Accelerating Probabilistic Volumetric Mapping using Ray-Tracing Graphics Hardware

Heajung Min, Kyung Min Han and Young J. Kim

***Abstract*— Probabilistic volumetric mapping (PVM) represents a 3D environmental map for an autonomous robotic navigational task. A popular implementation such as Octomap is widely used in the robotics community for such a purpose. The Octomap relies on an octree to represent a PVM and its main bottleneck lies in massive ray-shooting to determine the occupancy of the underlying volumetric voxel grids.**

In this paper, we propose GPU-based ray shooting to drastically improve the ray shooting performance in Octomap. Our main idea is based on the use of recent ray-tracing RTX GPU, mainly designed for real-time photo-realistic computer graphics and the accompanying graphics API, known as DXR. Our ray-shooting first maps leaf-level voxels in the given octree to a set of axis-aligned bounding boxes (AABBs) and employ massively parallel ray shooting on them using GPUs to find free and occupied voxels. These are fed back into the CPU to update the voxel occupancy and restructure the octree. In our experiments, we have observed more than three-orders-of-magnitude performance improvement in terms of ray shooting using ray-tracing RTX GPU over a state-of-the-art Octomap CPU implementation, where the benchmarking environments consist of more than 77K points and 25K∼34K voxel grids.

I. INTRODUCTION

3D mapping is an essential component for autonomous navigational tasks since the accuracy of 3D mapping significantly affects estimating the surroundings where the robot is deployed. The reconstructed map subsequently impacts the quality of trajectories predicted by a motion planner whose goal is correctly guiding the robot in the given environment. Probabilistic volumetric mapping (PVM) is a popular strategy for representing such 3D maps for two main reasons. First, in PVM, a map is represented by a set of voxels that adaptively subdivides a 3D space depending on the occupancy of the space. As such, the adaptive voxel representation reduces the substantial amount of memory required to represent the 3D environment. Second, the occupancy of a voxel is represented probabilistically [1], empowering the map to cope with uncertainties including sensor noise and dynamic scenarios.

As a choice for volumetric reconstruction, an octree is a *de facto* standard thanks to its practical benefits on both computational and memory efficiencies. An octree is a spatial subdivision that adequately represents the hierarchical nature of voxels, in particular where the time complexity for traversal is maintained in the logarithmic scale.

Octomap [2] is a state-of-the-art implementation for PVM using the octree representation that consists of the following steps to build a PVM:

The authors are with the department of computer science and engineering at Ewha womans university in Korea hjmin@ewhain.net, {hankm|kimy}@ewha.ac.kr

1) Scan and generate a point cloud for the environment.
2) Shoot rays toward each point in the point cloud.
3) Find and identify free or occupied voxels in the observed space.
4) Update the octree with updated occupancy.

Often, the second and third steps, i.e., the ray-shooting step, are the most time-consuming operation, especially when the input point cloud is large or the ray-length becomes long. Such cases potentially limit the PVM module to maintain the map at a coarse level, and the situation will get worse if a higher-resolution sensor is used to generate a point cloud. Furthermore, the slow ray-shooting process could hinder the robot from conducting online navigation.

Main Results: In this paper, we perform GPU-based ray shooting to drastically improve the ray shooting performance in Octomap, which is the main bottleneck in octree-based PVM. The main idea of our work is mapping leaf-level voxels in the given octree to a set of axis-aligned bounding boxes (AABBs) and employing massively parallel ray shooting on them using GPUs. The intersected AABBs are further subdivided into the finest resolution corresponding to free or occupied voxels. After that, voxel occupancy and octree restructuring are followed on the CPU after the voxels are readback from the GPU. In our experiments, we have observed more than three-orders-of-magnitude performance improvement in terms of ray shooting using ray-tracing RTX GPU over a state-of-the-art Octomap CPU implementation, where the benchmarking environments consist of 77K points and 25K∼34K adaptive voxel grids.

II. RELATED WORK

A. 3D Volumetric Mapping

The idea of subdividing a 2D planar floor into uniform grid cells, namely occupancy grid map (OGM), dates back to mid-80s [3]. Later, the grid map idea was expanded to reconstruct a 3D map where a volumetric space is subdivided into equal-sized volumetric grids, i.e., voxels [4], [5]. However, representing a volumetric space with uniform grids inevitably causes memory problems, preventing practical applications demanded large-scale and long-term operations. A well-known strategy to mitigate this problem is employing an octree data structure [2], [6], [7].

The voxel-based sampling assumes that the occupancy of each voxel is independent of its neighboring voxels. While this assumption is useful for simplifying the OGM, the resulting map suffers from inaccuracy to some extent. Gaussian Process (GP)-based methods [8], [9], on the other hand,

consider a continuous spatial domain rather than discretized grid cells. As a result, GP allows maps to estimate unknown terrains. Besides, the map can represent itself with various resolutions. Unfortunately, the drawback of this method is a large memory requirement and a cubic time complexity in terms of the number of cells. Subsequently, there have been researches to ameliorate the performance issue of GP in recent years [10].

When the application of the map is limited to particular purposes such as the legged robot's motion planning, a 2.5D height map [11], [12] is an efficient way to reconstruct the local surroundings of the robot. Normal Distributions Transform (NDT) is another approach to discretize a 3D volumetric space. In contrast to voxel-based sampling, a cell in NDT contains multiple points to form a local Gaussian distribution. For this reason, NDT is considered as a piece-wise continuous representation of a space, where the number of grid cells is much smaller than that of voxel grid maps. NDT was first proposed for 2D scan registration purposes [13]. Later, the idea was further developed to 3D scan registration methods [14]. [15] proposed to augment occupancy probability to NDT, which was followed by a real-time version [16].

B. GPU-based Octree Construction

The octree structure is predominantly used in computer graphics for various tasks including distance field generation, rendering, modeling, simulation, and model reconstruction [2], [17], [18]. [19] proposes a GPU-based octree construction for reconstructing surfaces on the GPU. For a large-scale volumetric scene, full or out-of-core style octree update to GPU was studied [20], [21]. Octree can be adjusted dynamically in real-time in GPU [22] as well as one-time construction or full reconstruction [23]. [24] studied streaming subtree data through CPU-GPU data transfer in a view-dependent manner with connectivity information. Recently, [17] supports dynamic topological updates on GPU.

To reduce the cost of searching neighbors during ray traversals on Octree, [24] has reduced the number of neighbors down to six per cell by pointing the parents of neighbors. [25] used three precomputed neighbors per cell that enable stackless ray casting and dynamic updating of Octree on a GPU. A sparse voxel octree (SVO) showed both high-quality rendering and efficient ray traversal of shallow tree topology for the static scene [26]. OpenVDB [20] used SVO data structures and was implemented in GPU [17], which enables efficient neighbor access using GPU-based ray casting for dynamic scenes.

C. GPU-based Ray Tracing

Ray tracing is a graphical technique to render a realistic scene using the physical properties of light. From a given camera viewpoint, many rays are generated and shot toward the virtual 3D scene and each ray-path is traced to determine the corresponding pixel color of the screen [27].

Graphics hardware-based ray tracing has been studied to accelerate each stage of the ray-tracing pipeline including constructing acceleration structure, ray generation, traversing acceleration structure, and ray-triangle intersection. The growing need for rendering dynamic scenes demanded online construction and update of the acceleration structure, such as kd-tree [28] or BVH [29]. [30] introduced dedicated graphics hardware for ray generation. The ray traversal is the bottleneck of the ray tracing pipeline. [31] explored the space of ray traversal on kd-trees, and [32] proposed a BVH traversal hardware using three parallel pipelines. After traversal, the process of identifying the closest intersection point through the ray-triangle hit test has been studied [32], [30].

In recent years, the RTX GPU platform for accelerating ray tracing has been introduced, which accelerates the BVH construction and traversal, and proposes a new rendering pipeline with a machine learning-based denoising technique to enable real-time ray tracing rendering [33], [34]. Moreover, RTX can be used as general-purpose computing (GPGPU) for non-rendering tasks, such as sampling for simulation [35], point location in tetrahedral meshes [36], or Monte Carlo particle transport [37].

III. Fast and Massively-Parallel Ray Shooting on RTX

Since ray-tracing is a computationally intensive technique, dedicated hardware support is highly beneficial. Timely, NVIDIA introduced and started to mass-produce ray tracing hardware, known as RTX. In CPU-based ray tracing, for each ray, thousands of serial CPU instructions need to be executed to check the ray-surface intersection using the bounding volume hierarchy (BVH) representing the scene geometry, often composed of many triangles. Many RT cores installed in the RTX graphics hardware consist of two units, one responsible for BVH traversal and the other for ray-triangle or ray-AABB intersection test [38].

In order to harness the power of RTX GPU, an application programming interface (API) such as DirectX's DXR has been also introduced. In DXR, scene geometry is divided into bottom/top-level acceleration structures, which is essentially a BVH. In the bottom level acceleration structure, geometric primitives such as triangles or AABBs are included, and in the top-level acceleration structure, transformation matrix and rendering material are included.

Also, shader programs in the RTX pipeline are executed in a massively-parallel fashion for all rays. Specifically, first, after a BVH is built, rays are generated in *the ray generation shader*, traverse the BVH, and check the intersection for hitting bounding volumes. Then, for the hit group, *the intersection shader* finds an intersection of the ray and the geometry, and *the closest hit shader* calculates shading information at the hit point. If the ray does not intersect any geometry, *the miss shader* would be executed [33]. In our application, we do not use the closest hit shader as no shading information is needed.

IV. GPU-ACCELERATED RAY SHOOTING

A. Octree Representation of PVM

The octree is a hierarchical data structure created by recursively subdividing three-dimensional space. Each octree node represents a small volumetric grid called a voxel and has eight sub-voxels as child nodes. The octree is also useful for labeling its subspace with different occupancy information using voxels.

A well-known PVM such as Octomap [2] relies on octree to map the environment into a set of voxels representing a *free* or a *occupied* state (obstacle space). Specifically, Octomap builds a 3D map by repeatedly relocating a proximity sensor at different positions and carrying out the following steps in turn:

1) A sensor observation is obtained in the form of a point cloud.
2) The sensor origin $\mathbf{o}$ and each point $\mathbf{p}$ in the point cloud constitute an individual ray $\overrightarrow{\mathbf{op}}$ with a fixed length $|\overrightarrow{\mathbf{op}}|$.
3) For all rays, a ray-shooting procedure is performed against the voxelized space using DDA [39] to discover free and occupied voxels. These labeled voxels correspond to the finest resolution of the octree.
4) The discovered voxels are merged into the octree and restructured depending on the cells' probabilistic occupancies.

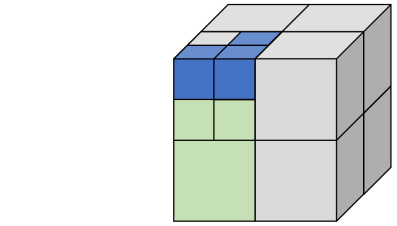
Note that the second and the third step take up about 90% of processing time for each scan and are the main bottleneck. The main goal of our work is to drastically improve it using massively parallel ray shooting on GPU.

B. Mapping an Octree on CPU to AABBs on GPU

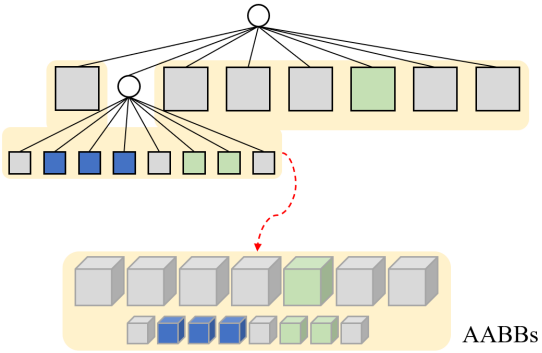
In order to leverage GPU-accelerated ray shooting for an octree, the voxel elements of the octree must be converted to geometric primitives that the ray-tracing GPU such as the RTX can process, which should be a set of triangles or AABBs in case of RTX. In our problem, we opt for AABBs as target primitives as their geometries are close to the shape of a voxel.

Specifically, we convert all leaf-level voxels in an octree, with occupied (blue), free (green), and unknown (gray) labels, to individual AABBs. As illustrated in Fig. 1, a leaf-level voxel is mapped to an AABB by matching their geometries including the position and size. Note that we also consider voxels with an unknown label during the conversion so that the AABBs need to fill the entire 3D workspace since a random ray can pass anywhere in the space and the traversed subspace needs to be labeled afterward.

After all leaf-level voxels are converted to a set of AABBs, they are uploaded to the GPU, and the RTX GPU builds a bounding volume hierarchy (BVH) of AABBs. Fig. 1 illustrates (a) spatial subdivision of space with different labels to represent occupied, free, or unknown voxel space, (b) a corresponding octree representation on CPU with leaf-level nodes highlighted in yellow, and a set of AABBs on GPU that correspond to the leaf-level voxel nodes on CPU.



(a) Spatial Subdivision

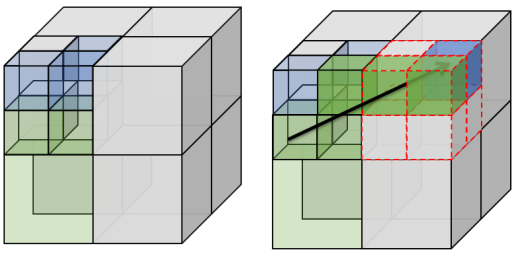


(b) Octree and the corresponding AABBs

Fig. 1: Mapping from an octree in CPU to a set of AABBs in GPU for GPU-based ray shooting. In the figures, blue, green, and gray boxes indicate occupied, free, unknown voxel space, respectively.

C. Massively-Parallel Ray Shooting

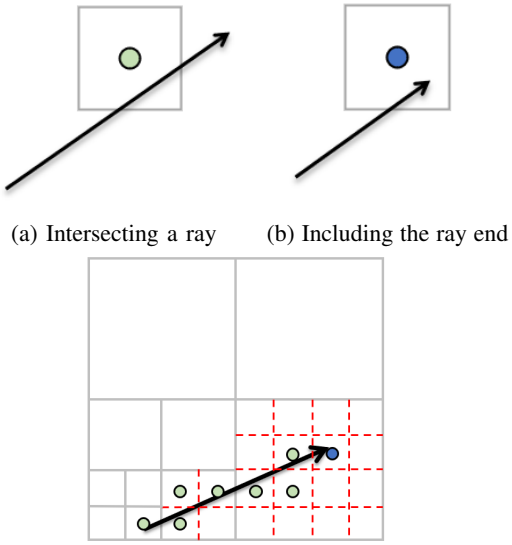
Once the BVH of AABB is computed on GPUs, we set up multiple rays in the ray generation shader and shoot them from the sensor origin to the environment obstacles, obtained as a point cloud by the sensor, to find occupied or free voxels of space. The direction of each ray is defined by the sensor origin (ray start) and the position of each point (ray end) in the cloud. Therefore, the number of rays is proportional to the number of points in the cloud, and the rays are shot in parallel. We set each ray's length equal to the distance from the sensor origin to the point cloud. This way, the ray end always corresponds to an occupied voxel and the interior of the ray to free voxels.



(a) Mapped AABBs (b) Subdivision

Fig. 2: Leaf-level voxels are mapped to a set of AABBs and subdivided to the finest resolution for occupancy labeling.

In our current implementation, we rely on GPUs to perform ray shooting, but still use CPU for octree and occupancy



(a) Intersecting a ray (b) Including the ray end

(c) Intermediate voxel

Fig. 3: Different cases of ray-AABB intersection. (a), (b) a ray hitting finest-resolution voxels, (c) a ray hitting a intermediate voxel that is not the finest resolution. (a) is labeled as free (green) and (b) is as occupied (blue). (c) is subdivided into finest resolution and labeled accordingly. The ray starts from the sensor origin and ends at a point in the cloud.

update, which is an effective strategy as the update step is not a dominant part of the entire pipeline. In particular, we use Octomap for the update part, and the Octomap expects voxels of the finest resolution in the octree as a result of ray shooting.

In order to meet this interfacing requirement, in our GPU-based ray shooting, if the size of an AABB intersected with a ray is greater than that of a finest-resolution voxel, we subdivide the subspace that the ray traverses to a set of sub-voxels in the finest resolution (typically 16) using DDA [39]. These voxels are labeled as free except that the voxel containing the endpoint of the ray is labeled as occupied. Fig. 2 illustrates this situation and Fig. 3 illustrates different cases of ray-AABB intersection. All AABBs intersected with a given ray are found within the ray's extent - i.e. the ray in our case is geometrically a line segment in 3D. When ray-AABB intersection occurs, determining the occupancy of the voxel is performed in the intersection shader. If the intersection of ray and AABB no longer occurs, the miss shader is executed. Since we do not need shading in our work, we execute the miss shader to simply terminate the ray shooting.

After a set of voxels hit by rays are found, their occupancy is updated, and the octree nodes are restructured. Typically, leaf-level nodes with the same occupancy are merged and promoted to a higher-level node. Currently, this step is executed on the CPU by sending the voxel data back to the CPU. However, this is also possible on GPUs as demonstrated by previous works in Sec. II-B. The pseudo-code for our whole ray shooting procedure is given in Algorithm 1.

Algorithm 1: GPU-based Ray Shooting

Input : sensorOrigin, rayDirections, a set of AABBs
Output: Intersected voxels with the rays

```
1  Build the BVH of AABBs in parallel;
2  do in parallel
3      /* using Ray Generation shader  */
4      Setup a ray;
5      while The ray hits AABBs within the ray extent
         during BVH traversal do
6          /* using Intersection shader  */
7          if The AABB size is greater than finest
             resolution voxels then
8              Subdivide the AABB to a set of finest
                 resolution voxels;
9          end if
10         forall Newly generated finest resolution
             voxels do
11             if voxel includes the ray's endpoint then
12                 Set it as occupied;
13             else if ray passed the voxel then
14                 Set it as free;
15         end forall
16     end while
17 end
```

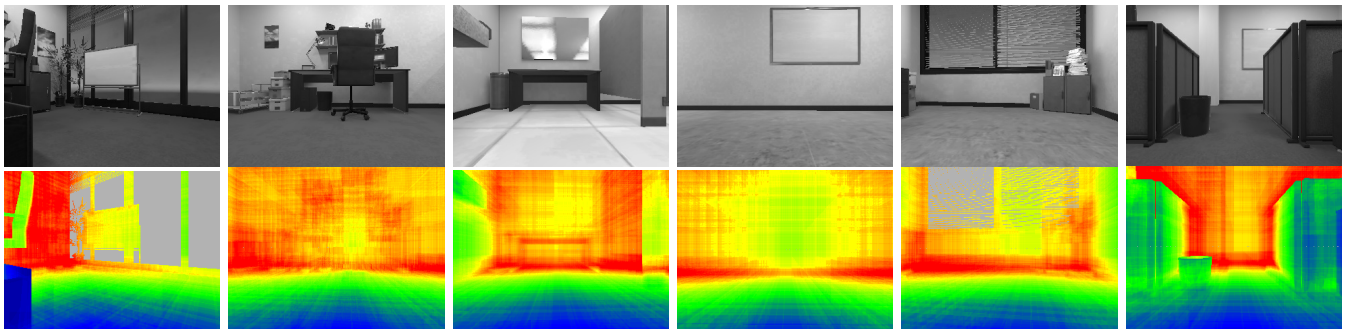
V. EXPERIMENTS AND RESULTS

In this section, we show our experimental results for GPU-based map building and also compare them against a state-of-the-art implementation.

A. Evaluation of Ray Shooting

GPU-based ray shooting for map building was implemented on a 64bit Windows 10 operating system and Microsoft Visual Studio 2017 C++ with AMD's Ryzen 7 3700X CPU, NVIDIA's RTX 2080 GPU, and 16GBs RAM. We used DirectX's DXR to drive GPU-accelerated ray tracing on RTX. As a benchmarking platform, we employed a virtual indoor environment built-in Tesse-Unity simulator [40] where a mobile robot equipped with a stereo camera navigated around this environment to collect a point cloud data set and build an octree-based map. Robot localization is assumed to be given and exact.

In order to test the benchmarks, while driving through a virtual building, at a random viewpoint, a point cloud is acquired from the scene using a range sensor. The acquired point cloud data is built into an octree with a maximum depth of 16, corresponding to 25K$\sim$34K leaf-level voxels. The ray-shooting step, which is the main bottleneck during map building, is implemented on GPU and the rest of the building steps is done on CPU using Octomap. Figure 4 illustrates six views from different viewpoints. Here, we shoot 320$\times$240 rays (76,800 rays) per view to collect and



(a) (b) (c) (d) (e) (f)

Fig. 4: Results of GPU-based ray shooting from different viewpoints navigating inside a complex virtual building. The top row is the target scene. The bottom row is the corresponding hit count of rays with voxels in the space; as the color changes from blue to red, more voxels are intersected with rays; gray indicates that the distance sensor did not obtain point cloud due to reflections in the environment.

TABLE I: Comparisons of Ray Shooting Performance on Octomap (CPU) and Ours (GPU), and Timing Breakdown

Benchmarking Views		(a)	(b)	(c)	(d)	(e)	(f)
# of Free, Occupied or Unknown Voxels in Octree		25,515	33,806	34,412	21,832	23,609	31,215
Octomap (CPU)	Ray Shooting (ms)	886.83	1,446.52	1,349.87	1,294.31	1,211.58	1,236.16
Ours (GPU)	Build BVH (ms)	0.59	0.63	0.58	0.55	0.54	0.68
	Ray Shooting (ms)	1.42	1.67	2.07	1.14	1.11	1.59
	Readback from GPU to CPU (ms)	14.19	14.63	14.39	14.23	14.33	14.29

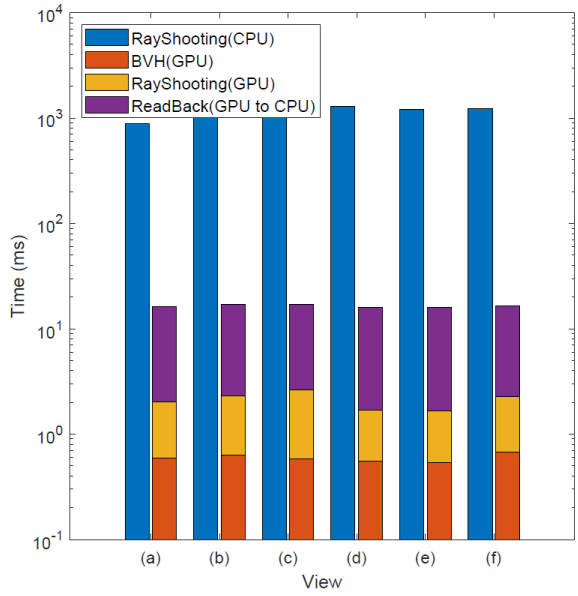


Fig. 5: Relative Performance Comparisons of Ray Shooting between Octomap (CPU in blue bars) and Ours (GPU in gold bars) in the Logarithmic Scale. BVH construction and readback from GPU to CPU are denoted in orange and purple bars.

identify octree cells. We measure the ray shooting time using the dispatchRay function of RTX, which queries the elapsed time of ray tracing performed on the GPU.

For the sake of comparison, ray-shooting time on CPU using Octomap was also measured with the same experimental condition with the same camera position, direction, and ray length using the same point cloud and the same number of voxels of the octree. The CPU implementation was measured on Ubuntu 20.04 with Intel Core i7-7700HQ CPU @ 2.8GHz and 16 GBs of RAM.

Table I is the result of evaluating ray shooting time for each view shown in Fig. 4. In the case of GPU-based ray shooting, we further included the BVH construction time on the GPU and the time for reading back the intersected voxels from GPU to CPU. Fig. 5 shows that GPU-based ray shooting can be performed three-orders-of-magnitude faster than CPU-based ray shooting on average excluding GPU-CPU readback. Even though GPU-CPU readback time is included, the performance improvement is still two-orders-of-magnitude faster than the CPU version, and the readback time may be removed by updating the octree and occupancy on the GPU using a technique like [18], [25].

VI. CONCLUSIONS

In this paper, we propose GPU-based ray shooting to improve the ray shooting performance, which is the main bottleneck, in the state-of-the-art PVM algorithm such as Octomap. The main idea is based on the use of recent ray-tracing RTX GPU and map voxel grids of an octree to a set of AABBs and employ massively parallel ray shooting on them using the GPUs to find free and occupied voxels. The observed speedup in ray-shooting is significant, for instance, to perform online navigation for the robot. There are limitations to our current work. First, the octree itself is not maintained on GPUs but CPU. Thus, it is required that the newly found voxels need to be readback from GPUs to the CPU, which takes almost 10X more than the ray shooting itself. However, this can be addressed in the future

using various GPU-based octree maintenance techniques as explained in Sec. II-B.

ACKNOWLEDGMENT

This project was supported in part by the ITRC/IITP program (IITP-2021-2020-0-01460) and the NRF (2017R1A2B3012701) in South Korea.

REFERENCES

[1] S. Thrun, W. Burgard, and D. Fox, *Probabilistic robotics*. Cambridge, Mass.: MIT Press, 2005.

[2] A. Hornung, K. M. Wurm, M. Bennewitz, C. Stachniss, and W. Burgard, "OctoMap: An efficient probabilistic 3D mapping framework based on octrees," *Autonomous Robots*, 2013, software available at http://octomap.github.com. [Online]. Available: http://octomap.github.com

[3] H. Moravec and A. Elfes, "High resolution maps from wide angle sonar," in *Proceedings. 1985 IEEE international conference on robotics and automation*, vol. 2. IEEE, 1985, pp. 116–121.

[4] Y. Roth-Tabak and R. Jain, "Building an environment model using depth information," *Computer*, vol. 22, no. 6, pp. 85–90, 1989.

[5] H. Moravec, "Robot spatial perception by stereoscopic vision and 3d evidence grids," robotics institute," Tech. Rep., 1996.

[6] K. Schauwecker and A. Zell, "Robust and efficient volumetric occupancy mapping with an application to stereo vision," in *2014 IEEE International Conference on Robotics and Automation (ICRA)*, 2014, pp. 6102–6107.

[7] E. Vespa, N. Nikolov, M. Grimm, L. Nardi, P. H. J. Kelly, and S. Leutenegger, "Efficient octree-based volumetric slam supporting signed-distance and occupancy mapping," *IEEE Robotics and Automation Letters*, vol. 3, no. 2, pp. 1144–1151, 2018.

[8] S. T. O'Callaghan and F. T. Ramos, "Gaussian process occupancy maps," *The International Journal of Robotics Research*, vol. 31, no. 1, pp. 42–62, 2012.

[9] S. Vasudevan, F. Ramos, E. Nettleton, H. Durrant-Whyte, and A. Blair, "Gaussian process modeling of large scale terrain," in *2009 IEEE International Conference on Robotics and Automation*, 2009, pp. 1047–1053.

[10] J. Wang and B. Englot, "Fast, accurate gaussian process occupancy maps via test-data octrees and nested bayesian fusion," in *2016 IEEE International Conference on Robotics and Automation (ICRA)*, 2016, pp. 1003–1010.

[11] P. Fankhauser, M. Bloesch, and M. Hutter, "Probabilistic terrain mapping for mobile robots with uncertain localization," *IEEE Robotics and Automation Letters (RA-L)*, vol. 3, no. 4, pp. 3019–3026, 2018.

[12] D. Belter, P. Łabcki, and P. Skrzypczyński, "Estimating terrain elevation maps from sparse and uncertain multi-sensor data," in *2012 IEEE International Conference on Robotics and Biomimetics (ROBIO)*, 2012, pp. 715–722.

[13] P. Biber and W. Strasser, "The normal distributions transform: a new approach to laser scan matching," in *Proceedings 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2003) (Cat. No.03CH37453)*, vol. 3, 2003, pp. 2743–2748 vol.3.

[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," *The International Journal of Robotics Research*, vol. 31, pp. 1377–1393, 09 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 *2013 IEEE International Conference on Robotics and Automation*, 2013, pp. 2233–2238.

[16] C. Schulz, R. Hanten, and A. Zell, "Efficient map representations for multi-dimensional normal distributions transforms," in *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2018, pp. 2679–2686.

[17] R. K. Hoetzlein, "Gvdb: Raytracing sparse voxel database structures on the gpu," in *Proceedings of High Performance Graphics*, 2016, pp. 109–117.

[18] F. Liu and Y. J. Kim, "Exact and adaptive signed distance fields computation for rigid and deformable models on gpus," *IEEE transactions on visualization and computer graphics*, vol. 20, no. 5, pp. 714–725, 2014.

[19] K. Zhou, M. Gong, X. Huang, and B. Guo, "Data-parallel octrees for surface reconstruction," *IEEE transactions on visualization and computer graphics*, vol. 17, no. 5, pp. 669–681, 2010.

[20] K. Museth, "Vdb: High-resolution sparse volumes with dynamic topology," *ACM transactions on graphics (TOG)*, vol. 32, no. 3, pp. 1–22, 2013.

[21] M. Hadwiger, J. Beyer, W.-K. Jeong, and H. Pfister, "Interactive volume exploration of petascale microscopy data streams using a visualization-driven virtual memory approach," *IEEE Transactions on Visualization and Computer Graphics*, vol. 18, no. 12, pp. 2285–2294, 2012.

[22] C. Crassin, F. Neyret, M. Sainz, S. Green, and E. Eisemann, "Interactive indirect illumination using voxel cone tracing," in *Computer Graphics Forum*, vol. 30, no. 7. Wiley Online Library, 2011, pp. 1921–1930.

[23] C. Crassin, F. Neyret, S. Lefebvre, and E. Eisemann, "Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering," in *Proceedings of the 2009 symposium on Interactive 3D graphics and games*, 2009, pp. 15–22.

[24] 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," *The Visual Computer*, vol. 24, no. 7-9, pp. 797–806, 2008.

[25] 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, no. 7. Wiley Online Library, 2018, pp. 179–190.

[26] S. Laine and T. Karras, "Efficient sparse voxel octrees," *IEEE Transactions on Visualization and Computer Graphics*, vol. 17, no. 8, pp. 1048–1059, 2010.

[27] P. Shirley and R. K. Morley, *Realistic ray tracing*. AK Peters/CRC Press, 2003.

[28] R. Woop, "A programmable ray processing unit for realtime ray tracing," *ACM Transactions on Graphics, SIGGRAPH*, vol. 5, pp. 1–11, 2005.

[29] M. J. Doyle, C. Fowler, and M. Manzke, "A hardware unit for fast sah-optimised bvh construction," *ACM Transactions on Graphics (TOG)*, vol. 32, no. 4, pp. 1–10, 2013.

[30] 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 Transactions on Graphics (TOG)*, vol. 33, no. 5, pp. 1–15, 2014.

[31] S. Woop, "A programmable hardware architecture for realtime ray tracing of coherent dynamic scenes," *Diss. Ph. D. Thesis, Sarrland University*, 2007.

[32] W.-J. Lee, Y. Shin, J. Lee, J.-W. Kim, J.-H. Nah, S. Jung, S. Lee, H.-S. Park, and T.-D. Han, "Sgrt: A mobile gpu architecture for real-time ray tracing," in *Proceedings of the 5th high-performance graphics conference*, 2013, pp. 109–119.

[33] M. Stich, "Introduction to NVIDIA RTX and DirectX Ray Tracing," 2018. [Online]. Available: https://devblogs.nvidia.com/introduction-nvidia-rtx-directx-ray-tracing/

[34] M.-K. Lefrancois and P. Gautron, "DX12 Raytracing tutorial," 2018. [Online]. Available: https://developer.nvidia.com/rtx/raytracing/dxr/DX12-Raytracing-tutorial-Part-2

[35] C. Gribble, "Multi-hit ray tracing in dxr," in *Ray Tracing Gems*. Springer, 2019, pp. 111–125.

[36] I. Wald, W. Usher, N. Morrical, L. Lediaev, and V. Pascucci, "Rtx beyond ray tracing: Exploring the use of hardware ray tracing cores for tet-mesh point location." in *High Performance Graphics (Short Papers)*, 2019, pp. 7–13.

[37] J. Salmon and S. McIntosh-Smith, "Exploiting hardware-accelerated ray tracing for monte carlo particle transport with openmc," in *2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. IEEE, 2019, pp. 19–29.

[38] Nvidia, "Nvidia turing gpu architecture," 2018. [Online]. Available: https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf

[39] J. Amanatides, A. Woo *et al.*, "A fast voxel traversal algorithm for ray tracing."

[40] D. Yadav, R. Jain, H. Agrawal, P. Chattopadhyay, T. Singh, A. Jain, S. B. Singh, S. Lee, and D. Batra, "Evalai: Towards better evaluation systems for ai agents," 2019.