Optimized Octtree Datastructure and Access Methods for 3D Mapping

Jann Poppinga, Max Pfingsthorn, Sören Schwertfeger, Kaustubh Pathak and Andreas Birk
School of Engineering and Science
Electrical Engineering and Computer Science (EECS)
Jacobs University Bremen
Campus Ring 1, D-28759 Bremen, Germany
a.birk@jacobs-university.de, http://robotics.jacobs-university.de

Abstract — 3D maps are becoming increasingly important for robots operating in real world scenarios. As occupancy grids are a popular standard in 2D mapping, it is a natural choice to extend them to spatial representation in 3D. But this approach suffers from large memory requirements that render 3D grids unfeasible for realistic mapping applications. A well-known alternative is quadtrees in 2D, respectively octtrees for 3D, where cells that contain identical information are collapsed into a single node in a tree. But octtrees only allow relative slow access to the spatial information stored in them. Here, a very efficient implementation of an octtree is presented. It is based on technical optimizations as well as newly introduced heuristics like the exploitation of typical access patterns. It is shown that the optimizations are indeed beneficial through experiments with real world 3D sensor data. In addition to basic access methods, a very fast nearest neighbor operation - as needed for example for registration in SLAM or map merging algorithms - for octtrees is presented. It is shown in experiments with real world data that this nearest neighbor operation outperforms a comparable operation on occupancy grids by far.

Keywords: *3D Mapping; Spatial Representation; Range Sensing; Safety, Security, and Rescue Robotics (SSRR)*

I. Introduction

3D range sensors are of particular interest for rescue robotics. In this domain, robots have to operate in complex, unstructured environments, which can not be reliably covered by standard range sensors that work best in normal environments with many walls, i.e., plain, perpendicular surfaces. In addition to providing local 3D range data to an operator, e.g., to asses the size of an opening, 3D range sensors can be used for advanced perception and modeling. One important application is 3D mapping, which is not only of obvious interest to Safety, Security, and Rescue Robotics (SSRR), but also of increasing importance to the whole mobile robotics community [1], [2], [3], [4], [5]. Further important applications are terrain classification to detect drivable areas, e.g., to assist a human operator or even for autonomous mobility, as well as map annotation [6], [7], [8], [9], [10].

For 2D maps, occupancy grids are the predominant form of representation [11] and they are hence also a natural candidate to be extended to the 3D case. An alternative option are octtrees, which provide in principle the same access operations as 3D grids. The big advantage of the octtree data structure is that it can store spatial occupancy information in a more memory-efficient manner than grids [12], [13]. The disadvantages are the computationally more expensive access operations, i.e., reads from and stores to a cell. Here, a special implementation is presented, which allows very fast cell access. First of all, an optimized coding using bit-arithmetics is used. Second, a heuristic is introduced, which exploits the fact that consecutive accesses are often to spatially adjacent cells. The benefits are demonstrated in experiments using real world 3D sensor data.

A further contribution is made by introducing an extremely fast nearest neighbor operation, i.e., an operation that given a cell finds the closest cell with same properties - like occupied or not - in a second dataset. According operations are very important for registration, for example for SLAM as well as map merging algorithms. As shown through exhaustive experiments with real world data, the operation is up to several orders of magnitude faster than its counterpart on a 3D grid.

II. A Fast Octtree Implementation

The well known octtree data structure is a means of storing spatial occupancy information very memory-efficiently, especially more efficiently than in the default solution, a 3D grid, which also occasionally called regular mesh. More precisely, we are interested in a datastructure where for every cell (x, y, z), a value $occ \in \{occupied, free\}$ has to be stored. The word "octtree" is formed from "oct" (short for "octant") and "tree". As the name suggests, information is stored in a tree. An octtree is able to *collapse* nodes. This happens when all sibling nodes are either "occupied" or "free". Then, this node will be represented in the parent by a single value, saving memory.

The **`FastOctTree`** (FOT) introduced here is an implementation of an octtree optimized for speed. The key design feature is the distinction between leafs of the octtree and leafs of the FOT: The latter store the values for eight of the former in a bitmap, thus saving one level of nodes. **`FastOctTreeNode`** is used for the nodes as well as for the leaves. It only has a single one-word member, a union. This union can either be used as a pointer or as a bitmap. As a pointer, it points to a struct containing all children as member variables. As a bitmap, it stores the occupancy of its

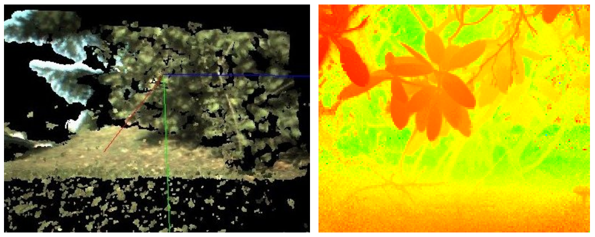


Fig. 1. Examples of 3D data delivered by the stereo camera (left) and the Swiss ranger (right).

children. Which role a node plays depends on whether it is a leaf or not, which can be determined by the value of the least significant bit of the union. As an inner node the variable is a pointer. The class **`FastOctTreeNode`** is 4 byte memory aligned, thus ensuring that at least the two least significant bits of pointers to objects of this class are always 0. For FOT leafs, on the other hand, the union stores a bitmap. As an inner octtree node always has eight children, only the most significant byte is used. The least significant bit is always set to 1.

The class **`FastOctTree`** manages the tree of **`FastOctTreeNode`**s. No coordinates and edge lengths are stored in the nodes, these have to be maintained by this class. Both iterative and recursive tree traversals are used, depending on the task at hand.

In addition to these technical optimizations in the implementation, a special conceptual optimization is introduced, namely the so-called *LineBurstAccess*. This heuristic starts the traversal to access a specific node from the last visited one. This strategy is more efficient than the default to start from the root as geometrically close points are usually consecutively accessed. As demonstrated in the experiments later on, this optimization is especially beneficial for typical mapping applications.

III. Performance Comparison

In this section we benchmark the performance of our optimized octtree implementation for core operations, i.e., read and store of cell values. One of the optimizations is the dual use of its single 1-word-variable as either a pointer to the array of its children or as a bitmap storing the occupancy of its children if the node is collapsed. For the comparison, an equally heavily optimized implementation of a 3D occupancy grid was realized. This class is called **`FastOccMesh`**. We implemented a one-dimensional array to hold all cells of the 3D grid. Each grid cell is exactly one bit in the array.

For the benchmark experiments, extensive 3D real world data sets are used. The data was collected outdoor with a Jacobs rugbot - from "rugged robot" [14]. The 3D sensors used for this purpose are a the SwissRanger SR-3000 and the Videre Stereo-on-a-chip (STOC). The SR-3000 is a time-of-flight camera, which uses modulated near-IR light to attribute a distance value to each of its ~25,000 pixels. The STOC is a stereo camera which generates distance information from difference of the features in the images of its two embedded

TABLE I

Properties of the two computers used in the test

Feature	Robot PC	Office PC
Processor type	Pentium M	Pentium 4 HT
Clock speed	1.6 GHz	3.0 GHz
Front side bus speed	400 MHz	800 MHz
Main memory	0.5 GB	1 GB
L2 cache	2 MB	2 MB
L1 cache	32 kB	16 kB

cameras. All computation is done in hardware, and so the STOC also directly delivers a 3D point cloud. Example data is shown in Figure 1.

The data sets for the experiments presented here were collected in nine different outdoor scenarios: *bush*, *car*, unobstructed *concrete*, *grass*, *hill*, *rubble*, grass and bushes while the robot was *moving*, and *tree*. Each scenario consists of several datasets with a point cloud each. In each scenario, a number (5 – 200) of point clouds for each of the sensors was produced. The point clouds were inserted into the two data structures as follows. First, only the point clouds themselves were entered. To make the measurement more exact, each point cloud was not only entered once, but 20 times. In a second experiment, a line was drawn to the origin for every point of a point cloud, i.e., a ray-tracing operation to determine free space was conducted. Note that this is a typical computation for mapping applications. In each test run, both data structures were kept in memory and each point cloud was entered into both in random order.

The program has been compiled with `g++` with optimization option `-O6`. The main part of the computation was done on the on-board robot PC. For comparison, some tests were also run on an office PC as noted later on. The properties of the PCs are listed in Table I.

The mapped area covered $1024 \times 1024 \times 1024$ cells. This relatively small resolution is due to restrictions of the two data structures. On the one hand, the edge length of the mapped cube has to be a power of two for the octtree. On the other hand, the 3D grid consumes a lot of memory. 1024 was the greatest power of two such that a 3D grid of that size (equivalent to 128 MB) would fit into main memory. Note that this is already a knock-out criterion for the use of 3D grids for many realistic mapping applications. The octtree in contrast does not suffer from these large memory requirements as also shown in the following experiments.

A. Results for Cell-Access

The results for raw point cloud data are shown in Table II. One interesting fact is that the speed advantage for the occupancy 3D grid is considerably higher for TOF data. This is due to the structure of the stereo data and the LineBurstAccess heuristic of entering consecutive points into the octtree (see Section II). Usually, the stereo camera distance image is not dense but consists of patches of very similar distances where features have been found and of gradients of distances along the edges of large objects. Furthermore, errors produced by the

TABLE II

PERFORMANCE OF BASIC ACCESS METHODS. TIME IS AVERAGED FOR EACH POINT CLOUD AND MEASURED IN CLOCK CYCLES.

Scene	Stereo camera data				TOF camera data			
	time		$\#_P$	$\#_{pc}$	time		$\#_P$	$\#_{pc}$
	FT	GR			FT	GR		
Bush	187,843	95,686	2,649,956	51	109,423	43,654	1,279,417	52
Bush 2	84,412	40,294	758,821	34	81,714	34,286	634,258	35
Car	99,773	66,364	1,724,173	44	74,894	30,638	818,429	47
Concrete	29,846	21,692	865,453	65	46,620	19,718	807,431	71
Grass	36,000	22,000	59,607	5	53,151	23,699	1,008,215	73
Hill	375,192	198,077	5,374,168	52	38,387	13,226	447,213	62
Rubble	28,039	24,188	714,171	51	59,474	25,790	853,781	57
Moving	129,500	73,833	6,882,265	180	61,100	20,900	2,351,969	200
Tree	31,800	16,800	464,843	50	86,226	40,000	1,246,244	53
Average speed advantage factor of the 3D grid (GR) over the fast octtree (FT):								
	1.697				2.470			

$\#_P$: total number of points; $\#_{pc}$: number of point clouds for the scene

TABLE III

THE RESULTS FOR THE RAY-TRACED DATA. TIME AND WRITE OPERATIONS ARE AVERAGED FOR EACH POINT CLOUD, TIME IS IN CLOCK TICKS.

Scene	Stereo camera data				TOF camera data			
	time		$\mathcal{S}$	$\#_w$ $(\times 10^6)$	time		$\mathcal{S}$	$\#_w$ $(\times 10^6)$
	FT	GR			FT	GR		
Bush	564,314	192,941	2.925	7.255	149,808	60,769	2.465	1.828
Bush 2	703,824	292,941	2.403	13.744	124,571	53,143	2.344	1.522
Car	1,826,140	960,000	1.976	46.954	104,894	38,723	2.709	1.054
Concrete	1,096,770	532,462	2.060	28.121	57,042	23,239	2.454	.666
Grass	536,000	240,000	2.233	11.699	68,630	27,671	2.480	.779
Hill	1,227,880	421,923	2.910	16.127	59,032	20,000	2.951	.499
Rubble	770,000	279,000	2.760	13.500	67,543	29,825	2.265	.787
Running	876,889	425,444	2.061	19.212	78,700	38,950	2.021	.886
Tree	125,600	37,000	3.395	1.331	84,340	34,717	2.429	.918
average			2.517				2.458	

$\mathcal{S} = \Delta t_{FT} / \Delta t_{GR}$

stereo camera often take the form of a straight line from the origin. The **FastOctTree** exploits this circumstance with the LineBurstAccess.

In Figure 2, the execution time is plotted against the average number of points per point cloud. It becomes clear that the time needed linearly depends on the number of points and that it increases faster for the octtree.

In addition to raw point cloud data, ray tracing from the camera origin to each point representing occupancy is used. This computation is needed to determine the free space in mapping applications. The results of the ray-traced point clouds are shown in Table III and Figure 3. Again, the 3D occupancy grid is faster than the octtree. However, there are no observable differences between the two sensors other than the delivered number of points. Also, the constant factor for the linear depence is equal for both storage methods. This is due the fact that many straight lines are traced, for which the LineBurstAccess-heuristic is especially beneficial. Furthermore, they result in a high structual similarity between the data from the two sensors.

TABLE IV

THE OCTTREE'S SIZE IN KB AND THE RELATED NUMBER OF POINTS

Scene	stereo camera		TOF camera		ray-traced size	
	size	$\#_P$	size	$\#_P$	stereo c.	TOF c.
Bush	121	2,649,956	125	1,279,427	1050	105
Bush 2	110	758,821	190	634,258	1680	385
Car	178	1,724,173	297	818,429	1199	923
Concrete	34	865,453	106	807,431	2170	454
Grass	45	59,607	164	1,008,215	1683	712
Hill	325	5,374,168	584	447,213	1193	1910
Rubble	307	714,171	153	853,781	1353	637
Moving	2570	6,882,265	3210	2,351,969	5603	4544
Tree	48	464,843	136	1,246,244	3768	650

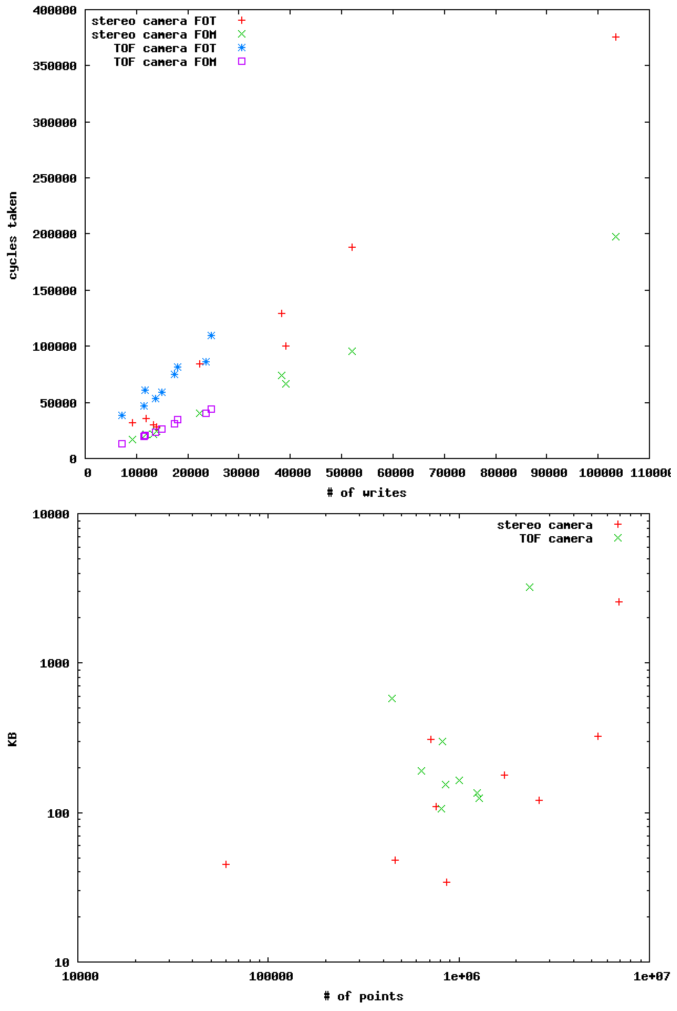


Fig. 2. Results for the raw 3D data in terms of runtime (top) and memory size (bottom). Note the logscale on both axes in the graph for the memory usage.

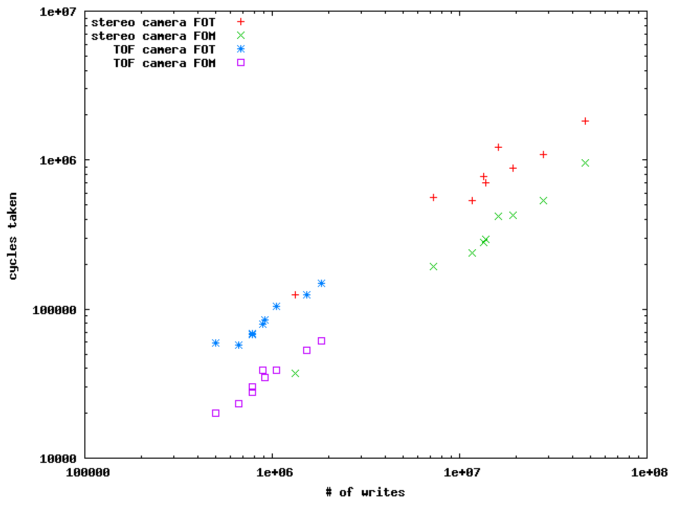


Fig. 3. Performance on the ray-traced data; note the log scale on both axes.

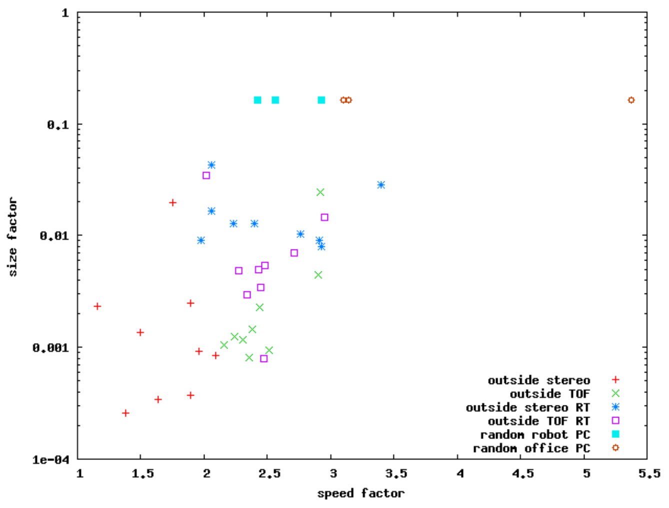


Fig. 4. Comparison between 3D grid and octtree (logarithmic y-axis)

The size of the octtree in the experiments with real-world data is given in Table IV. For comparison, the size of the 3D grid was always 131,072 KB. Although the FOT is never faster than or as fast as the fast grid implementation, the difference in speed is not as high as one might have expected. This is to some extent due to the fact that the FOT can be kept in cache memory because of its small size, whereas the grid has to be swapped to main memory. Figure 4 summarizes all data in this section. Note the significant difference in size. While a 3D grid's size linearly depends on the size of the mappable area, the size of an octtree (almost) entirely depends on the data entered into the map. Consequently, the 3D grid put the limits on resolution/mappable area in this test. Another advantage is that the size and position of the octtree is dynamic. They do not have to be fixed beforehand, e.g. by specifying that the robot always starts at the origin, which might render large parts of the map useless if it only moves in one direction. Furthermore, the map will only be as big as required by the mission, not using up valuable memory. For an octtree, the operations of resizing and repositioning easily and effortlessly fit in with the standard operations. For a 3D grid, they are very expensive. Last but not least, the LineBurstAccess heuristic plays a significant role. This holds especially for typical mapping related access patterns like consecutive storage of range image from typical 3D sensors and even more with respect to ray-tracing operations to determine free space.

IV. Nearest Neighbor Operation on Octtrees

Many important map-related algorithms like various approaches to SLAM or map merging require the computation of nearest neighbor operations. This means that given a particular cell in one data set, the closest cell with the same property must be determined in a second data set, such that a distance metric is minimized. One example is the distance function ψ, which is based on accumulated minimal Manhattan distances between cells in the two data sets that share the same

```
Algorithm IV.1: NearestTreeNeighbor($node$)

procedure NearestTreeNeighbor($node$)
 if $node.hasChildren()$ and not at maximum depth
   then { $node \leftarrow getNextChild(node, point)$
        { NearestTreeNeighbor($node, point$)
   else $upperBound \leftarrow 4 \cdot edgeLength(node)$
 SearchChilden($node, upperBound$)

procedure SearchChilden($node, upperBound$)
 if not $node.hasChildren()$
   then { $upperBound \leftarrow$ manhattan distance to filled cell closest to$point$
        { $nearestPoint \leftarrow$ said cell
   else if any point within this node falls within current upper bound
   then { $children =$ list of children, ordered by distance to point
        { for all $child \in children$
        { SearchChildren($child, upperBound$)
```

properties [15]. For each cell in a grid P_1 with a certain value v, ψ finds the cell in grid P_2, which has the same value and the smallest Manhattan distance to the cell in P_1. This is done for all cells and the distances are accumulated. For symmetry of the ψ metric, the same is done in reverse, meaning starting with all pixels in grid P_2. The two results are then added.

$$\psi(P_1, P_2) = \sum_{v \in V} d(P_1, P_2, v) + d(P_2, P_1, v) \quad (1)$$

$$d(P_1, P_2, v) = \sum_{P_1(p_1)=v} min\{d_{manhattan}(p_1, p_2) | P_2(p_2) = v\} \quad (2)$$

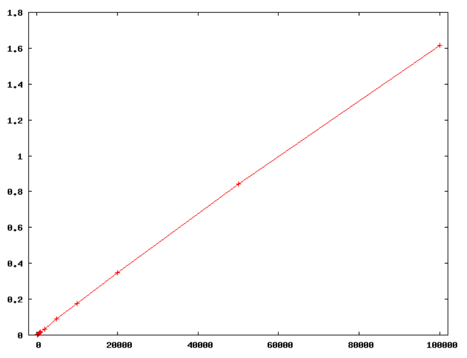
The ψ metric is computed using a lookup table for all coordinates. This table, called a distance map or $dmap_v$, has to be computed for all values v and contains the distances for each coordinate to the nearest pixel with a specific value. The algorithm is linear in the number of cells. Independent of the concrete definition of ψ, its computation on the occupancy grid is typical for this type of problem.

For octtrees, a much more efficient technique can be used. Due to the hierarchical structure of the octree, we can quickly find all filled cells and only compute the nearest neighbor distances for only those points. Therefore, this way to evaluate ψ only depends on the number of filled cells instead of on the region the octree covers. This in itself is a significant performance boost. Furthermore, an algorithmic optimization is introduced, which extents the work of Hoel and Samet on nearest neighbor search on line segments in a quadtree [16]. In a dynamically growing octree, the presence of a node implies that within the boundaries of that node, there is at least one leaf node. Thus, we can immediately deduce an upper bound to the volume we have to search in the tree. Potentially, a lot of subtrees can be pruned from the search that lie completely outside this bound. While searching the nodes which do fall within this bound, we can progressively tighten said bound when we encounter leaf nodes in lower levels. This way we prune even more nodes from the search. Exploiting this property of octrees makes this algorithm very efficient. There are two phases for our recursive implementation of this algorithm: First, finding the octree node from which we want to start the search and finding the upper bound on the subsequent search. Second, we search this node's children and this node's parents children, and so forth, until we reach the root node. During the search, we progressively tighten the upper bound to correspond to the closest filled cell found so far, ensuring the pruning of subtrees that lie outside of the current bound. In the first phase, we first need to find the node which contains the point from where we start our search. In order to do so, we traverse the tree downwards, choosing the child node which contains that point. We do this until there are no more children, or until we reached the maximum depth. We then set the estimate of the upper bound to six times the edge length of the node we are at (six times because we want the Manhattan distance across the parent in 3D). This procedure is implemented recursively and shown in Algorithm IV.1. The upward phase actually performs the search. Going up the stack generated by the recursive downward phase, we search the area around the query point outwards. To ensure this search patter, the children of the current node are sorted by the manhattan distance to that point. In order not to replicate work, we do not search the child of the current node which contains the original point, if there is such a child, since we would have visited it before. The children of the current node are only searched if any of them lie within the current upper bound. This phase is shown in Algorithm IV.1.

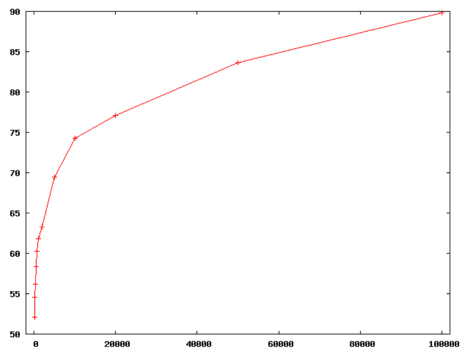
V. Performance of Nearest Tree Neighbor

Both the *nearest tree neighbor* method and the $dmap_v$ method to evaluate ψ are compared in terms of execution speed. The experiment consists of two parts, one focusing on real world sensor data and one focusing on the scalability properties of each method. The real sensor data is the same as used in the performance comparison of the octree implementation itself. The scalability experiment varies the number of points in a random point cloud to understand what the impact is on the execution time. It is important to note the inherent difference between such random and real data. Real data is governed by very complex distributions which can not be accurately modeled with random point clouds. It is thus very interesting, how the methods behave under real world constraints which results in sparse but clustered data. Random data is drawn from a uniform distribution, so we mainly test the efficiency tradeoff between point density and the amount of points in the region.

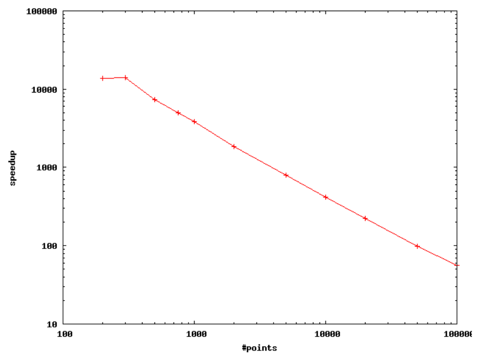
To achieve better timing accuracy, the similarity function was run five times per comparison. This was especially important for the nearest neighbor method since it achieved execution times very close to the granularity of the clock. Each data set produced up to 15 point clouds that were compared. The mean and variance listed in the results are computed over these several comparisons. The range of the point cloud size in the second experiment includes the normal sizes as measured by the two sensors. We generate points clouds from 100 points up to 100000 points. The TOF camera delivers at most 25000 points while the stereo camera can deliver up to 307200 points. However, the stereo camera relies on color patterns in the images, so it will never return as many points. Usually, the stereo camera returns between 300 and 100,000 points. Thus, the range chosen for this experiment covers all important sizes.



(a)



(b)



(c)

Fig. 5. Execution times vs. number of points for the *nearest tree neighbor* method (top). Execution times vs. number of points for $dmap_v$ method (center). Speedup by using *nearest tree neighbor* rather than $dmap_v$ vs. number of points. Note the log scales of the x and y axis (bottom).

The computer used in the experiment contained a Pentium D 2.8GHz dual core CPU with 16KB L1 and 1MB L2 cache per core, and 1GB of RAM. The program has been compiled with GNU `g++` and optimization option `-O6`.

Table V shows the results of the first experiment using real sensor data. The second column shows the average number of voxels in the octree that were filled. It is evident that this number has an impact on the execution time of both methods. In the case of $dmap_v$, this is mainly due to the lookup operations in the octree necessary for the algorithm. The most eye catching result, however, is the significant speedup achieved with the nearest neighbor method. The speed-up is

		Nearest Neighbor		$dmap_v$		
Scene	Voxels	Mean time	Variance	Mean time	Variance	speedup
rubble (St)	13776	0.030200	0.000248	41.569000	0.902541	1376.5
rubble (TOF)	10566	0.016400	0.000002	40.748800	0.013036	2484.7
grass (St)	448	0.001000	0.000002	39.255000	3.080162	39255
grass (TOF)	10594	0.017600	0.000002	40.367800	0.007788	2293.6

TABLE V
RESULTS FOR NEAREST NEIGHBOR.

several orders of magnitude ranging from 1,000 and up to almost 40,000 times faster.

The second experiment included the comparison of varying amounts of random points. Figures 5(a) and 5(b) show the results. The speedups are similarly impressive, ranging from 55 up to 14,000, decreasing with the number of points. This is shown graphically in Figure 5(c). Interestingly, $dmap_v$ scales sublinearly, while *nearest neighbor* scales only linearly in the number of points. Actually, $dmap_v$ should always take exactly the same time because the grid size does not vary. However, it does need to access the octree while it constructs the lookup table, so what is visible in the graph is the sublinear scaling of the octree access. Our previous observation that the *nearest tree neighbor* method only depends on the number of occupied cells is also validated. Additionally, the results from Table V indicate that the *nearest tree neighbor* method can work up to 10 times faster on real data than on random data. Thus, clustering of points and the highly non-uniform distribution of points in real data facilitate an even faster comparison of point data.

VI. CONCLUSION

Several significant optimizations to the well-known octtree for spatial representation were presented. The optimizations consist of several technical feats in the implementation of the data structure itself as well as conceptual improvements for the access methods. The technical implementation aspects include for example the minimized memory size: the optimized nodes just consume 4 bytes of memory; thus many if not all nodes of a fast octtree (FOT) fit into the cache. The conceptual improvements are the LineBurstAccess for basic read/write operations and a special nearest neighbor method. The LineBurstAccess, i.e., exploiting the high likelihood of spatial proximity of consecutive accesses, is a particular useful heuristic for mapping applications. It is especially efficient with storage of range images from typical 3D sensors and in particular in ray-tracing operations as needed for determining free space from range data. This is supported by extensive experiments with real-world 3D data from a stereo camera and a time-of-flight range camera, a so-called Swissranger. Furthermore, a highly efficient nearest neighbor operation for octtrees is presented. This operation performs orders of magnitude faster than standard naive implementations.

ACKNOWLEDGMENTS

The authors gratefully acknowledge the financial support of *Deutsche Forschungsgemeinschaft* (DFG).

Please note the name-change of our institution. The Swiss Jacobs Foundation invests 200 Million Euro in **International University Bremen (IUB)** over a five-year period starting from 2007. To date this is the largest donation ever given in Europe by a private foundation to a science institution. In appreciation of the benefactors and to further promote the university's unique profile in higher education and research, the boards of IUB have decided to change the university's name to **Jacobs University Bremen**. Hence the two different names and abbreviations for the same institution may be found in this article, especially in the references to previously published material.

REFERENCES

[1] A. Howard, D. F. Wolf, and G. S. Sukhatme, "Towards 3d mapping in large urban environments," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Sendai, Japan, 2004.

[2] S. Thrun, D. F. D. Haehnel, M. Montemerlo, R. Triebel, W. Burgard, C. Baker, Z. Omohundro, S. Thayer, and W. Whittaker, "A system for volumetric robotic mapping of abandoned mines," in *Proc. IEEE International Conference on Robotics and Automation (ICRA)*, Taipei, Taiwan, 2003.

[3] D. Hähnel, W. Burgard, and S. Thrun, "Learning compact 3D models of indoor and outdoor environments with a mobile robot," *Robotics and Autonomous Systems*, vol. 44, no. 1, pp. 15–27, 2003.

[4] J. Davison and N. Kita, "3d simultaneous localisation and map-building using active vision for a robot moving on undulating terrain," in *IEEE Conference on Computer Vision and Pattern Recognition*, Hawaii, Dec 8-14, 2001.

[5] Y. Liu, R. Emery, D. Chakrabarti, W. Burgard, and S. Thrun, "Using em to learn 3d models of indoor environments with mobile robots," in *18th Conf. on Machine Learning*, Williams College, 2001.

[6] R. Unnikrishnan and M. Hebert, "Robust extraction of multiple structures from non-uniformly sampled data," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, vol. 2. IEEE Press, 2003, pp. 1322–1329 vol.2.

[7] S. Lacroix, A. Mallet, D. Bonnafous, G. Bauzil, S. Fleury, M. Herrb, and R. Chatila, "Autonomous rover navigation on unknown terrains: Functions and integration," *International Journal of Robotics Research*, vol. 21, no. 10-11, pp. 917–942, 2002.

[8] ——, "Autonomous rover navigation on unknown terrains functions and integration," in *Experimental Robotics Vii*, ser. Lecture Notes in Control and Information Sciences, 2001, vol. 271, pp. 501–510.

[9] S. Thrun, W. Burgard, and D. Fox, "A real-time algorithm for mobile robot mapping with applications to multi-robot and 3d mapping," in *ICRA*, 2000, pp. 321–328.

[10] D. B. Gennery, "Traversability analysis and path planning for a planetary rover," *Autonomous Robots*, vol. 6, no. 2, pp. 131–146, 1999.

[11] S. Thrun, "Proper label: Thrunmapsurvey; robotic mapping: A survey," in *Exploring Artificial Intelligence in the New Millenium*, G. Lakemeyer and B. Nebel, Eds. Morgan Kaufmann, 2002.

[12] C. L. Jackins and S. L. Tanimoto, "Oct-trees and their use in representing three-dimensional objects," *Computer Graphics Image Process*, vol. 14, no. 3, pp. 249–270, 1980.

[13] D. Meagher, "Geometric modelling using octree encoding," *Computer Graphics Image Process*, vol. 19, no. 2, pp. 129–147, 1982.

[14] A. Birk, K. Pathak, S. Schwertfeger, and W. Chonnaparamutt, "The iub rugbot: an intelligent, rugged mobile robot for search and rescue operations," in *IEEE International Workshop on Safety, Security, and Rescue Robotics (SSRR)*. IEEE Press, 2006.

[15] A. Birk, "Learning geometric concepts with an evolutionary algorithm," in *Proc. of The Fifth Annual Conference on Evolutionary Programming*. The MIT Press, Cambridge, 1996.

[16] E. G. Hoel and H. Samet, "Efficient processing of spatial queries in line segment databases," in *Advances in Spatial Databases, Second International Symposium, SSD'91, Zürich, Switzerland, August 28-30, 1991, Proceedings*, ser. Lecture Notes in Computer Science, O. Günther and H.-J. Schek, Eds., vol. 525. Springer, 1991, pp. 237–256.