Efficient Processing of Large 3D Point Clouds

Jan Elseberg, Dorit Borrmann, Andreas Nüchter
School of Engineering and Science, Jacobs University Bremen gGmbH, 28759 Bremen, Germany
Email: {j.elseberg|d.borrmann|a.nuechter}@jacobs-university.de

***Abstract*—Autonomous robots equipped with laser scanners acquire data at an increasingly high rate. Registration, data abstraction and visualization of this data requires the processing of a massive amount of 3D data. The increasing sampling rates make it easy to acquire Billions of spatial data points. This paper presents algorithms and data structures for handling this data. We propose an efficient octree to store and compress 3D data without loss of precision. We demonstrate its usage for fast 3D scan matching and shape detection algorithms. We evaluate our approach using typical data acquired by mobile scanning platforms.**

***Index Terms*—3D Scanning, Data Structures, Octree, RANSAC, Nearest Neighbor Search**

I. Introduction

Laser range scanning provides an efficient way to actively acquire accurate and dense 3D point clouds of object surfaces or environments. 3D point clouds provide a basis for rapid modeling in industrial automation, architecture, agriculture, construction or maintenance of tunnels and mines, facility management, and urban and regional planning. Modern terrestrial and kinematic laser scan systems acquire data at an astonishing rate. For example, the Faro Focus^{3D} delivers up to 976000 3D points per second and the Velodyne HDL-64E yields 1.8 million range measurements per second. Kinematic laser scan systems often use multiple line scanners and also produce a huge amount of 3D data points. A common way to deal with the data is to process only a small subset of it. While this is an acceptable way of handling the data for some applications it calls into question why so many measurements were acquired in the first place.

Using innovations from the computer graphics community, we develop an octree implementation with advantageous properties. First, we prefer a data structure that stores the raw point cloud over a highly approximate voxel representation. While the latter is perfectly justifiable for some use cases, it is incompatible with tasks that require exact point measurements like data visualization and scan matching. Second, the octree ought to be fast, i.e., access, insert and delete operations must be in $O(\log n)$, where n is the number of stored points. Last and most important, the data structure has to be memory efficient. We present an easy to implement octree encoding that fulfills these requirements and is capable of storing one billion points in 8 GB of memory. In principle, it is also possible to employ disk caching with the the data structure, i.e., larger data sets can be streamed from a mass storage device when processing it.

In addition to the octree, this paper presents algorithms exploiting the properties of our data structure: An efficient RANSAC implementation for shape detection and the nearest neighbor search for ICP-based scan matching. Not discussed in this paper is the fast adaptable visualization using frustum culling as in [?]. The presented data structure and algorithms are available as open-source. The octree, as a data structure heavily employed in computer graphics, is also capable of very efficient ray tracing. This is important for occupancy grid mapping, which relies on ray casting for "simulating" laser beams. Even though neighbor pointers are not stored in our particular octree, ray casting can still be done efficiently and with a constant number of floating point operations per ray.

II. State of the Art

Since its introduction in the early 80's by Meagher [12], the octree data structure has experienced widespread use among various fields that deal with large quantities of 3 dimensional data, especially computer graphics [9], [10], [7], [11] but also theoretical physics [1] and, of course, robotics [17].

In computer graphics and visualization the octree has recently had a resurgence under the term *sparse voxel octree*. It is used for efficient ray tracing and casting, since an octree can represent large data sets with a small memory footprint and simultaneously provides ray casting in $O(\log n)$. In this application octrees are representations of a voxel map only, i.e., leaf nodes do not store other data, apart from the properties needed to visualize a voxel such as its color and normal. Laine and Karras [11] have presented an octree encoding for the GPU which is similar to ours. Their implementation also stores contours in each node in addition to the usual color and normal information. Knoll et al. have developed powerful ray tracing algorithms on octrees [9], [10]. They employ the fast indexing scheme as presented by Frisken and Perry [4] and improve upon standard ray traversal with slice-based coherent octree traversal.

Furthermore, octrees are used in the color quantization algorithm as designed by Gervautz and Purgathofer [6] to minimize memory requirements. Each level in their tree represents one bit of the colors contained therein, beginning with the most significant bit in the first level. This is similar to how we propose to compress point clouds by implicitly by storing the most significant bits in our compact octree data structure.

Roboticists have utilized the octree/quadtree data structure mainly for mapping applications. The advantage of efficiently representing uniform space has been most useful for occupancy grid (cf. [3]) based mapping approaches [17], [13].

III. Octrees for storing 3D point clouds

An octree is a tree data structure that is used for indexing three dimensional data. It is the generalization of binary trees and quadtrees, which store one and two dimensional data

respectively. Each node in an octree represents the volume formed by a rectangular cuboid, often, also in our implementation, simplified to an axis aligned cube. This is analogous to representing a line segment, or rectangle for binary and quadtrees. Consequently an octree node has up to eight children, each corresponding to one octant of the overlying cube/node. A node having no children usually implies that the corresponding volume can be uniformly represented, i.e., no further subdivision is necessary to disambiguate. This convention is not completely applicable when storing points, which are dimensionless, i.e., there is no volume associated with them. When storing a point cloud, we must therefore define a stopping rule for occupied volumes. We define both a maximal depth and a minimal number of points as a stopping criteria. If either the maximal depth is exceeded or the number of points is below the given limit leaf nodes, instead of inner nodes, are generated. Defining a maximal depth is equivalent to defining the smallest possible leaf size, also referred to as the voxel size. A list of points is stored in each occupied leaf. By applying two simple criteria we avoid building a perfect octree, i.e., an octree where all leaves are at the same depth and all other nodes have exactly 8 children. First and foremost the uniformity criteria above is applied to volumes without points, such that subdivision is not necessary in empty nodes. In fact, we only create child nodes for octree volumes with points. Nodes without children are empty and represent empty space. Second, we do not subdivide a volume further that contains only a single point. Laser scanners sample only the surface of objects, and usually provide only a single distance measurement per angle pair. This leads to a 3-dimensional point cloud that is not fully volumetric. Consequently, most space is not occupied and most octree nodes will only have few children. The octree data structure is therefore ideally suited to store and retrieve 3D data efficiently.

A. Memory efficient encoding of an octree

We prioritize memory efficiency in our octree implementation. This is not usually done, since the compression achieved by representing uniform subvolumes as a single node is sufficient for many applications. Not subdividing uniform volumes will compress the data to a very large degree as compared to a 3D grid stored as an array.

Many implementations store redundant information in each octree node. In computer graphics, for example, neighbor pointers as well as a parent pointer are used to facilitate extremely fast ray tracing at the cost of additional memory. Another encoding, that redundantly stores the position and size in each node is given in Fig. 1, where `center` and `size` store the position and size of the node while `child` is an array of pointers to the 8 children. This allows to stop subdivision for empty nodes, thus potentially reducing the number of nodes required for storage. On a standard 64-bit architecture, each node requires 100 bytes of memory. This implementation will be the implementation that we will compare against.

We create an efficient octree implementation that is free of redundancies and is nevertheless capable of fast access operations. Serialized pointer-free octrees are the most memory efficient encoding. However, accessing the data structure is then in $O(n)$, where n is the size of the data (number of points/cells). Such a serialization is only useful when storing the data structure for later use or when communicating over channels where bandwidth is an issue. Our implementation allows for access operations in $O(\log n)$. Add and delete operations are also in $O(\log n)$, even though in the worst case longer blocks of memory will have to be allocated or deallocated.

```
struct OcTree {
  float center[3];
  float size[3];
  OcTree *child[8];
  int nr_points;
  float **points;
};
```

Fig. 1. Definition of an octree with redundant information. Each node contains position, size and eight pointers to child nodes. The size of this node in a C/C++ implementation on a 64 bit architecture is at least 100 bytes. Similar implementations are found with $(72 + x)$ bytes in *OctoMap – 3D occupancy mapping* [17], [8] and with $(72 + x)$ bytes in the *Point Cloud Library (PCL ver. 1.1.1)* [15], [14]. There is also a low memory variant of the octree in the PCL with $(25 + x)$ bytes per node. x varies due to the use of C++ templates.

Most information about inner nodes of an octree is computed when recursing through the structure. The depth of a node is calculated as the depth of its parent plus one. Due to the properties of the regular octant subdivisions the size of a node is a function of its depth. Similarly the position of a node is computed by displacing the position of the parent node by half of the cell size in the appropriate direction. In the same manner parent pointers may be computed, or rather remembered, by pushing visited parents onto a stack. It is even possible to compute neighbor pointers by a fast indexing scheme. However, this operation requires limited backtracking along the parent stack so that it is somewhat less efficient.

For the sake of memory efficiency, we omit any information that is computable by traversing the tree. However, referring to the baseline implementation in Fig. 1, the removed redundancy accounts for only 24 of 100 bytes. 64 remain for child pointers and 12 bytes for the point storage. We downsized this by moving the information about whether or not a node exists from the node itself into its parent. We add a single byte, where each bit corresponds to one octant of the node. This allows us to remove the constraint to always store 8 children, so that only those child nodes need to exist that contain valuable information in the first place. We can therefore remove 56 further bytes by storing only a single pointer to all children. Adding another byte, where each bit signals whether the corresponding octant is a leaf allows the removal of the point information that is unnecessary in inner nodes. The resulting encoding is presented in Fig. 2.

Our encoding consists of three parts. The child pointer is the largest part of each node and is implemented as a relative pointer to the first child. All other valid children are arranged linearly in memory as shown in Fig. 2. The pointer can vary in size for different systems. For 64 bit architectures we have

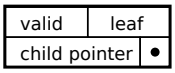

Fig. 2. The proposed encodings of an octree node optimized for memory efficiency. The child pointer as the relative pointer is the largest part of an octree node, but varies in size to accommodate different systems.

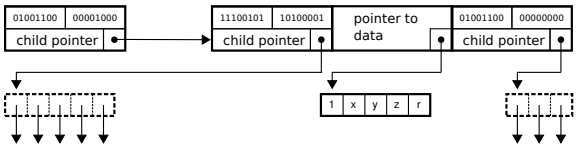

Fig. 3. An example of a simple octree as it is stored using the proposed encoding. The node in the upper left has three valid children, one of which is a leaf. The leaf node is a simple pointer to an array storing both the number of points and their attributes.

chosen 6 bytes. There is no need for an additional bit signaling for a far pointer as proposed in [11], since this is sufficient to address a total of 256 terabyte, Using a far pointer flag would require more sophisticated memory management, but it would enable one to reduce the size of the child pointer to two or fewer bytes. There is a second, easier way to reduce the number of bytes required for the child pointer, if we are willing to sacrifice $O(\log n)$ add and delete operations. In this case, the octree can be stored in a linear array in breadth first order, with each child pointer simply indexing the array. However, further discussion will focus on using this 6-byte pointer implementation.

The attributes `valid` and `leaf` are each a single byte large, one bit for each subvolume. The `valid` bits signal whether the corresponding octant is present, while the `leaf` bits signal whether the corresponding child is a leaf node. This encoding is somewhat redundant, as non-valid children cannot be leaf nodes. There are only $3^8 = 6561$ combinations possible. These could be compressed and represented with only 13 instead of 16 bits. Due to concerns about the runtime efficiency and the relatively minor reduction of the memory requirements, we decided against such a compression. It is possible to remove the leaf byte, by enforcing a constant depth of the octree. This reduces the size of an octree node but at the same time increases the number of nodes to obtain the defined depth. Point clouds acquired by laser scanners are much too sparse for this to still allow for a reduction in overall size.

Our implementation stores points in the leaf nodes, thus they need to be represented differently from inner nodes. In Fig. 3, leaf nodes are pointers to arrays of points. The first entry is always the total number of points, then sequentially the information for each point, i.e., the coordinates and additional attributes such as reflectance. In that representation, leaf nodes would be n bytes larger than inner nodes, where n is the number of bytes used to encode the number of points. In our case it is more than sufficient to reserve $n = 4$ bytes for this purpose. Such a point list representation is then already more memory efficient than the usual `float**`, as it cuts down on a pointer.

B. Octree based compression of 3D point clouds

Our octree encoding drastically decreases the overhead for obtaining the data structure itself (cf. Table I). As opposed to the reference implementation the memory for the point cloud exceeds now the overhead (cf. Fig. 5). Therefore, we seek to compress the point list as well. For a simple technical reason we like to store each point coordinate using only two bytes. Two bytes are exactly the resolution at which most laser scanners measure additional point attributes, such as reflectance, deviation. To store floating point coordinates in only two bytes without significant loss of precision, we use each bit of the two byte coordinate as $s/2^{16}$ increments to the lower left front corner of the rectangular cuboid of the leaf node, where s is the side length of the cuboid. This is similar to color quantization as used for example in [6].

Data of terrestrial laser scanners represented as four byte floating point value has a precision of approx. 100 μm (100 micrometer) at the maximal distance of 500 m. At a smaller distance, e.g., at 1.5 m, the precision increases to 1 μm. To achieve the same 1 μm precision the smallest volume in the octree must have a side length of 6.5 cm. Assuming a desired precision of 10 μm, which is still 2 orders of magnitude smaller than typical specified measurement precisions, the largest node is allowed to have a side length of 65 cm. At this voxel size the octree overhead is minimal even for large scans.

IV. Efficient algorithms on octrees

A. RANSAC for efficient parameter estimation

The Random Sample Consensus (RANSAC) algorithm is an approach for estimating parameters of a model that best describes a set of sample points [5]. While it is traditionally used for line and plane detection RANSAC can be used for any parameterized model. It is an iterative algorithm that repeatedly draws a small number of samples from the data to be modeled. From this subset of samples the parameters of the model are computed and the number of points in the entire data set that intersect with the model is calculated. As the process is repeated the model with the largest number of points is selected as the result.

The most evident drawback of RANSAC is its computational complexity. The number of iterations required to detect a model with any certainty can easily reach impracticable levels. In addition, verifying a model on a large data set is itself a time consuming task. For shape detection, the octree offers ways to mitigate both problems to a drastic degree. Schnabel et al. [16] have shown that by carefully selecting samples that are in proximity to each other, the number of iterations required to detect a shape with a certain probability can be reduced by several orders of magnitude. This is done by first selecting a sample in a random leaf l, and then selecting further samples only from children of a randomly selected parent node of l.

In an unorganized point cloud, determining the number of points that agree with the candidate model requires iterating over all points. We employ the octree data structure for a significant speedup of this step. The candidate model is recursively intersected with the octree nodes to determine which nodes may contain points on the model. The process is depicted in 2D in Fig. 4. After a candidate line has been

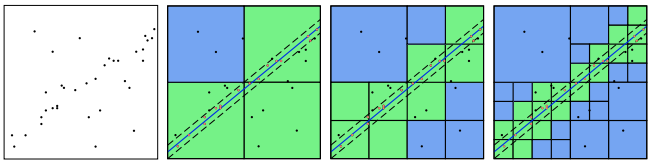

Fig. 4. How an octree can speed up RANSAC. Top left: The initial sample set in which a plane should be detected. A line has been generated from a subsample and is intersected with the octree. The dashed lines signify the maximal distance threshold of RANSAC. Nodes colored in blue are outside of the model, green nodes intersect the line. The intersection test continues only in the children of the nodes that are known to intersect the model.

generated, intersection tests are performed on the children of nodes known to be at least partially on the line. This process is executed from top to bottom, until the points are counted in the leaves. In this way a large number of points are automatically excluded from being checked against the model thus leading to a massive speedup.

B. Nearest neighbor search for scan matching

Nearest neighbor search (NNS) is a part of many scan matching algorithms for establishing corresponding points. The most prominent example of this is the Iterative Closest Point (ICP) algorithm, which spends most of its processing time in the lookup of closest points. It is therefore of utmost importance for this application to reduce the computing time of this task. Very naively implemented, finding a closest point requires iterating over the entire data set, i.e., it is in $O(n)$, where n is the number of points in the point cloud. This expensive running time is avoided by employing metric data structures. By far the most popular data structure to speed up NNS in scan matching is the k-d tree. Since k-d trees are binary trees they allow for an efficient implementation of NNS. Principally, octrees should allow for the same efficiency. In fact, due to the regular subdivision in an octree it ought to be better suited for NNS than the k-d tree. The complication arises during the implementation of NNS on an octree. The key to an efficient traversal to the node containing the nearest neighbor for both tree variants is the order in which children are visited. The number of nodes that we need to visit is best reduced by the closest child first criteria, i.e. the order of traversal is determined by the distance to the query point. This is trivial to do for the binary k-d tree, but somewhat more involved for an octree which may have one to eight child nodes.

For any octree node with 8 children there is a total of 48 possible sequences in which to traverse the children. Every child corresponds to an octant of the entire coordinate space. The query point may fall into any of those 8 octants. For each of those cases there are 6 possible traversals determined by the order of proximity of the query point to the 3 split planes. Therefore, NNS in an octree has to make proximity checks to 3 split planes, sort them and select the appropriate sequence of traversal for a closest child first search for every traversed node. Compared to this, the order of traversal in a k-d tree is instantly determined by a single proximity check,

Algorithm 1 FindClosest

Input: query point q, maximal allowed distance d
lookup deepest node N containing bounding box of q
convert q to octree coordinate i
return FindClosestInNode(N, q, i, d)

Algorithm 2 FindClosestInNode

Input: query point q and its coordinate i
Input: maximal allowed distance d and the current node N
compute child index c from i
for $j = 0$ to 8 **do**
  get next closest child $C = N$.children[`sequence[c][j]`]
  if C is outside of bounding ball **then**
    return currently closest point p
  else
    if C is a leaf **then**
      FindClosestInLeaf(C, q, d)
      update currently closest point p
    else
      FindClosestInNode(C, q, i, d)
      update currently closest point p
    end if
  end if
end for
return currently closest point p

thereby avoiding unnecessary computations if nodes need not be visited.

However, the regular subdivisions of an octree can still be leveraged for an NNS that is in most cases faster or as fast as on a k-d tree. The biggest benefit is that fast indexing is possible in an octree. Any real-valued point coordinate can immediately be converted to (x, y, z) integer coordinates valid on the deepest level of the octree. Using these integer coordinates, it is possible to very efficiently traverse the octree using only bit operations as explained in [4].

This allows us to directly traverse to the deepest octree node, which contains the bounding sphere of the query point, with a constant number of floating point operations. The full NNS with closest child first and backtracking is then performed on this node. The initial lookup for finding the deepest octree node that contains the bounding box around the query point is a modified indexed lookup. For this purpose the two diametrically opposed corners of the bounding box are converted into integer coordinates. The tree is then traversed until both indices disagree as to what child is to be traversed next.

The initial lookup is considerably faster than the equivalent operation in a k-d tree, which is essentially a lookup of a point already in the tree. However, the speedup gained by this is clearly dependent on the maximal allowed distance to the query point. The smaller this distance is, the deeper the node enclosing the bounding sphere is on average. The deeper said node is, the fewer steps need to be performed in the following NNS.

For ease of implementation and to further reduce the number of floating point operations, we only employ 8 orders of traversal instead of all 48. Since the order of traversal can now

be decided by the octant into which the query point falls, there is no need for proximity checks or sorting. Consequently, no floating point operations are required in our NNS implementation except in the leaves of the octree, where stored points are checked against the query point. The approach is summarized in Algorithm 1. We use the function `FindClosestInLeaf`, which is a trivial check of the points stored in the leaf.

V. Experiments and results

The following experiments were conducted using the data sets displayed in Fig. 5. Data sets are representative of their corresponding domain, each with a different point density.

To demonstrate the effectiveness of the proposed octree encodings, we computed the required memory for the octree data structure (without the points) with different depths. The data is given in Table I to III for both the new compact and the reference representation. The given leaf size is half of the side length of the leaf nodes. For all tests, the root volume and therefore all octree volumes are axis aligned cubes. The size and position of the root is such that it represents the smallest cube possible to contain the entire data set. For smaller and sparse data sets the benefits of the compact encoding are less apparent than for the denser point clouds. For the large data set (city) the reduced memory requirements are indispensable. In this case the octree with all points and their reflectance attributes required 121 MB as compared to the 242 MB the unorganized raw list of points requires.

We further compare the computing time of standard RANSAC for plane detection with RANSAC sped up solely by fast model checking in Table IV. Results were averaged over 100 runs, each with 5000 iterations and include the construction time for the octree. The latter is only significant for the city data set (≈ 88 s). Even including the construction time of the octree we achieve significant decrease in computing time.

To evaluate the performance of the NNS for varying maximal distances, we performed ICP scan matching employing the octree and the k-d tree NNS using default parameters (i.e. a maximal distance of 25 cm and 50 ICP iterations). We averaged results over 100 runs, except for the city, due to time concerns. Results can be found in Table V and Fig. 6, where the maximal distance was varied. Experiments were done with the original point cloud as well as with reduced data sets. For this purpose we created an octree with the same voxel size as is used in NNS. In leaves that contained more than 10 points, we randomly selected 10 points. All other points remained. This creates a uniformly subsampled point cloud, which is often used to speed up and improve upon ICP's matching accuracy. Since the k-d tree stops subdividing when a node contains less than or exactly 10 points, both data structures will, on average, have roughly the same number of points per leave.

The octree based NNS does not suffer considerably more from larger maximal distances than the k-d tree based NNS. We observe however, that there is an increase in the variance for the computing time for the NNS using the octree. Conversely, the variance of the k-d tree based NNS is stable over all distances. This suggests that the octree is more vulnerable to the combination of large maximal distances and unfavorable starting pose estimates.

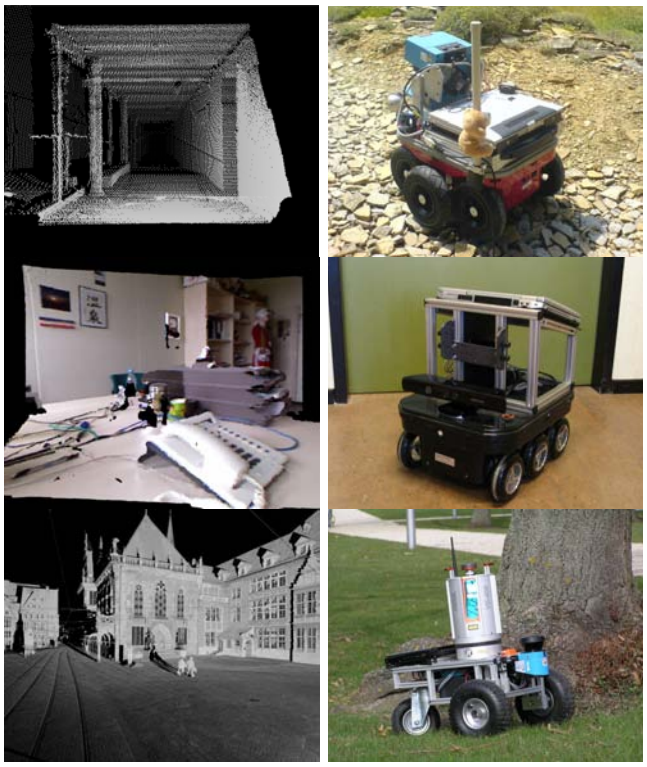

Fig. 5. Three point clouds are used for the following analysis. The top point cloud is a 3D scan that was acquired by the mobile robot Kurt3D using an actuated SICK LMS200 laser scanner in an office environment with 81360 points ($\approx$ 1.5 MB). Statistics for this data set are given in Table I. The middle point cloud has been acquired by the Microsoft Kinect in an office environment. The cloud contains 293772 points and is therefore relatively dense. Data courtesy of N.N.. The scan on the bottom is a high resolution scan taken in a historic city center using the Riegl VZ-400 3D scanner. The point cloud contains 15896875 points ($\approx$ 303 MB). Refer to Table III for data on this point cloud.

VI. Conclusions and outlook

We have implemented an efficient data structure for 3D point clouds. All presented algorithms are available under the GPL license and can be downloaded. The software contains a small viewer application that is capable of processing 1 billion points while still enabling the user to navigate smoothly through the point cloud.

This paper has further presented novel algorithms for the efficient processing of very large point clouds. In addition to storing and visualizing 1 billion points on modern hardware, we are capable of fast shape detection and scan matching.

In future work we will continue using our octree for efficient 3D point cloud processing, e.g., for globally consistent scan registration [2], for automatically deriving semantic information, for dynamic maps, i.e., maps that can handle changes of the scene, and for next-best-view planning.

Acknowledgment

This work was partially supported by the SEE-ERA.NET project ThermalMapper under the project number ERA 14/01.

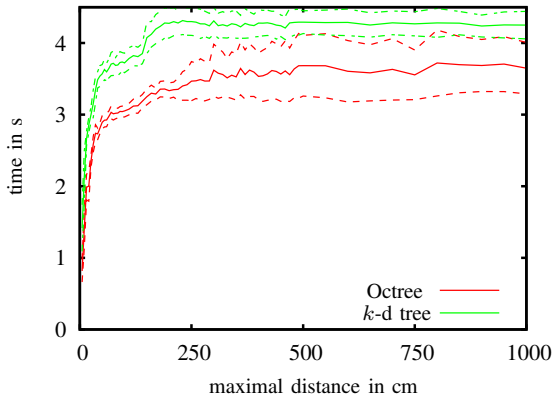

Fig. 6. Plot of the average and standard deviation of the computing time of ICP using octrees and k-d trees over the maximal allowed matching distance for the Kurt3D data set . Noise was applied to the initial starting pose estimate ($\pm$25cm translational, $\pm 10^\circ$ rotational) for each run.

REFERENCES

[1] J. Bielak amd O. Ghattas and E. J. Kim. Parallel octree-based finite element method for large-scale earthquake ground motion simulation. *Computer Modeling in Engineering and Sciences*, 10(2):99 – 112, 2005.

[2] D. Borrmann, J. Elseberg, K. Lingemann, A. Nüchter, and J. Hertzberg. Globally consistent 3d mapping with scan matching. *Journal Robotics and Autonomous Systems (JRAS)*, 56(2):130–142, February 2008.

[3] A. Elfes. Using occupancy grids for mobile robot perception and navigation. *Computer*, 22:46–57, June 1989.

[4] S. F-Frisken and R. N. Perry. Simple and Efficient Traversal Methods for Quadtrees and Octrees. *Journal of Graphics Tools*, 7(3), 2002.

[5] M. A. Fischler and R. C. Bolles. Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography. *Communications of the ACM*, 24:381 – 395, 1981.

[6] M. Gervauts and W. Purgathofer. A simple method for color quantization: octree quantization. *Graphics Gems I*, pages 287–293, 1990.

[7] E. Gobbetti, F. Marton, G. Iglesias, and A. Jose. A single-pass gpu ray casting framework for interactive out-of-core rendering of massive volumetric datasets. *Visual Computing*, 24:797–806, July 2008.

[8] K. M. Wurm et al. Octomap. http://octomap.sourceforge.net/, May 2011.

[9] A. Knoll, I. Wald, S. Parker, and C. Hansen. Interactive isosurface ray tracing of large octree volumes. In *Proc. of the IEEE Symposium on Interactive Ray Tracing*, pages 115–124, September 2006.

[10] A. M. Knoll, I. Wald, and C. D. Hansen. Coherent multiresolution isosurface ray tracing. *Visual Computing.*, 25:209–225, February 2009.

[11] S. Laine and T. Karras. Efficient sparse voxel octrees. In *Proceedings of the ACM SIGGRAPH symposium on Interactive 3D Graphics and Games (I3D '10)*, pages 55–63, New York, NY, USA, 2010. ACM.

[12] D. Meagher. Geometric modeling using octree encoding. *Computer Graphics and Image Processing*, 19(2):129 – 147, 1982.

[13] P. Payeur, P. Hebert, D. Laurendeau, and C.M. Gosselin. Probabilistic octree modeling of a 3d dynamic environment. In *Robotics and Automation, 1997. Proceedings., 1997 IEEE International Conference on*, volume 2, pages 1289 –1296 vol.2, April 1997.

[14] Radu Bogdan Rusu et al. Point Cloud Library 1.1.1 revision 2440. http://pointclouds.org/, September 2011.

[15] Radu Bogdan Rusu and Steve Cousins. 3D is here: Point Cloud Library (PCL). In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA '11)*, Shanghai, China, May 2011.

[16] R. Schnabel, R. Wahl, and R. Klein. Efficient RANSAC for Point-Cloud Shape Detection. *Computer Graphics Forum*, 2007.

[17] K. M. Wurm, A. Hornung, M. Bennewitz, C. Stachniss, and W. Burgard. OctoMap: A Probabilistic, Flexible, and Compact 3D Map Representation for Robotic Systems. In *Proceedings of the IEEE ICRA Workshop on Best Practice in 3D Perception and Modeling for Mobile Manipulation*, Anchorage, AK, USA, 2010.

TABLE I
MEMORY REQUIREMENTS FOR THE OCTREE STRUCTURE OF THE SPARSE KURT3D POINT CLOUD.

Leaf size *cm*	# Nodes	# Leaves	compressed size	reference size
876	1	8	104 B	954 B
219	34	77	1.19 kB	11.8 kB
54.76	282	505	8.31 kB	86.8 kB
13.69	1777	3688	58.4 kB	621.0 kB
3.423	9327	19064	303.3 kB	3.57 MB
0.855	27668	52836	855.3 kB	12.85 MB

TABLE II
MEMORY REQUIREMENTS FOR THE OCTREE STRUCTURE OF THE KINECT DATA SET.

Leaf size *cm*	# Nodes	# Leaves	compressed size	reference size
143	1	6	80 B	742 B
35.9	24	64	960 B	9.3 kB
8.98	336	909	13.5 kB	131.9 kB
2.24	4593	10794	166.2 kB	1.63 MB
0.56	46510	86560	1.41 MB	14.10 MB
0.14	323479	253281	5.62 MB	61.13 MB

TABLE III
MEMORY REQUIREMENTS FOR THE OCTREE STRUCTURE OF THE DENSE CITY DATA SET.

Leaf size *cm*	# Nodes	# Leaves	compressed size	reference size
8560	6	12	192 B	1.9 kB
2140	46	86	1.4 kB	13.9 kB
535	408	800	12.8 kB	130.1 kB
133	3616	7993	124.8 kB	1.25 MB
33.4	33965	75999	1.18 MB	11.91 MB
8.35	302573	687529	10.67 MB	109.53 MB
2.08	1927234	4166979	65.42 MB	742.98 MB
0.52	5592151	10142923	166.45 MB	2.643 GB

TABLE IV
AVERAGE COMPUTING TIME IN *ms* OF RANSAC.

Data set	no octree	octree	speedup
Kurt3D	1666.57	176.69	9.43
Kinect	6905.94	429.32	16.08
city	388551.55	11084.81	35.05

TABLE V
AVERAGE COMPUTING TIME IN MS FOR ICP. NOISE TO THE INITIAL POSE ESTIMATE WAS ADDED AS FOR FIG. 6.

Data set	k-d tree	octree	speedup
Kurt3D	3043.099	2386.881	1.27
Kinect	19545.277	14507.198	1.34
city	355848.476	314506.905	1.13
Kurt3D reduced	757.514	625.683	1.21
Kinect reduced	169.079	152.356	1.10
city reduced	91735.667	74706.85	1.22