OctoMap-RT: Fast Probabilistic Volumetric Mapping Using Ray-Tracing GPUs

Heajung Min, Kyung Min Han, and Young J. Kim, *Senior Member, IEEE*

***Abstract*—A 3D occupancy map that is accurately modeled after real-world environments is essential for reliably performing robotic tasks. Probabilistic volumetric mapping (PVM) is a well-known environment mapping method using volumetric voxel grids that represent the probability of occupancy. The main bottleneck of current CPU-based PVM, such as OctoMap, is determining voxel grids with occupied and free states using ray-shooting. In this letter, we propose an octree-based PVM, called OctoMap-RT, using a hybrid of off-the-shelf ray-tracing GPUs and CPUs to substantially improve CPU-based PVM. OctoMap-RT employs massively parallel ray-shooting using GPUs to generate occupied and free voxel grids and to update their occupancy states in parallel, and it exploits CPUs to restructure the PVM using the updated voxels. Our experiments using various large-scale real-world benchmarking environments with dense and high-resolution sensor measurements demonstrate that OctoMap-RT builds maps up to 41.2 times faster than OctoMap and 9.3 times faster than the recent SuperRay CPU implementation. Moreover, OctoMap-RT constructs a map with 0.52% higher accuracy, in terms of the number of occupancy grids, than both OctoMap and SuperRay.**

***Index Terms*—Mapping, simulation and animation, hardware -software integration in robotics.**

I. Introduction

3D MAPPING reconstructs the spatial data acquired by a sensor in a virtual space similar to the real world, and critically influences the reliability and safety of robot deployment. There are various robotic applications of 3D mapping [1], such as 3D mapping for large-scale indoor spaces [2], real-time mapping for underground exploration [3], and end-to-end 3D online simultaneous localization and mapping (SLAM) [4]. For these applications, it is crucial that 3D mapping be executed robustly and quickly.

A popular strategy for building a volumetric map in 3D is to reconstruct the environment from a sensor-driven 3D point cloud [5], [6]. Various sensors, like RGB-D cameras and LiDAR, are used to accurately generate 3D point clouds by representing the environment with high-resolution geometric information. There are a number of different methods for building maps from a point cloud considering their information representations, such as grid-based maps [7], normal distributions transform maps [8], signed distance fields [5], and adaptive spatial subdivision using octree [9]. These maps are generated based on an occupancy estimation of the environment from noisy sensor measurements. The occupancy map represents the environment using the probabilistic occupancy of obstacles to classify the environment with *occupied* space (by obstacles), *free* space, and *unknown* space that has not yet been explored.

The octree-based probabilistic volumetric mapping (PVM) represents a space using a set of voxels. The 3D space is subdivided into voxels depending on the probabilistic occupancy [10], considering uncertainties such as sensor noise and dynamic obstacles. After the range or vision sensor scans the environment and obtains a point cloud, the sensor origin and each point in the point cloud define the virtual ray. The voxels that are intersected by the ray are found and identified as having an occupied or a free state. The octree is restructured using the found voxels along with the updated occupancy. In the CPU-based PVM using octree [6], the voxel determination step that identifies each voxel's occupancy status through ray-casting is the most time-consuming, particularly when the input point cloud is large, or the sensor range is long. Such cases limit the PVM to a coarse-level map. Furthermore, the slow voxel determination process may hinder the robots from conducting online tasks.

Graphics processing units (GPUs) are designed to perform highly parallel tasks with large datasets and minimal dependencies between data. Notably, dedicated ray-tracing GPUs, such as RTX GPUs, demonstrate fast performance for massively parallel ray shooting. In this study, we effectively exploit the ray-tracing GPUs to mitigate the performance bottlenecks of the CPU-based PVM based on ray-shooting while still utilizing the CPU to update the PVM.

Main Results: We propose OctoMap-RT, the octree-based CPU-GPU hybrid approach that uses dedicated ray-tracing graphics hardware, to improve the CPU-based PVM substantially. A core idea of this new hybrid approach is to effectively distribute the CPU and GPU workload depending on the characteristics of processing units and the mapping tasks involved. Specifically, we employ ray-tracing GPUs to generate voxels with occupancy states fast and in parallel, and we utilize the CPU to update the occupancy probability and maintain the octree. In order to accelerate the voxel determination using ray-shooting on the GPU, we first determine the compact ray space where the ray-shooting needs to be performed and then subdivide it with uniform voxel grids. We then represent the voxels using a small set of localized axis-aligned bounding

Manuscript received 30 March 2023; accepted 16 July 2023. Date of publication 31 July 2023; date of current version 4 August 2023. This letter was recommended for publication by Associate Editor P. Gao and Editor A. Bera upon evaluation of the reviewers' comments. The work of Kyung Min Han was supported by the NRF under Grant 2021R1I1A1A01048639. This work was supported in part by the ITRC/IITP Program under Grant IITP-2023-2020-0-01460 and in part by NRF under Grants 2021R1A4A1032582 and 2022R1A2B5B03001385 in South Korea. *(Corresponding author: Young J. Kim.)*

The authors are with the Department of Computer Science and Engineering, Ewha Womans University, Seoul 03760, South Korea (e-mail: hjmin@ewhain.net; hankm@ewha.ac.kr; kimy@ewha.ac.kr).

This letter has supplementary downloadable material available at https://doi.org/10.1109/LRA.2023.3300227, provided by the authors.

Digital Object Identifier 10.1109/LRA.2023.3300227

boxes (AABBs) that the GPU can process. With the AABBs, we build a bounding volume hierarchy (BVH) on the GPU for accelerated ray traversal and further optimize it using novel instanced and shared BVH schemes. Using massively parallel ray-shooting based on the GPU, we find voxels with occupied and free states, and we consistently update them by properly encoding their state information. After the voxels are read back from the GPU to the CPU, the voxel's occupancy probability and the octree are recursively updated in a bottom-up fashion from the leaf node toward the root node on the CPU. With our new voxel representations and generation techniques, we achieve performance improvements up to 41.2 times and 9.3 times over OctoMap and SuperRay CPU implementations in OctoMap-RT on real-world indoor and outdoor benchmarking environments. Furthermore, OctoMap-RT constructs a better PVM with 0.52% higher accuracy, in terms of the number of free occupancy grids, than both OctoMap and SuperRay owing to the GPU-based ray shooting. Finding free voxels more accurately induces more accurate PVM since a robot can access free space more reliably with accurate occupancy probability. Moreover, OctoMap-RT can generate voxels 5.3 times faster than our prior work [11].

The rest of this letter is organized as follows. In Section II, we summarize previous work that is relevant to ours and provide an overview of our approach in Section III. We describe our voxel and BVH representation in Section IV. We describe our voxel determination in Section V and explain how to update the occupancy map on the CPU in Section VI. In Section VII, we present our experiments and results, and we draw the conclusion in Section VIII.

II. Related Work

This section discusses previous work on PVM for map representation, GPU-based octree construction, and ray-tracing methods based on GPU.

A. *Pvm*

2D [10] and 3D [7] grids are popular representations for occupancy maps. However, uniform grids inevitably cause memory problems, which hinder large-scale and long-term operations. A well-known strategy for mitigating this problem is to employ an octree data structure [12]. The authors in [12] proposed occupancy probability based on octree to consider sensor noise. OctoMap [6] represented a volumetric occupancy map that labeled the grid as occupied, free, or unknown states. Grid-based sampling assumes that each grid's occupancy is independent of its neighboring grids. Although this assumption helps simplify the map, the resulting map suffers from inaccuracy due to the discrete nature of grids [13]. However, Gaussian process-based methods [13] consider a continuous spatial domain rather than discretized space. The normal distributions transform (NDT) [14] is another approach to discretizing a 3D volumetric space. In contrast to voxel-based sampling, a cell in the NDT contains multiple points to form a local Gaussian distribution. For this reason, NDT is considered a piece-wise continuous representation of a space where the number of grid cells is much smaller than that of grid maps. The NDT occupancy map (NDT-OM) [15] proposed to augment occupancy probability to NDT, followed by a real-time version [8] of NDT-OM. NDT-TM [16] extended NDT-OM using traversability mapping by computing the permeability of the rays within a cell, and [17] proposed the decay rate map using the ray length for out-of-range sensor measurements. Voxfield [5] adopted truncated signed distance fields (TSDFs) to find the nearest surface using ray casting. Particle-based mapping represents the occupancy and the dynamic state of a grid map using a number of particles or particle weights [18], [19]. Multi-view fusion (MVF) [20] proposed dynamic voxelization to circumvent memory problems by dynamically changing the voxel positions and buffer sizes without loss of point information.

There have been several research efforts to speed up PVM and improve its performance by reducing ray-casting time. UFOMap [21] adjusts the depth level of the octree during the ray-casting stage in an adaptive manner, leading to a more efficient marching of rays on free voxels that are nonadjacent to the occupied points. SuperRay [22] reduces the number of ray traversals by defining a representative ray for a group of adjacent rays traversing the same voxels. Moreover, this method culls the unnecessary traversed regions based on the occupancy state without compromising the mapping accuracy. The occupancy homogenous mapping (OHM) [7] proposes GPU-based implementations using ray splitting and rasterization to improve occupancy grid maps, NDT algorithms, and TSDF. However, OHM employs a uniform grid as the underlying voxel representation rather than an adaptive grid, unlike the approaches mentioned above. Another GPU-based method is NanoMap [23], which employs an octree-based voxel representation. However, NanoMap does not consider the consistent occupancy update employed by OctoMap-RT, OctoMap, and SuperRay, and limits the occupancy probability to a narrow range. As a result, NanoMap builds a different occupancy map than OctoMap and SuperRay. Therefore, in Section VII, we compare the performance of OctoMap-RT to two baseline methods, OctoMap and SuperRay, based on octree-based voxel representations. Our prior work [11] improved voxel generation performance using a ray-tracing GPU over a state-of-the-art OctoMap CPU implementation but was not fully integrated into the map-building pipeline. Compared with [11], we propose a more efficient generation scheme using novel voxel and BVH representations, which are better suited for large-scale scan data. Further, we introduce fast, consistent voxel determination during ray-shooting and integrate a full map-building pipeline using a novel CPU-GPU hybrid approach.

B. *GPU-Based Octree Construction*

The octree structure is predominantly used in the computer graphic domain for various tasks, including distance field generation, rendering, modeling, simulation, and model reconstruction [6], [24]. [25] proposed a GPU-based octree construction for reconstructing surfaces on the GPU. For a large-scale volumetric scene, a full or out-of-core style octree update to GPU was studied [26]. Octree can be adjusted dynamically in real-time in the GPU, as well as one-time construction or full reconstruction [27]. [28] studied streaming subtree data through CPU-GPU data transfer in a view-dependent manner. Recently, [25] supported dynamic topological updates on GPU. During ray traversals, in order to reduce the cost of neighbors searching on the octree, [29] used three pre-computed neighbors per cell to enable stackless ray casting and dynamic updating of the octree on a GPU. A sparse voxel octree (SVO) showed both high-quality rendering and efficient ray traversal of shallow tree topology for a static scene [26]. OpenVDB [26] used SVO data structures and was implemented on GPU [25], which enabled efficient neighbor access using GPU-based ray casting for dynamic scenes.

C. GPU-Based Ray Tracing

Ray tracing is a graphical technique that enables the rendering of a photo-realistic scene. From a given viewpoint, many rays are fired toward screen pixels until they hit objects in the scene, and each ray path is traced back to determine the pixel color of the screen [30]. GPU-based ray tracing has been studied to accelerate each stage of the ray-tracing pipeline using various acceleration data structures, such as kd-tree [31] or BVH [32], traversing acceleration structures like BVH in parallel [33], and ray-triangle intersections [33]. Recently, a dedicated GPU platform for accelerating ray tracing was introduced [34]. Moreover, a ray-tracing GPU can be used as a general-purpose GPU (GPGPU) for computing non-rendering tasks, such as sampling for simulation, Monte Carlo particle transport, or simulation for sound propagation in water [35]. Our work also utilizes ray-tracing GPUs as a GPGPU.

III. Preliminaries and Overview

In this section, we identify the main bottleneck of CPU-based PVM, such as OctoMap, and briefly explain graphics hardware accelerated ray-shooting. We then provide an overview of our approach.

A. CPU-Based PVM Using Octree

A sensor scans and generates a 3D point cloud corresponding to the obstacle surface in the environment. This scanning process is continuously repeated by changing the sensor location until the mapping is completed. OctoMap [6] and its variants like SuperRay [22] discretize the mapped area using the octree voxels. Each voxel indicates the occupancy state probabilistically. OctoMap consists of the following steps to build a PVM:

1) *Ray-shooting:* The voxel that includes each point of the 3D point cloud corresponds to an *occupied* voxel. Ray shooting to find intersected voxels is performed in the 3D uniform grid of voxels where a ray traverses from the sensor origin to the point cloud. All intersected voxels correspond to *free* voxels.
2) *Consistent occupancy update:* As multiple rays can intersect the same voxel, the voxel's occupancy information should be decided consistently from the multiple rays.
3) *Octree update:* The voxel occupancy probability and the octree are updated using intersected voxels.

In our experiments, we observe that the processing time to determine the ray-intersected voxels for a consistent occupancy update (*i.e.,* the first and second steps of the above process) takes up about 90% of the map building. To mitigate this bottleneck, our approach uses a ray-voxel intersection and consistent voxel determination accelerated by the GPU.

B. Graphics Hardware Accelerated Ray-Shooting

A ray-tracing GPU, such as RTX, is dedicated hardware for accelerating computationally-intensive ray tracing for photo-realistic image synthesis. Our mapping problem uses the ray-tracing GPU as a GPGPU instead of rendering the scene. The RTX GPUs support parallel BVH construction/traversal and ray-triangle or ray-AABB intersection tests. In order to leverage the ray-tracing GPU, an application programming interface (API) such as DirectX's DXR has also been introduced [34]. Given the scene geometry, a BVH is built, and various programmable shaders are launched using the BVH during ray tracing:

1) *The ray generation shader* generates rays by specifying the ray origins, directions, and parametric interval in which intersections occur along the interval.
2) During BVH traversal, whenever the ray intersects the leaf nodes of the BVH, *the intersection shader* is called, which performs the ray-geometry intersection test to find the closest hit. If the closest hit is found, it is reported; otherwise, the hit is ignored to find the next hit.
3) When the entire traversal finishes, if the closest hit is reported in the intersection shader, *the closest hit shader* is called. The closest hit shader calculates shading data at the hit point and launches the additional ray.
4) If the ray does not intersect any geometry and the closest hit is absent, *the miss shader* is executed.

C. Overview of Our Approach

With large point cloud data of high resolution, voxel determination becomes the bottleneck of a CPU-based PVM, such as OctoMap. We focus on leveraging ray-tracing GPUs that support massively parallel ray shooting to speed up CPU-based PVM. Moreover, we distribute the workload of the CPU and GPU according to the characteristics of each device and required tasks. Specifically, a ray-tracing GPU is used for intensive and regular streaming tasks, such as the BVH construction, ray-voxel intersection, and occupancy update. In contrast, the CPU handles irregular workflows, such as updating the voxel's occupancy probability and octree. As shown in Fig. 1, OctoMap-RT consists of the following steps to build a PVM, and each step respectively corresponds to a blue-boxed enumeration in the figure:

1) *Voxel representation (CPU):* We estimate the *local* extent of sensor measurements to set up the *shared ray space* for ray-shooting in order to minimize the size of BVH for voxels. We subdivide the shared ray space with uniform voxel grids, map the voxels to AABBs, and build a *shared BVH* of AABBs.
2) *Ray-shooting (GPU):*
    a) *BVH instancing:* When we construct a BVH for a dense set of AABBs of uniform size, we build a BVH of a subset of the AABBs and instance it to multiple copies of BVHs to speed up the construction of the full BVH and reduce GPU memory consumption.
    b) *Voxel intersection:* We launch rays in a massively-parallel fashion to find intersected AABBs. AABBs containing ray endpoints correspond to *occupied* voxels. However, AABBs intersected with the rays correspond to *free* voxels.
3) *Consistent occupancy update (GPU):* When a voxel contains ray endpoints and is intersected by other rays, its state can be classified as both occupied and free due to parallel processing. We must ensure that such a voxel is consistently classified as occupied.
4) *Voxel readback (GPU $\rightarrow$ CPU):* All voxels with consistent occupancy information are read from the GPU back to the CPU.

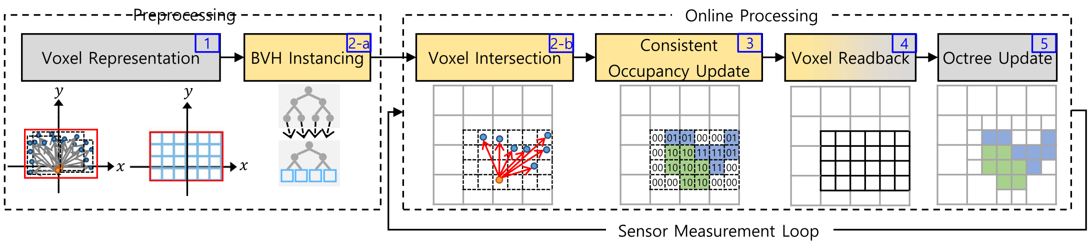

Fig. 1. Pipeline of OctoMap-RT consists of intertwined CPU-based serial tasks (gray box) and GPU-based parallel tasks (yellow box). Steps 1 to 2a correspond to preprocessing and run only once, but steps 2b to 5 are repeated online as new sensor data are measured. In step 3, unknown, free, and occupied voxels are colored white, green, and blue, respectively, which are encoded as 00 (unknown), 01 (occupied), 10 (free), and 11 (free or occupied).

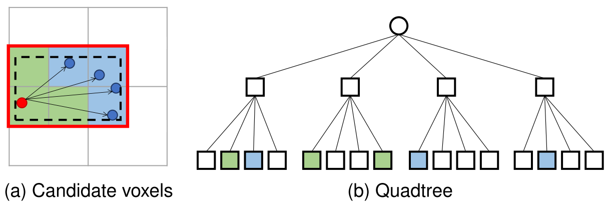

Fig. 2. Localized voxel mapping. (a) The black-dashed box bounds the sensor origin (red circle) and point cloud (blue circles). The colored cells are the candidate voxels for the ray-voxel intersection (green for free and blue for occupied voxels) and define the ray space (red box) for ray shooting. (b) The quadtree is updated using the colored cells as leaf nodes.

5) *Octree update (CPU):* The read voxels are used to update the occupancy probability and octree.

The voxel representation 1) and BVH instancing 2a) steps in the pipeline are preprocessed, and the rest are repeated online as new sensor data is fed into the update loop. In the remaining sections, we provide detailed explanations of each step.

IV. Voxel and BVH Representations

This section introduces how to specify the 3D uniform grid space filled with voxels where ray-shooting is performed, and how to build it into a BVH structure that the ray-tracing GPU can process.

A. Localized Voxel Mapping

Since the ray-tracing GPU can shoot rays against such geometric primitives as triangles or AABBs [36], the voxels in the octree are converted to AABBs, as an AABB accurately represents a voxel geometry; we can define the AABB's geometry using the voxel's position and size. However, we do not convert all voxel nodes in the octree into AABBs for ray shooting; instead, we convert only the leaf-level voxels that may be intersected by the rays based on the current sensor position. These leaf-level voxels are used for probabilistic state updates in the octree in a bottom-up fashion.

Fig. 2 illustrates how to map voxels to AABBs on a GPU using a quadtree in 2D, instead of an octree, for the sake of simplicity of illustration. As illustrated in Fig. 2(a), we find the bounding volume (black-dashed box) that encloses both the sensor origin in red and the point cloud in blue. Then, we determine a minimal number of leaf-level voxels that contain the bounding volume. These voxels are the candidate voxels that are potentially intersected with rays, and they define the *ray space* (red box). Fig. 2(b) illustrates the corresponding voxel nodes in the quadtree. We convert the candidate voxels to AABBs and upload them to the GPU. Then, the GPU builds a BVH for ray traversal.

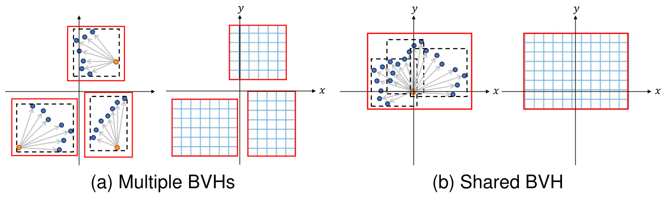

Fig. 3. Shared BVH. (a) Multiple ray spaces and separate BVH for each ray space. (b) Shared ray space and a single shared BVH.

B. Shared BVH

As described in Section IV-A, we define the ray space where the ray-shooting is performed; whenever the sensor changes its location, so does the ray space. For instance, as illustrated in Fig. 3(a), as the sensor scans at the three different locations, three independent ray spaces (red rectangles) can be defined, and three BVHs of AABBs (blue grids) need to be constructed. However, this strategy is inefficient, as the number of different sensor locations can be arbitrarily high, which makes both the number of AABBs and their BVH construction time high. Instead, as shown in Fig. 3(b), we define the *shared ray space*. The shared ray space can be pre-computed depending on the extent of the maximum sensor range and frequency. The extent of the space can be more tightly determined if the sensor locations are known beforehand: for example, if the trajectory of sensing is pre-planned. In this case, we can pre-compute AABBs and their BVH for the shared ray space, cache the BVH on the GPU, and reuse the same BVH for the varying sensor locations. This shared BVH strategy can considerably reduce the memory usage for AABBs and BVH construction time on the GPU.

C. BVH Instancing

A large scanning environment or high-resolution sensor can induce a large number of voxel (or AABB) candidates. This can significantly impact performance in BVH building, as well as memory consumption by the GPU. In order to alleviate this issue, we propose a BVH-instancing approach to build the BVH of AABBs efficiently in terms of both computation time and memory usage. The main idea is based on the observation that our candidate voxels have a uniform size of AABBs, and a small number (typically $2^3 \sim 2^9$) of AABBs can be instanced to a large number of AABBs.

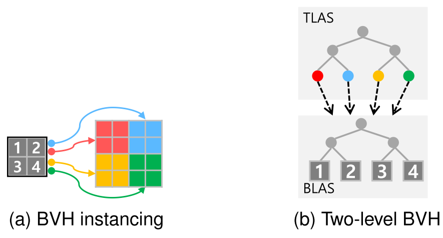

Fig. 4. Two-level BVH instancing. (a) A group of four gray AABBs is instanced into four copies of red, blue, yellow, and green AABBs with different transformations. (b) Two-level BVH of the sixteen AABBs. TLAS has four instances of BVH as leaf nodes (red, blue, yellow, and green nodes), and each of them points at the BLAS with a different transformation.

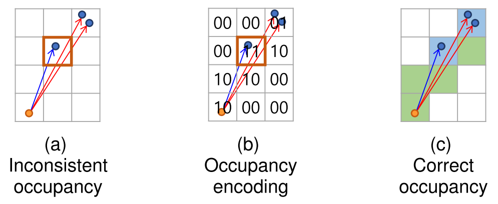

Fig. 5. Consistent voxel state. (a) In the orange voxel, three-ray intersections occur. The blue ray intersection labels the voxel as occupied because the ray endpoint is included in the voxel, whereas the red ray intersections label the voxel as free. (b) The voxel has two bits to encode the 00 (unknown), 01 (occupied), 10 (free), and 11 (free or occupied) states. (c) When there is an occupancy conflict, the occupied state (01) takes precedence over the free state (10). Thus, the orange voxel in (b) is colored blue (occupied) in (c) instead of green (free).

We use a two-level BVH to realize BVH instancing; the bottom-level acceleration structure (BLAS) is a BVH of AABBs, and the top-level acceleration structure (TLAS) has a BVH of instanced BVHs (*i.e.,* pointers to the BLAS) with transformations. For example, as illustrated in Fig. 4, we group 2×2 adjacent AABBs into one BLAS. The entire set of AABBs is instanced from the BLAS with proper transformations; for example, the four BVHs of red, blue, yellow, and green AABBs are instanced from the gray BVH in Fig. 4.

V. Voxel Determination

In this section, we describe an approach to fast voxel determination using a ray-tracing GPU, and we determine the state of the voxels.

A. Voxel Intersection

The workspace defining the ray space is discretized into uniform voxel grids, each mapped to an AABB. Then, the BVH of the AABBs is built on the GPU, and ray-shooting is performed to determine the intersected AABBs, or equivalently the voxels, using various programmable shaders dedicated to GPU-based ray tracing. In the ray generation shader, rays are massively cast in parallel from the sensor origin to each point in the scanned point cloud. The voxels intersected by the rays are determined using parallel BVH traversal and ray-AABB intersection tests on the GPU. Whenever a ray intersects an AABB, the intersection shader is called to determine the intersected AABB. We ignore the closest hit shader, as we do not calculate shading information. We label the state of the intersected AABBs containing the ray endpoints as occupied and the rest of intersected AABBs as free.

B. Consistent Voxel Occupancy State

As illustrated in Fig. 5(a), during the ray-AABB intersection test, adjacent rays may intersect the same voxel, and each intersection can yield a different state classification for the same voxel. This inconsistent voxel classification jeopardizes the reliability of the built occupancy map. For instance, the inconsistent classification may create a non-existing void in the obstacle space. This phenomenon is apparent for the voxels close to the sensor origin or when the range sensor scans at a shallow angle. In [6], this problem was handled by sequentially comparing and removing voxels on the CPU, which causes a bottleneck in the map updates, as described in the second step in Section III-A.

We address this inconsistent classification during our GPU-based parallel voxel determination, as illustrated in Fig. 5. The main idea is that when a voxel is classified as both occupied and free by different rays, the occupied state takes precedence over the free state. Specifically, we use two bits to encode the voxel state information, each representing occupied (least significant bit) or free (most significant bit) states. In other words, a voxel can have four state labels 00 (unknown), 01 (occupied), 10 (free), and 11 (free or occupied). Each ray-voxel intersection modifies only one of the two bits. When the state information of each voxel is read from the GPU back to the CPU, an overlapped occupancy state of 11 is interpreted to the occupied state.

VI. Occupancy Map Update

Because the CPU is designed to work well for random data access and handling irregular workflows, we update the occupancy probability of the voxel and octree on the CPU. After reading the voxels back from the GPU, we update the occupancy map on the CPU as follows. For each voxel with an occupancy label, if the most significant bit of the label is one, the voxel is updated as occupied to prioritize the occupied state over the free state. Otherwise, the voxel is determined to be free and is updated when the least significant bit of the label is one. We also update the occupancy probability based on the voxel's prior probability and the probability from the current measurement using [10]. Using these updated voxels as leaf-level nodes, the entire octree nodes are recursively updated in a bottom-up fashion [6]. To safely implement the CPU/GPU hybrid mechanism in OctoMap-RT, it is crucial to enforce synchronized readback between the GPU and CPU. Specifically, the octree update on the CPU should start only after the reading of the voxel occupancy data from the GPU is finished. We use the *fence* operation available in the modern graphics API [37] to make the CPU cycle wait for the GPU cycle to finish sending the voxel data. This occupancy map update process is repeated whenever new voxel data are generated.

VII. Experiments and Results

In this section, we provide our experimental setup, various results of the experiments, and discussions of our approach. All the experiments were conducted on Intel's i9-13900 K CPU with 64 GB of RAM and NVIDIA's RTX 4090 GPU. We used DirectX's DXR as a ray-tracing API with the Microsoft Visual Studio 2017 C++ programming language under 64-bit Windows 10.

To evaluate OctoMap-RT, we used both real (Fig. 7) and virtual environment (Fig. 8) datasets. As shown in Fig. 7 and Table I,

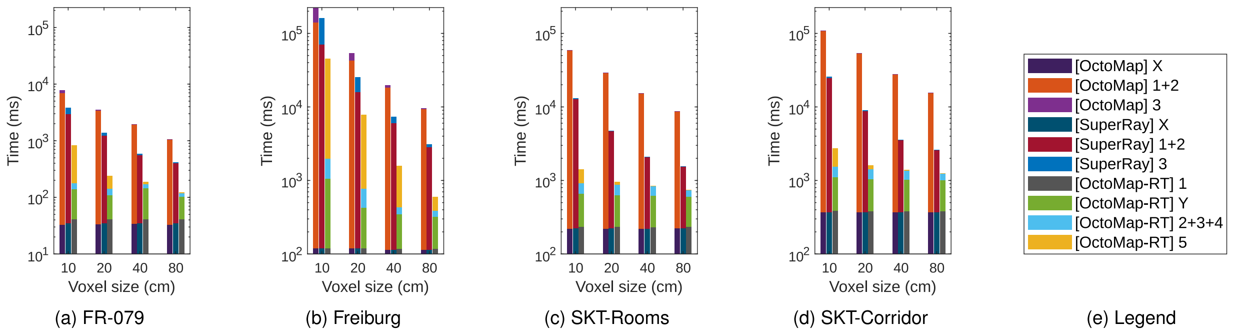

Fig. 6. Performance breakdown of OctoMap-RT and its comparisons against OctoMap and SuperRay in log scale with various voxel dimensions using the datasets in Fig. 7. For OctoMap and SuperRay, the labels in the stacked bars represent the times for *ray data preparation* (X), *ray-shooting* (1), *consistent occupancy update* (2), and *octree update* (3), respectively. For OctoMap-RT, the labels in the stacked bars represent the times for *voxel representation* (1, CPU), *voxel/ray data upload* (Y, from CPU to GPU), *voxel intersection* (2, GPU), *consistent occupancy update* (3, GPU), *voxel readback* (4, from GPU to CPU), and *octree update* (5, CPU), respectively. The labels also coincide with the steps in Sections III-A and III-C.

TABLE I
DATASET STATISTICS FOR FIG. 7

Dataset	FR-079 corridor	Freiburg campus	Ewha-SKT B/D rooms	Ewha-SKT B/D corridor
Dimension of the scene (m^3)	46.48×36.45×4.55	292.35×167.05×27.65	16.53×24.11×3.14	45.52×33.31×4.55
Dimension of the shared ray space (m^3)	20.09×17.95×3.22	60.41×60.30×21.59	16.13×18.22×3.05	19.41×18.59×4.49
Average # of points (rays) per scan (K)	89	247	270	270
# of scans	66	81	172	305

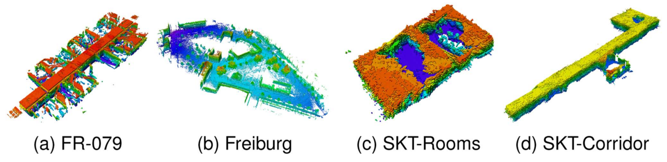

Fig. 7. PVM results of OctoMap-RT. The voxels are color-coded depending on the vertical height from the floor, and the dimension of each voxel is $10\,\mathrm{cm}^3$.

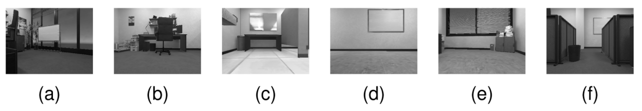

Fig. 8. Different sensor positions inside a complex virtual building. The average number of rays per frame is 76 K.

we used four real environment datasets; the FR-079 corridor and Freiburg campus datasets, based on Lidar sensors, were obtained from [6], and Ewha-SKT B/D rooms and Ewha-SKT B/D corridor were acquired using a PrimeSense Carmine 1.09 RGB-D sensor attached to a Fetch mobile robot. The physical dimension of leaf-level voxels in PVM ranged from 10 cm to 80 cm, reflecting their spatial extent in the real-world environment. Fig. 8 is a virtual office environment dataset obtained by a stereo camera attached to a mobile robot [11]. The point cloud scanned from all datasets is built into an octree with a maximum depth of 16.

A. Overall Performance and Comparisons

We measured the map-building performance of OctoMap-RT system using the dataset shown in Fig. 7 and compared it against two octree-based PVM approaches, OctoMap [6] and SuperRay [22], as shown in Fig. 6.

Overall improvements: The average performance in building PVMs using OctoMap-RT compared with OctoMap improved by a factor of 10.7, 10.1, 25.4, and 26.2, respectively, as shown in Fig. 6(a)–(d). OctoMap-RT is also 4.2, 4.1, 4.7, and 4.9 times faster than SuperRay in Fig. 6(a)–(d). The performance improvement depends on the number of rays (or points). Specifically, the more points are scanned, the more voxels need to be determined, and thus the greater the performance difference between OctoMap-RT and others becomes. This is evident from Table I and Fig. 6. The performance improvements shown in Fig. 6(c) and (d) are greater than that of Fig. 6(a) because the number of points (or rays) is higher. Meanwhile, the ray length also impacts the map-building performance of OctoMap-RT as it will determine the size of the shared ray space, as discussed in Section IV-B; *i.e.,* long rays create a large ray space. For example, the number of points in Fig. 6(b) is three times greater than that in Fig. 6(a), but the performance improvement of shown in Fig. 6(b) is slightly less than that in Fig. 6(a), as Fig. 6(b) contains many points far from a sensor, which creates long rays. Therefore, we anticipate that OctoMap-RT performs well when a high-resolution sensor scans a dense environment that does not contain many long passages.

Performance breakdown: Fig. 6 shows that the main bottlenecks in CPU-based PVM, such as OctoMap and SuperRay, are ray-shooting and the consistent occupancy update, as shown in the "[OctoMap] 1+2" dark orange bar and "[SuperRay] 1+2" dark red bar in Fig. 6. This step corresponds to the very small "[OctoMap-RT] 2+3+4" cyan bar in Fig. 6, which is significantly reduced in comparison with OctoMap and SuperRay. The ray data preparation time "X" indicates the duration required for transforming point clouds from the sensor coordinate system to the world coordinate system.

Voxel size factor: Fig. 6 also shows performance improvements within the same dataset depending on the voxel size while the number of scanned points is fixed; within the same dataset, the smaller the voxel size, the more voxels need to

TABLE II
COMPARISONS OF VOXEL GENERATION PERFORMANCE USING FIG. 8

	Sensor Positions	(a)	(b)	(c)	(d)	(e)	(f)
Ours	# of Voxels (10^3)	185	250	172	229	392	313
	BVH Build (ms)	0.38	0.42	0.38	0.44	0.59	0.51
	Intersection (ms)	0.27	0.31	0.29	0.33	0.35	0.28
[11]	# of Voxels (10^3)	34	35	22	35	32	33
	BVH Build (ms)	0.66	0.59	0.54	0.67	0.64	0.62
	Intersection (ms)	1.62	1.79	1.57	1.54	1.48	1.71

be subdivided for the same ray space, and thus the greater the performance difference becomes between CPU-based PVM and OctoMap-RT. For example, in Fig. 6(c), when the voxel size is 10 cm and 80 cm, OctoMap-RT is 41.2 times and 11.6 times faster than OctoMap, respectively, and 9.3 times and 2.1 times faster than SuperRay, respectively. Therefore, we expect OctoMap-RT to perform better than CPU-based PVM when a higher voxel resolution is needed; for example, a small robot is used to build a map or a narrow passage in the environment.

B. Performance Comparison of Voxel Generation

Min et al. [11] proposed GPU-based voxel generation techniques for OctoMap, which is the main performance bottleneck in OctoMap. We compare the performance of our OctoMap-RT against that of Min et al. [11] in terms of voxel generation using the same benchmark (Fig. 8) provided in [11], where the voxel generation performance was measured for six different sensor positions.

As shown in Table II, the voxel generation performance in OctoMap-RT is 5.3 times faster than in Min et al. [11] on average, even though the number of voxel grids needed for the ray-shooting is 7.6 times higher in OctoMap-RT than in [11]; a higher number of voxel grids in OctoMap-RT is needed since we need to voxelize the shared ray space at the leaf level, whereas [11] does so at different levels, *i.e.,* voxels with different resolutions. Even so, building BVH in OctoMap-RT is 1.4 times faster than in [11] thanks to the BVH instancing described in Section IV-C. Moreover, the voxel intersection in OctoMap-RT still runs 5.3 times faster than in [11] since [11] must refine the voxel to the leaf level after finding the intersection.

C. Comparisons of Voxels With Occupancy States

It is interesting to note that the numbers of determined voxels in OctoMap-RT and OctoMap are different due to the different ray shooting mechanisms; OctoMap-RT uses geometric ray shooting, whereas OctoMap uses a volume rasterization technique [38]. Because of this difference, OctoMap may miss some voxels when the ray intersects their boundaries, while OctoMap-RT reports all the intersected voxels. For example, as illustrated in Fig. 9(a) and (b) in 2D, for a single sensor scan, OctoMap-RT and OctoMap find the same number of occupied voxels in blue that include the ray endpoints. However, OctoMap-RT finds more free voxels in green than OctoMap.

Moreover, with more sensor scans, OctoMap-RT may find more free voxels but fewer occupied voxels than OctoMap. This is because as OctoMap-RT finds more free voxels, it reduces the accumulated occupancy probability for the previously *believed-to-be*-occupied voxels. This situation is illustrated in Fig. 9(c) and (d). The thickly bordered voxels in Fig. 9(c) are occupied because they contain the endpoints of the rays fired from position

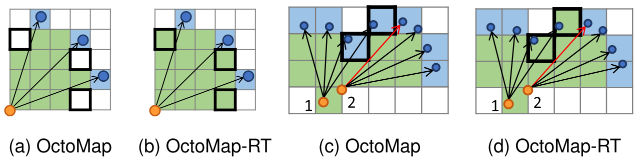

Fig. 9. Differences in the number of intersected voxel grids and their occupancy states in OctoMap and OctoMap-RT. (a), (b) Single sensor scan. (c), (d) Sensor scans from position one followed by position two.

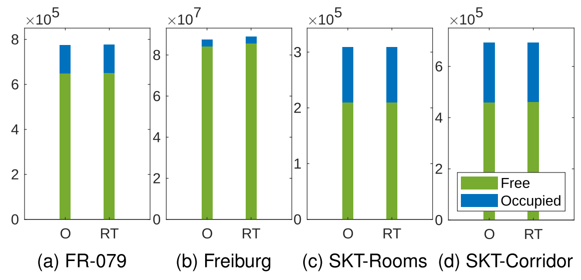

Fig. 10. Number of free/occupied voxels generated using OctoMap (O) vs. OctoMap-RT (RT) using the benchmarks in Fig. 7. (a) 649 K/126 K vs. 651 K/125 K. (b) 84 M/3.2 M vs. 85 M/3.1 M. (c) 209 K/99.3 K vs. 210 K/99.2 K. (d) 460 K/232.5 K vs. 461 K/232.1 K.

TABLE III
COMPARISONS OF THE GPU MEMORY USAGE IN SHARED BVH

Voxel size (cm)		10	20	40	80
Shared BVH size (MB)	FR-079	54.54	7.37	1.04	0.17
	Freiburg	7.21	0.91	0.29	0.19
	SKT-Rooms	42.13	5.92	0.84	0.14
	SKT-Corridor	77.17	9.63	1.41	0.25

one, but the red ray fired from position two does not intersect them. However, the same voxels in Fig. 9(d) are switched from occupied to free because the red ray intersects them, reflected in the accumulated occupancy probability.

Consequently, OctoMap-RT builds the PVM with more voxels than are used in OctoMap, particularly with more free voxels. Fig. 10 supports this finding using the datasets in Fig. 7 with a voxel size of 10 cm. The leaf-level voxels consist of occupied and free voxels. Compared with OctoMap and SuperRay, on average, OctoMap-RT has 0.52% more leaf-level voxels in total, 0.59% fewer occupied voxels, and 0.65% more free voxels. As OctoMap-RT maps the free space more accurately than does OctoMap, robotic navigation tasks that are more delicate, such as navigating a narrow passage, are achievable based on the mapping.

D. GPU Memory Usage in Shared BVH

We conducted measurements of the GPU memory usage for shared BVH with different voxel sizes as shown in Table III using the dataset of Fig. 7. As the voxel size decreases in each environment, the number of AABBs filling the shared ray space increases, resulting in a proportional increase in the size of the shared BVH memory. Despite such an increase, BVH instancing has effectively mitigated memory consumption for the large-scale outdoor environment. For instance, when the voxel size is 10 cm, even though the shared ray space of Freiburg is 87.5 times larger than that of SKT-Rooms, as shown in Table I, the shared BVH's memory of the former is 5.8 times even smaller than that of the latter.

VIII. Conclusion

In this letter, we propose OctoMap-RT, an octree-based PVM using a CPU-GPU hybrid approach based on dedicated ray-tracing graphics hardware. Our experiments show that OctoMap-RT builds PVM significantly faster than do OctoMap and SuperRay, and it constructs a map with higher accuracy. There are a few limitations to our current work. Because our system is a CPU-GPU hybrid, it suffers from data traffic overhead between two processing units, which could be resolved by migrating the CPU task (*i.e.,* octree maintenance) to the GPU using GPU-based octree construction techniques, as discussed in Section II-B. Another implementation issue of our current system is that it relies on the DXR API for GPU-based ray tracing, which is currently available only under the Windows OS. However, it is still possible to reproduce our system using vendor-independent ray tracing APIs such as OptiX [39]. As per another future work, we plan to release our codes and are interested in extending our system to manage a large point cloud on a city scale. We also would like to apply our GPU-based ray tracing technique to diverse types of PVMs, including NDT, TSDF, and decay rate maps. Further improvements using GPU-based octree construction to the OctoMap-RT are to be explored in future work.

References

[1] C. Cadena et al., "Past, present, and future of simultaneous localization and mapping: Toward the robust-perception age," *IEEE Trans. Robot.*, vol. 32, no. 6, pp. 1309–1332, Dec. 2016.
[2] D. Lee et al., "Large-scale localization datasets in crowded indoor spaces," in *Proc. IEEE/CVF Comput. Vis. Pattern Recognit.*, 2021, pp. 3227–3236.
[3] H. Azpúrua, M. F. M. Campos, and D. G. Macharet, "Three-dimensional terrain aware autonomous exploration for subterranean and confined spaces," in *Proc. IEEE Int. Conf. Robot. Automat.*, 2021, pp. 2443–2449.
[4] D. Droeschel and S. Behnke, "Efficient continuous-time SLAM for 3D LiDAR-based online mapping," in *Proc. IEEE Int. Conf. Robot. Automat.*, 2018, pp. 5000–5007.
[5] Y. Pan, Y. Kompis, L. Bartolomei, R. Mascaro, C. Stachniss, and M. Chli, "Voxfield: Non-projective signed distance fields for online planning and 3 D reconstruction," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst.*, 2022, pp. 5331–5338.
[6] A. Hornung, K. M. Wurm, M. Bennewitz, C. Stachniss, and W. Burgard, "OctoMap: An efficient probabilistic 3D mapping framework based on octrees," *Auton. Robots*, vol. 34, no. 3, pp. 189–206, 2013.
[7] K. Stepanas, J. Williams, E. Hernández, F. Ruetz, and T. Hines, "OHM: GPU based occupancy map generation," *IEEE Robot. Autom. Lett.*, vol. 7, no. 4, pp. 11078–11085, Oct. 2022.
[8] C. Schulz, R. Hanten, and A. Zell, "Efficient map representations for multi-dimensional normal distributions transforms," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst.*, 2018, pp. 2679–2686.
[9] C. Yuan, W. Xu, X. Liu, X. Hong, and F. Zhang, "Efficient and probabilistic adaptive Voxel mapping for accurate online LiDAR odometry," *IEEE Robot. Autom. Lett.*, vol. 7, no. 3, pp. 8518–8525, Jul. 2022.
[10] H. Moravec and A. Elfes, "High resolution maps from wide angle sonar," in *Proc. IEEE Int. Conf. Robot. Automat.*, 1985, pp. 116–121.
[11] H. Min, K. M. Han, and Y. J. Kim, "Accelerating probabilistic volumetric mapping using ray-tracing graphics hardware," in *Proc. IEEE Int. Conf. Robot. Automat.*, 2021, pp. 5440–5445.
[12] P. Payeur, P. Hébert, D. Laurendeau, and C. M. Gosselin, "Probabilistic octree modeling of a 3D dynamic environment," in *Proc. IEEE Int. Conf. Robot. Automat.*, 1997, pp. 1289–1296.
[13] S. T. O'Callaghan and F. T. Ramos, "Gaussian process occupancy maps," *Int. J. Robot. Res.*, vol. 31, no. 1, pp. 42–62, 2012.
[14] T. Stoyanov, M. Magnusson, H. Andreasson, and A. Lilienthal, "Fast and accurate scan registration through minimization of the distance between compact 3D NDT representations," *Int. J. Robot. Res.*, vol. 31, pp. 1377–1393, 2012.
[15] J. Saarinen, H. Andreasson, T. Stoyanov, J. Ala-Luhtala, and A. J. Lilienthal, "Normal distributions transform occupancy maps: Application to large-scale online 3D mapping," in *Proc. IEEE Int. Conf. Robot. Automat.*, 2013, pp. 2233–2238.
[16] J. Ahtiainen, T. Stoyanov, and J. Saarinen, "Normal distributions transform traversability maps: LiDAR-only approach for traversability mapping in outdoor environments," *J. Field Robot.*, vol. 34, no. 3, pp. 600–621, 2017.
[17] A. Schaefer, L. Luft, and W. Burgard, "An analytical LiDAR sensor model based on ray path information," *IEEE Robot. Autom. Lett*, vol. 2, no. 3, pp. 1405–1412, Jul. 2017.
[18] D. Nuss et al., "A random finite set approach for dynamic occupancy grid maps with real-time application," *Int. J. Robot. Res.*, vol. 37, no. 8, pp. 841–866, 2018.
[19] G. Chen, W. Dong, P. Peng, J. Alonso-Mora, and X. Zhu, "Continuous occupancy mapping in dynamic environments using particles, 2022, *arXiv:2202.06273*.
[20] Y. Zhou et al., "End-to-end multi-view fusion for 3D object detection in LiDAR point clouds," in *Proc. Conf. Robot Learn.*, 2020, pp. 923–932.
[21] D. Duberg and P. Jensfelt, "UFOmap: An efficient probabilistic 3D mapping framework that embraces the unknown," *IEEE Robot. Autom. Lett.*, vol. 5, no. 4, pp. 6411–6418, Oct. 2020.
[22] Y. Kwon, D. Kim, I. An, and S.-e. Yoon, "Super rays and culling region for real-time updates on grid-based occupancy maps," *IEEE Trans. Robot.*, vol. 35, no. 2, pp. 482–497, Apr. 2019.
[23] V. Walker, F. Vanegas, and F. Gonzalez, "NanoMap: A GPU-accelerated openVDB-based mapping and simulation package for robotic agents," *Remote Sens.*, vol. 14, no. 21, 2022, Art. no. 5463.
[24] F. Liu and Y. J. Kim, "Exact and adaptive signed distance fields computation for rigid and deformable models on GPUs," *IEEE Trans. Vis. Comput. Graph.*, vol. 20, no. 5, pp. 714–725, May 2014.
[25] R. K. Hoetzlein, "GVDB: Raytracing sparse Voxel database structures on the GPU," in *Proc. High Perform. Graph.*, 2016, pp. 109–117.
[26] K. Museth, "VDB: High-resolution sparse volumes with dynamic topology," *ACM Trans. Graph.*, vol. 32, no. 3, pp. 1–22, 2013.
[27] C. Crassin, F. Neyret, M. Sainz, S. Green, and E. Eisemann, "Interactive indirect illumination using Voxel cone tracing," in *Computer Graphics Forum*, vol. 30. Hoboken, NJ, USA: Wiley, 2011, pp. 1921–1930.
[28] E. Gobbetti, F. Marton, and J. A. I. Guitián, "A single-pass GPU ray casting framework for interactive out-of-core rendering of massive volumetric datasets," *Vis. Comput.*, vol. 24, no. 7/9, pp. 797–806, 2008.
[29] Y. Kim, B. Kim, and Y. J. Kim, "Dynamic deep octree for high-resolution volumetric painting in virtual reality," in *Computer Graphics Forum*, vol. 37. Hoboken, NJ, USA: Wiley, 2018, pp. 179–190.
[30] P. Shirley and R. K. Morley, *Realistic Ray Tracing*. Natick, MA, USA: AK Peters/CRC Press, 2003.
[31] R. Woop, "A programmable ray processing unit for realtime ray tracing," *ACM Trans. Graph., SIGGRAPH*, vol. 5, pp. 1–11, 2005.
[32] M. J. Doyle, C. Fowler, and M. Manzke, "A hardware unit for fast SAH-optimised BVH construction," *ACM Trans. Graph.*, vol. 32, no. 4, pp. 1–10, 2013.
[33] W.-J. Lee et al., "SGRT: A mobile GPU architecture for real-time ray tracing," in *Proc. 5th High-Perform. Graph. Conf.*, 2013, pp. 109–119.
[34] M. Stich, "Introduction to NVIDIA RTX and DirectX ray tracing," 2018. [Online]. Available: https://devblogs.nvidia.com/introduction-nvidia-rtx-directx-ray-tracing/
[35] M. Ulmstedt and J. Stålberg, "GPU accelerated ray-tracing for simulating sound propagation in water," M.S. Thesis, Dept. Elect. Eng., Linköping Univ., Linköping, Sweden, 2019.
[36] Microsoft, "DirectX-Spec," 2022. [Online]. Available: https://microsoft.github.io/DirectX-Specs/d3d/Raytracing.html
[37] F. Luna, *Introduction to 3D Game Programming With DirectX 12*. Herndon, VA, USA: Mercury Learn. Inf., 2016.
[38] J. Amanatides et al., "A fast voxel traversal algorithm for ray tracing," *Eurographics*, vol. 87, no. 3, pp. 3–10, 1987.
[39] NVIDIA, "NVIDIA OptiX 7.6," 2022. [Online]. Available: https://raytracing-docs.nvidia.com/optix7/guide/index.html#preface#