a,
18        oVector.a, ray.d3.a);
19    v1 = vsetq_lane_f32(cVector.b, bVector.b,
20        oVector.b, ray.d3.b);
21    v2 = vsetq_lane_f32(cVector.c, bVector.c,
22        oVector.c, ray.d3.c);
23    v3 = vsetq_lane_f32(dCrosso.a, dCrosso.a,
24        cCrossb.a, cCrossb.a);
25    v4 = vsetq_lane_f32(dCrosso.b, dCrosso.b,
26        cCrossb.b, cCrossb.b);
27    v5 = vsetq_lane_f32(dCrosso.c, dCrosso.c,
28        cCrossb.c, cCrossb.c);
29
30    SIMDScalarFloat4 tmp1, tmp2, tmp3, tmp4;
31    tmp1 = vmulq_f32(&v0, &v3);
32    tmp2 = vmulq_f32(&v1, &v4);
33    tmp3 = vaddq_f32(&tmp1, &tmp2);
34    tmp4 = vmulq_f32(&v2, &v5);
35    intermediate_results = vaddq_f32(&tmp3, &tmp4);
36
37    // intermediate_results -> {m, t, beta, gamma}
38    // scalar operations to determine whether the
39    // ray intersects the triangle.
40    // ...
41 }

```

The precomputed triangle data is utilized to accelerate dot product computations between the ray's origin and direction vectors and the precomputed vectors using SIMD operations (lines 3-9). The next step requires a set of dot products; however, its SIMD implementation is not straightforward. To address this, we restructure the input into six SIMD vectors during computation (lines 11-17) and convert four scalar dot products into three vector multiplications and two vector additions (lines 19-24). By combining these preprocessing and SIMD computation techniques, we significantly accelerate the ray-triangle intersection test. Finally, similar to the original algorithm [43], scalar operations are performed to verify these values and determine whether the ray intersects the triangle.

C. CASE 2: USING MBVHS AS THE ACCELERATION STRUCTURE

In this subsection, we introduce the method of using an MBVH instead of a kd-tree as the acceleration structure, and how it facilitates vectorizing the entire T & I step in the sound propagation stage. To utilize the MBVH, we referenced Ernst and Greiner's method [13].

**FIGURE 5.** The SAH-based BVH and MBVH build process.

The following describes how the MBVH is built. First, similar to the kd-tree, we generate BVH nodes using the SAH method, as shown in Figure 5. BVH nodes are split based on the axis with the lowest cost, calculated using the centroids of primitives [26], which are provided as input data. This process is recursively applied to form the hierarchical structure of the binary BVH. Then, the BVH is converted into an MBVH with four child nodes by collapsing and merging BVH nodes.

During the construction of BVH nodes for an MBVH, Ernst and Greiner [13] ensured optimal SIMD utilization for the MBVH by limiting the number of primitives per node to match the SIMD size. This is necessary to prevent additional loops for intersection tests in a leaf node, which would complicate the data structure. To ensure the intended number of primitives, we applied a simple method during the construction process of the binary BVH. If the number of reference primitives in a leaf node is four or fewer, the node subdivision is terminated. Even if further subdivision of nodes with more than four primitives does not reduce the SAH cost, we force the subdivision to ensure that the number of primitives per node is four or fewer. The split candidate with the lowest cost is selected for the subdivision. This policy is not optimal for binary BVHs in terms of traversal performance but simplifies the data layout and traversal algorithm for MBVHs. Additionally, we rearrange the array of structures (AoS) memory layout to the structure of arrays (SoA) layout for MBVH node data to optimize SIMD operations.

Finally, to determine the traversal order of nodes, we experimented with visiting child nodes in the accurate front-to-back order by comparing t-values, which represent the distance between the intersection point and the bounding box, in a binary BVH, as suggested by Lee et al. [44], and applied this method to an MBVH as well. This resulted in a slight reduction in intersection counts compared to the predefined traversal order. However, due to the overhead of sorting four t-values at each traversal step, the overall performance decreased by approximately 10%. For this reason, we decided to adopt the predefined traversal order proposed by Ernst and Greiner [13].

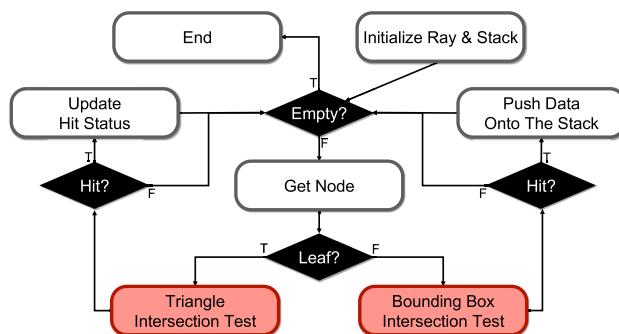


FIGURE 6. Flowchart of MBVH traversal and our optimization part (highlighted in red).

After the construction of the MBVH, ray-AABB and ray-triangle intersection tests during the traversal stage are accelerated using ARM Neon SIMD operations, leveraging data calculated in the preprocessing phase. The information of a single ray is duplicated into four, and up to four preprocessed AABBs or triangles are processed together in parallel to determine intersections. This is a different parallelization approach compared to kd-trees from Section III-B, where vector operations inside the ray-triangle intersection algorithm were accelerated using SIMD intrinsics. The reason for employing different methods is that, while one triangle may belong to multiple leaf nodes in the space partitioning method of kd-trees, in the object partitioning method of MBVHs, each triangle belongs to only one leaf node. Therefore, arranging data for up to four triangles in the SoA format in an MBVH allows easier parallelization using SIMD intrinsics. Consequently, while our kd-tree implementation reduces the cost of ray-triangle intersection tests, the MBVH implementation focuses on reducing the number of intersection tests. After these SIMD intersections, the final intersection checking is performed by using scalar operations.

D. THREAD AFFINITY AND POWER CONSUMPTION

Using significant CPU resources for sound rendering may lead to performance degradation in other tasks or unnecessary increases in power consumption. To address these issues, we have added an option to allocate sound rendering threads exclusively to the little cores on a multi-core CPU. Since the number of little cores may vary depending on the CPU model, we read CPU information from the files ‘proc/cpuinfo’ and ‘/sys/devices/system/cpu/cpuX/cpufreq/cpuinfo_max_freq’ (where X denotes a CPU ID) to allocate threads. We then sorted the cores by their maximum frequency in ascending order and assigned threads to the little cores, which operate at lower frequencies compared to the big cores. To implement this feature in our software, we referred to Jamsoft’s CCpuAffinity [45]. Of course, if higher frame rates are required due to a large number of sound sources or complex scenes, this option can be disabled, allowing the use of the big and middle cores.

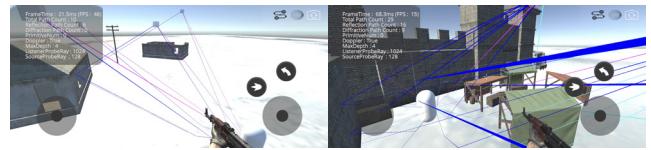


FIGURE 7. Captured screenshots of the running Android app. Blue lines represent reflection paths, while sky-colored lines depict diffraction paths, thereby visualizing sound propagation from the sound source.

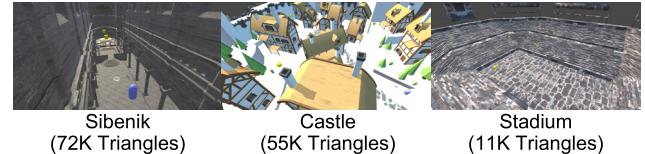


FIGURE 8. Three scenes used for the experiments. In each scene, yellow boxes depicted the positions of sound sources, while purple capsules represented the listener.

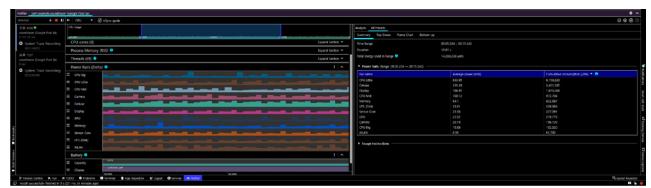


FIGURE 9. Example of energy consumption measurement using the power rail of the Android Studio Iguana.

TABLE 1. Specification of the application processor (AP) in the Pixel 8a.

AP	CPU	GPU
Google Tensor G3	1 × ARM Cortex-X3 2.91 GHz 4 × ARM Cortex-A715 2.37 GHz 4 × ARM Cortex-A510R 1.70 GHz	ARM Mali-G715 MP7

As demonstrated by prior studies on smartphone power consumption characteristics [11], computation-intensive applications constitute a major factor that can substantially reduce battery life. Accordingly, we focused on SIMD-based optimizations for each acceleration structure [46], as SIMD enables the effective reduction of computational cost. By utilizing SIMD intrinsics, traversal and intersection operations were vectorized, thereby allowing the same computations to be performed with fewer CPU cycles. Furthermore, when only the aforementioned little cores are employed, the computational throughput of a single thread is limited, which can impede the attainment of interactive frame rates. Leveraging SIMD instructions plays a critical role in enhancing single-thread performance while concurrently mitigating unnecessary power consumption.

IV. RESULTS

A. EXPERIMENT SETUP

For our experiments, we followed a methodology similar to that of the multi-threaded sound propagation algorithm [15]. We generated naive plugins in the form of .so and .lib files, integrating them into the Android application (see Figure 7),

which was developed using Unity Engine version 2020.3.47. We conducted sound propagation using each acceleration structure across three scenes, as illustrated in Figure 8. These test scenes consisted entirely of static environments, with evaluations conducted using arbitrary audio data. Sibenik is a standard indoor test scene, with the feature that sound sources were biased to one side. Castle is an outdoor test scene, characterized by many obstacles such as trees, rocks, and buildings. Lastly, Stadium is an outdoor test scene characterized by a relatively wide space. Each scene contained four sound sources. To evaluate performance numerically, we compared metrics such as the number of node traversals per ray (TRV), triangle intersection tests (IST), and frames per second (FPS) against prior sound rendering methods mentioned in Section III-A. In our experiments, the ray depth required for sound rendering was fixed at 4, and the number of the listener probe rays and the source probe rays were fixed at 1024 and 128, respectively.

We used a mobile device for our experiments: the Google Pixel 8a, which is equipped with Android 14, 8GB of RAM, and the Google Tensor G3 application processor. Table 1 describes the CPUs and GPUs integrated into the Tensor G3. The sound propagation component of the software utilizes two threads, which are allocated to two of the CPU's little cores (Cortex-A510R), or one big core (Cortex-X3) and one of the middle cores (Cortex-A715), according to the thread affinity configuration. Additionally, the development of applications on mobile or embedded devices involves a trade-off between energy consumption and performance. Unlike desktops, even if application performance improves, an increase in energy consumption may cause critical issues such as excessive battery usage or heat dissipation. Therefore, to measure energy consumption before and after exploiting SIMD units, we utilized the power profiler introduced in Android Studio Iguana. Figure 9 shows an example of the energy consumption measurement method. When calculating energy consumption for sound rendering, we referred to the profiler values. Specifically, when only little cores were used, we considered the numerical values of 'CPU Little' and 'Memory'. In contrast, when both big and middle cores were used, we considered the numerical values of 'CPU Big', 'CPU Middle', and 'Memory'.

B. EXPERIMENTAL RESULTS OF KD-TREES

In this subsection, we evaluate the impact of applying well-known optimization techniques, alongside our method, to kd-tree traversal in the context of sound rendering on a mobile device. Figure 10 presents the average FPS measured across all test scenarios on the Google Pixel 8a. Both the combined big and middle cores ("B+M") and little cores only ("L") were evaluated.

For the B+M configuration, on-the-fly pruning led to slight improvements in FPS. In the Sibenik scene, FPS increased from 26.9 to 27.1 (1.01×), while in the Castle and Stadium scenes, FPS improved from 38.1 to 41.6 (1.09×) and from 57.2 to 59.8 (1.05×), respectively. For the little core

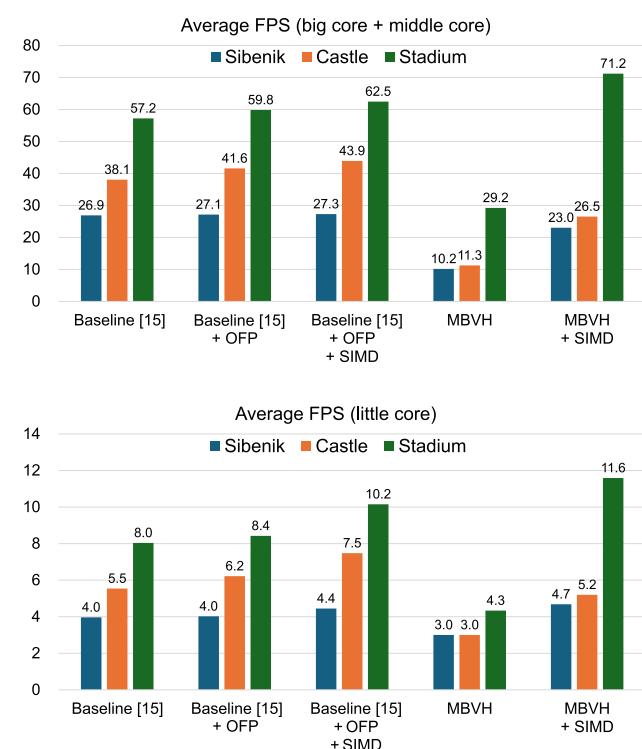


FIGURE 10. Average FPS over 1000 frames across three test scenes under different optimization methods. The upper graph presents the FPS when both big and middle cores are utilized, whereas the lower graph shows the FPS when execution is confined to little cores only. OFP denotes on-the-fly pruning.

only configuration, the Sibenik scene FPS remained at 4.0, whereas the Castle and Stadium scenes increased from 5.5 to 6.2 (1.12×) and from 8.0 to 8.4 (1.05×), respectively.

Applying SIMD intrinsics further improved performance. For B+M, the Sibenik, Castle, and Stadium scenes increased to 27.3 (1.02×), 43.9 (1.15×), and 62.5 (1.09×), respectively. For the little cores, FPS increased to 4.4 (1.12×) in Sibenik, 7.5 (1.35×) in Castle, and 10.2 (1.26×) in Stadium. These results indicate that SIMD acceleration complements on-the-fly pruning by reducing ray-triangle intersection computation overhead. Additional energy consumption due to SIMD usage remained below 2% when only little cores were used, whereas it increased by up to 5.8% when both big and middle cores were utilized.

C. EXPERIMENTAL RESULTS OF MBVHS

We next evaluate the impact of applying the MBVH acceleration structure and SIMD, compared to the kd-tree. Figure 10 presents the average FPS for both B+M and little core configurations.

For naive MBVH without SIMD, FPS decreased substantially. For B+M, the Sibenik and Castle scenes dropped to 10.2 (0.38×) and 11.3 (0.3×), respectively, compared to baseline kd-trees. The Stadium scene decreased to 29.2 (0.51×). For little cores only, the Sibenik and Castle scenes dropped to 3.0 (0.76×) and 3.0 (0.54×), respectively, and the

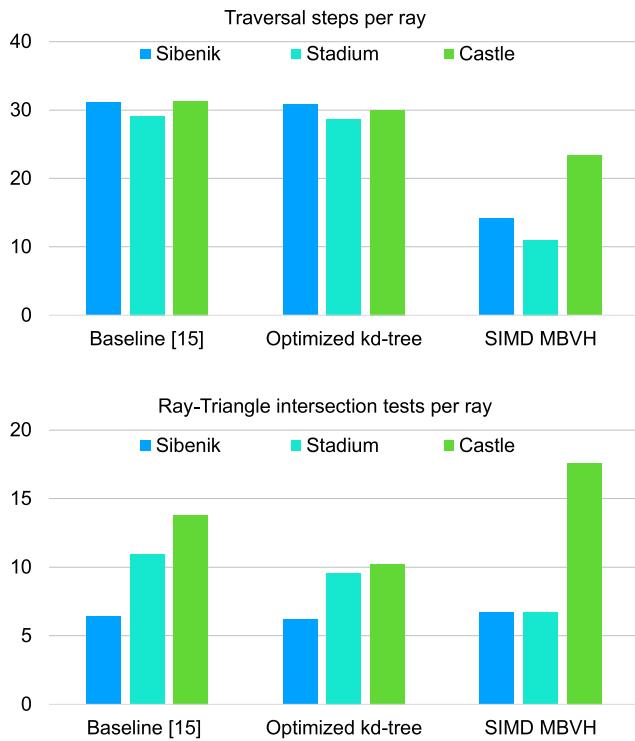


FIGURE 11. Comparison of each acceleration structure in terms of numerical values for TRV and IST. Higher values indicate more computational operations, but note that the computational costs per operation differ between kd-trees and MBVHs.

Stadium scene decreased to 4.3 (0.54×). This performance drop was caused by the naive MBVH traversal, which required additional loop iterations and intersection tests.

When SIMD acceleration was applied, FPS improved substantially. For B+M, FPS increased to 23.0 (0.86×) in Sibenik, 26.5 (0.7×) in Castle, and 71.2 (1.24×) in Stadium. For little cores only, FPS increased to 4.7 (1.18×), 5.2 (0.94×), and 11.6 (1.44×) in Sibenik, Castle, and Stadium, respectively. These results indicate that SIMD effectively alleviates the traversal bottleneck in MBVH, particularly in open environments. However, several failure cases were observed, which are further discussed in Section IV-D. Additional energy consumption remained below 2% when only little cores were used, and increased by up to 1.3% when both big and middle cores were utilized.

D. COMPARISON IN TERMS OF TRAVERSALS (TRV) AND INTERSECTIONS (IST)

In the next analysis, we aim to compare each acceleration structure based on the number of TRV and IST per ray across different scenes. Figure 11 shows the TRV and IST values measured for each test scene, categorized by acceleration structure. The key focus is on the efficiency of each acceleration structure, considering the characteristics of the scene. In summary, across all test scenes, the optimized kd-tree yielded reductions of 0.9%–4.1% in TRV and

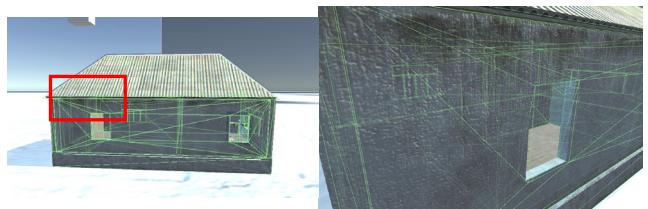


FIGURE 12. To visualize the wireframe of the example of the overlap problem scenario, the right image depicts an enlarged view of the red box on the left. This enlargement reveals that the walls are modeled with double-layered triangles, leading to overlapping AABBs.

0.4%–25.9% in IST, whereas the SIMD MBVH showed reductions of 25.4%–62.2% in TRV and 4.1%–38.6% in IST.

According to our experimental results, the optimized kd-tree exhibited only minor changes in TRV and IST values for test scenes such as Sibenik and Stadium, which are relatively simpler than Castle. Nevertheless, an improvement in FPS was observed, as the optimization primarily focused on accelerating ray-triangle intersection tests using SIMD. This highlights the necessity of employing SIMD on mobile or embedded devices with limited computational resources. In contrast, for more complex scenes such as Castle, where a large number of objects are present, on-the-fly pruning proved to be effective, as it partitions space not only using bounding boxes but also split planes. During kd-tree traversal, the likelihood of bounding box overlaps increases in complex scenes, where a larger number of objects can be rendered, as discussed in the Section III-B. This issue similarly arises in dynamic scenes. However, by applying the on-the-fly pruning method, which refits the bounding boxes of triangles, unnecessary ray-triangle intersection tests can be reduced, allowing for more precise traversal. As a result, the method demonstrated particularly favorable performance in the Castle test scene. In other words, while the benefits of the on-the-fly pruning method may be negligible in very simple scenes, its importance increases with scene complexity. Therefore, depending on the scene's complexity, the kd-tree with the on-the-fly pruning method can be a more efficient choice.

In contrast, the MBVH achieved a significant reduction in TRV and IST for all test scenes compared to the optimized kd-tree. These results demonstrate that multiple calculations can be handled at once using SIMD intrinsics, reducing repetitive operations during traversal and intersection tests. The SIMD MBVH handles TRV and IST more efficiently than the optimized kd-tree, especially in open environments such as Stadium and scenes with fewer obstacles, such as Sibenik. As a result, higher FPS was observed compared to the optimized kd-tree. However, the IST and TRV values for the Castle scene using SIMD MBVH were significantly higher compared to other test scenes. To explain this, Figure 12 provides an example. Due to the presence of large or overlapping triangles in the scene, the bounding boxes of BVH nodes are often misaligned with the actual sizes of the

TABLE 2. Memory footprints of the trees and triangles in the test scenes (unit: MB).

Acceleration structure	Test scene	Tree data	Triangle data
Baseline kd-tree	Sibenik	3.42	3.18
	Castle	4.84	2.42
	Stadium	0.89	0.50
Optimized kd-tree	Sibenik	3.07	7.80
	Castle	3.67	5.95
	Stadium	0.70	1.23
SIMD MBVH	Sibenik	1.49	6.36
	Castle	1.22	4.85
	Stadium	0.24	1.00

primitives. In complex scenes such as Castle, which contain numerous obstacles, the bounding boxes of BVH tend to overlap extensively. These overlapping regions considerably increase the TRV and IST values. When searching for the closest hit point, if a node's bounding box overlaps with others such that the ray may intersect multiple nodes, the ray often performs unnecessary operations by traversing nodes that do not contain the optimal primitive associated with the closest hit point. As a result, such scenes tend to exhibit relatively higher TRV and IST values compared to other scenes.

The experimental results demonstrate that the performance difference between the two acceleration structures varies depending on scene complexity. Specifically, in scenes with numerous obstacles or overlapping triangles, the overlap problem of MBVHs leads to unnecessary additional computations, causing performance degradation. In such cases, using the kd-tree proves to be a more efficient choice. On the other hand, in scenes with fewer obstacles and minimal overlap, the MBVH outperforms the kd-tree in terms of parallel processing performance. This is because the MBVH structure is well-suited for parallelization, allowing for increased computational speed. The choice of acceleration structure significantly impacts performance, especially on mobile devices, where computational resources are limited compared to desktop environments.

E. MEMORY FOOTPRINTS

As a final point, we compare the memory footprint of each acceleration structure. For the test scenes used in the experiments, the memory footprint of the acceleration structures ranged from 1.2 to 10.8 MB, which is not large. However, for larger scene sizes, this could become a significant factor. The measured results are shown in Table 2. First, the optimized kd-tree increased the memory footprint by 39% to 69%. The on-the-fly pruning algorithm for the kd-tree reduced the number of nodes and triangle lists. However, as mentioned in Section III-B, the additional precomputed triangle data increased the memory size for the triangles. Second, an MBVH had a significantly lower memory footprint for the tree data compared to a kd-tree. This is attributed to the shallower tree levels and the removal of the triangle lists. Although the triangle data size increases due to the inclusion of edge and vertex information, the reduction in

tree data results in the total memory usage of MBVHs ranging from 16% lower to 19% higher compared to baseline kd-trees. These results are more memory-efficient than our optimized kd-trees, making MBVHs a more advantageous choice for handling large-scale scenes.

V. CONCLUSION

In this paper, we compared two key acceleration structures, kd-trees and MBVHs, for sound rendering on mobile/embedded devices by combining various optimization techniques. The experimental results demonstrate that the performance difference between the two acceleration structures varies depending on scene complexity.

This paper highlights the importance of selecting the appropriate acceleration structure to optimize sound rendering performance in mobile environments. By demonstrating the effectiveness of flexible acceleration structure selection based on scene characteristics, this research provides valuable insights for the development and optimization of sound rendering applications on mobile devices. Not only can these findings be applied to sound rendering, but they are also relevant to other domains, such as collision detection, physical simulation, robotics, and data structure search, where efficient spatial queries are critical.

VI. FUTURE WORK

As the next step in this research, we would like to prioritize efficiently handling both static and dynamic environments. For static objects, spatial splits [27] can be applied during MBVH construction, reducing unnecessary traversals—the main cause of MBVH performance degradation in the Castle scene. In dynamic scenes, tree decomposition can be a useful technique for fast and partial tree updates, as demonstrated in gKDtrees [28].

MBVHs can also be constructed using a simple two-level hierarchy, as employed in Vulkan and DirectX Ray Tracing [47], [48], for dynamic scenes. Specifically, only the top-level acceleration structure (TLAS) needs to be rebuilt, while the bottom-level acceleration structures (BLAS) can simply be refitted. However, in highly dynamic environments, frequent changes in object positions may reduce the effectiveness of these optimizations or introduce additional overhead. Future work will explore strategies to efficiently update acceleration structures in real time, enabling the proposed methods to be applied to dynamic mobile scenarios without performance degradation.

Finally, we are interested in architectural improvements to existing hardware architectures for sound rendering [41]. We believe that our kd-tree optimizations can be directly applied to kd-tree-based architectures, and the MBVH optimizations can provide insight into designing a new hardware architecture. These dedicated hardware approaches could offer more energy-efficient solutions, thereby enabling fully immersive virtual reality experiences on embedded systems. We are also interested in converting the core part of our sound propagation algorithm into Vulkan code to run

on GPUs, utilizing ray-tracing acceleration units in recent mobile/embedded GPUs (such as Qualcomm Adreno, ARM Mali, Samsung Xclipse, etc.).

REFERENCES

- [1] T. Takala and J. K. Hahn, "Sound rendering," in *Proc. 19th Annu. Conf. Comput. Graph. Interact. Techn.*, 1992, pp. 211–220.
- [2] S. Liu and D. Manocha, *Sound Synthesis, Propagation, and Rendering*. San Rafael, CA, USA: Morgan & Claypool.
- [3] A. Krokstad, S. Strom, and S. Sørsdal, "Calculating the acoustical room response by the use of a ray tracing technique," *J. Sound Vibrat.*, vol. 8, no. 1, pp. 118–125, Jul. 1968.
- [4] T. Whitted, "An improved illumination model for shaded display," *Commun. ACM*, vol. 23, no. 6, pp. 343–349, Jun. 1980.
- [5] H. Zhou, Z. Ren, and K. Zhou, "Adaptive geometric sound propagation based on A-weighting variance measure," *Graph. Models*, vol. 116, Jul. 2021, Art. no. 101109.
- [6] Apple. (2024). *Apple Vision Pro*. [Online]. Available: <https://www.apple.com/apple-vision-pro/>
- [7] M. Vinkler, V. Havran, and J. Bittner, "Performance comparison of bounding volume hierarchies and Kd-trees for GPU ray tracing," *Comput. Graph. Forum*, vol. 35, no. 8, pp. 68–79, Dec. 2016.
- [8] S. G. Parker, J. Bigler, A. Dietrich, H. Friedrich, J. Hoberock, D. Luebke, D. McAllister, M. McGuire, K. Morley, A. Robison, and M. Stich, "OptiX: A general purpose ray tracing engine," *ACM Trans. Graph.*, vol. 29, no. 4, pp. 1–13, 2010, Art. no. 66.
- [9] J.-H. Nah, J.-S. Park, C. Park, J.-W. Kim, Y.-H. Jung, W.-C. Park, and T.-D. Han, "T&I engine : Traversal and intersection engine for hardware accelerated ray tracing," *ACM Trans. Graph.*, vol. 30, no. 6, pp. 1–10, Dec. 2011, Art. no. 160.
- [10] I. Wald, S. Woop, C. Benthin, G. S. Johnson, and M. Ernst, "Embree: A kernel framework for efficient CPU ray tracing," *ACM Trans. Graph.*, vol. 33, no. 4, pp. 1–8, Jul. 2014, Art. no. 143.
- [11] E. Ahmadoh and L. A. Tawalbeh, "Power consumption experimental analysis in smart phones," in *Proc. 3rd Int. Conf. Fog Mobile Edge Comput. (FMEC)*, Apr. 2018, pp. 295–299.
- [12] A. Soupikov, M. Shevtsov, and A. Kapustin, "Improving kd-tree quality at a reasonable construction cost," in *Proc. IEEE Symp. Interact. Ray Tracing*, Aug. 2008, pp. 67–72.
- [13] M. Ernst and G. Greiner, "Multi bounding volume hierarchies," in *Proc. IEEE Symp. Interact. Ray Tracing*, Aug. 2008, pp. 35–40.
- [14] ARM. (2024). *Neon*. [Online]. Available: <https://developer.arm.com/Architectures/Neon/>
- [15] J. E. Kim, S. Choi, C. G. Kim, and W.-C. Park, "Multi-threaded sound propagation algorithm to improve performance on mobile devices," *Sensors*, vol. 23, no. 2, pp. 1–17, Jan. 2023, Art. no. 973.
- [16] Arm. (2025). *Big.LITTLE: Balancing Power Efficiency and Performance*. [Online]. Available: <https://www.arm.com/technologies/big-little>
- [17] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen, "Single -ISA heterogeneous multi -core architectures :The potential for processor power reduction," in *Proc. 22nd Digit. Avionics Syst. Conf.*, Jun. 2003, pp. 81–92.
- [18] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Commun. ACM*, vol. 18, no. 9, pp. 509–517, Sep. 1975.
- [19] J. D. MacDonald and K. S. Booth, "Heuristics for ray tracing using space subdivision," *Vis. Comput.*, vol. 6, no. 3, pp. 153–166, May 1990.
- [20] I. Wald and V. Havran, "On building fast kd-trees for ray tracing, and on doing that in O(n log n)," in *Proc. IEEE Symp. Interact. Ray Tracing*, Sep. 2006, pp. 61–69.
- [21] M. Shevtsov, A. Soupikov, and A. Kapustin, "Highly parallel fast kd-tree construction for interactive ray tracing of dynamic scenes," *Comput. Graph. Forum*, vol. 26, no. 3, pp. 395–404, Sep. 2007.
- [22] S. Popov, J. Gunther, H.-P. Seidel, and P. Slusallek, "Experiences with streaming construction of SAHKD-trees," in *Proc. IEEE Symp. Interact. Ray Tracing*, Sep. 2006, pp. 89–94.
- [23] J. H. Clark, "Hierarchical geometric models for visible surface algorithms," *Commun. ACM*, vol. 19, no. 10, pp. 547–554, Oct. 1976.
- [24] I. Wald, "On fast construction of SAH-based bounding volume hierarchies," in *Proc. IEEE Symp. Interact. Ray Tracing*, Sep. 2007, pp. 33–40.
- [25] C. Lauterbach, S.-E. Yoon, D. Manocha, and D. Tuft, "RT-DEFORM: Interactive ray tracing of dynamic scenes using BVHs," in *Proc. IEEE Symp. Interact. Ray Tracing*, Sep. 2006, pp. 39–46.
- [26] I. Wald, S. Boulos, and P. Shirley, "Ray tracing deformable scenes using dynamic bounding volume hierarchies," *ACM Trans. Graph.*, vol. 26, no. 1, pp. 1–28, Jan. 2007, Art. no. 6.
- [27] M. Stich, H. Friedrich, and A. Dietrich, "Spatial splits in bounding volume hierarchies," in *Proc. Conf. High Perform. Graph.* New York, NY, USA: Association for Computing Machinery, Aug. 2009, pp. 7–13.
- [28] Y.-S. Kang, J.-H. Nah, W.-C. Park, and S.-B. Yang, "gKDtree: A group-based parallel update kd-tree for interactive ray tracing," *J. Syst. Archit.*, vol. 59, no. 3, pp. 166–175, Mar. 2013.
- [29] C. Wächter and A. Keller, "Instant ray tracing: The bounding interval hierarchy," in *Rendering Techniques*, 2006, pp. 139–149.
- [30] D. Lin, K. Shkurko, I. Mallett, and C. Yuksel, "Dual-split trees," in *Proc. ACM SIGGRAPH Symp. Interact. 3D Graph. Games*, 2019, pp. 1–9, Art. no. 3.
- [31] I. Wald, "Realtime ray tracing and interactive global illumination," Ph.D. dissertation, Comput. Graph. Group, Saarland Univ., Saarbrücken, Germany, 2004.
- [32] H. Dammertz, J. Hanika, and A. Keller, "Shallow bounding volume hierarchies for fast SIMD ray tracing of incoherent rays," *Comput. Graph. Forum*, vol. 27, no. 4, pp. 1225–1233, Jun. 2008.
- [33] I. Wald, C. Benthin, and S. Boulos, "Getting rid of packets—Efficient SIMD single-ray traversal using multi-branching BVHs," in *Proc. IEEE Symp. Interact. Ray Tracing*, Aug. 2008, pp. 49–57.
- [34] D. Meister, P. Kulkarni, A. Vasishta, and T. Harada, "HIPRT: A ray tracing framework in HIP," *Proc. ACM Comput. Graph. Interact. Techn.*, vol. 7, no. 3, pp. 1–18, Aug. 2024.
- [35] C. Schissler and D. Manocha, "GSound: Interactive sound propagation for games," in *Proc. 41st Int. Conf. Audio Eng. Soc. Conf., Audio Games*, 2011, pp. 1–6, Art. no. P2-6.
- [36] C. Schissler and D. Manocha, "Interactive sound propagation and rendering for large multi-source scenes," *ACM Trans. Graph.*, vol. 36, no. 4, pp. 1–12, Jul. 2017, Art. no. 114c.
- [37] C. Schissler, G. Mückl, and P. Calamia, "Fast diffraction pathfinding for dynamic sound propagation," *ACM Trans. Graph.*, vol. 40, no. 4, pp. 1–13, Jul. 2021, Art. no. 138.
- [38] Z. Tang, H.-Y. Meng, and D. Manocha, "Learning acoustic scattering fields for dynamic interactive sound propagation," in *Proc. IEEE Virtual Reality 3D User Interfaces (VR)*, Mar. 2021, pp. 835–844.
- [39] A. Ratnarajah and D. Manocha, "Listen2Scene: Interactive material-aware binaural sound propagation for reconstructed 3D scenes," in *Proc. IEEE Conf. Virtual Reality 3D User Interfaces (VR)*, Mar. 2024, pp. 254–264.
- [40] E. Kim, J. Yun, W. Chung, J.-H. Nah, Y. Kim, C. G. Kim, and W.-C. Park, "Effective algorithm to control depth level for performance improvement of sound tracing," *J. Web Eng.*, vol. 21, no. 3, pp. 713–728, Feb. 2022.
- [41] E. Kim, S. Choi, J. K. Kim, J. Nah, W. Jung, T.-H. Lee, Y.-K. Moon, and W.-C. Park, "An architecture and implementation of real-time sound propagation hardware for mobile devices," in *Proc. SIGGRAPH Asia Conf. Papers*, 2023, pp. 1–9, Art. no. 81.
- [42] J.-H. Nah, H.-J. Kwon, D.-S. Kim, C.-H. Jeong, J. Park, T.-D. Han, D. Manocha, and W.-C. Park, "RayCore: A ray-tracing hardware architecture for mobile devices," *ACM Trans. Graph.*, vol. 33, no. 5, pp. 1–15, Sep. 2014, Art. no. 162.
- [43] I. Quilez. (2018). *Hacking a Generic Ray-Intersector*. [Online]. Available: <https://iquilezles.org/articles/hackingintersector/>
- [44] J. Lee, W.-J. Lee, Y. Shin, S. Hwang, S. Ryu, and J. Kim, "Two-AABB traversal for mobile real-time ray tracing," in *Proc. SIGGRAPH Asia Mobile Graph. Interact. Appl.*, 2014, pp. 1–5, Art. no. 14.
- [45] Jamsoft. (2021). *Moving an Android Thread to a Specific CPU Core*. [Online]. Available: <https://jamsoft.tistory.com/225>
- [46] H. Inoue, "How SIMD width affects energy efficiency: A case study on sorting," in *Proc. IEEE Symp. Low-Power High-Speed Chips (COOL CHIPS XIX)*, Apr. 2016, pp. 1–3.
- [47] Khronos Group. (2020). *Vulkan Ray Tracing Extensions Specification*. [Online]. Available: <https://www.khronos.org/blog/ray-tracing-in-vulkan>

- [48] Microsoft. (2020). *DirectX Raytracing (DXR) Functional Spec.* [Online]. Available: <https://microsoft.github.io/DirectX-Specs/d3d/Raytracing.html>



HYEON-KI LEE received the B.S. and M.S. degrees in computer science from Sangmyung University, Seoul, South Korea, in 2023 and 2024, respectively, where he is currently pursuing the Ph.D. degree in computer science. His research interests include texture compression and global illumination.



WOO-CHAN PARK received the B.S., M.S., and Ph.D. degrees in computer science from Yonsei University, Seoul, South Korea, in 1993, 1995, and 2000, respectively. From 2001 to 2003, he was a Research Professor with Yonsei University. He is currently a Professor of computer engineering with Sejong University, Seoul. His current research interests include ray-tracing processor architecture, 3-D rendering processor architecture, real-time rendering, advanced computer architecture, computer arithmetic, lossless image compression hardware, and application-specific integrated circuit design.



HYEJU KIM received the B.S. and M.S. degrees in computer science from Sangmyung University, Seoul, South Korea, in 2023 and 2025, respectively. Her research interests include AI, image processing, super-resolution, and texture compression.



DONG-YUN KIM received the B.S. degree in computer science from Sangmyung University, Seoul, South Korea, in 2025. His research interests include computer vision and rendering.

JAE-HO NAH received the B.S., M.S., and Ph.D. degrees in computer science from Yonsei University, in 2005, 2007, and 2012, respectively. He is currently an Assistant Professor with Sangmyung University. Prior to joining academia, he was a Professional with LG Electronics. He has co-authored over 30 technical papers published in international journals and conferences, including *ACM Transactions on Graphics*, *IEEE TRANSACTIONS ON VISUALIZATION AND COMPUTER GRAPHICS*, *Computer Graphics Forum*, High-Performance Graphics, and others. His research interests include ray tracing, rendering algorithms, and graphics hardware.

