UFOMap: An Efficient Probabilistic 3D Mapping Framework That Embraces the Unknown

Daniel Duberg and Patric Jensfelt

Abstract—**3D models are an essential part of many robotic applications. In applications where the environment is unknown a-priori, or where only a part of the environment is known, it is important that the 3D model can handle the unknown space efficiently. Path planning, exploration, and reconstruction all fall into this category. In this letter we present an extension to OctoMap which we call UFOMap. UFOMap uses an explicit representation of all three states in the map, i.e., unknown, free, and occupied. This gives, surprisingly, a more memory efficient representation. We provide methods that allow for significantly faster insertions into the octree. Furthermore, UFOMap supports fast queries based on occupancy state using so called indicators and based on location by exploiting the octree structure and bounding volumes. This enables real-time colored octree mapping at high resolution (below 1 cm). UFOMap is contributed as a C++ library that can be used standalone but is also integrated into ROS.**

Index Terms—**Mapping, RGB-D perception, motion and path planning.**

I. Introduction

MANY robot tasks require a 3D model of the environment to be completed. For navigation and manipulation tasks the model is often used to calculate collision free paths and in exploration to determine where new information can be found and how to get to it. 3D SLAM updates the model and localizes the robot using it.

A number of different methods for modeling 3D environments exists: point clouds, elevation maps [1], multi-level surface maps [2], octrees [3], and signed distance fields [4]. All of these methods are able to represent occupied and free space. However, only octrees and signed distance fields are able to represent the unknown space. Moreover, point clouds, elevation maps, and multi-level surface maps are not able to represent arbitrary 3D environments.

OctoMap [5], one of the most popular mapping frameworks, is based on octrees. It provides a readily available, reliable, and efficient 3D mapping framework. OctoMap provides researchers, also those not focusing on mapping, an off the shelf solution for

Manuscript received February 24, 2020; accepted July 9, 2020. Date of publication August 4, 2020; date of current version August 14, 2020. This letter was recommended for publication by Associate Editor L. Paull and Editor S. Behnke upon evaluation of the reviewers' comments. This work was supported in part by SSF FACT and in part by VR XPLORE3D. *(Corresponding author: Daniel Duberg.)*

The authors are with the Division of Robotics, Perception and Learning (RPL), KTH Royal Institute of Technology, Stockholm 114 28, Sweden (e-mail: dduberg@kth.se; patric@kth.se).

Digital Object Identifier 10.1109/LRA.2020.3013861

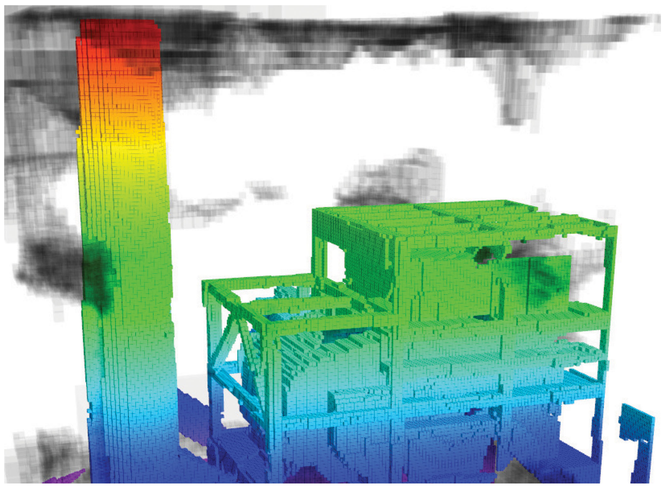

Fig. 1. UFOMap representation of in-progress exploration of a power plant from Gazebo model library (https://bitbucket.org/osrf/gazebo_models/src). The colored voxels are occupied space. The black voxels are unknown space which are represented explicitly in UFOMap.

mapping. While OctoMap excels in many use cases, there exist certain areas where it can be improved. In this letter we focus on two areas.

First, in OctoMap, unknown space is not modeled explictly like occupied and free space are. In algorithms where the unknown space is used extensively, OctoMap's implicit representation of unknown space can be a bottleneck. Use cases where this can be a problem are, for example, collision checking, path planning where we want to know if it is possible for a robot to move from one location to another and next-best view exploration methods such as [6], [7], [8], and [9]. In the first two cases the common workaround is to treat unknown space as free space [10], [11]. This way, only occupied space has to be considered in the collision checking. However, this increases the probability of collisions. Regarding the third use case, [9] states that "*computing volumetric information gains can account for up to 95% of a planners run-time.*" The information gain for a certain sensor/robot pose in an exploration scenario is typically calculated as a function of the unknown space. In [7] the information gain is approximated, leading to a reduced need to access unknown space in the OctoMap and it is one of the major contributing factors of the performance improvement over the baseline [6].

Second, manipulating the content of an OctoMap is time consuming and limited. For example, inserting a single point cloud takes on the order of a second. This means that real-time mapping with an OctoMap is prohibitively slow in many applications.

In this letter we present an extension to OctoMap taking these shortcomings into account. The contributions of this work are as follows:

- An explicit representation of unknown space.
- Indicator values for the inner nodes summarizing the state of children for faster state based queries.
- A separate threshold for classifying space as free, allowing for unknown space to be defined also based on the occupancy value.
- Morton codes for faster traversal of the octree.
- A fast ray casting approach for integrating data into the octree.

The remainder of this letter is organized as follows. Related work is presented in Section II, while in Section III the UFOMap mapping framework is presented. Implementation details are given in Section IV, followed by an evaluation of the proposed framework and case studies in Sections V and VI, respectively. Finally, Section VII summarizes the letter.

II. Related Work

This section gives a brief overview of different 3D map representations used in robotics.

A popular approach to model 3D environments is to discretize the world into equal sized cubic volumes [12], called voxels. One of the major shortcomings of fixed grid structures is that the size of the area to be mapped has to be known *a-priori*. Memory requirements can also be a problem when mapping large areas at a high resolution. Voxel hashing [13] is one approach to overcome these shortcomings, where fixed sized blocks are allocated on demand.

The mapping framework Voxblox [14] uses a signed distance field [4] voxel grid, with voxel hashing for dynamic growth, as representation. It was mainly developed for planning or trajectory optimizations in the context of micro aerial vehicles (MAVs). The signed distance field representation makes trajectory optimizations faster by storing the distance to the closest obstacle in each voxel. Voxblox builds on [13] where they use a spatial hashing scheme and allocate blocks of fixed size when needed. This means that the size of the area to be mapped does not need to be known beforehand.

One of the most popular mapping frameworks is OctoMap [5]. OctoMap uses an octree-based data structure, as proposed in [3], to do occupancy mapping. The octree structure allows for delaying the initialization of the grid structure. It is also often more memory efficient compared to Voxblox or a fixed grid structure since the information can be stored at different resolutions in the octree, without losing any precision. If the inner nodes of the octree are updated correctly, it is possible to do queries at different resolutions. This can be especially beneficial in systems where multiple algorithms are using the same map but have different computational and time requirements. These are important properties and we will therefore use an octree representation and build on OctoMap.

III. UFOMap Mapping Framework

This section presents the UFOMap mapping framework, which is based on OctoMap. We highlight the difference between the two frameworks to make comparison easier.

UFOMap uses an octree data structure, just like OctoMap. An octree is a tree data structure where each node represents a cubic volume of the environment being mapped. A node can recursively be split up into eight equal sized smaller nodes, until a minimum size has been reached. The resolution of the octree is the same as the minimum node size. A visual representation of an octree can be seen in Fig. 2(b).

Each node has an occupancy value indicating if the cubic volume represented by the node is occupied, free, or unknown. A probabilistic occupancy value is used in order to better handle sensor noise and dynamic environments. More specifically the log-odds occupancy value is stored in the nodes. Sensor readings can thus be fused in using only additions instead of multiplications. The log-odds occupancy value is clamped to allow for pruning the tree, as in OctoMap. Pruning can be applied when all eight of a node's children are the same. This leads to a smaller tree, which is beneficial both in terms of memory usage and efficiency when traversing the tree.

As mentioned in Section II, the octree data structure allows for queries at different resolutions. This can speedup applications, as well as allow for different applications that require different resolutions to share the same map. As in OctoMap, a node is defined to have the same occupancy value as the maximum occupancy value of its children. Using the maximum value is conservative and it enables doing path planning, or similar, at whichever resolution without missing any obstacles.

The nodes in UFOMap store three indicators, i_f, i_u, and i_a, which are not found in OctoMap. i_f and i_u indicates if a node contains free space or unknown space, respectively. No indicator is needed for the occupied space, since the occupancy value of a node is enough to tell if it contains occupied space. When updating the octree the indicator values are propagated upwards in the tree along with the occupancy values.

The third indicator, i_a, indicates whether all of a node's children are the same. i_a is useful in cases were automatic pruning has been disabled, but you still want the same behaviour as if it were enabled. Automatic pruning means that the octree is automatically pruned when data is deleted or inserted into the tree. The main reason for disabling automatic pruning is for applications where you want to access and insert/delete data concurrently. It is possible to manually specify when the pruning should occur in this case. The indicators allow for increased efficiency when traversing the tree, by allowing certain branches to be ignored, for example, when only looking for unknown space.

The state of a node, n, in OctoMap is determined by a threshold, t_o:

$$\text{state}(n) = \begin{cases} \text{unknown} & \text{if } n \text{ is null pointer} \\ \text{occupied} & \text{if } t_o \leq \text{occ}(n) \\ \text{free} & \text{else.} \end{cases}$$

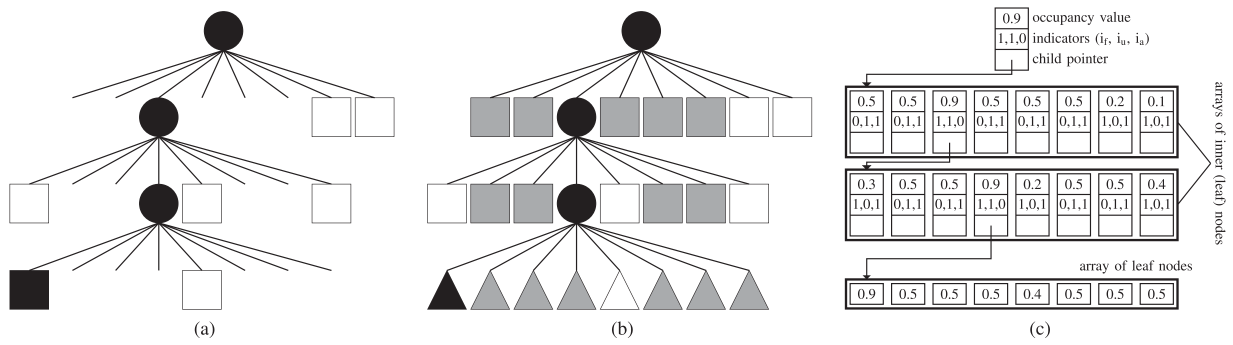
0.9 occupancy value
1,1,0 indicators (i_f, i_u, i_a)
child pointer
arrays of inner (leaf) nodes
array of leaf nodes
(a)
(b)
(c)

Fig. 2. Example of an octree. The OctoMap tree representation is shown in (a) and the corresponding tree representation in UFOMap is displayed in (b). Black indicates occupied space, white free space, and grey unknown space. Circles means that it is an inner node, square an inner leaf node, and triangle a leaf node. Note that OctoMap does not make a distinction between leaf nodes types. (c) shows how UFOMap stores that octree in memory.

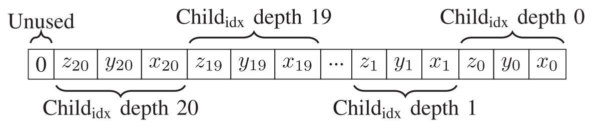

Fig. 3. Bit string representation of Morton code used in UFOMap.

where $\text{occ}(n)$ is the occupancy value of the node n. In UFOMap this has been expanded to include a threshold, t_f, for when a node is free as well:

$$\text{state}(n) = \begin{cases} \text{unknown} & \text{if } t_f \leq \text{occ}(n) \leq t_o \\ \text{occupied} & \text{if } t_o < \text{occ}(n) \\ \text{free} & \text{if } t_f > \text{occ}(n), \end{cases}$$

these two thresholds together can be useful in certain applications. In OctoMap's case any part of space that has ever been observed is forced to be classified as either occupied or free, unless you explicitly delete a node to restore the null pointer. By setting $t_f = 0.1$ and $t_o = 0.9$ in exploration and reconstruction, the nodes stay in the unknown state until they have high confidence that they are either occupied or free. This can simplify the reconstruction or exploration algorithm since only unknown nodes have to be considered and these can be accessed fast with the help of the indicators.

Alternatively, if you use $t_f = 0.5$ and $t_o = 0.9$, the exploration algorithm would focus its attention on the occupied space, meaning you would target higher certainty about the occupied space. This can be convenient in applications requiring reconstruction. Note that you do not have to change anything in the exploration method, and instead use these parameters to modify the exploration behaviour.

Finally, Morton codes [15] are used to speedup traversal of the octree. To generate a Morton code, we first convert the cartesian coordinate $c = (c_x, c_y, c_z) \in \mathbb{R}^3$ to the octree node index tuple $k = (k_x, k_y, k_z) \in \mathbb{N}_0^3$ that c falls into $k_j = \lfloor \frac{c_j}{\text{res}} \rfloor + 2^{d_{\text{max}}-1}$ where d_{max} is the depth of the octree. By interleaving the bits representing k_j as shown in Fig. 3, the Morton code m is created for c. From the root node it is then possible to traverse down to the node by simply looking at the three bits of m that correspond to the depth of the child and moving to that child.

The Morton code generation has been accelerated by using integer dilation and contraction [16].

IV. Implementation Details

This section covers implementation details and highlights differences compared to OctoMap.

A. Nodes

UFOMap has three different types of nodes; inner nodes, inner leaf nodes, and leaf nodes. In contrast, OctoMap has only what corresponds to the inner nodes and inner leaf nodes.

1) Inner Nodes: The inner nodes are all the nodes that have children. An inner node contains a log-odds occupancy value, the three indicators mentioned in Section III, and a pointer to an array of its children. In UFOMap the children are stored directly in the array, instead of the array holding pointers to the children like in OctoMap. This way, 64 bytes are saved for an inner node with 8 children in UFOMap compared to OctoMap. However, this also means that a node in UFOMap either must have no children or 8 children.

2) Inner Leaf Nodes: Inner leaf nodes are exactly the same as the inner nodes. The only distinction is that the inner leaf nodes have no children, meaning the child pointer is a null pointer. Once an inner leaf node gets a child it is considered an inner node. In OctoMap this is the only kind of leaf node that exist.

3) Leaf Nodes: The leaf nodes are more simplistic. Only the log-odds occupancy value is stored. Compared to an inner leaf node, it is not possible for a leaf node to have children. In OctoMap the corresponding node is the inner leaf node.

The three kinds of nodes can be seen in Fig. 2(c). The nodes which have an occupancy value, indicators, and a child pointer are inner nodes. Nodes where the child pointer is empty are inner leaf nodes. The nodes with only an occupancy value are the leaf nodes.

As stated in [5], and also seen in Table I, 80-85% of the octree nodes are (inner) leaf nodes in OctoMap. In UFOMap, where a distinction is made between inner lead nodes and leaf nodes, this value drops to 71-81%. This highlights that the majority of the nodes in the tree are leaf nodes. Therefore, having the dedicated leaf node data structures and by storing the children of an inner node directly in an array in UFOMap significantly reduces the memory usage over OctoMap, by a factor of 3, even though the total amount of nodes in the tree increases (see Table I).

TABLE I
MEMORY CONSUMPTION AND NUMBER OF NODES COMPARISON BETWEEN UFOMAP AND OCTOMAP ON THE OCTOMAP 3D SCAN DATASET

Dataset	Memory usage			Number of nodes			
	UFOMap (MB)	OctoMap (MB)	Reduction (%)	UFOMap	OctoMap	UFOMap leaves (%)	OctoMap leaves (%)
FR-078 tidyup	7.42	21.49	65.47	1 642 113	1 369 165	80.51	85.01
FR-079 corridor	19.70	51.86	62.01	2 823 713	1 829 134	75.20	80.70
Freiburg campus	58.71	155.46	62.23	8 402 193	5 515 178	75.10	80.96
freiburg1_360	15.98	42.05	62.00	2 161 849	1 547 112	71.72	82.53
New College	29.41	75.40	60.99	4 157 217	2 633 701	74.37	80.27

B. *Integrating Point Cloud Sensor Measurements*

Ray casting is used to integrate point cloud sensor measurements. When integrating a point cloud it is important that points that should be occupied gets an increased occupany value and that all points between the sensor origin and each point of the point cloud gets a lower occupancy value. In UFOMap there are three methods for this, each faster than the previous but with less accurate results:

1) Simple Integrator: For each point in the point cloud we trace a ray, using [17], and decrease the occupancy value for all nodes from origin to the point. The occupancy value for the node in which the point lies is increased. Same as in OctoMap.

2) Discrete Integrator: By discretizing the point cloud first it is possible to utilize that a number of points in the point cloud fall inside the same node. The ray casting is only done once for each node the points fall into. The ray casting is performed towards the center of the node which means the result may vary compared to the simple integration. However, this is significantly faster for large point clouds with many points falling into the same node. This also exists in OctoMap.

3) Fast Discrete Integrator: In the fast discrete integrator, the ray casting and discretization is performed at multiple different depths of the octree. First, a simple ray marching algorithm with a fixed step size equal to the resolution corresponding to depth d is applied. Ray marching is performed until n nodes, at that resolution, away from the end node. From that node we perform a new ray marching at lower depth until we reach depth 0 (leaf node depth) where we perform the same ray casting as in the discrete integration. The parameters n and d allow us to trade speed against accuracy. The special case $d = 0$ corresponds to the discrete integrator above. In the special case $n = 0$, space is cleared only at depth d and thus some space behind the occupied space is cleared (see Fig. 4(f)).

To compensate for the larger surface[1] where the occupancy value is decreased in the fast discrete integrator; the clearing is reduced by a factor $\frac{1}{2d+1}$. The fast discrete integrator enables mapping applications requiring higher resolution and/or update rate such as exploration and reconstruction (see Section V-B).

The occupied nodes are updated at the highest resolution in all three integrators. Thus, the occupied space will look mostly the same for all three methods.

A comparison of the integrators can be seen in Fig. 4. The effect of them is evaluated in Section V-F.

[1]Boundary surface for the volume spanned by the point cloud. See Fig. 4.

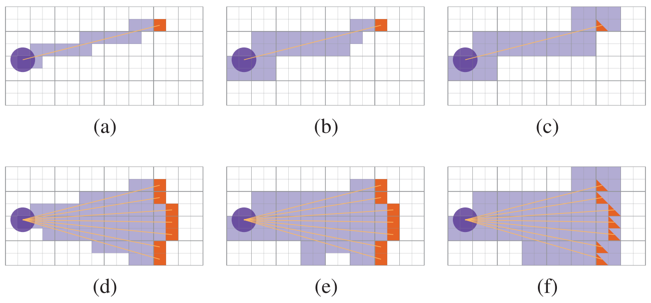

Fig. 4. Comparison of the integrators, mentioned in Section IV-B, for a single ray (top) and a point cloud sensor measurement (bottom). The sensor is located to the left (the circle). The lines are the lines being traced, from sensor to obstacle. The violet and orange cells will be marked free or occupied by the integrator, respectively. (left) Simple/discrete integrator. (middle) Fast discrete integrator with $d = 1$ and $n = 2$. (right) Fast discrete integrator with $d = 1$ and $n = 0$. Note that the occupancy values of the end points will be both reduced, at a lower resolution, and increased, at the leaf level.

C. *Accessing Data*

To access the data in UFOMap, iterators are used. They are fast and they can be used to specify axis-aligned bounding boxes (AABBs), oriented bounding boxes (OBBs), bounding spheres, bounding frustums, or any combination of them. Only AABBs are available in OctoMap. Only the nodes that intersects the bounding volume specified will be retrieved. Iterators can also specify the type of nodes that should be retrieved, e.g., only leaves, occupied, free, unknown, contains occupied/free/unknown, or a combination of them. In comparison, OctoMap always retrieves both free and occupied nodes. It is not possible to get the unknown nodes, and it is not possible to get only occupied or only free nodes, in OctoMap.

Both the bounding volumes and indicators significantly increases the performance of the iterators since whole branches of the octree can be ignored. This is shown in Section V-D.

D. *Availability*

UFOMap is available as a self-contained C++ library at https://github.com/UnknownFreeOccupied/UFOMap. Packages for integration with the Robot Operation System (ROS) [18] are also available. There are functions for reading and writing OctoMap files and converting between UFOMap and OctoMap. This facilitates the transition from OctoMap to UFOMap in an already existing system, and allows to utilize both mapping frameworks for different parts of the same system.

TABLE II
INSERTION TIMINGS ON THE COW DATASET[3]

Method	Voxel size (cm)	Total (ms)	Ray casting (ms)	Insertion (ms)
UFOMap	16	4.98 ± 1.30	4.64 ± 1.08	0.34 ± 0.27
OctoMap		5.52 ± 1.66	4.86 ± 1.18	0.66 ± 0.56
UFOMap	8	12.3 ± 7.4	10.4 ± 5.8	1.9 ± 1.6
OctoMap		16.3 ± 10.4	12.2 ± 6.9	4.1 ± 3.7
UFOMap*		8.1 ± 3.4	6.7 ± 2.3	1.4 ± 1.1
UFOMap†		6.5 ± 2.2	6.0 ± 1.8	0.5 ± 0.4
UFOMap	4	60.9 ± 44.7	46.8 ± 32.7	14.1 ± 12.2
OctoMap		104.6 ± 82.2	71.8 ± 53.0	32.9 ± 30.1
UFOMap*		21.1 ± 12.0	14.3 ± 6.7	6.8 ± 5.4
UFOMap†		10.9 ± 4.7	9.3 ± 3.5	1.6 ± 1.3
UFOMap	2	371 ± 254	264 ± 176	107 ± 79
OctoMap		745 ± 548	521 ± 369	224 ± 188
UFOMap*		74 ± 44	42 ± 22	32 ± 22
UFOMap†		28 ± 15	20 ± 9	9 ± 7

TABLE III
TIME TO DO COLLISION CHECKING FOR UFOMAP AND OCTOMAP

Dataset	Voxel size (cm)	UFOMap (µs/pose)	OctoMap (µs/pose)	UFOMap$^{\text{octo}}$ (µs/pose)
FR-078 tidyup	5	**2.7**	23.1	14.5
FR-079 corridor	5	**3.0**	15.7	10.4
Freiburg campus	20	2.4	1.4	**0.8**
freiburg1_360	2	**3.1**	163.3	121.8
New College	20	2.5	1.4	**0.8**

V. EVALUATION

We compare our proposed mapping framework, UFOMap, against OctoMap when it comes to memory consumption, insertion time, and in three different use cases. The impact of the indicators, i_f and i_u, the thresholds, t_f and t_o, and the accuracy of the different integrators are also evaluated.

A. Memory Consumption and Node Count

In the first experiment, we compare the memory consumption between UFOMap and OctoMap on the OctoMap 3D scan dataset.[2] We analyze the memory usage when the tree has been pruned. Table I shows that UFOMap is around 38% of the size of OctoMap, even though UFOMap contains more nodes in total, as seen in the same table. The increase in the number of nodes is a result of the unknown space being explicitly represented, meaning an inner node must have either 0 or 8 children. The decrease in memory usage is because of i) the smaller data structures for the leaf nodes mentioned in Section IV-A and ii) storing the child nodes directly in the child array of the inner nodes, instead of storing pointers to the children as in OctoMap.

B. Insertion

A point cloud integration time comparison between UFOMap and OctoMap is shown in Table II, using the cow dataset.[3] For both UFOMap and OctoMap the discrete integrator (see Section IV-B2) is used. UFOMap* uses the fast discrete integrator (see Section IV-B3) with $n = 2$ and d corresponding to the depth of the voxels at 16 cm voxel size. UFOMap† uses the fast integrator with $n = 0$ and again d corresponding to the depth of the voxels at 16 cm voxel size.

The total time represents the average time for a single point cloud to be integrated. Ray casting shows the average time per point cloud for doing the ray casting part of the insertion. Insertion shows the average time per point cloud for integrating the points calculated from the ray casting step into the octree. The standard deviation is included for all.

When only looking at the discrete integrator for UFOMap, we see that it is about two times faster at the insertion part of the integration compared to OctoMap. The ray casting is between 1 to 2 times faster in UFOMap, which is, most likely, due to different implementation factors since both utilize the same algorithm. As the voxel size decreases, the difference between the two frameworks increases, in favour of UFOMap. At a voxel size of 4 cm and below, the need for the faster integrators becomes very apparent.

UFOMap* and UFOMap† provides just that, fast integration while still clearing free space. With UFOMap† scaling a lot better with the resolution and being 26 times faster than OctoMap at 2 cm voxel size.

C. Collision Checking

In the first of the use cases we check if the robot, can safely be at a certain position in the map. We thus check that there is no occupied or unknown space in that region. This is an operation that is heavily used in sampling-based motion planners, such as RRTs. By not allowing any unknown space in this region we are more conservative and safe. We sampled 1000000 poses where at least the center of the pose was in free space. For each of the samples we check for collisions in a radius of 25 cm. On average around 50% of the sampled poses were in collision. The results are presented in Table III. We can see that UFOMap allows for faster collision checking than OctoMap in all cases but the ones with large voxels. This is most likely because of the overhead for constructing the iterators in UFOMap in these cases. The last column, UFOMap$^{\text{octo}}$, shows the result when traversing the octree for collisions as in OctoMap in UFOMap. We see that this is faster than OctoMap for all resolutions. However, it is significantly slower than the default method in UFOMap, which better exploits the octree structure, for the higher resolutions.

The time for collision checking changes only marginally with voxel size for UFOMap compared to OctoMap. This can be because of the indicators i_f and i_u (see Section III) together with the iterators (see Section IV-C). Specifying the bounding sphere and that only occupied and unknown voxels should be retrieved, UFOMap can directly move to a octree node that is either occupied or unknown inside the radius.

To check the speedup from explicitly representing unknown space, we also tested against only occupied space. The result for UFOMap was largely unchanged. OctoMap was significantly

[2][Online]. Available: http://ais.informatik.uni-freiburg.de/projects/datasets/octomap

[3][Online]. Available: https://projects.asl.ethz.ch/datasets/doku.php?id=iros2017

TABLE IV
TIME TO DO COLLISION CHECKING ALONG A LINE SEGMENT

Dataset	UFOMap (ms/line)	OctoMap (ms/line)	OctoMap simple (ms/line)	UFOMap No indicators (ms/line)
FR-078 tidyup	**0.100**	8.867	5.161	0.162
FR-079 corridor	**0.149**	15.524	7.510	0.237
Freiburg campus	**0.088**	95.286	13.926	0.115
freiburg1_360	**0.102**	38.679	21.856	0.182
New College	**0.094**	96.438	11.193	0.130

TABLE V
SAME AS TABLE IV EXCEPT ONLY W.R.T. OCCUPIED SPACE

Dataset	UFOMap (ms/line)	OctoMap (ms/line)	OctoMap simple (ms/line)
FR-078 tidyup	**0.074**	1.316	0.308
FR-079 corridor	**0.127**	2.145	0.461
Freiburg campus	**0.111**	6.246	1.332
freiburg1_360	**0.086**	3.390	0.819
New College	**0.102**	3.678	0.725

faster, still always slower than UFOMap, but now at most a factor of 2.4 (compare to Table III).

D. Collision Checking Along a Line Segment

In the second use case we perform collision checking along a line to see if a robot can move from one position to another. For simple RRT path planning, the operations described in Sections V-C and V-D are combined.

As in Section V-C, we are conservative and require that there is no occupied or unknown space along the line. The line is defined by two randomly sampled points, $4m$ apart. A check is first performed to see if the robot can be at these two points using the method mentioned in Section V-C. If not, two new points are sampled.

In the collision checking along the line UFOMap can specify an OBB and use the indicators for traversing the octree. With OctoMap, lacking an explicit representation of unknown space and an efficient way to query cells in an OBB we have to query the octree as if it was a fixed size grid instead. As seen in Table IV UFOMap is between 88 to 1080 times faster than OctoMap. In OctoMap simple, third column, we only check if the distance between the node center of an occupied or unknown node and the line segment is less than the size of the robot. This common simplification speeds up the calculations but leads to around 5% incorrect answers. In the last column of Table IV we performed the same experiment but without the indicators in UFOMap. In this case the indicators provides up to almost a factor of 2 in speedup.

In Table V we present results from collision checking along a line but only considering occupied space. This allows us to isolate the influence of the OBB (see Section IV-C) from the influence of the explicit representation of unknown space. UFOMap is only slightly faster, compared to in Table IV, which means that most of the gain comes from the OBB and the Morton codes. OctoMap shows a speedup of roughly an order of magnitude but it is still more than an order of magnitude slower than UFOMap. In OctoMap we must first use an AABB to get all the leaf nodes and then check all of these against an OBB. In OctoMap simple, the check against the OBB has been replaced by a simpler check, as mentioned earlier. This shows how substantial the OBB calculation is and the benefit of having it directly integrated into the UFOMap framework.

TABLE VI
TIME TO COMPUTE INFORMATION GAIN AT A POSE

Dataset	UFOMap (s)	UFOMap$^{\text{fast}}$ (s)	OctoMap (s)	UFOMap$^{\text{octo}}$ (s)
FR-078 tidyup	1.51 ± 0.85	$\mathbf{0.68 \pm 0.34}$	5.39 ± 0.85	3.25 ± 0.50
FR-079 corridor	1.71 ± 0.82	$\mathbf{0.75 \pm 0.17}$	5.37 ± 1.37	3.24 ± 0.79
Freiburg campus	0.01 ± 0.00	$\mathbf{0.01 \pm 0.00}$	0.04 ± 0.00	0.02 ± 0.00
freiburg1_360	47.2 ± 44.3	$\mathbf{12.4 \pm 12.8}$	183.6 ± 65.4	105.0 ± 36.0
New College	$\mathbf{0.01 \pm 0.00}$	0.01 ± 0.00	0.04 ± 0.00	0.02 ± 0.00

E. Calculate Information Gain

In reconstruction and exploration applications, next-best-view [19] planning is a popular approach. The next-best-view is often obtained by calculating the information gain from being at a specific pose in the map. The information gain is a measure of how much new information can be collected from a certain pose. In exploration, where the goal is to turn each voxel into either occupied or free space, the information gain can simply be how many unknown nodes can be seen from a certain pose. In this experiment we compare the performance when calculating the information gain in UFOMap compared to OctoMap. For the sensor we use a horizontal field of view of 115° and vertical field of view of 60°. The minimum and maximum range of the sensor were set to 0m and 6.5m, respectively.

The results from this experiment can be seen in Table VI. OctoMap and UFOMap$^{\text{octo}}$ compute the information gain similar to how it is done in [6]. Both not exploiting the octree structure.

For UFOMap we exploit the octree structure to quickly find the unknown voxels inside the region of interest, using a frustum bounding volume. For each node found we do ray casting from the sensor to the node, at the same depth in the octree as the node. If the node is not blocked by any occupied space we add it to the total gain, otherwise we recurs down to the node's children and do the ray casting for each child at their depth instead. We do this until the node is either not blocked by an occupied node or until we are at the leaf depth.

UFOMap$^{\text{fast}}$ also exploits the octree structure. For each unknown node found, we recurs down to the leaf nodes right away and do the ray casting. Once a single of the children is not blocked we assume that we can see all children and add all of them to the total gain. Therefore, this approach gives more of an approximate answer than the other.

As seen in Table VI the information gain computation is 1.5 to 15 faster with UFOMap compared to OctoMap, depending on which UFOMap method is used.

F. Effect of Integrators on Mapping

In order to investigate the effect of the integrators, introduced in Section IV-B, we created a Gazebo environment containing a

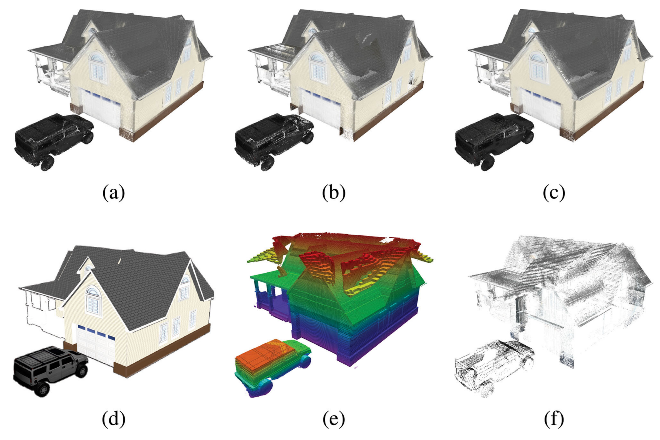

Fig. 5. Comparison of the integrators mentioned in Section IV-B. (a) Discrete integrator. (b) Fast discrete integrator with $d = 3$ and $n = 2$. (c) Fast discrete integrator with $d = 3$ and $n = 0$. (d) Gazebo environment. (e) and (f) shows extra added free space and occupied space, respectively, in (g) compared to (h), bounded around the house and car.

house and a car shown in Fig. 5(d). We let an unmanned aerial vehicle with a RGB-D sensor fly around in the environment at predefined setpoints to map the environment with the different integrators. The final map produced with the discrete integrator is seen in Fig. 5(a). Results with the fast discrete integrator for two settings are shown in Fig. 5(b) ($d = 3$ and $n = 2$) and Fig. 5(c) ($d = 3$ and $n = 0$).

Fig. 5(e) shows the free space that exists in Fig. 5(c) but not in Fig. 5(a), i.e. the extra free space added by the fast discrete integrator with $n = 0$. Since this integrator clears space behind obstacles the outline of the house and car can be seen in the additional free space. Fig. 5(f) shows the occupied space that exists in Fig. 5(c) but not in Fig. 5(a). Since the clearing of free space in scaled down in the fast discrete integrator it contains additional occupied nodes around the surfaces.

The comparisons above compared the integrators against each other, but not against ground truth. The result of mapping depends on, e.g., the sensor, sampled view points, sensor positioning accuracy and the state the of the world in unseen volumes. Defining the ground truth for the map is thus not well-defined. In this letter there is no sensor noise nor sensor positioning error. The RGB-D point clouds themselves are thus as close to perfect samples of the ground truth as we get. We build a ground truth octree model by merging all point clouds using the ground truth sensor positions. A voxel is classified as occupied if at least one point falls in it.

For the discrete integrator 1698999 of the voxels matched as being occupied, while 455068 voxels were misclassified as being free. The corresponding values for the fast discrete integrator with $n = 2$ were 1441556 and 712521 respectively. Lastly, the fast discrete integrator with $n = 0$ resulted in 1746293 and 407784 respectively.

We can conclude that the fast discrete integrator with $n = 0$ performs similar to the discrete integrator, while being a lot faster. Using the fast discrete integrator with something other than $n = 0$ is discouraged, since it performed worse in every situation tested. For these experiments we used the same parameter values for all methods. Some improvements for $n \neq 0$ might be achievable by tuning.

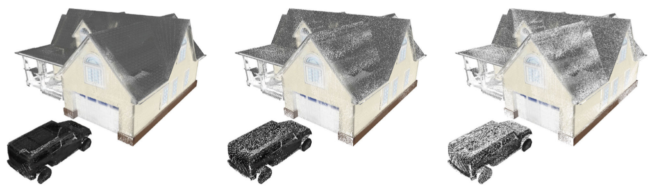

Fig. 6. Resulting map for different t_o, mentioned in Section III. For all three cases t_f was set to 0.5.

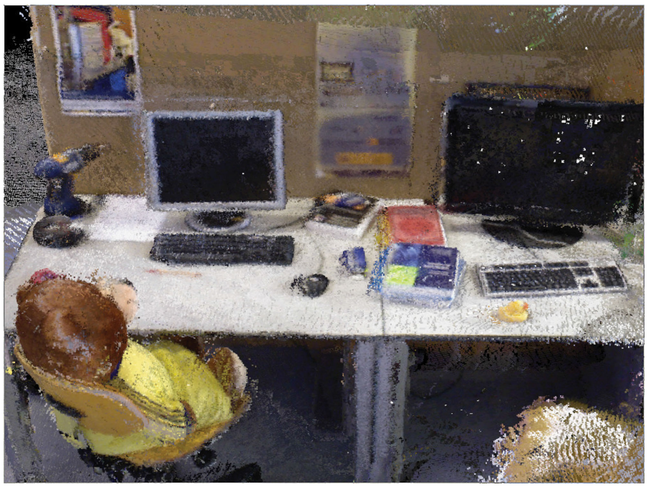

Fig. 7. UFOMap real-time volumetric color mapping at 4 mm voxel size.

G. Effect of the Additional Threshold t_f

Lastly, the effect of the threshold t_f is investigated. As mentioned in Section III the additional threshold allows for classifying the occupancy value into unknown, free, and occupied. Fig. 6 illustrates that by increasing t_o, but keeping t_f, constant we can see which part of the environment needs to be further updated in order to have a map with high probability in the occupied regions. If the same change to t_o would have been done in OctoMap all the nodes that disappear in Fig. 6(b) and Fig. 6(c) compared to Fig. 6(a) would be classified as free. UFOMap classify them as unknown, since t_f has not changed.

VI. Case Studies

A. Real-Time Volumetric Mapping

The first case compares the volumetric mapping performance on the freiburg3_long_office_household sequence from [20], a 86 s office sequence with point cloud data at 2 Hz.

UFOMap managed to incorporate all point clouds and create a 3D map with 4 mm voxel size with color information at 2 Hz, seen in Fig. 7. Without color, 2 mm voxel size without missing any of the point clouds was the limit for UFOMap. In both cases the fast discrete integrator was used, mentioned in Section IV-B3, with $n = 0$ and $d = 4$. OctoMap was only able

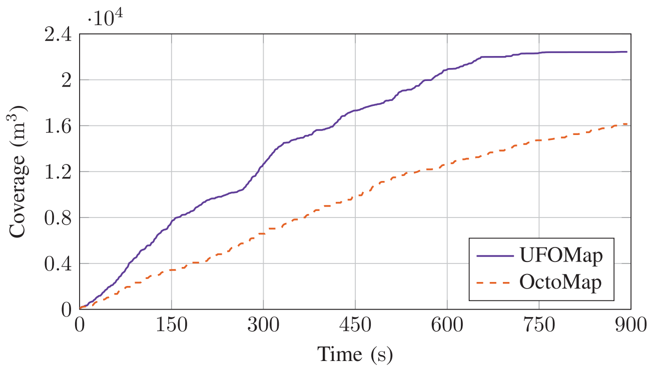

Fig. 8. Comparision of exploration progress between UFOMap and OctoMap in the power plant scenario.

to process all point clouds with 3 cm voxel size and without color.

B. Exploration

In the second case, we compare UFOMap with OctoMap in a next-best-view exploration scenario. We have chosen this scenario since it incorporates all of the above use cases in a realistic setting. We have chosen to use the receding horizon next-best-view exploration method, proposed in [6], for this comparison. For each node that is being sampled for the RRT, there is a check if the node is in free space. There is also a check if the path between the newly sampled node and prospective parent node is clear. Lastly, when selecting where to move next we calculate a score for each node in the RRT. This score depends on the distance and the information gain along the branch to the node.

In UFOMap, as the exploration proceeds the benefit of the proposed approach increases. As more of the space gets classified as free space, all of the above calculations are accelerated in UFOMap compared to OctoMap. This means that when the environment is almost fully explored you can discard nodes that give very little new information quickly. This shows that UFOMap is especially well suited for exploration, compared to OctoMap where the necessary calculations are not affected to the same degree by how much is explored.

The power plant scenario from the Gazebo model library was used for the exploration test, seen in Fig. 1. 16 cm voxel size was used. As can be seen in Fig. 8, switching from OctoMap to UFOMap makes a significant difference to the exploration rate. UFOMap managed to finish the exploration after around 650s, while OctoMap had completed 70% of the exploration after 900s, which was the maximum allowed time.

VII. Conclusion

We present UFOMap, an open source framework for 3D mapping. UFOMap was built on OctoMap, which is one of the basic building blocks in a number of different robotics applications. Just like OctoMap, the underlying data structure is an octree. OctoMap only explicitly models occupied and free space, while UFOMap explicitly models unknown, free, and occupied space. This representation, together with the fact that every node in the octree stores indicators for what kind of space their children contains, results in a significant performance boost compared to OctoMap for use cases where unknown space is extensively used.

Along with these improvements, we introduce new ways of integrating data into the octree. We show that this leads to significant reductions in the time to insert new measurement data, such as point clouds, into the map.

The UFOMap mapping framework is freely available at https://github.com/UnknownFreeOccupied/UFOMap and can be easily integrated with robotics systems. It is written in C++ and can be run as a standalone package or integrated into ROS [18].

References

[1] M. Hebert, C. Caillas, E. Krotkov, I. S. Kweon, and T. Kanade, "Terrain mapping for a roving planetary explorer," in *Proc. IEEE Int. Conf. Robot. Autom.*, 1989, pp. 997–1002.

[2] R. Triebel, P. Pfaff, and W. Burgard, "Multi-level surface maps for outdoor terrain mapping and loop closing," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst.*, 2006, pp. 2276–2282.

[3] D. Meagher, "Geometric modeling using octree encoding," *Comput. Graph. Image Process.*, vol. 19, no. 2, pp. 129–147, 1982.

[4] H. Oleynikova, A. Millane, Z. Taylor, E. Galceran, J. Nieto, and R. Siegwart, "Signed distance fields: A natural representation for both mapping and planning," in *Robot.: Sci. Syst.*, 2016.

[5] 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, pp. 189–206, 2013.

[6] A. Bircher, M. Kamel, K. Alexis, H. Oleynikova, and R. Siegwart, "Receding horizon "Next-Best-View" planner for 3D exploration," in *Proc. IEEE Int. Conf. Robot. Autom.*, 2016, pp. 1462–1468.

[7] M. Selin, M. Tiger, D. Duberg, F. Heintz, and P. Jensfelt, "Efficient autonomous exploration planning of large-scale 3-D-environments," *IEEE Robot. Autom. Lett.*, vol. 4, no. 2, pp. 1699–1706, Apr. 2019.

[8] F. S. Barbosa, D. Duberg, P. Jensfelt, and J. Tumova, "Guiding autonomous exploration with signal temporal logic," *IEEE Robot. Autom. Lett.*, vol. 4, no. 4, pp. 3332–3339, Oct. 2019.

[9] L. Schmid, M. Pantic, R. Khanna, L. Ott, R. Siegwart, and J. Nieto, "An efficient sampling-based method for online informative path planning in unknown environments," *IEEE Robot. Autom. Lett.*, vol. 5, no. 2, pp. 1500–1507, Apr. 2020.

[10] M. Pivtoraiko, D. Mellinger, and V. Kumar, "Incremental micro-UAV motion replanning for exploring unknown environments," in *Proc. IEEE Int. Conf. Robot. Autom.*, 2013, pp. 2452–2458.

[11] J. Chen, T. Liu, and S. Shen, "Online generation of collision-free trajectories for quadrotor flight in unknown cluttered environments," in *Proc. IEEE Int. Conf. Robot. Autom.*, 2016, pp. 1476–1483.

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

[13] M. Nießner, M. Zollhöfer, S. Izadi, and M. Stamminger, "Real-time 3D reconstruction at scale using voxel hashing," *ACM Trans. Graph.*, vol. 32, no. 6, pp. 169-1–169-11, 2013.

[14] H. Oleynikova, Z. Taylor, M. Fehr, R. Siegwart, and J. Nieto, "Voxblox: Incremental 3D Euclidean signed distance fields for on-board MAV planning," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst.*, 2017, pp. 1366–1373.

[15] G. M. Morton, "A computer oriented geodetic data base and a new technique in file sequencing," Ottawa, Canada: IBM Ltd., Tech. Rep., 1966.

[16] L. Stocco and G. Schrack, "Integer dilation and contraction for quadtrees and octrees," in *Proc. IEEE Pacific Rim Conf. Commun., Comput., Signal Process.*, 1995, pp. 426–428.

[17] J. Amanatides and A. Woo, "A fast voxel traversal algorithm for ray tracing," in *Eurographics*, vol. 87, no. 3, pp. 3–10, 1987.

[18] M. Quigley *et al.*, "ROS: An open-source robot operating system," in *Proc. ICRA Workshop Open Source Softw.*, 2009, pp. 1–6.

[19] C. I. Connolly, "The determination of next best views," in *Proc. IEEE Int. Conf. Robot. Autom.*, 1985, pp. 432–435.

[20] J. Sturm, N. Engelhard, F. Endres, W. Burgard, and D. Cremers, "A benchmark for the evaluation of RGB-D SLAM systems," in *IEEE/RSJ Int. Conf. Intell. Robots Syst.*, 2012, pp. 573–580.