Εθνικο Μετσοβιο Πολυτεχνειο
Σχολη Ηλεκτρολογων Μηχανικων
και Μηχανικων Υπολογιστων
Τομεας Σηματων Ελεγχου και Ρομποτικης

Τριδιάστατη Αναπαράσταση για Ρομποτική Χαρτογράφηση σε Αστικό Περιβάλλον

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Αθανάσιος Χ. Δομέτιος

Επιβλέπων : Κωνσταντίνος Σ. Τζαφέστας
Επίκουρος Καθηγητής

Αθήνα, Ιούλιος 2013

Εθνικο Μετσοβιο Πολυτεχνειο
Σχολη Ηλεκτρολογων Μηχανικων
και Μηχανικων Υπολογιστων
Τομεας Σηματων Ελεγχου και Ρομποτικης

Τριδιάστατη Αναπαράσταση για Ρομποτική Χαρτογράφηση σε Αστικό Περιβάλλον

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Αθανάσιος Χ. Δομέτιος

Επιβλέπων : Κωνσταντίνος Σ. Τζαφέστας
Επίκουρος Καθηγητής

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 15η Ιουλίου 2013.

....	....	....
Κωνσταντίνος Τζαφέστας	Πέτρος Μαραγκός	Στέφανος Κόλλιας
Επίκουρος Καθηγητής	Καθηγητής	Καθηγητής

Αθήνα, Ιούλιος 2013

....
Αθανάσιος Χ. Δομέτιος
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Copyright © Αθανάσιος Δομέτιος 2013.
Με επιφύλαξη παντός δικαιώματος. All rights reserved.

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς τον συγγραφέα.

Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτό το έγγραφο εκφράζουν τον συγγραφέα και δεν πρέπει να ερμηνευθεί ότι αντιπροσωπεύουν τις επίσημες θέσεις του Εθνικού Μετσόβιου Πολυτεχνείου.

3D REPRESENTATION FOR MAPPING URBAN ENVIRONMENT

eingereichte
MASTERARBEIT
von

cand. ing. Athanasios Dometios

geb. am 15.01.1990
wohnhaft in:
Landsbergerstrasse 289
80687 München
Tel.: 0176 98420963

Lehrstuhl für
STEUERUNGS- UND REGELUNGSTECHNIK
Technische Universität München
Univ.-Prof. Dr.-Ing../Univ. Tokio Martin Buss
Univ.-Prof. Dr.-Ing. Sandra Hirche

Betreuer: M.Sc. Sheraz Khan
Beginn: 15.10.2012
Zwischenbericht: 06.02.2013
Abgabe: 15.04.2013

Περίληψη

Η κατασκευή μίας ακριβούς αναπαράστασης του περιβάλλοντος είναι μία από τις βασικότερες εργασίες για ένα αυτόνομο κινητό ρομπότ, δεδομένου ότι η ύπαρξη ενός τέτοιου χάρτη είναι ουσιαστικά βασική προϋπόθεση για την ανάπτυξη αποτελεσματικών στρατηγικών ρομποτικής εξερεύνησης και πλοήγησης. Οι περισσότερες ρομποτικές εφαρμογές βασίζονται σε χάρτες 2D ή υψομετρικούς χάρτες (2.5D height maps), αφού οι τρισδιάστατοι χάρτες απαιτούν μεγάλα ποσά υπολογιστικού χρόνου και κατανάλωσης μνήμης. Ωστόσο, η χρήση 3D χαρτών για την αναπαράσταση ενός περιβάλλοντος είναι πιο κοντά στην ανθρώπινη διαίσθηση και πιο λεπτομερής. Οι περισσότερες τεχνικές 3D χαρτογράφησης χρησιμοποιούν απευθείας την μέθοδο του νέφους σημείων (point clouds), ωστόσο, άλλες στρατηγικές, που βασίζονται σε διάφορες δομές δεδομένων, μπορούν να επιτύχουν γρήγορους χρόνους εισαγωγής και προσπέλασης των δεδομένων. Η συγκεκριμένη διπλωματική εργασία επικεντρώνεται στην αναζήτηση και τη σύγκριση διαφόρων δομών δεδομένων όσον αφορά υπολογιστικούς χρόνους και κατανάλωση μνήμης, όπως οκτάδεντρα (octrees), πίνακες κατακερματισμού (hash-tables) και δέντρων K-d για την ακριβή 3D αναπαράσταση του περιβάλλοντος, χρησιμοποιώντας 3D σημεία. Επιπροσθέτως, παρουσιάζεται η ιδέα της προσέγγισης του περιβάλλοντος με χρήση ορθογωνίων (rectangles) ως "RMAP" και εξετάζεται η χρήση της δομής δεδομένων R-δέντρου (R-tree). Η προσέγγιση RMAP θα παρουσιαστεί τόσο με πιθανοτικό τρόπο ως πλέγμα κατάληψης (occupancy grid) όσο και μία προσέγγιση βασισμένη στην πυκνότητα των σημείων για 3D pointclouds και επίπεδα τμήματα.

Λέξεις Κλειδιά

Τριδιάστατη αναπαράσταση, Χαρτογράφηση, Αυτόνομα κινητά ρομπότ, Ρομποτική, Δομές δεδομένων, Οκτάδεντρα (octree), Πίνακας κατακερματισμού (hash-tables), Δέντρο K-d (K-d tree), Ορθογώνιο, R-δέντρο (R-tree)

Abstract

Building an accurate representation of an environment is essential for a robot since an accurate map can lead to efficient and precise exploration and navigation strategies. Most robotic applications rely on 2D maps or 2.5D height maps since 3D maps require a large amount of computation time and memory consumption. However, the use of 3D maps for the representation of an environment is much more intuitive and accurate. Most 3D mapping techniques directly utilize a point cloud. However, using several other strategies, which rely on different data structures, both fast insertion/extraction times and efficient memory usage can be achieved. This topic focuses on searching and comparing, in terms of computational time and memory complexity, different data structures such as octrees, hash-tables and K-d trees to build an accurate 3D representation of the environment using the basic primitive of points. Furthermore, the concept of an environmental approximation referred to as "RMAP" is presented, which utilizes rectangles as basic geometric primitives, and the use of the R-tree data structure will be considered. The RMAP approach will be presented both in a probabilistic manner as an occupancy grid and as a point density-based approximation for 3D point-clouds and planar segments.

Key Words

3D representation, Mapping, Autonomous mobile robots, Robotics, Data structures, Octree, Hash- table, K-d tree, Rectangle, R-tree

Contents

1 Introduction **5**

2 Environment representation using points 7
2.1 Basic structures for storing points 7
2.1.1 List 7
2.1.2 Array 8
2.1.3 Hash Table 8
2.1.4 Trees 9
2.2 Theoretical comparison 12
2.3 Practical comparison 13

3 RMAP: Environment representation using rectangles **21**

3.1 Occupancy grid framework 23
3.2 3D Rectangle Approximation 26

4 RMAP: Environment representation using planar convex polygons **37**

4.1 Rectangle approximation of convex polygons 37
4.2 2D Rectangle approximation 42

5 Applications in robotics **47**

6 Conclusion and future work **49**

List of Figures **51**

List of Tables **55**

Bibliography **57**

1 Introduction

An accurate 3D map of the environment is an essential requirement for all autonomous robots. It can be considered a prerequisite for the autonomy of the robot since it is utilized in most navigation algorithms and the accuracy of the map is crucial for most collision avoidance algorithms. Also, many mobile robotic applications including land vehicles, planetary and underwater explorers require high environmental representation accuracy.

One of the most commonly used environment representation is an occupancy grid. 2D occupancy grids [1], [2], [3] can be considered the de facto standard. Besides 2D environment representations, some grid structures also store the height corresponding to each cell leading to the so called 2.5D height maps [4]. 2D NDT [5] is another grid representation which models cell centers and their uncertainty using Gaussian distributions. In addition to grid representations some approaches model the 2D environment using geometric primitives such as lines [6] which are computationally less expensive compared to grid representations.

The recent surge in the 3D sensing technology with the influx of Kinect and Velodyne has shifted the focus of the robotics society from 2D to 3D environment representations which are more intuitive, natural and useful in cases the environment is no longer planar. The most common approach for 3D environment representation is utilizing raw point clouds and operating directly on them or utilizing a 3D occupancy grid.

This thesis is mainly divided in two parts. The first part considers the representation of 3D environments utilizing points. Due to the fact that a great variety of data structures are offered in the computer science field and no rule of thumb exists to decide which data structure to use and in which scenario, a search and comparison of different data structures that can correspond to the aforementioned requirements will be conducted. The comparison is conducted in terms of computation time, memory consumption and accuracy of mapping result. A detailed presentation of data structures that are suitable for spatial applications can be found in [7]. In [8] a comparison of strategies that utilize different data structures for the nearest-neighbor-search (NNS) problem is featured.

The second part of the thesis considers a rectangular approximation of environments titled “RMAP”. The presented framework is very generic as it can be used as an occupancy grid to represent the environment in a probabilistic framework or to find rectangular approximations of environments using a splitting algorithm based on point density. In addition, the approach can be utilized for approximation of polygons as well. In case of polygonal approximation, the presented approach is used as a post processing step of standard segmentation algorithms. The proposed approach is highly flexible and offers the advantage that any arbitrary rectangular cuboid can be used as a grid cell for

environment representation. Also the resolution can be adapted according to the application (navigation or registration) or computational resources available.
Chapter 2 deals with environment representation using points. Data structures that store points are briefly presented and compared theoreticaly. Also, a practical comparison in terms of time and memory complexity between an octree, a hash table and a K-d tree is conducted.
In Chapter 3 we present the RMAP rectangular approximation of an environment which considers the environmental representation using rectangles. Firstly, the probabilistic aspect of the approach is featured, as it is used as an occupancy grid. Secondly, a rectangle approximation of 3D environments is suggested, employing a splitting algorithm based on the point density. The approach is tested both on simulation and real world data sets.
Chapter 4 deals with an extension of the RMAP approach to 2D pointclouds and convex polygons.
Finally, in Chapter 5 we discuss several robotic applications for which an accurate environmental representation is required and Chapter 6 presents conclusions and future work.

2 Environment representation using points

In this section a brief presentation of the basic data structures that are suitable for storing points will be attempted, such as lists, hash tables and several types of trees. Then a comparison will follow, both theoretically and practically using experimental results. The comparison will be conducted in terms of memory consumption and computational complexity for several robotic applications, such as 3D mapping and navigation techniques, and procedures they include, such as inserting and accessing the points in the data strucure.

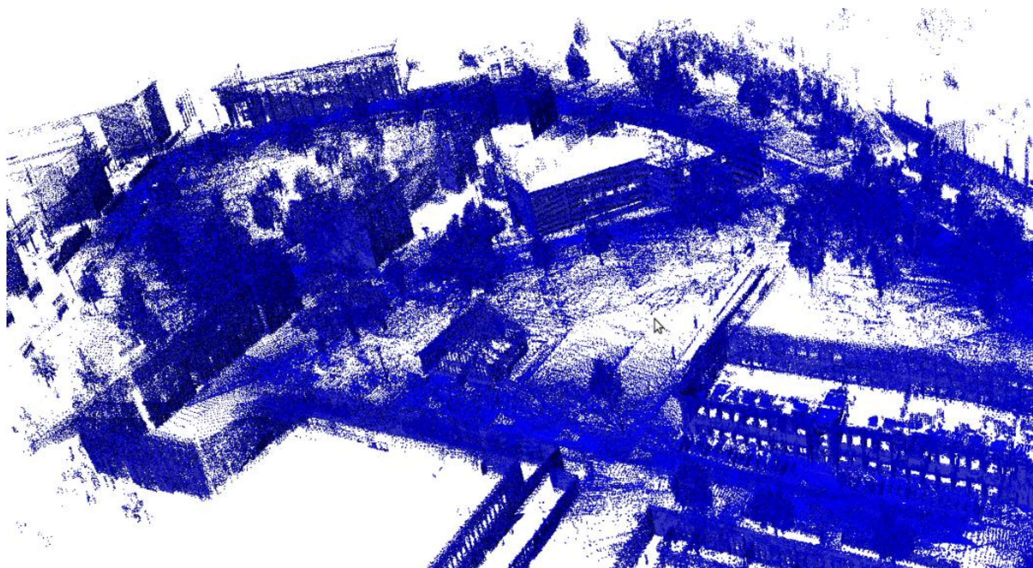


Figure 2.1: Representation of the Freiburg campus using an octree data structure

2.1 Basic structures for storing points

2.1.1 List

A list or sequence is an abstract data type that implements an ordered collection of values, where the same value may occur more than once. Each instance of a value in the list is called *item, entry* or *element.* In our case, an entry may consist of a 3D vector that

contains the coordinates of the 3D point that is to be stored. Typical procedures include insertion, access and deletion of a point. Every time a point is to be inserted, a check has to be made to see whether the point is already in the list or not, so that it is not inserted again. Therefore, for the insertion proceure the whole list would have to be traversed making the total time O(n), where n is the total number of entries already stored in the list. For the accessing and deletion procedure, when a certain point has to be found, the worst-case scenario suggests that the point will be the last entry in the list, making the running time again O(n). The total memory consumed is proportional to the number of entries, that is, O(n). More specifically, assuming that three floats (4 bytes each) and 8 bytes are allocated for the point coordinates and the pointer respectively, the total amount of memory would be $20 \cdot n$ bytes. A double-linked list could be employed to reduce the time for these procedures, using for example, two directional search algorithms, without having significant improvement though. More detailed information about the list data structure can be found in [9].

2.1.2 Array

An array is a data structure consisting of a collection of elements (values or variables), each identified by at least one array index or key. An array is stored so that the position of each element can be computed from its index tuple by a mathematical formula. In our case, three instances of the array data structure could be used. The first one considers an array that contains pointers to vectors of the 3D points, which is similar to the list data structure described above. Secondly, a 2,5D array could be used storing the z-coordinate in specific positions defined by the other two coordinates x and y which are the two indices of the array. Finally, the third array that could be employed is a 3D one, where all three indices correspond to the three coordinates x,y and z of the 3D point. This latter array instance is closer to a 3D grid and can conceive better the concept of 3D space. In the first case, due to the fact that the data structure is like a linked list, the times for insertion, accessing and deletion and the memory consumption will be the same, O(n) in all three procedures. In the second and third case, all that it needs to be done is to find the corresponding indices x and y (and z in the third case). So the insertion time would be constant, O(1), as only one multiplication and one addition are required. The same process is followed in the accessing and deletion procedures as well, requiring O(1) running time. However, accessing all the elements in the array would take O(n) time. The memory that needs to be consumed is O(n) by allocating space for $n \cdot k$, where k is the size of the type of the entry in bytes. For example, if integers are stored in the array, $4 \cdot n$ will be the total memory amount required. More information about the array data structure can be found in [10].

2.1.3 Hash Table

A hash table (also hash map) is a data structure used to implement an associative array, a structure that can map keys to values. A hash table uses a hash function to compute an index into an array of buckets or slots, from which the correct value can be found. In our

case, the values in the buckets could be a vector of the 3D points. The running time for inserting a point is constant, like in the array data structure case, O(1). The accessing time is O(1), simply because in our case the number of entries is the same as the number of buckets. The same stands for the deletion time. One can conclude that employing a simple hash-table data structure like the above, the complexity of the running times is significantly low. The memory consumed is proportional to the number of entries, O(n). If the entries describe the coordinates of a 3D point as assumed above, total memory of $n \cdot 3 \cdot 8 = 24 \cdot n$ has to be allocated, as 3 doubles of 8 bytes each are required for each entry . Modifications of the standard hash-table structure can also be used for better performance, such as seperate chaining, linked lists at each bucket, open addressing and coalesced hashing. A detailed analysis regarding the hash table data structure can be found in [10] and [11]. In [12] hash-tables have been used in intra-logistics tasks by mobile robots.

2.1.4 Trees

Binary Trees

Binary trees are the most common tree data structures used in computer science. A binary tree is a tree data structure in which each node has at most two child nodes, usually distinguished as "left" and "right". Nodes with children are parent nodes, and child nodes may contain references to their parents. Outside the tree, there is often a reference to the "root" node (the ancestor of all nodes), if it exists. Any node in the data structure can be reached by starting at root node and repeatedly following references to either the left or right child. General information about binary trees can be found in [13] and [14]. There exists a great variety of binary trees, from which the most commonly utilized in the field of robotics is the k-d tree, which is briefly presented below.

K-d Tree

A k-d tree is a space-partitioning (SP) binary tree data structure for organizing points in a k-dimensional space. Each leaf node is a k-dimensional point and every non-leaf node can be thought of as implicitly generating a splitting hyperplane that divides the space into two parts. Points to the left of this hyperplane represent the left subtree of that node and points to the right of the hyperplane are represented by the right subtree. The hyperplane direction is chosen in the following way: every node in the tree is associated with one of the k-dimensions, with the hyperplane perpendicular to that dimension's axis. So, for example, if for a particular split the "x" axis is chosen, all points in the subtree with a smaller "x" value than the node will appear in the left subtree and all points with larger "x" value will be in the right subtree. In such a case, the hyperplane would be set by the x-value of the point, and its normal would be the unit x-axis. In our

case, a 3-d tree could be used for the storage of 3D points at the leaf nodes, conceiving in that way the sense of space. The time complexity is O(log*n*) for the insertion, access and deletion procedures both in the average and worst case, where *n* is the total number of nodes. The memory complexity is O(*n*). In particular, the branch nodes contain two pointers to their two children, that is, 8 bytes. The leaf nodes contain a value whose type depends on the application that uses the tree. In [15] they exploit k-d trees for ray-tracing. In [16] and [17] it is described explicitly how k-d trees are involved in applications in the field of robotics and especially in spatial pattern search and the ICP algorithm. More detailed information can also be found in [18].

Octree

An Octree is a space-partitioning tree data structure in which the root (father) and each internal node have exactly eight children. Octrees are mostly used to partition three dimensional spaces by recursively subdividing them into eight octants, as depicted in Fig. 2.2. Each octant is considered to be a node and each node represents the space contained in a cubic volume. So each node contains pointers to eight other nodes, its children, and a value, depending on the needs of its use. The recursive subdivision stops as soon as a minimum size is reached, which determines the resolution of the octree. In our case, when a 3D point needs to be stored, the value of the node that corresponds to the x,y,z – coordinates is increased, or set.

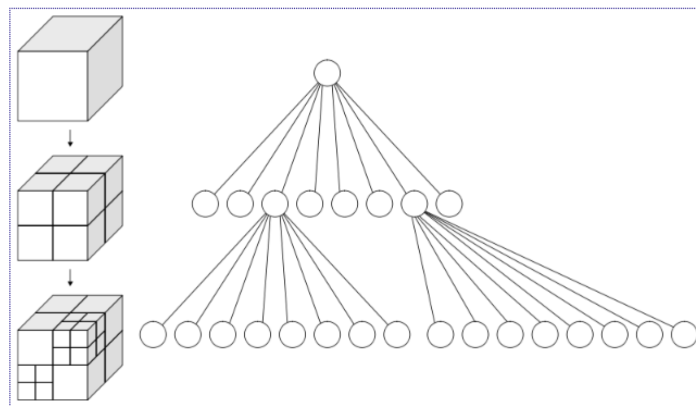


Figure 2.2: The octree data structure

Depending on the resolution of the octree, more than one point may correspond to the same node, carrying out in that way a down sampling of the whole amount of the received points. In Fig. 2.3, 2.4 and 2.5 the first scan of the Freiburg campus dataset[1] is featured for three different resolutions, 0.05m, 0.2m and 0.8m, respectively. The time complexity here would be the same as in the k-d tree, O(logn) both in the average and in the worst case for the three standard procedures, with *n* being the total number of nodes.

[1] Courtesy of B. Steder and R Kuemmerle, available at http://ais.informatik.uni-freiburg.de/projects/datasets/octomap/

The memory requirements are similar to that of the k-d tree, with the main difference lying in the fact that the branch nodes have eight pointers that point to their children, instead of two. The use of octrees for modeling was originally proposed in [19]. In [20] a probabilistic way of modeling occupied and free space was introduced. A similar approach titled "Octomap" is also considered in [21]. Furthermore, in [22] an octree-based 3D map representation was designed that can efficiently perform map updates and copies, especially in the context of particle filter SLAM. More detailed analysis regarding the octree data structure can be found in [23].

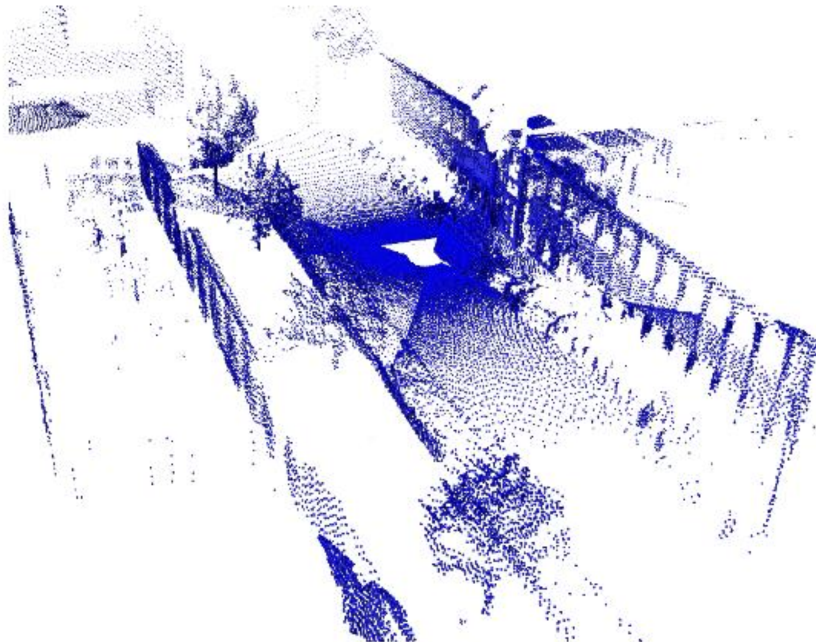


Figure 2.3: 1st scan of the Freiburg campus data set using resolution 0.05

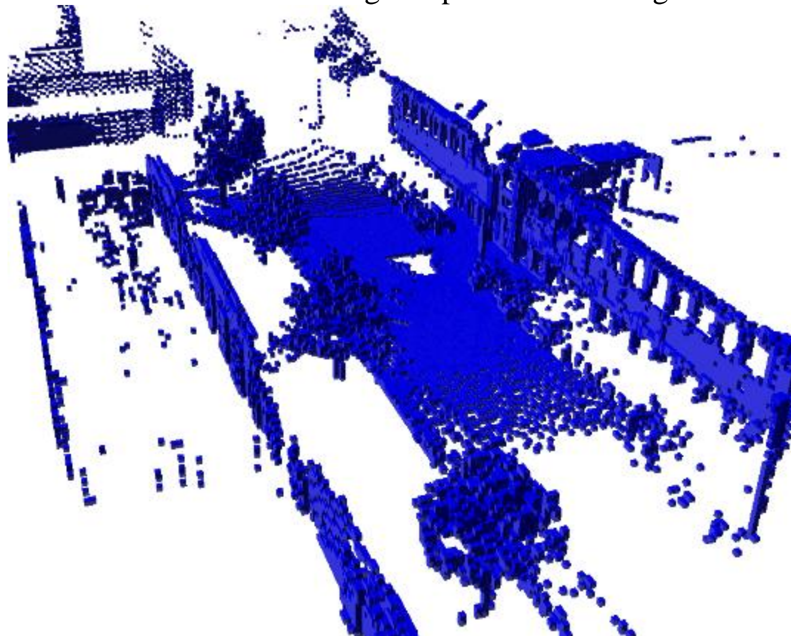


Figure 2.4: 1st scan of the Freiburg campus data set using resolution 0.2

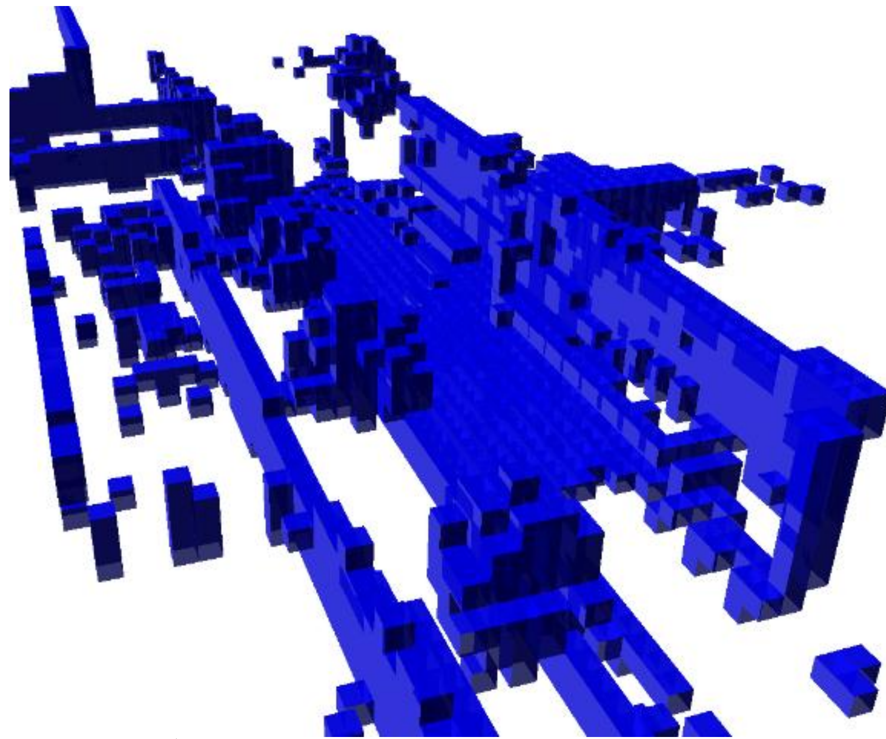


Figure 2.5: 1st scan of the Freiburg campus data set using resolution 0.8

2.2 Theoretical comparison

As one can conclude from the above, there exists a great variety of data structures that can be used for storing points. For this reason, a comparison is necessary in order to be able to choose the most appropriate one for the needs of 3D mapping. That is, low insertion and access times are preferable, as well as low memory consumption, especially in dynamically changing environments, such as urban ones and on-line applications where points are received at very high frequencies. Moreover, of high significance is a multi-resolutional representation of the environment, as it offers adaptability of the computer systems employed, and navigational advantages. In Table 2.1 such a comparison is presented in terms of big O notation for the needs of the specific topic. The notation n is used to describe the number of points stored in each data structure. The access time depicted applies for the access of one single point in the data structure.

It can be seen that the list data structure has the worst insertion and access times. This is an expected result, as lists are naive data structures and multiple traversals need to be conducted for each procedure. The computational times for the array data structure are constant, as discussed in Section 2.1. Although this is a great advantage, the size of a 3D array data structure must be set in advance and memory is allocated for all of its elements. Hence, the memory consumption is expected to be high, since memory for both occupied and free cells will be allocated. This can be partially avoided by using the hash table data structure, for which the average cases of insertion and access times are

also constant, making it computationally efficient. However, its performance is highly dependent on the hash function and the general implementation employed. The tree data structure gives the advantage of finding a needed entry in O(log*n*) time both in the average and the worst case. If n is the number of points stored in the tree, then its height h is log*n*. K-d trees are the adaptation of binary trees in the requirements of storing 3D points. However, due to the fact that they are binary trees, many splits have to be carried out in order to reach the level of a point and that results in a large tree height. The implementation of a K-d tree, however, has a great impact on its performance. On the other hand, an octree data structure is expected to have smaller height, as each inner node has eight children, resulting in lower insertion and access times. Moreover, the greatest advantage of the octree towards the other data structures is that it conceives at the maximum the sense of 3D space due to the fact that the division of each node is carried out by eight ($2^3 = 8$). For that reason, a multi-resolutional approach to the representation of the environment can be achieved, traversing the tree at a certain level l. For example, if the leaf nodes are at $l = 1$ and each cell describes a cubic volume of edge 0.05m (Fig. 2.3), at $l = 4$ each node will describe a cubic volume of edge $4 \cdot 0.05 = 0.2$m and the representation of the environment would be like in Fig. 2.4. Finally, Table 2.1 also shows that the memory consumption is the same in terms of big O notation for all the data structures. For the latter three structures, the memory is highly dependent on the implementation utilized, as will be featured in the next Section.

Data Structure	**Insertion**		**Access**		**Memory**
	Average case	**Worst case**	**Average case**	**Worst case**	
List	O(*n*)	O(*n*)	O(*n*)	O(*n*)	O(*n*)
3D Array	O(1)	O(1)	O(1)	O(1)	O(*n*)
Hash Table	O(1)	O(*n*)	O(1)	O(*n*)	O(*n*)
K-d tree	O(log*n*)	O(log*n*)	O(log*n*)	O(log*n*)	O(*n*)
Octree	O(log*n*)	O(log*n*)	O(log*n*)	O(log*n*)	O(*n*)

Table 2.1: Data structures theoretical comparison (big O notation) in terms of inserton, access time and memory consumption

2.3 Practical comparison

In order for the theoretical results that are shown above to be verified, some experiments were carried out. More specifically, three implementations of an octree, a k-d tree and a

hash table were employed, briefly described below. In all three approaches, the probabilistic manner of describing the 3D space was incorporated, making the data structures appropriate for dynamically changing environments. In each implementation, an occupancy grid structure is modeled. In the tree structures, the nodes where the information is stored represent a 3D cubic cell of the environment, whose size depends on the resolution used. The same applies for the entries of the hash table. That implies that a cell is represented by the coordinates of a 3D point that describe its centroid. Ray tracing is done along the path beam to update the occupancy values of each cell. Let z represent the observation and the lower subscripts represent the time instances. The probability of any grid r_i of being occupied can be estimated by the formula:

$$P(r_i|z_{1:t}) = \left[1 + \frac{1 - P(r_i|z_t)}{P(r_i|z_t)} \frac{1 - P(r_i|z_{1:t-1})}{P(r_i|z_{1:t-1})}\right]^{-1},$$

given the case that the initial occupancy of each cell being occupied or free is the same. In order to prevent each cell of being over confident about its state we utilize a clamping/saturation threshold a after which the cell is not updated. The values of $a_{\min}$ and $a_{\max}$ were set to 0.05 and 0.95 respectively. After the brief description of the data structures below, an evaluation of them is conducted for randomly generated points and for the real data sets of Freiburg campus and Bremen city center.

Hash Table

The main structure of the hash table that was implemented is an one-dimensional array. The x-coordinates of the cells that describe a certain 3D space are addressed to the indices of this array, using the resolution of the map. Each element of the array contains a vector of y-nodes. A y-node consists of a key that corresponds to the y-coordinate of the cell and another vector of z-nodes. Finally, a z-node is also identified by its key that describes the z-coordinate of the cell and contains the probability of this cell being occupied. The y- and z- keys are obtained from the original y- and z- coordinates, respectively, employing a hash function that utilizes the resolution of the map. During the insertion procedures, if two 3D points correspond to the same cell, the probability of this cell being occupied is updated according to the above formula. To access a cell, the x,y,z- coordinates of a point inside this cell are given as input. Using the x-coordinate, the corresponding element of the array is determined and then an exhaustive search of the corresponding y- and z- nodes is conducted. The matching is done using the keys in each node. That is, only the x-array is an associative one. The y- and z- values are arbitrarily stored as keys in the y- and z-nodes respectively. If this specific cell does not exist, it is considered as unknown space. An example for an access procedure is shown in Fig. 2.6. The memory consumption is given by the formula:

$$Memory_{Hash\ Table} = 24 \text{ x } x\text{-}nodes + 28 \text{ x } y\text{-}nodes + 8 \text{ x } z\text{-}nodes$$

The structure of a vector (contained in a bucket of the x-array) contains 24 bytes. So a y-node will contain, except for these bytes, an integer for the value of the y-coordinate (4

bytes). Finally, each z-node contains an integer for the value of the z-coordinate (4 bytes) and a float for the probability of being occupied (4 bytes).

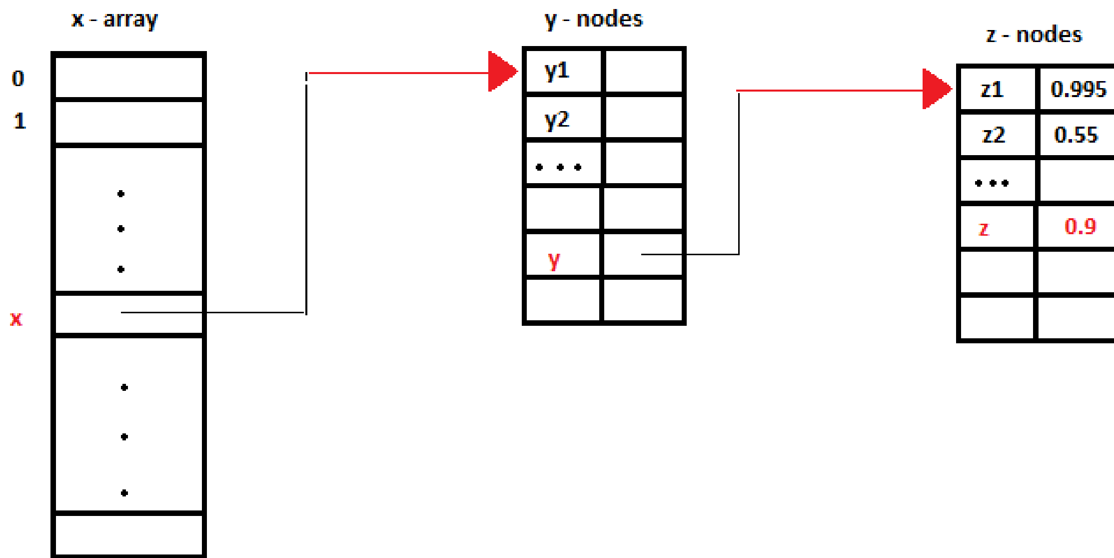


Figure 2.6: Example of accessing a cell in the hash table

K-d Tree

The K-d tree implementation used can be found in [24]. We modified it appropriately to adopt the probabilistic properties described above. Each node of the tree contains a 3D point that is the centroid of the corresponding cell and its probability of being occupied. For inserting and accessing a single node, a heuristic function is used. More specifically, every node can be thought of as implicitly generating a splitting hyperplane that divides the space into two parts, known as half-spaces. Points to the left of this hyperplane represent the left subtree of that node and points right of the hyperplane are represented by the right subtree. The fact that the information is stored in each node of the tree makes the data structure much more efficient in terms of memory. Other K-d tree variants could store the information only in the leaf nodes giving in that way to the data structure a sense of space. However, due to the fact that the K-d tree structure is binary, many splits have to be made resulting in a great number of inner nodes and large tree height. The memory requirements in this case would be higher. In addition, an octree data structure conceives much better the sence of 3D space, having a greater performance than a K-d tree implementation that stores all the information in the leaf nodes. For this specific implementation, the formula below gives the memory consumption in bytes:

$$Memory_{K\text{-}d\ Tree} = 64 \text{ x } Nodes + 88,$$

where each node contains 2 pointer to its children (16 bytes), a float for the probability of being occupied (4 bytes), an integer that determines the axis that the division is made (4 bytes), a pointer to a specific data of the node and an array of 3 doubles for the 3D point storage (1 pointer and 3 doubles = 32 bytes as an array structure contains also a

pointer) resulting in 64 bytes. For the initilization of the tree some structures are required such as an integer that determines the dimension (4 bytes), a pointer pointing to the root node (8 bytes) and a pointer to a hyperectangle structure (8 bytes), which contains the same dimension integer (4 bytes) and two arrays of doubles for the minimum and maximum values of the hyperectangle (16 + 48 = 64 bytes), resulting in 88 bytes.

Octree

A publically available octree implementation [25] was also slightly modified to a probabilistic approach for the needs of the experiments. As described in Chapter 2.1, each node divides the cubic cell it describes into eight nodes that correspond to eight smaller and equally sized cells. The resolution of the octree and the fact that each node contains exactly eight children (3D space, $2^3 = 8$) offer a multiresolutional representation of the environment depending on the level of traversal of the tree. The leaf level renders the most detailed representation that the resolution of the tree imposes. In contrast to the other data structures, the probability of each non-leaf node of the octree has to be updated with the mean value of the probabilities of its children, each time a new insertion is carried out. The memory requirements of this implementation is computed as follows:

$$Memory_{Octree} = 80 \text{ x } InnerNodes + 8 \text{ x } LeafNodes + 16$$

An inner node contains an integer and a float determining its type and its probability of being occupied respectively (4+4 = 8 bytes) and an array of 8 pointers to its children nodes (1+8 pointers = 8x9 = 72 bytes) resulting in 80 bytes. A leaf node contains a float determining its probability of being occupied and an integer determining its type (4+4 = 8 bytes). For the initialization of the tree, a pointer to the root node is needed (8 bytes), an integer for the size of the tree (4 bytes) and a float to determine non-occupied nodes (4 bytes), resulting in 16 bytes.

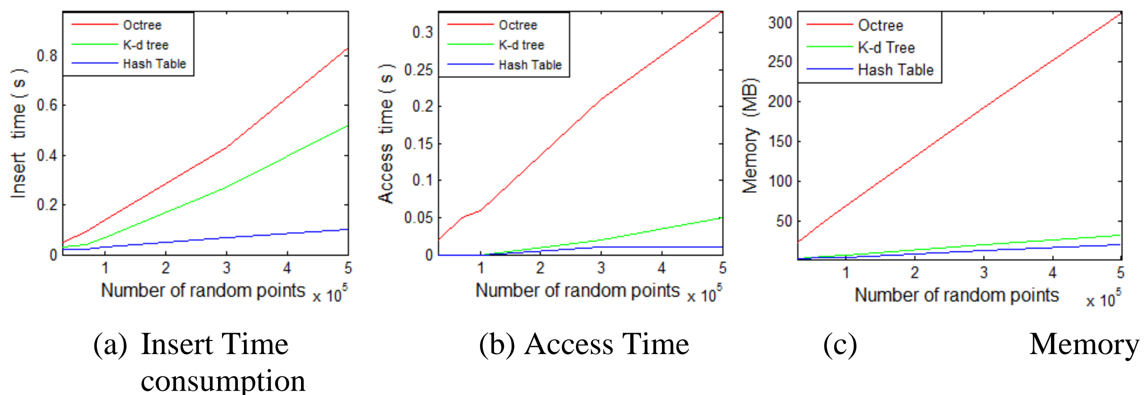
Both real-world data and randomly distributed points were used as input data. In the first case, the Freiburg campus dataset from [21] and the Bremen city center dataset from [REF] were used, whereas in the latter case a stepwise modifying number of random points (simulation) was generated. In both cases the memory consumption and the times for inserting and accessing the total number of points were examined.
The experiments mentioned in this section, as well as the rest evaluations of this thesis, were carried out on an Intel(R) Core i5-2500K, 3.30 GHz processor with 16 GB memory.

1) **Simulation data:**

The focus of this test is to evaluate and compare insertion and access times as well as the memory consumption between the three different data structures described above. Different number of points (30,000, 70,000, 100,000, 300,000 and 500,000) were generated using different resolution values. Moreover, two different ranges of points were used ([-1000,1000] and [-150, 150]).

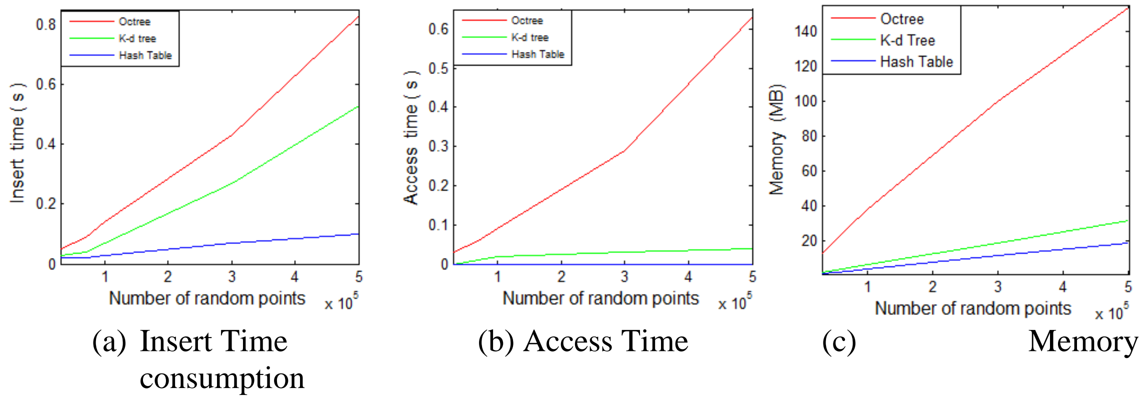
Fig. 2.7 (a)-(c) shows the total insertion and access times of the points in the data structures as well as the memory consumption for random points in range [-1000,1000]m and grid resolution 10cm. The insert times for the hash table are remarkably low, as it is an array structure as discussed in Section 2.2. The corresponding values for the octree data structure are the highest. This can be attributed to the fact that the octree has the sense of the 3D space and stores all the information in the leaf nodes. In that way many inner nodes are built, increasing the height of the tree and making its traversal slower. For example, whenever a point is to be inserted, the octree will generate all the inner nodes to reach it, whereas the other data structures would arbitrarily store it. This is the reason why the K-d tree insertion times are lower, as the information is stored in every node for the implementation used. The above arguments apply also for the access times, which feature a similar behaviour to the insertion ones. However, one could notice that both insertion and access times are significantly low, making these data structures appropriate for online applications. Furthermore, it makes total sense the fact that the octree structure consumes the largest amount of memory, for the aforementioned reason regarding the conception of 3D space. The small memory difference between the K-d tree and the hash table can be attributed to the more complicated structure of the first (e.g., pointers pointing to decendant nodes).

Fig. 2.8 (a)-(c) shows the same results for random points in range [-150,150]m. Remarkable is the difference in memory consumption for the octree structure. As we can see from the graph, for the same number of points, the corresponding values are lower. That is an expected performance, as the total space is smaller and so the inner nodes that the octree creates are less. Moreover, using the same argument as above, one could explain how different grid resolution values affect the time and memory complexity of the octree. Using lower resolution values, the size of each occupancy cell is smaller. In order then to describe the same 3D map, more inner and leaf nodes are created. This can be verified by Fig. 2.9(a)-(c).



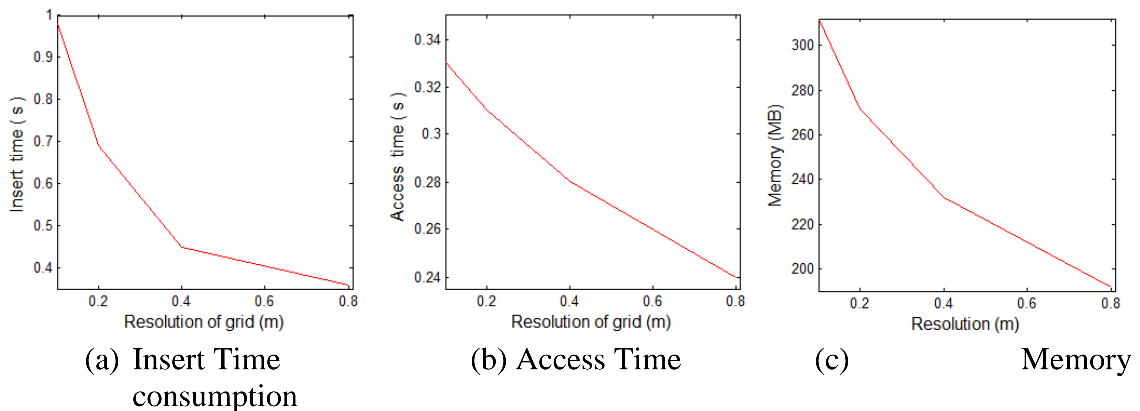
(a) Insert Time consumption (b) Access Time (c) Memory

Figure 2.7: (a)-(c) Computational time and memory comparison between Octree, K-d tree and Hash Table (10cm resolution) for randomly generated points between [-1000, 1000]m



(a) Insert Time consumption (b) Access Time (c) Memory

Figure 2.8: (a)-(c) Computational time and memory comparison between Octree, K-d tree and Hash Table (10cm resolution) for randomly generated points between [-150, 150]m



(a) Insert Time consumption (b) Access Time (c) Memory

Figure 2.9: (a)-(c) Computational time and memory for the Octree data structure for different grid resolutions and 500,000 randomly generated points between [-1000,1000]m

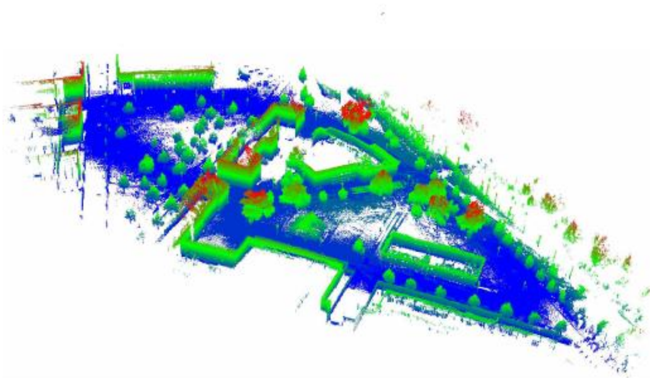
2) Real World Data Sets:

This section focuses on the comparison of the three data structures in terms of computational time and memory consumption for the real world data sets of Freiburg campus and Bremen city center[2] (Fig 2.10(a) and 2.10(b). The Freiburg campus data set consists of 77 files, each one containing approximately 155829 points. The Bremen city center data set consists of 13 files, each one containing approximately 295000 points.
Fig. 2.11(a) shows the average insertion time for each file of the Freiburg campus data set regarding the three data structures. Different grid resolution values are used.

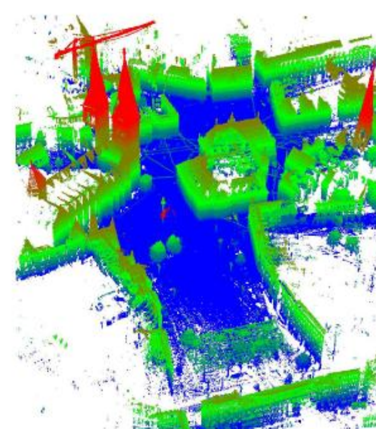
[2] Courtesy of Dorit Borrmann and Jan Elseberg available at the Osnabrueck robotic 3D scan repository, http://kos.informatik.uniosnabrueck. de/3Dscans/

Remarkable here is the performance of the octree data structure, which, in contrast to the simulation tests, achieves the lowest insertion times. The data set corresponds to a structured environment and the octree capability of modelling the 3D space significantly decreases these times. More specifically, each time a new point is to be inserted, the corresponding leaf node of the octree will be automatically reached. If the node is already occupied then its probability will be updated. On the other hand, the K-d tree and hash table structures, not having this advantage, will have to search for the new point more exhaustively to check whether it already exists or not, leading in larger complexity. More explicitly, regarding the hash table, there are many points that have similar x-coordinate (e.g., part of the ground) whose y-value will be stored as key in the vector corresponding to this x-coordinate. This can be seen in Fig. 2.6 above. In that way, the search for such points may be more complex in terms of time. However, the direct associativity of the hash table makes it faster than the K-d tree. For this reason, the time to access the whole data set is much lower for the hash table strucure, as shown in Fig. 11(b). Fig. 2.11 (c) shows the memory performance of the three data structures. The behaviour is similar to the one shown for the simluation tests above, as the same arguments apply. Finally, the K-d Tree has proven to be more efficient for nearest neighbor serch algorithms as mentioned in [48].

In Fig. 2.12(a)-(c) we can see the same results for the Bremen city center data set. In comparison with the Freiburg data set, we notice that the average insert times per file are higher, since the number of points per file is larger. However, the total number of points of this data set is less than of the Freiburg one, contributing to lower memory consumption and access times for the K-d tree and hash table. Nevertheless, noteworthy is the higher memory consumption of the octree structure. This can be attributed to the bigger map of the Bremen city center. Thus, the number of the inner nodes of the octree will be greater.

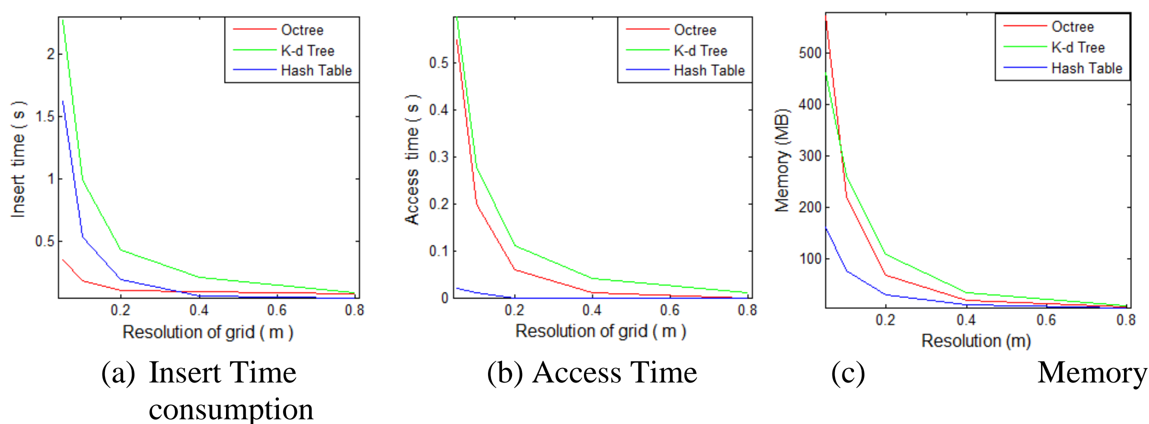


(a) Freiburg campus at 2cm resolution (292m x 167m x 28m)



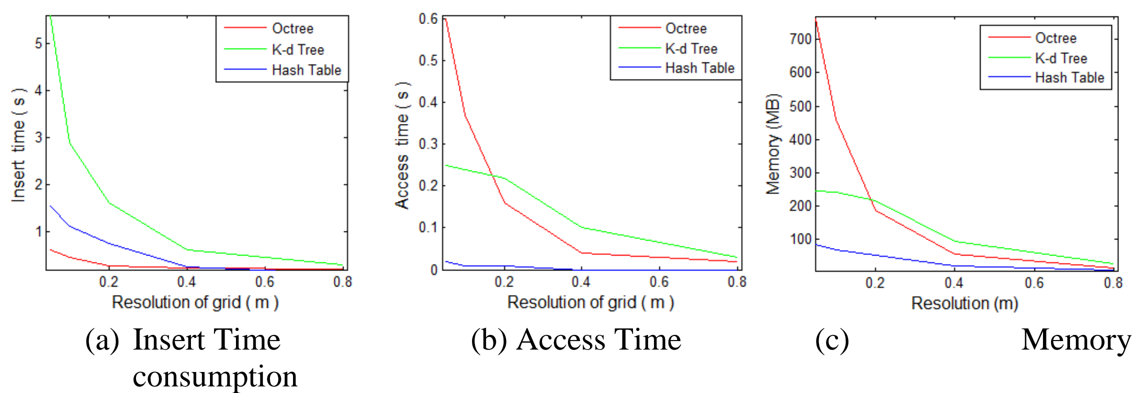
(b) Bremen city center at 2cm resolution (778m x 870m x 154)

Figure 2.10: (a),(b) Freiburg campus and Bremen city center visualizations at 2cm resolution



(a) Insert Time consumption (b) Access Time (c) Memory

Figure 2.11: (a)-(c) Computational time and memory comparison between Octree, K-d tree and Hash Table for the Freiburg campus with different grid resolution values



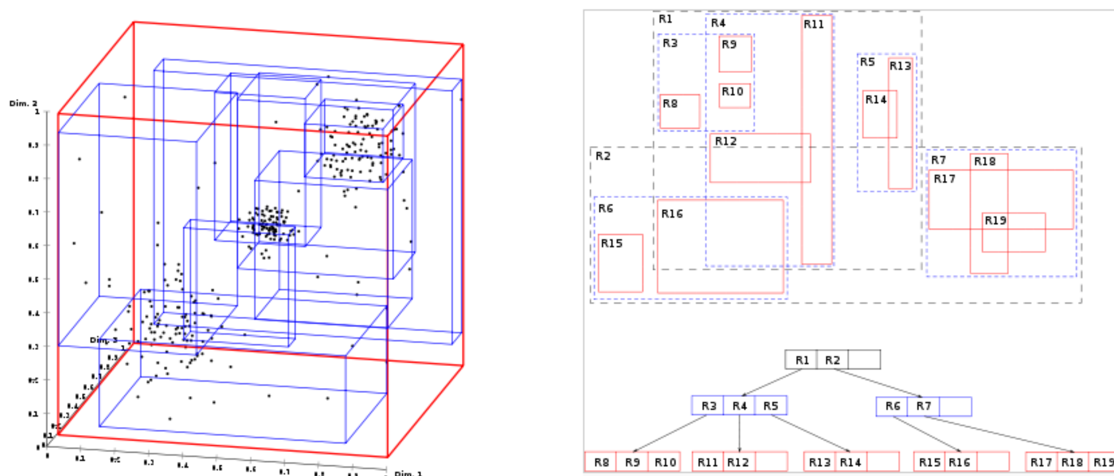
(a) Insert Time consumption (b) Access Time (c) Memory

Figure 2.12: (a)-(c) Computational time and memory comparison between Octree, K-d tree and Hash Table for the Bremen city center with different grid resolution values

3 RMAP: Environment representation using rectangles

The most common approach for 3D environment representation is utilizing raw point clouds and operating directly on them or utilizing a 3D occupancy grid. The extension of 2D occupancy grid concepts directly to 3D leads to a large over head in terms of memory and computational cost due to the explicit calculation of free space in standard approaches. In this Chapter a rectangle approximation of 3D pointclouds titled “RMAP” is proposed. The presented approach can be used either as an occupancy grid for the representation of the environment or as an approximation using point density. The capability of grouping points in arbitrary shapes, such as rectangles, both in 2D and 3D space can lead to high computational efficiency. We use the concept of the bounding box to form rectangles around groups of points. In that way, except for the savings we achieve in terms of time and memory, as the number of rectangles is significantly less than the total number of points, the environment can be approximated to any arbitrary resolution, according to the application (navigation or registration) or computational resources available. Moreover, flexibility is gained as any arbitrary axis aligned rectangle can be utilized. Appropriate data structures for the needs of environment representation with rectangles would be the R-trees with its variants [26], [27]. A brief description about R-tree is presented below.

Section 3.1 considers the framework of the occupancy grid for representing 3D environments, whereas Section 3.2 deals with the approximation of 3D pointclouds by introducing a density-based splitting algorithm. The concept of RMAP is extended to rectangular approximation of 2D pointclouds and polygonal convex segments, presented in Chapter 4.



(a) Example of 3D R-tree

(b) Example of 2D R-tree

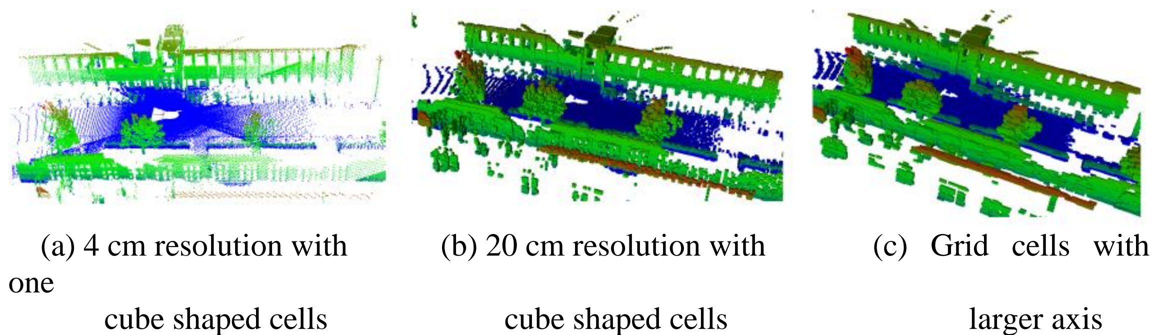
Figure 3.1: (a), (b) 3D and 2D R-tree structure

R-trees

R-trees are tree data structures used for spatial access methods. The idea here is to group nearby objects (e.g. nearby points) and represent them with their minimum bounding rectangle in the next higher level of the tree. At the leaf level, each rectangle describes a single object whereas at higher levels the aggregation of an increasing number of objects. The R-tree is a balanced search tree, so all the leaves are at the same height, and every node (apart from the leaf nodes) contains information about its bounding box and pointers to its children, whose bounding boxes lie inside the bounding box of their parent node. The bounding box is defined by four values (the minimum and maximum x, y coordinates) for a rectangle in the 2D space or six values (the minimum and maximum x, y, z coordinates) for a cube in the 3D space. These values are sufficient for describing all the vertices of a rectangle or a cube, since we deal with axis-aligned bounding boxes. The insertion of a new rectangle is done by using a heuristic such as choosing the rectangle that requires least enlargement and the search is done based on overlapping rectangles. The searching algorithms of an R-tree (usually based on intersection) use the bounding boxes to decide whether or not to search inside a subtree. The time complexity here is O(log$n$) for the procedures of insertion, accessing and deletion of a rectangle both in the average case and in the worst case, where n is the number of nodes (e.g. rectangles) in the tree. The memory complexity is O(n). Furthermore, variants of the R-tree could be employed, like the R*-tree and the R+-tree, which try to minimize the overlap at leaf and internal level, respectively. [26] and [27] give more detailed information about R-trees and its variants. The use of R*-trees for storing points and rectangles efficiently can be found in [28].

3.1 Occupancy grid framework

For this approach, a small rectangle is fitted around each point and stored in the R-tree. The size of the rectangle depends on the desired resolution. Each rectangle represents an occupancy cell with a certain probability, which is updated, either by ray-casting or by encountering a 3D point that belongs to the same occupancy rectangle (that is, the rectangles of two different points overlap) in a similar way to the octree data structure presented in Section 2.3. That means that if the rectangle to be inserted overlaps with a rectangle in the leaf nodes, it is not inserted. Instead, the probability of the specific node is updated. Fig. 3.2(a) and 3.2(b) show the first scan of the Freiburg campus at different resolutions with cubic grid cells. Fig. 3.2(c) shows the case when the basic grid cell has a greater length along one axis. This flexibility property can be exploited in cases where there is a prior information about the structure of the environment and the movement of the robot.



(a) 4 cm resolution with cube shaped cells | (b) 20 cm resolution with cube shaped cells | (c) Grid cells with one larger axis

Figure 3.2: Different resolution views of the 1st scan of the Freiburg campus data set

Experimental Evaluation

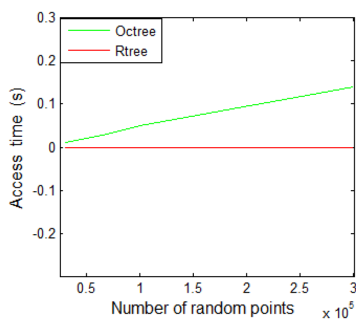
This section focuses on the evaluation of the approach, mainly in terms of insertion and extraction times as well as memory consumption. Furthermore, a comparison will be conducted to the implementation of the octree structure employed in Section 2.3. A publically available implementation of the R-tree data structure [29] was employed. It was also properly modified to integrate probabilistic properties, as the data structures discussed in Section 2.3. The evaluation was conducted both for simulation data by inserting randomly generated (30,000, 70,000, 100,000 and 300,000) points in range [-1000, 1000]m and retrieving all occupied grid cells and for real world data sets (Freiburg campus data set). The memory of the R-tree in both cases was computed according to the following formula (in bytes):

$$Memory_{R\text{-}tree} = InnerNodes\ x\ (18 + Branches \text{ x } 36) + LeafNodes \text{ x } (18 + Branches \text{ x } 28)$$

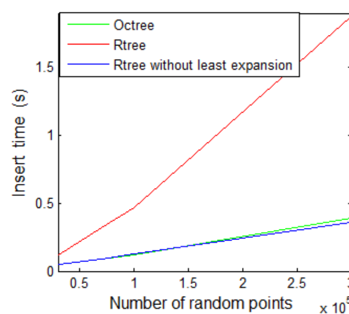
and was derived as follows. Each node consists of 2 integers (one for the number of its branches and one for the current level (height) of the Rtree), 2 booleans (to check whether it is an inner node or a leaf node) and a pointer to an array of branches, leading to 18 bytes. Each branch contains 6 integers describing the bounding box, 1 integer describing the probability of being occupied and a pointer to the next node, if the branch is inner. In total, inner branches consume 36 and leaf branches 28 bytes.

1) Simulation results:

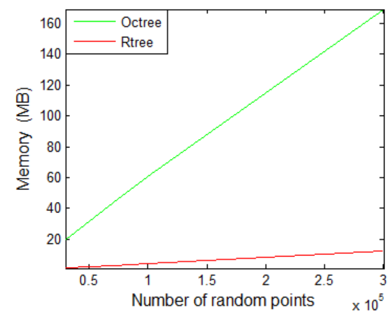
Fig. 3.3(b) shows the insertion times for the R-tree and the octree structures for randomly distributed points. For small number of points the times for the two structures are comparable but as the number of points increases the insertion time for the R-tree becomes larger. This can be mainly attributed to more computationally expensive heuristics for the insertion procedure such as the least expansion, used in the R-tree data structure. As discussed in the data structure description, each time a rectangular cuboid is inserted, the structure tries to find the best inner node which leads to minimum expansion of the tree structure. If there exist no overlapping bounding boxes at the inner nodes, the choice is obvious, but if bounding boxes of inner nodes overlap, this process can be computationally expensive. Secondly if an inner node overflows (the number of branches exceeds the maximum), the node has to be split and the bounding boxes of all intermediate nodes need to be recomputed. Elimination of the 'least expansion' process in this approach can lead to large computational savings (however it also leads to an unbalanced tree) as can be seen from Fig. 3.3(b). There exist variants of the basic R-tree structure such as the R+ tree and the R* tree [30], [31], [32], which can handle this scenario better and can lead to better computation times. Most 3D sensors (such as Kinect) provide 300,000 points per frame, however in most cases the data is downsampled or ignored after some distance (mainly 4-5 m for Kinect due to increasing error with distance), leading to a more manageable number of points (assuming between 30,000 to 100,000) for the RMAP approach. Fig. 3.3(c) shows the memory consumption (with increasing number of random points) of both approaches which is directly dependent on the depth/height of the tree structure. It can be seen from the figure that RMAP utilizes less memory than the Octree approach. This can be explained by a simple example by considering the insertion of a single point. The Octree would generate all intermediate branches and nodes to insert the point at a specific resolution whereas the R-tree structure would insert a single rectangular cuboid around the point.



(a) Access Time consumption



(b) Insert Time

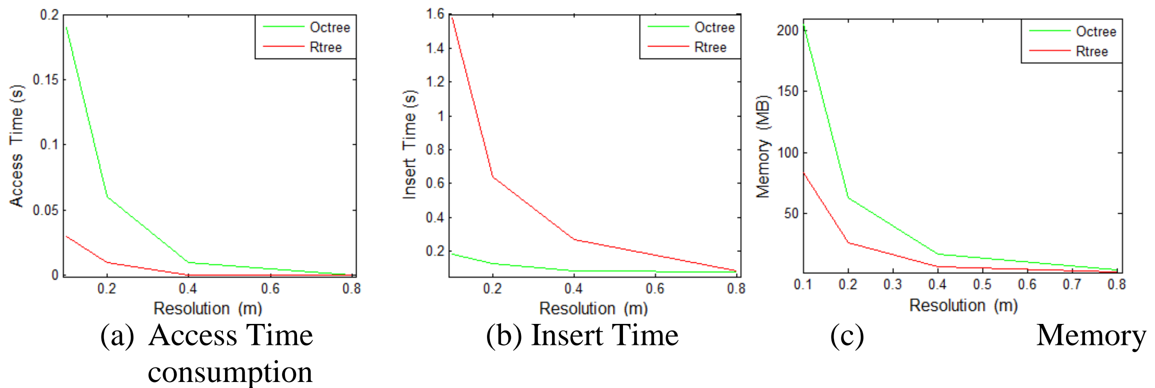


(c) Memory

Figure 3.3: (a)-(c) Computational time and memory comparison between Octree and R-tree (20cm resolution) for randomly generated points between [-1000, 1000]m

2) Real World Datasets:

The first section dealt with simulated data. In order to extensively evaluate the approach, different scans of the Freiburg campus were used. A comparison of RMAP with the same implementation of an octree structure was done for computational costs on 70 scans of the Freiburg campus. Fig. 3.4(a) presents the access time of all occupied cells (for an average number of points i.e. 155829) for different resolutions and reinforces the results of the simulation (presented in previous subsection) setup that RMAP can be used for online motion planning in 3D environments. Fig. 3.4(b) shows the insertion time (for the same number of points) for different resolutions. At low resolutions, the computational times are quite comparable, but at high resolutions the least expansion aspect can lead to larger computational costs. However, in a real world environment like the Freiburg campus data set the least expansion procedure cannot be avoided, since it organizes the inner nodes in such a way that the structure of the environment is conceived in the tree. Fig. 3.4(c) shows the memory (as per the formula in the previous section) usage of both approaches for different resolutions. The difference in memory consumption can be explained based on the same argument given in the previous subsection. To be more explicit, consider the memory consumption corresponding to 10 cm resolution shown in Fig. 3.4(c) for which the height of the Octree structure is 13 in comparison to 7 for the RMAP approach.



(a) Access Time consumption (b) Insert Time (c) Memory

Figure 3.4: (a)-(c) Computational time and memory comparison between Octree and R-tree with respect to grid resolution for the first 70 files of the Freiburg campus data set

The proposed approach does not differentiate between free and unknown space for the entire grid (only dynamic cells). This is not a critical issue for most robotic applications such as navigation and registration, however it can be problematic for some exploration algorithms which generally use the unknown and free space in calculating the utility of a frontier. This issue can be addressed from two perspectives. Firstly, most robotic architectures utilize two layers for navigation (global and local representations of the

environment). RMAP can be used to make a global map, whereas the exploration strategy can be based on generating frontiers on the local map which can explicitly model free space and unknown space. These frontiers can be marked on the global map to stop the algorithm from exploring already explored areas. Secondly there exist exploration algorithms [33] which place frontiers on the global map corresponding to a certain radius based on the robot position or the maximal field of view of the sensor and thus do not require explicit free and unknown space modeling.

3.2 3D Rectangle Approximation

A formulation like the one discussed in the previous Section allows a probabilistic framework for 3D mapping, however it does not justify the full potential of the RMAP approach. The occupancy grid framework from Section 3.1 could be considered a naive one, since despite the fact that flexibility is achieved, storing a rectangle for each single point (according to the grid resolution) does not necessarily lead to great computational savings in time and memory. In contrast, the extraction of large rectangular sections from a 3D point cloud would be a much more efficient approach in comparison to an occupancy grid. Rectangles would be formed by groups of points, hence their number would be smaller, resulting in lower computational times and memory consumptions. In that way, the basic R-tree property and advantage of storing any arbitrary axis-aligned rectangles would be fully exploited.
In this section we discuss a formulation that splits the maximal bounding box of the pointcloud corresponding to its density to obtain rectangular approximations of the environment. Since the algorithm is dependent on density, we utilize a fully registered pointcloud. This approach can be used as a post processing step after registration to reduce computational and memory complexity of point cloud storage.
The pseudocode used for the 3D rectangle approximation is shown in Fig. 3.5. The input to the algorithm is the point cloud p to be approximated and the density and minimum volume ε, ω respectively. The threshold ε is transformed into the threshold β, as discussed below. The output is a set of rectangles R which approximate the point cloud. The algorithm starts by checking if the number of points is less than four in line 1 (at least four points are required to define a volume). If so, these points are considered to be undefined. If not, the maximal bounding box for the point cloud is calculated based on the minimal and maximal points (line 2). In addition the algorithm calculates the volume V of the bounding box and the point density (line 3 and 4). The volume and density are compared to the density and minimum volume threshold β, ω. If the density is greater than β or the volume is less than ω, the bounding box is stored in the set R, otherwise the algorithm splits the point cloud into 8 equal parts with respect to the center (line 7). Each split point cloud is recursively passed to the algorithm (Fig. 5) until the threshold is satisfied (line 8). The formulation presented in this section can be extended to work on point cloud data provided by segmentation algorithms. An incremental version of this approach can also be formulated however it would be more sensitive as the 3D point cloud density would change as more scans are accumulated and would require merging

along with splitting, further increasing the computation time of the algorithm. β is derived from the initial density (points/m^3) of the pointcloud according to the formula:

$$\beta = initial\ density\ /(1-\varepsilon),$$

where ε is the input normalized threshold and takes values in the range (0,1). In the case where $\varepsilon = 0$, then it would be $\beta = initial\ density$ and no approximation would be conducted. In the case where $\varepsilon = 1$, it would be $\beta = \infty$ and the threshold would never be satisfied. Hence, the algorithm would continue running until the threshold ω is reached and all the points until that point would be undefined. So we can conclude that the higher the threshold ε is, the more accurate is the approximation.

```
3DRectangle_approximation(p, ε, ω)
Input:
      Point cloud p,
      Approximation percentage ε,
      Density threshold β = initial density of p / (1 − ε),
      Minimum volume threshold ω
Output:
      Set of rectangles R = {r1, r2.., rn}
Procedure:
1    if (points in p ≤ 3)
         store as undefined points;
2    Compute bounding box b of point cloud p;
3    Compute volume V of bounding box b;
4    Compute point density d, where d = points in p / V;
5      if (d ≥ β) or (V ≤ ω) then
6            R = R + b
       else
7            split p into 8 equal parts {p1, p2.., p8}
             with respect to the center
             for all pj do
8                 3DRectangle_approximation(pj, ε, ω);
             end
       end
```

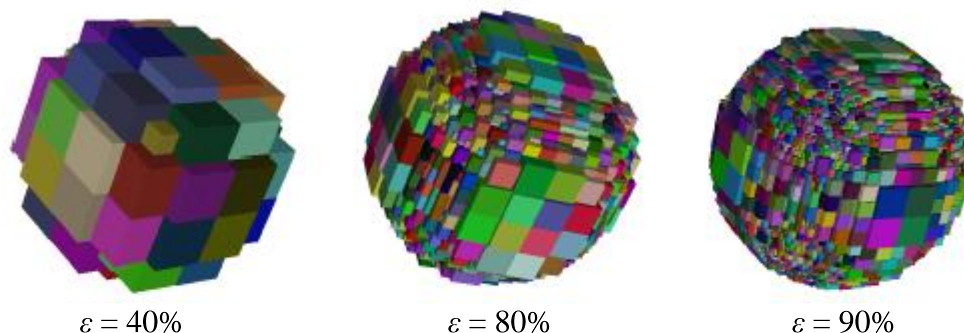
Figure 3.5: 3D rectangle approximation pseudocode

Experimental evaluation

In this section we evaluate the approach on simulation and real datasets based on memory and computational complexity. The algorithm is capable of approximating complex 3D environments however the computational complexity for now limits it for offline application.

1) Simulation datasets:

The algorithm was evaluated on simulated 3D pointclouds. Fig. 3.6 shows the approximation of sphere utilizing different thresholds ε which define the detail of the approximation. It can be seen from the images that as ε is increased the rectangles generate a more accurate approximation.

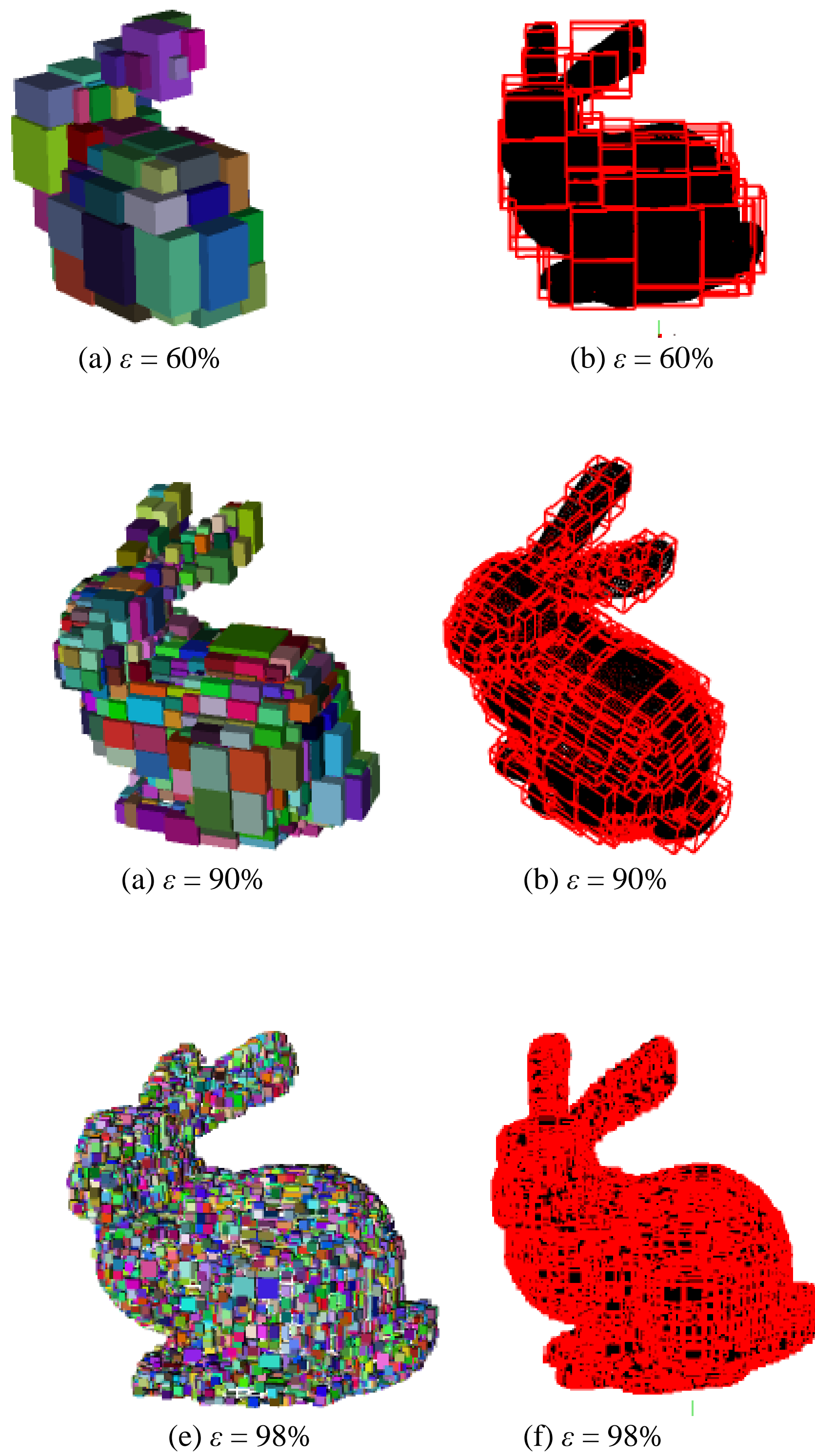


$\varepsilon = 40\%$ $\varepsilon = 80\%$ $\varepsilon = 90\%$

Figure 3.6: Sphere shaped point cloud rectangle approximation using different thresholds

Apart from the artificially created sphere, the algorithm was also tested on the Stanford repository bunny and dragon data sets[3] which consist of 35947 and 437645 points respectively. Fig. 3.7 and 3.9 show the approximation results for these two data sets for different approximation thresholds ε, leading from a coarse to a finer approximation. Fig. 3.8 and 3.10 depict the approximation time, the memory consumption, the number of rectangles and the number of undefined points (Fig. 3.5 -line 1) for these two data sets. Regarding the approximation time, we can tell from Fig. 3.8(a) and 3.10(a) that, except for the threshold, it is highly dependent on the number of points, as for the dragon data set the time needed can be over 45 seconds for high ε values whereas for the bunny dataset the time does not exceed the 1 second. The memory consumption for the two datasets depends on the number of rectangles that need to be stored which increases as the threshold ε increases. However, the total number of rectangles (maximum of 60000 for the dragon data set) is much lower than the points that consist the initial 3D pointcloud achieving great memory savings. Fig. 3.8(c) and 3.10(c) depict the loss of information (undefined points) for the two data sets, which also increases as the threshold ε increases. That is an expected outcome, as discussed above in the pseudocode explanation. Nevertheless, we can see that the number of undefined points is not large, especially in high density pointclouds such as the dragon data set.

[3] Courtesy of Stanford University Computer Graphics Laboratory, available at The Stanford 3D Scanning repository, http://graphics.stanford.edu/data/3Dscanrep/

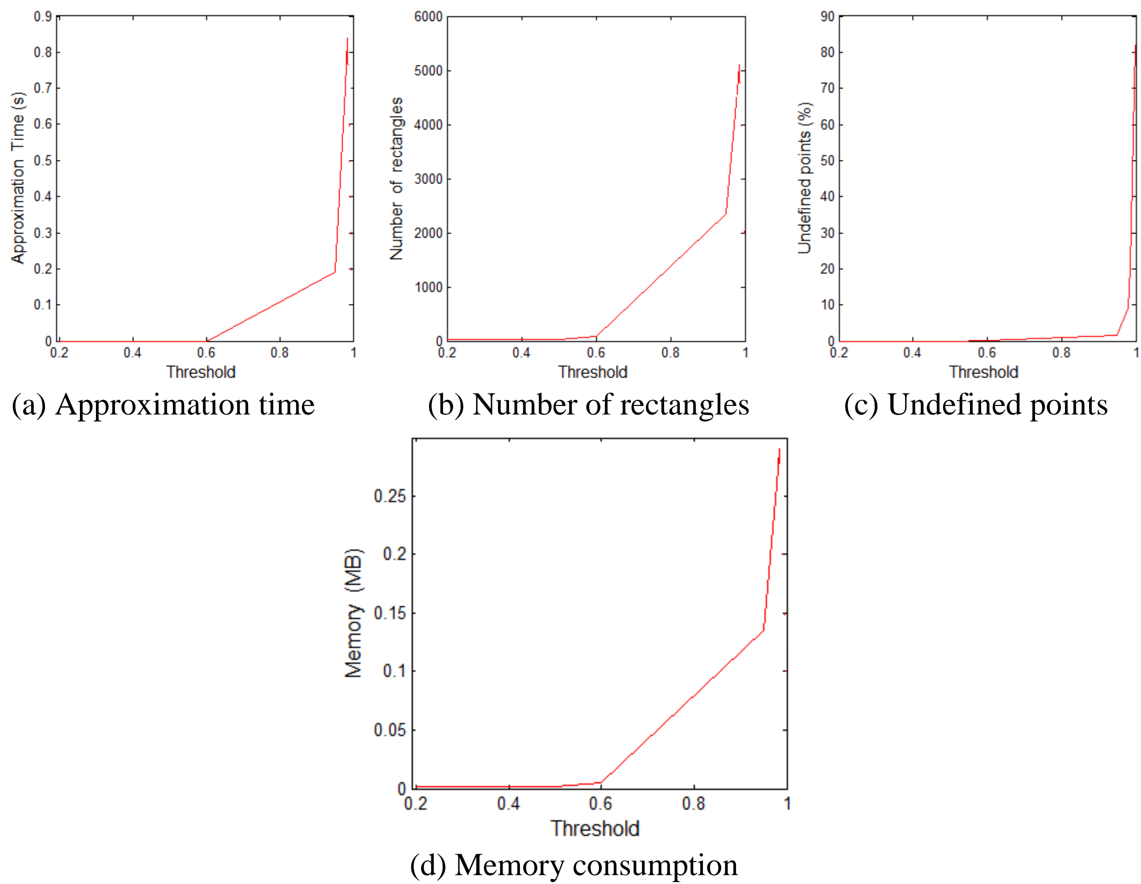


(a) $\varepsilon = 60\%$ (b) $\varepsilon = 60\%$

(a) $\varepsilon = 90\%$ (b) $\varepsilon = 90\%$

(e) $\varepsilon = 98\%$ (f) $\varepsilon = 98\%$

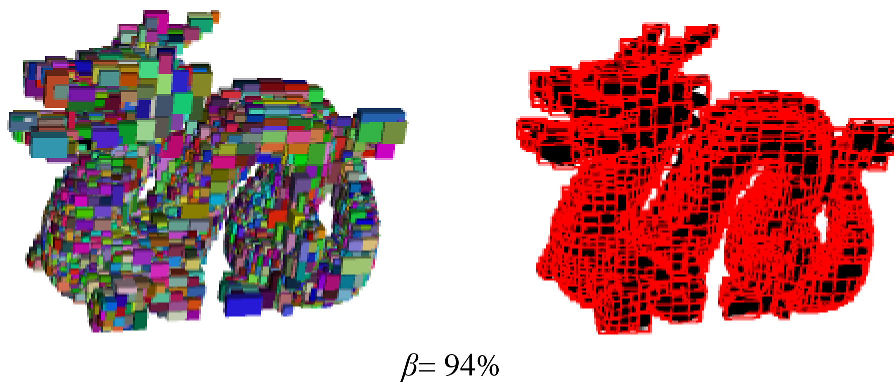
Figure 3.7: Rectangle approximation of the Stanford repository bunny for different density thresholds.



(a) Approximation time (b) Number of rectangles (c) Undefined points

(d) Memory consumption

Figure 3.8: (a) Approximation time, (b) Total number of rectangles, (c) Number of undefined points and (d) Memory consumption with respect the threshold β for the Stanford repository bunny



β= 94%

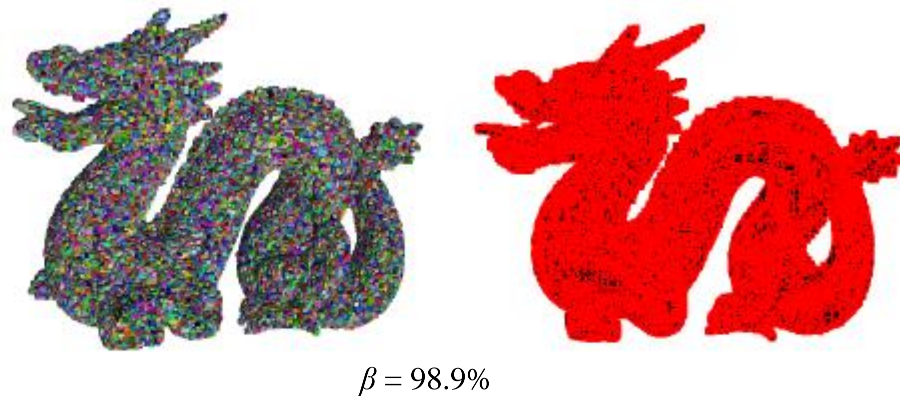
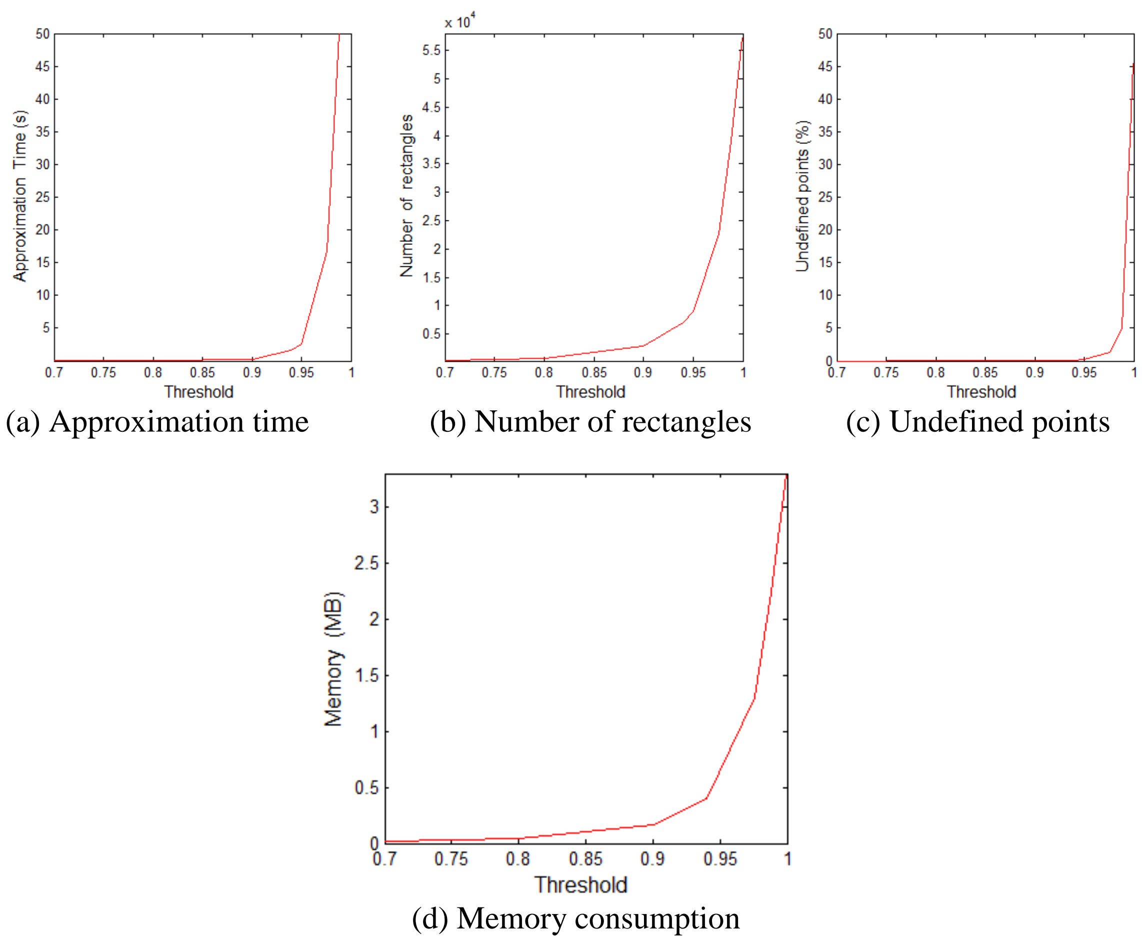


Figure 3.9: Rectangle approximation of the Stanford repository dragon for different density thresholds.



(a) Approximation time (b) Number of rectangles (c) Undefined points

(d) Memory consumption

Figure 3.10: (a) Approximation time, (b) Total number of rectangles, (c) Number of undefined points and (d) Memory consumption with respect the threshold β for the Stanford repository dragon

2) Real world datasets:

The algorithm was also evaluated on real world data sets. More specifically, different scans of the Freiburg campus and the Bremen city center data sets were utilized. Fig 3.11 shows the rectangular approximation of a specific section (a tree and pole) of the Freiburg campus. The image shows that the algorithm adapts the size of the rectangles according to the density of the point cloud. Fig 3.12(a) and Fig 3.12(b) show the rectangular approximation of 10 scans of the Freiburg campus for ε = 98% and ω = 1mm^3 and 4 scans of the Bremen city center for ε = 95% and ω = 1 mm^3 respectively. Fig. 3.13 shows the approximation time, the memory consumption, the number of rectangles required for representation and the undefined points for the first 10 scans of the Freiburg campus dataset. As the approximation threshold is increased the algorithm is able to build a finer approximation of the environment as can be observed with the increase in the number of rectangles and memory. Another important observation is that the loss of information is at most 3.5% of the actual point cloud size.
It is interesting to compare the RMAP rectangular approximation framework with the occupancy grid framework to determine the advantages/disadvantages of each formulation. However defining a criterion for comparison is quite difficult, since the former approach utilizes different sized grid cells based on density whereas a fixed cell size (for a specific resolution) is used in the occupancy grid framework making it difficult to determine the equivalency of representation. Another important factor is that the rectangular approximation framework also rejects points, hence making it difficult
to quantify the loss of information in comparison to the loss of information of a multi resolution occupancy grid. Hence in order to roughly compare both approaches, a conclusion which follows intuition and is based on a wide variety of possible approximations achievable by both approaches can be considered. Fig. 3.14 shows the computation time, number of rectangles required for representation and the memory consumption for the both approaches on 70 files of the Freiburg campus which contains 10984515 points (almost 292m x 167m x 28m). The approximation time in Fig. 3.14(a) shows the computation time for the presented approach as well as for the insertion of the rectangles into the R-tree. The comparison shows that for large point clouds, the rectangular approximation is computationally expensive (limiting it to an offline approach) compared to an occupancy grid formulation (computation time based on average number of points per file). The number of rectangles required for the rectangular approximation Fig. 3.14(c) is less than the number of rectangles required for an occupancy grid formulation (Fig. 3.14(d)) which is fairly intuitive if Fig. 3.14(a) is kept in mind. A similar intuition follows for the memory consumption as can be seen in Fig. 3.14(e) and Fig. 3.14(f).

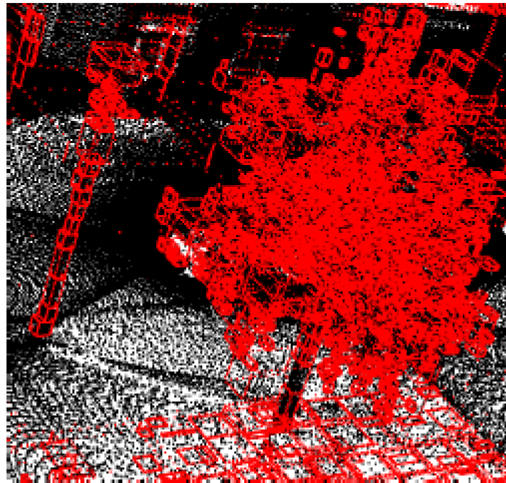
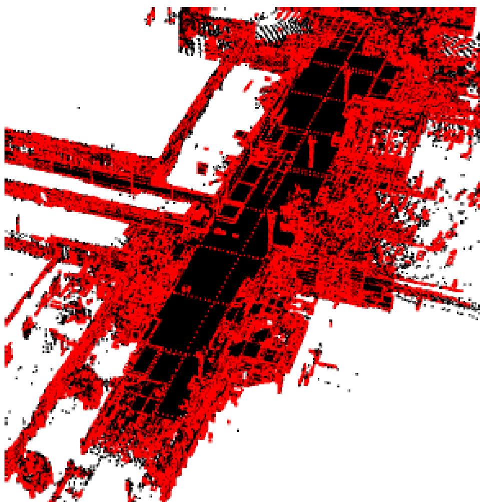
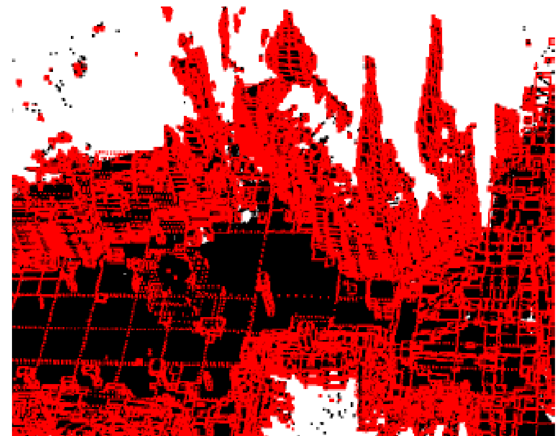


Figure 3.11: Rectangular approximation of a tree and a pole from the first scan of the Freiburg campus data set ($\varepsilon = 98\%,\ \omega = 1\text{mm}^3$)

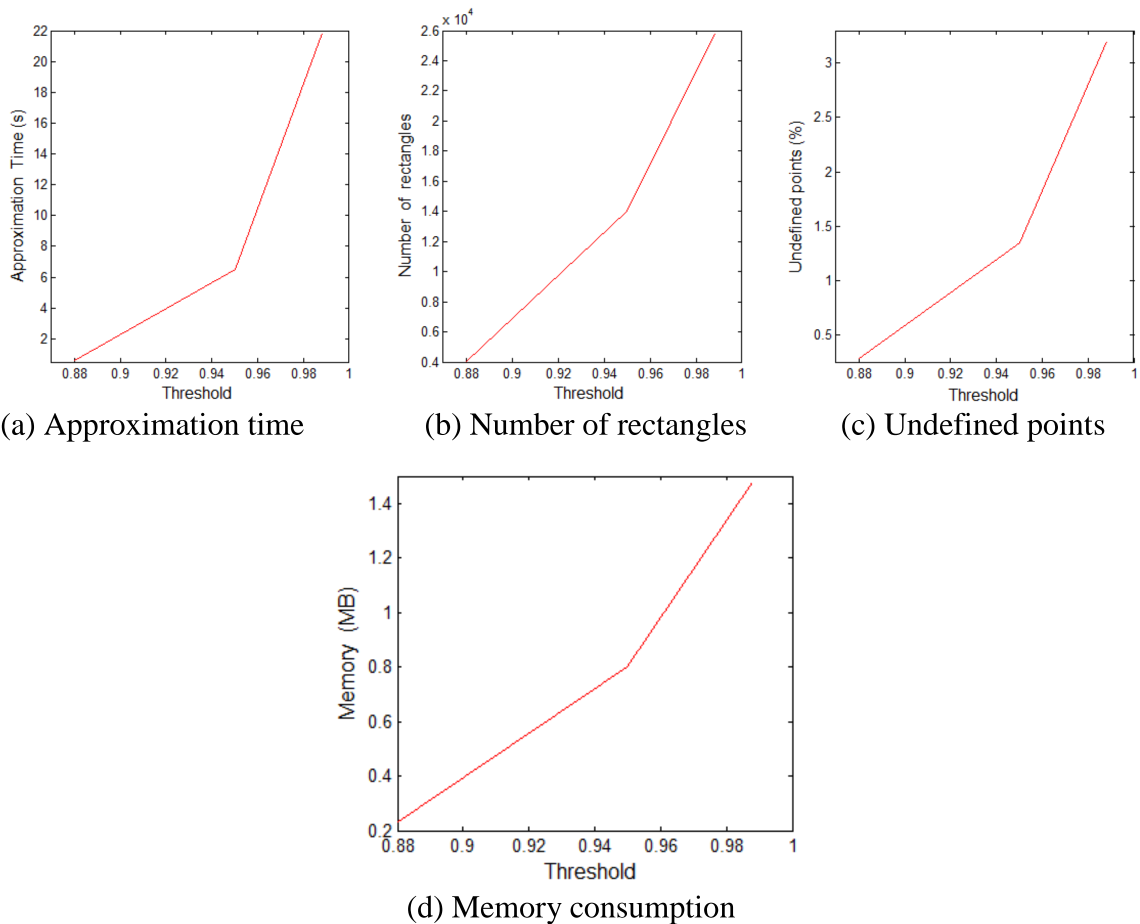


(a) Rectangular approximation of the first 10 scans of the Freiburg campus dataset ($\varepsilon = 98\%,\ \omega = 1\text{mm}^3$)



(b) Rectangular approximation of the first 4 scans of the Bremen city center dataset ($\varepsilon = 95\%,\ \omega = 1\text{mm}^3$)

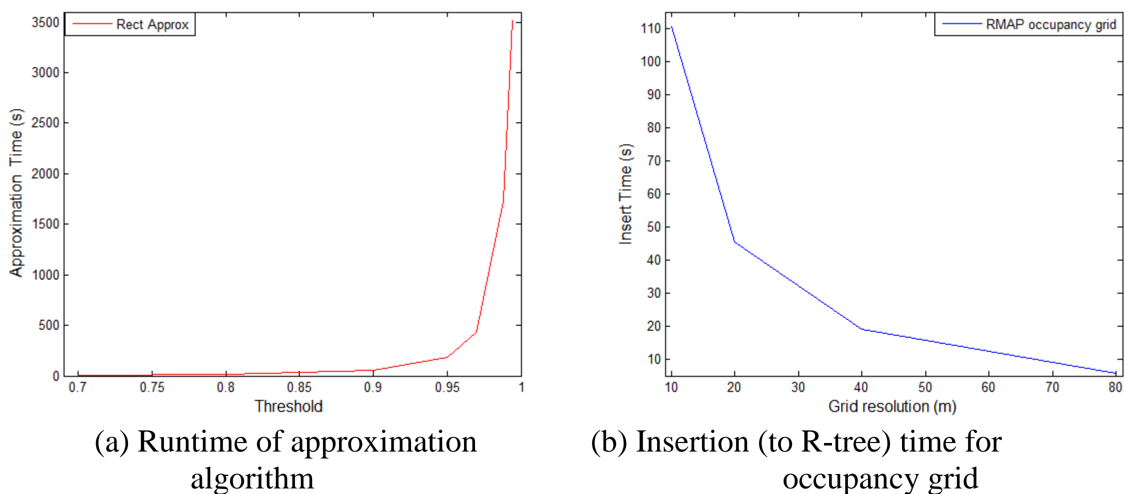
Figure 3.12: Rectangular approximation of the first (a) 10 files of the Freiburg campus dataset and (b) 4 scans of the Bremen city center dataset



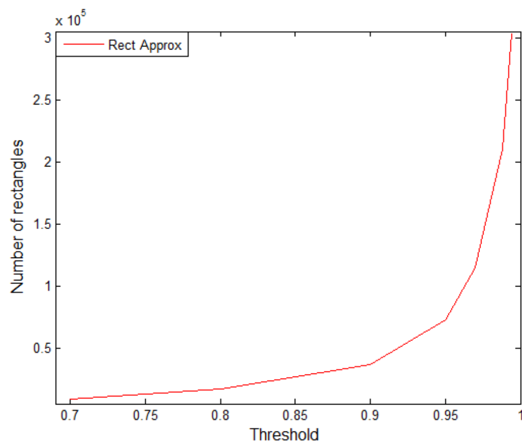
(a) Approximation time (b) Number of rectangles (c) Undefined points

(d) Memory consumption

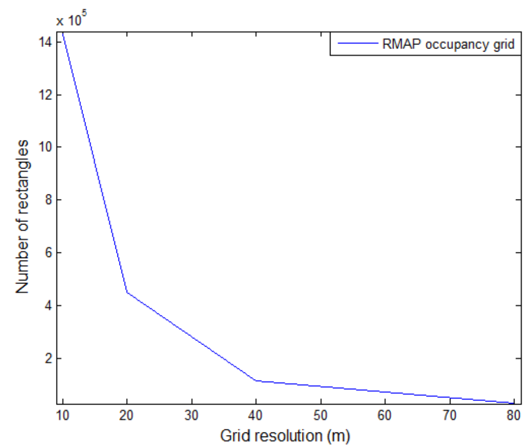
Figure 3.13: (a) Approximation time, (b) Total number of rectangles, (c) Number of undefined points and (d) Memory consumption with respect the threshold β for the first 10 files of the Freiburg campus data set



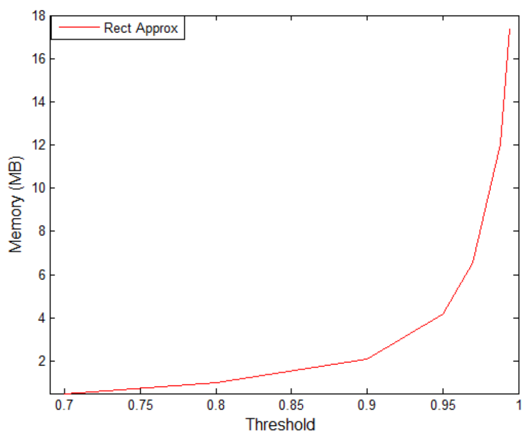
(a) Runtime of approximation algorithm (b) Insertion (to R-tree) time for occupancy grid



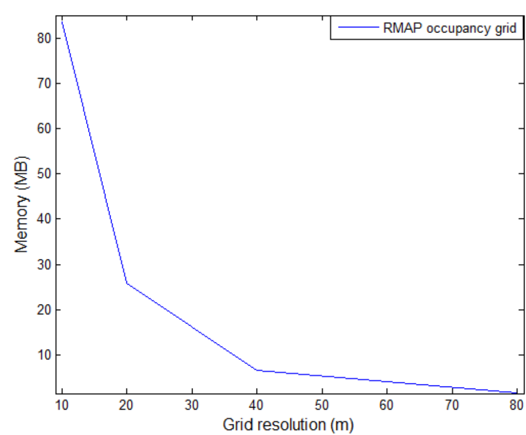
(c) Number of rectangles of approximation algorithm



(d) Number of rectangles for occupancy grid



(e) Memory consumption of approximation algorithm



(f) Memory consumption for occupancy grid

Figure 3.14: Computation time, number of rectangles and memory consumption in comparison to an occupancy grid formulation for 70 files of the Freiburg campus dataset

4 RMAP: Environment representation using planar convex polygons

Many robotic applications utilize plane extraction algorithms [34] from 3D pointclouds. These planar segments are used in several scenarios, such as 3D mapping, navigation and registration. These procedures require high representation accuracy and efficiency in terms of computational time and memory. The points that form the planar segments are utilized for their polygonization, calculating the convex hull and formulating the planar convex polygons. The latter are described by the planar attributes, such as the normal vector and the offset and the vertices of the polygon. Hence, their use leads to computational savings in memory in comparison to the use of 3D points. Furthermore, representing an environment using such polygons can be more intuitive in the sense that a human being does not perceive the real-world as points, but more like planar surfaces and polygons, especially in structured environments.

However, due to the complexity of convex polygons (inconstant and large number of vertices required for the description of the convex hull), until now no data structure that can store them efficiently has been found. In this Chapter we discuss the effectiveness of the extension of the RMAP approach to approximate convex polygons using rectangles, exploiting the advantages of the R-tree data structure. This approach is divided in two main parts. In Section 4.1 we introduce a splitting algorithm to approximate efficiently a convex polygon and in Section 4.2 we discuss an extension of the 3D approximation algorithm presented in Section 3.2 to planar segments.

4.1 Rectangle approximation of convex polygons

The application of RMAP approach presented in Section 3.1 is not just limited to an occupancy grid formulation using rectangular cuboids. It can be extended to approximate polygonal regions extracted by standard segmentation algorithms [34], [35], [36] in order to represent the 3D environment. Quadtrees (2D environments) have been used before for approximation of convex polygons, however R-trees have been shown to be more efficient in terms of memory and computation time [37]. Although the discussion presented in this section and the experimental evaluation section are in context of convex planar polygons (hence we use the word rectangle instead of rectangular cuboid as one axis is degenerate), RMAP is more general and can be utilized for any polygonal set. A polygonal approximation of convex polygons is also considered in [38].

Fig. 4.1 shows the polygonal approximation pseudocode utilized in the RMAP approach. The robot utilizes a segmentation algorithm to extract the convex hull of planar segments denoted as p_i (i-th planar segment). The bounding box denoted as b_i for the convex hull can be created from the minimum and maximum values of the segment in each axis. Since the R-tree structure only allows axis aligned rectangles, the planar polygons are first projected along a specific axis (storing the values required for projection) and then calculating the bounding box. The arguments passed to the pseudocode are the convex hull of the i-th planar segment p_i, the area ratio threshold ζ and the minimum permissible area γ. The threshold ζ denotes the ratio between the area of the bounding box and the convex hull of the polygon. Hence ζ can be termed as the resolution parameter, since varying this parameter allows a very fine or rough approximation of the planar polygons. The parameter γ defines the minimum area that can be used to represent the regions of the convex hull. A large value for this parameter leads to a very coarse approximation. The algorithm starts by computing the bounding box of the input segment (line 1). If the area of the segment's convex hull is larger than γ (line 2), the convex hull is compared to the area of the bounding rectangle (line 3). If the overlapping area between them is lower than the resolution parameter ζ, the planar segment is split into four parts (line 5) with respect to its centre and the function is called recursively for all of them (lines 6). The process continues until the bounding boxes approximate the planar segments desirably (depending on the resolution parameter) or until the minimum area γ is reached. In general the overall detail captured by the polygonal approximation algorithm is highly dependent on the results of the segmentation algorithm.

Approximate_polygon($\mathbf{p}^i$, ζ, γ)
Input:
i^{th} Planar segment $\mathbf{p}^i$,
Area ratio threshold ζ,
Minimum segment area γ
Output:
Polygon approximation $\mathbf{b}^{\text{approx}} = \{\mathbf{b}_1, \mathbf{b}_2 .., \mathbf{b}_n\}$
Procedure:
1 Compute bounding box $\mathbf{b}$ of planar segment $\mathbf{p}^i$;
2 **if** area($\mathbf{p}^i$) $\geq \gamma$ **do**
3 **if** area_ratio($\mathbf{p}^i$,$\mathbf{b}$) $\geq \zeta$ **do**
4 store $\mathbf{b}$
else
5 split $\mathbf{p}^i$ into 4 equal parts $\left\{\mathbf{p}_1^i, \mathbf{p}_2^i .., \mathbf{b}_4^i\right\}$
for all $\mathbf{p}_j^i$ **do**
6 Approximate_polygon($\mathbf{p}_j^i$,ζ,γ);

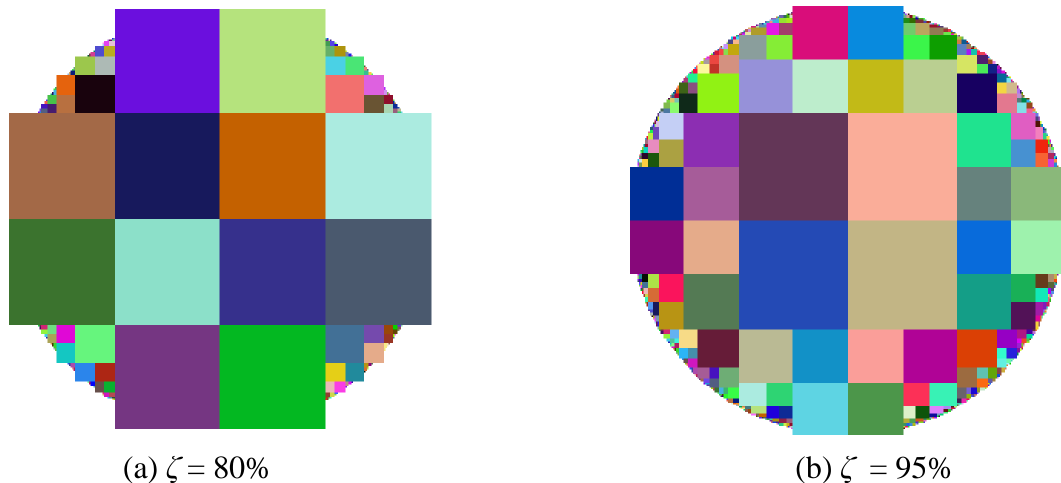
Figure 4.1: Polygonal approximation pseudocode

Experimental evaluation

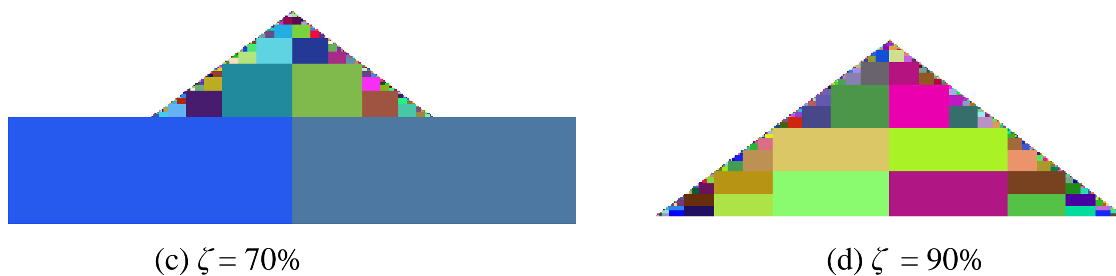
The RMAP polygonal approximation framework is evaluated for simulation as well as real world datasets. The details captured by this framework are dependent on the results of the segmentation algorithm as well as on ζ and γ thresholds.

1) Simulation results

Triangles and circles were chosen for this subsection as they can be considered to be the most difficult shapes to check the effects of different thresholds (ζ and γ). Fig. 4.2(a), 4.2(b) and Fig. 4.2(c), 4.2(d) show the results of varying ζ threshold for a fixed value of γ. As ζ is increased the polygonal approximation becomes more accurate. Fig. 4.3 shows the effect of γ, which leads to a coarse approximation of the polygon as can be seen by comparing the edges of the circle and the triangle of Fig. 4.2 (γ = 10 cm^2) to the circles and triangles in Fig. 4.3 (where γ = 10 mm^2). The colors of the rectangles shown in these images are assigned randomly to aid visualization.
In order to evaluate the time complexity of the algorithm, an increasing number of circles was approximated. Fig. 4.4(a)-(b) shows the runtime of the algorithm for this simulation scenario, for two different values of γ (10cm^2 and 10mm^2) and fixed $\zeta = 95\%$. We can see that the time increases linearly with respect to the number of circles. Furthermore, even for the case where 500 circles are approximated for γ = 10mm^2, the time does not exceed the value of two seconds, which verifies the computational efficiency of the algorithm.

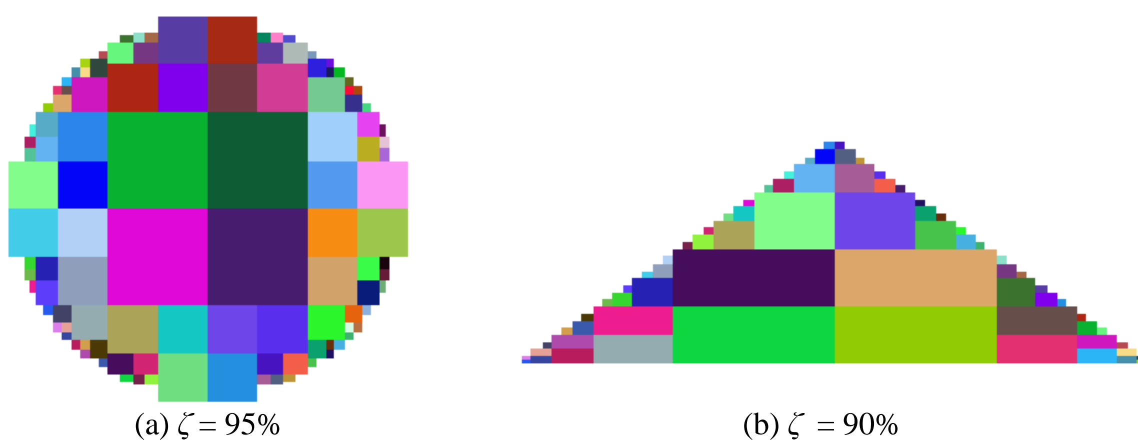


(a) $\zeta = 80\%$ (b) $\zeta = 95\%$



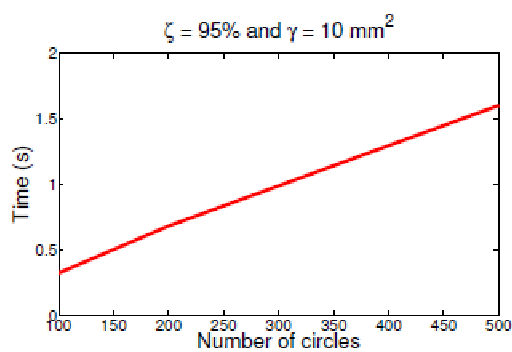
(c) $\zeta = 70\%$ (d) $\zeta = 90\%$

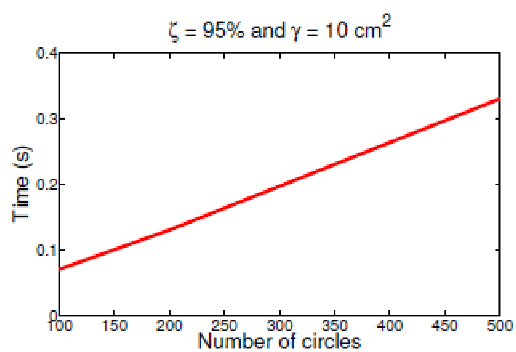
Figure 4.2: Effect of ζ for fixed $\gamma = 10\text{mm}^2$ for 2D circle and triangle



(a) $\zeta = 95\%$ (b) $\zeta = 90\%$

Figure 4.3: Effect of $\gamma = 10\text{cm}^2$ for 2D circle and triangle



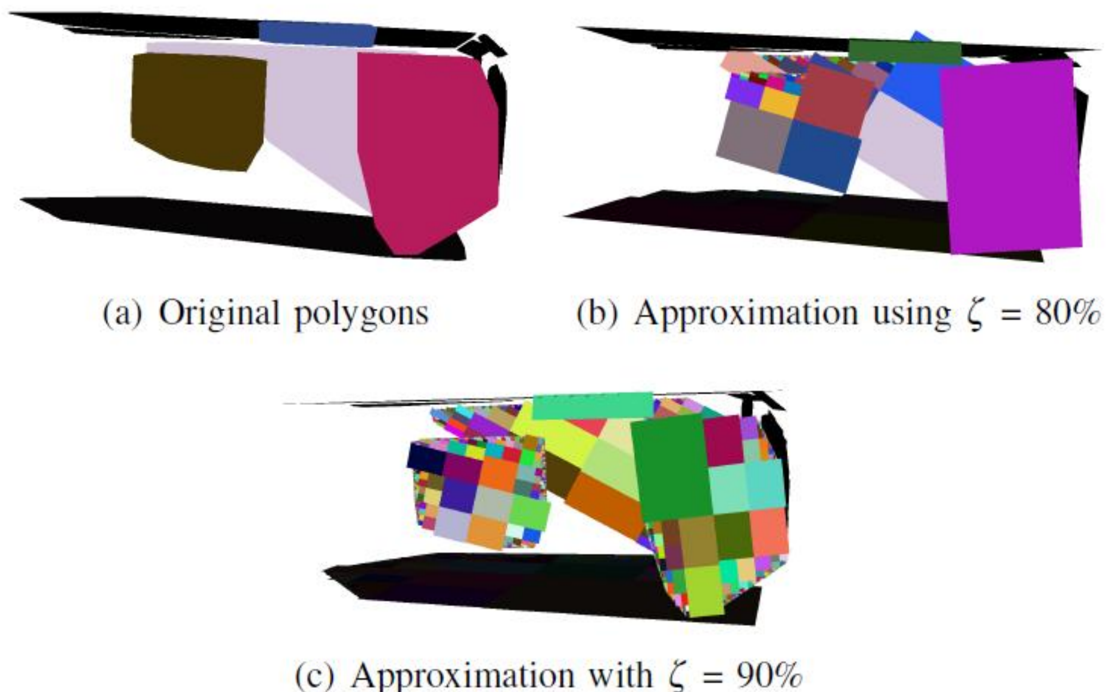


(a) $\zeta = 95\%$, $\gamma = 10\ mm^2$ (b) $\zeta = 95\%$, $\gamma = 10\ cm^2$

Figure 4.4: (a)-(b) Time for splitting algorithm (Fig. 4.1) with increasing number of circles (fixed ζ and varying γ)

2) Real world data sets

The RMAP formulation was also tested on real world datasets. Fig. 4.5 shows results of a few accumulated scans in a specific scenario for the indoor data set[4] to make the effect of the approximation more apparent (since larger point clouds lead to a very cluttered visualization). It can be seen that adapting the threshold ζ leads to better approximation for a fixed value of $\gamma = 10$ mm^2.



(a) Original polygons (b) Approximation using $\zeta = 80\%$

(c) Approximation with $\zeta = 90\%$

Figure 4.5: (a)-(c) Approximation of polygons using rectangles (varying ζ for fixed γ = 10 mm^2)

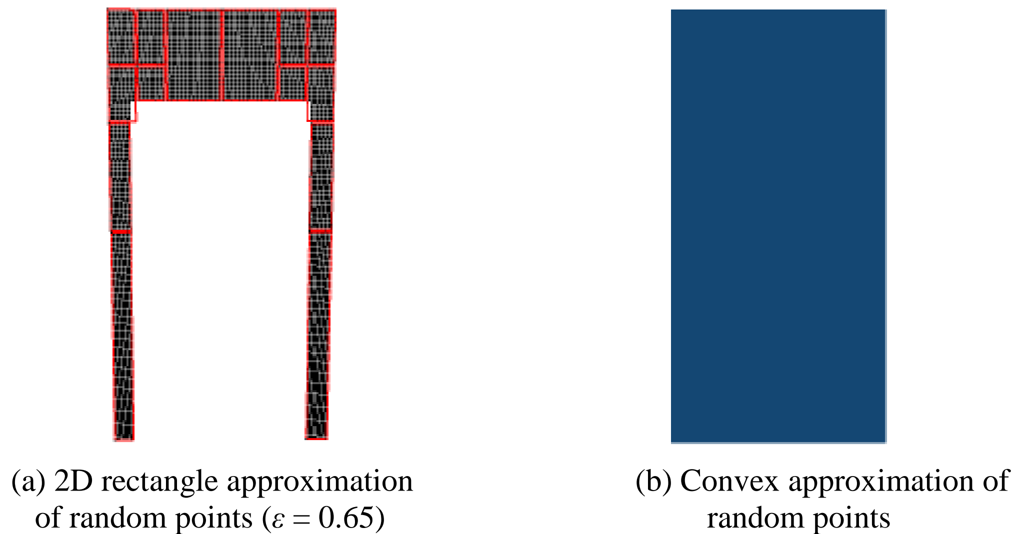
The polygonal approximation framework can be very useful for navigation in a polygonal map. In general most navigation algorithms utilize a parameter corresponding to ‘inflation of obstacles’ which increases the size of the obstacles to bias the trajectory away from the obstacles. In case of RMAP the parameter ζ can be used to vary the size of the ‘inflated obstacles’. An extended maximal bounding box (corresponding to the maximal inflation required) can be put around the obstacles. A low value of ζ would bias the trajectories away from the obstacles, whereas high values would lead to a very close approximation of the polygons and allow the trajectories to get closer to the obstacles.

[4] Courtesy of Martin Magnusson, available at the Osnabrueck robotic 3D scan repository, http://kos.informatik.uni-osnabrueck.de/3Dscans/

4.2 2D Rectangle approximation

Most segmentation algorithms do not consider convexity constraints and the most common approach to deal with output of planar segmentation algorithms is to form a convex planar polygonal representation [35], [39]. In cases the output point cloud of a segmentation algorithm is not convex a convex planar polygonal representation leads to an over approximation. In this Section we discuss an extension of the 3D rectangle approximation algorithm presented in Section 3.2 which is applied to the points of the planar segments extracted from a segmentation algorithm. In that way planar segments that are not convex are approximated more accurately. Like the 3D rectangle approximation, the rectangle approximation is done based on the density of points. In this approach, however, instead of the volume of the initial pointcloud, the area is considered, since we are dealing with 2D points.
Fig. 4.6(a) shows the assumed point cloud output of a segmentation algorithm and the approximation developed by the algorithm. Fig. 4.6(b) shows the convex approximation which leads to an over approximation since the shape in not convex. In such case the presented approach gives better approximation in comparison to convex planar polygonal approximation. This case could simulate passages of real world scenarios, such as doors in indoor environments, in which the better approximation would give also navigational advantages in comparison to the convex hull approximation.

.

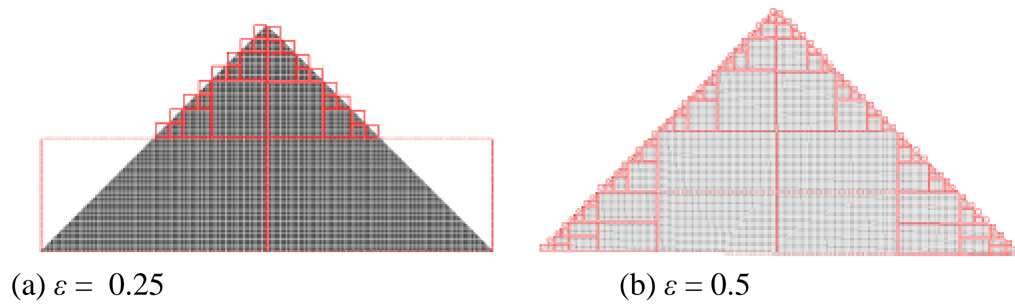


(a) 2D rectangle approximation of random points ($\varepsilon = 0.65$)

(b) Convex approximation of random points

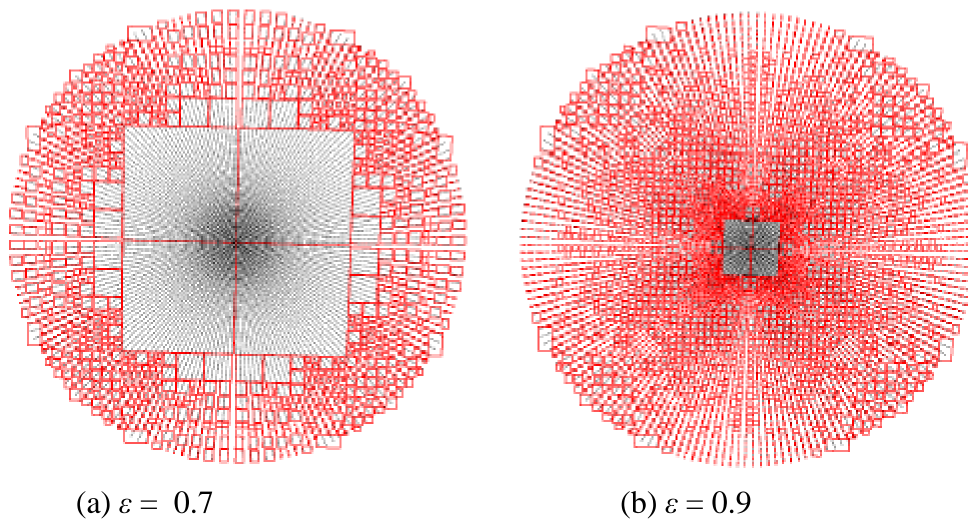
Figure 4.6: Example test case for 2D approximation in comparison to convex hull

In Fig. 4.7 and 4.8 two examples of simulation data are shown. Fig. 4.7 shows two approximations of a triangle shaped pointcloud with different threshold values and Fig. 4.8 shows two approximations of a disc shaped pointcloud with different threshold values.



(a) $\varepsilon = 0.25$ (b) $\varepsilon = 0.5$

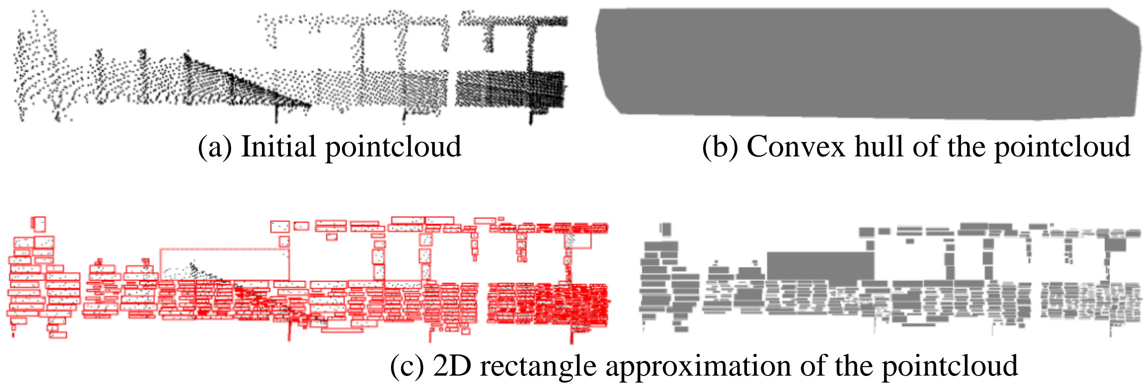
Figure 4.7: 2D rectangle approximation of triangle shaped pointcloud using different thresholds



(a) $\varepsilon = 0.7$ (b) $\varepsilon = 0.9$

Figure 4.8: 2D rectangle approximation of disk shaped pointcloud using different thresholds

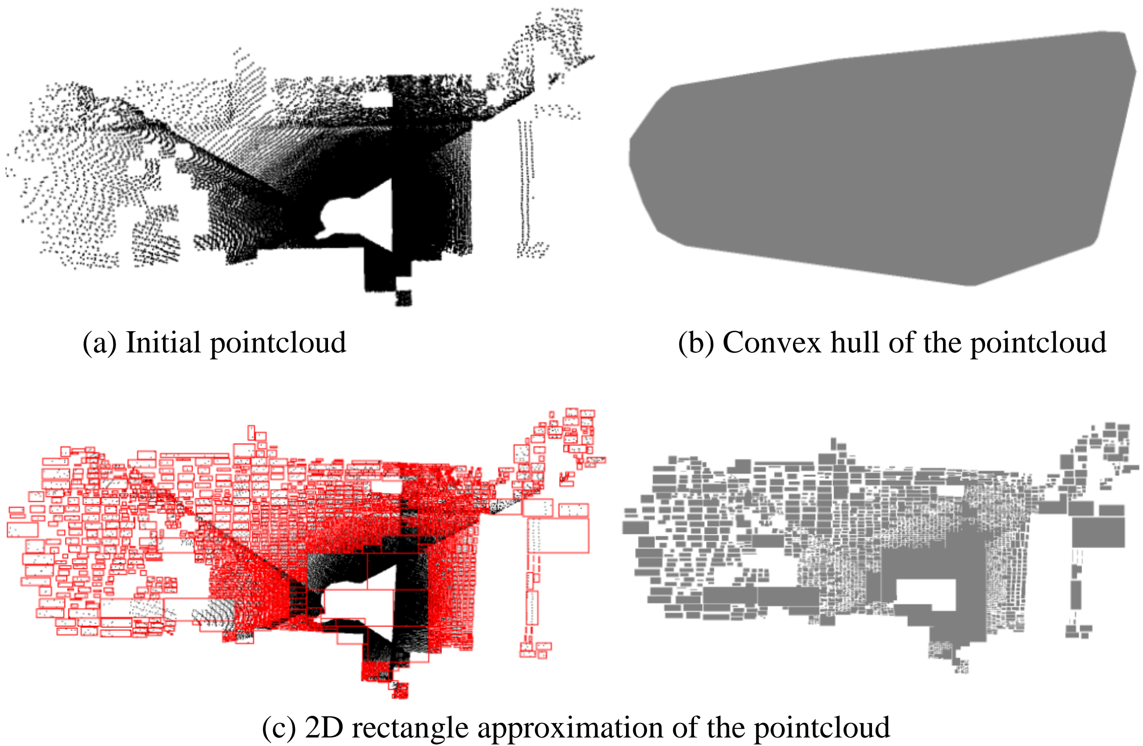
Furthermore, the algorithm was tested in real world scenarios as well. In Fig. 4.9(a) and 4.10(a) we see the points that form two different planar segments from the 1st scan of the Freiburg campus data set. These planar segments were extracted using a segmentation algorithm. Fig. 4.9(b) and 4.10(b) depict the convex hull of these points which over-approximates the initial pointcloud. The results of our algorithm are depicted in Fig. 4.9(c)-(d) and 4.10(c)-(d) utilizing two different visualizations. It is clear that our approach approximates much more accurately the pointclouds than the convex hull does.



(a) Initial pointcloud (b) Convex hull of the pointcloud

(c) 2D rectangle approximation of the pointcloud

Figure 4.9: (a) Initial pointcloud, (b) Convex planar polygonal approximation of the pointcloud, (c) 2D rectangle approximation of the pointcloud using different visualizations

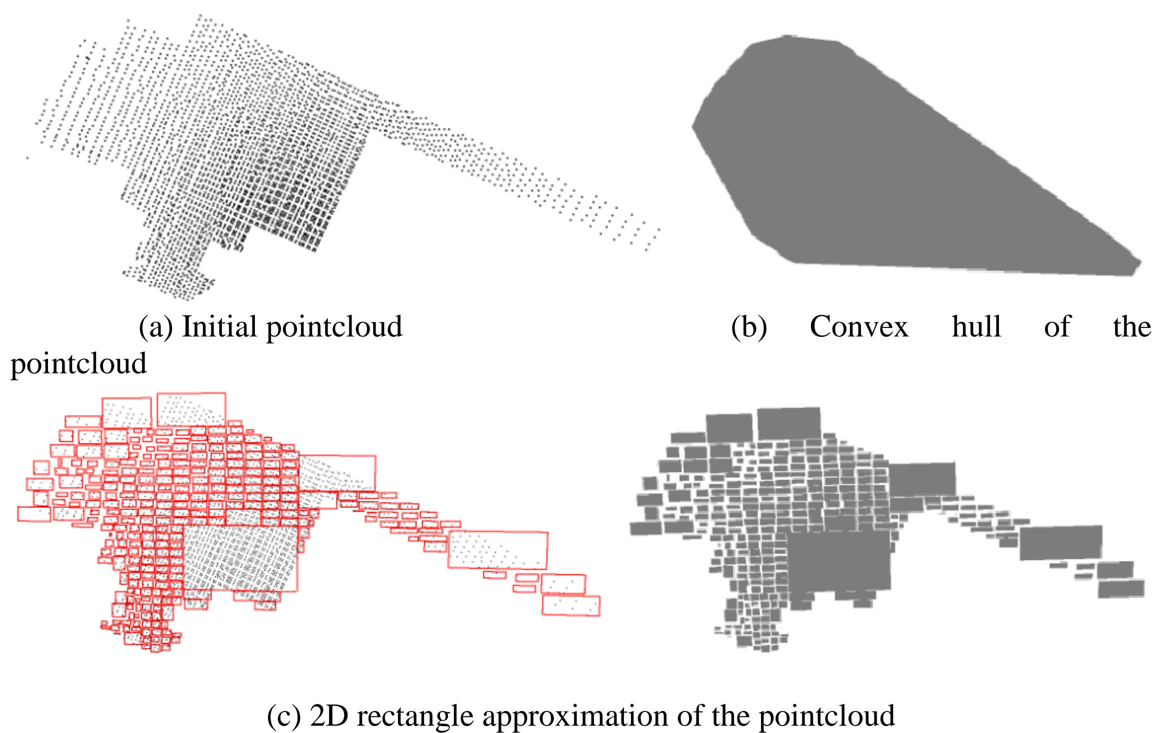


(a) Initial pointcloud (b) Convex hull of the pointcloud

(c) 2D rectangle approximation of the pointcloud

Figure 4.10: (a) Initial pointcloud, (b) Convex planar polygonal approximation of the pointcloud, (c) 2D rectangle approximation of the pointcloud using different visualizations

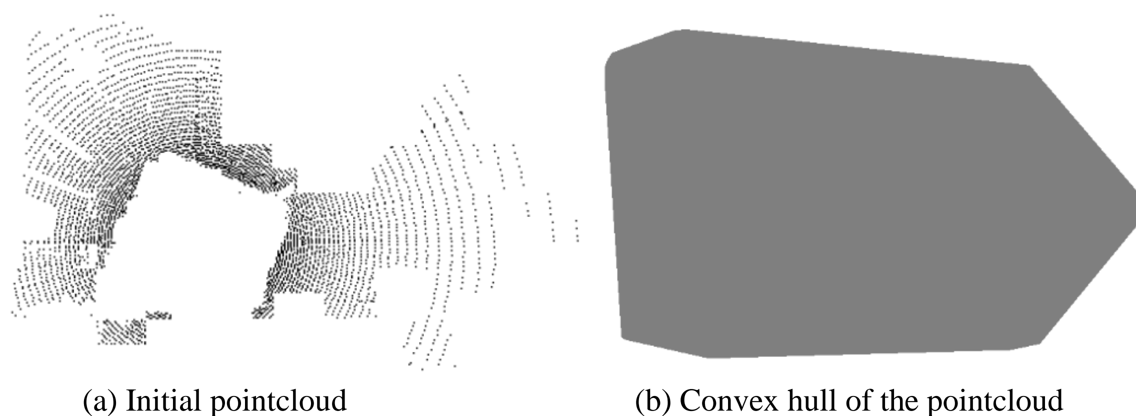
In Fig. 4.11 and 4.12 the same procedure is depicted for the 1st scan of the indoor data set. More specifically, the algorithm is applied to the points of two of the planar segments that were extracted utilizing a segmentation algorithm.



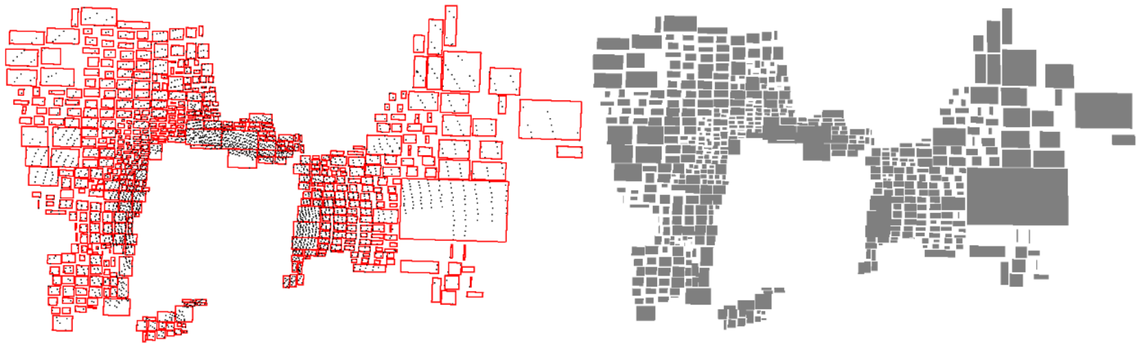
(a) Initial pointcloud (b) Convex hull of the pointcloud

(c) 2D rectangle approximation of the pointcloud

Figure 4.11: (a) Initial pointcloud, (b) Convex planar polygonal approximation of the pointcloud, (c) 2D rectangle approximation of the pointcloud using different visualizations



(a) Initial pointcloud (b) Convex hull of the pointcloud



(c) 2D rectangle approximation of the pointcloud

Figure 4.12: (a) Initial pointcloud, (b) Convex planar polygonal approximation of the pointcloud, (c) 2D rectangle approximation of the pointcloud using different visualizations

5 Applications in robotics

The progress that has been made the recent years in the field of robotics has allowed the integration of mobile robots in more and more applications. Many of these applications include the use of mobile robots in environments where a human being could not survive or could be exposed to several dangers, such as disaster areas, volcano and underwater exploration or even outer space missions. In such cases it is crucial that the robot can build a 3D map of the surrounding environment, as it is the most important prerequisite for all the procedures it is called to do. In [40] one can find useful information about NASA's rover for the exploration of Mars. In [41] an efficient representation in 3D environment modeling for planetary robotic exploration is presented and in [4] they consider terrain mapping for a roving planetary explorer. Moreover, mobile rescue robots were employed for the fall of the twin towers at 11 of September 2001, for which information can be gained in [42] and [43]. In [44] it is described how mobile robots advance on Disaster City in Texas. In [45] an algorithm of 3D-plane SLAM experiment was performed at Disaster city , whereas in [46] a 3D-NDT test was conducted using an Atlas Copco drill rig in a mining environment. More information about Atlas Copco robots can be found in [47]. Utilizing efficient data structures in terms not only of time and memory but also of accurate environmental representation for the mapping in scenarios as the above ones is very critical for the tasks of a mobile robot. On-line procedures in environments that are not known in advance, such as those mentioned above, require quick insertion and extraction of the primitives used (e.g. points) making the choice of the data structure that is to be used very significant.

6 Conclusion and future work

In this thesis the concept of representing 3D environments is considered. Firstly, the use of 3D points is featured. Different data structures that can store points are compared both theoretically and practically in terms of insertion and access times and memory consumption. Hash tables provide fast computational times exploiting the array's direct mapping property and low memory consumption since the allocation for modelling the free space is avoided. Octrees however, except for being efficient in terms of time and memory, also provide a multi-resolutional representation of a 3D environment which is crucial for several applications. Furthermore, a rectangular approximation of the 3D environment is presented. This is done firstly by utilizing an occupancy grid framework which allows a large amount of flexibility in terms of shape of the grid cell that can be chosen for the environmental representation and resolution adaption. Secondly, the idea of approximating the environment with rectangles based on point density is presented, which can lead in great memory savings, being however limited to an offline procedure due to high computational time. Moreover, this concept is extended to planar segments extracted from a segmentation algorithm by approximating efficiently both points of concave shapes and any arbitrary shaped convex hull. All the instances of the approach allow adaptation of the environment representation using different parameters. Finally, the above ideas were extensively tested in simulation and real world data sets for computational and memory costs as well as other criterion (such as resolution), verifying their flexibility and advantages in 3D mapping.

As future work the further development or modification of the RMAP framework could be considered. More specifically, the approach could be extended to apply also for oriented bounding boxes (OBB) instead of axis-aligned ones (AABB), since the axis-aligned constraint makes the approximation of 3D environment objects that have a certain orientation very coarse. However, OBBs cause an increase in computational and memory complexity. An increase in memory complexity occurs, since in RMAP only the minimum and maximum point of the bounding box are stored whereas in case of OBBs either all points have to be stored or transformation angles need to be stored. An increase in computational cost occurs as the test for bounding box overlap becomes more complex in case of OBBs. The choice between AABBs and OBBs is a mere trade-off between computational and memory complexity. Moreover, the concept of bounding boxes can be extended to moving boxes using TPR*-trees which can model moving objects and dynamic environments [30].

List of Figures

Figure 2.1: : Representation of the Freiburg campus using an octree data structure......... 7
Figure 2.2: The octree data structure 10
Figure 2.3: 1st scan of the Freiburg campus data set using resolution 0.05 11
Figure 2.4: 1st scan of the Freiburg campus data set using resolution 0.2. 11
Figure 2.5: 1st scan of the Freiburg campus data set using resolution 0.8 12
Figure 2.6: Example of accessing a cell in the hash table 15
Figure 2.7: Computational time and memory comparison between Octree, K-d tree and Hash Table (10cm resolution) for randomly generated points between [-1000, 1000]m 17
Figure 2.8: Computational time and memory comparison between Octree, K-d tree and Hash Table (10cm resolution) for randomly generated points between [-1000, 1000]m 18
Figure 2.9: Computational time and memory for the Octree data structure for different grid resolutions and 500,000 randomly generated points between [-1000,1000]m. 18
Figure 2.10: Freiburg campus and Bremen city center visualizations at 2cm resolution 19
Figure 2.11: Computational time and memory comparison between Octree, K-d tree and Hash Table for the Freiburg campus with different grid resolution values 20
Figure 2.12: Computational time and memory comparison between Octree, K-d tree and Hash Table for the Bremen City with different grid resolution values 20
Figure 3.1: 3D and 2D R-tree structure 22
Figure 3.2: Different resolution views of the 1st scan of the Freiburg campus data set .. 23
Figure 3.3: Computational time and memory comparison between Octree and R-tree (20cm resolution) for randomly generated points between [-1000, 1000]m 24
Figure 3.4: Computational time and memory comparison between Octree and R-tree with respect to grid resolution for the first 70 files of the Freiburg campus data set 25
Figure 3.5: 3D rectangle approximation pseudocode 27

Figure 3.6: Sphere shaped point cloud rectangle approximation using different thresholds 28
Figure 3.7: Rectangle approximation of the Stanford repository bunny for different density thresholds. 29
Figure 3.8: (a) Approximation time, (b) Total number of rectangles, (c) Number of undefined points and (d) Memory consumption with respect the threshold β for the Stanford repository bunny 30
Figure 3.9: Rectangle approximation of the Stanford repository dragon for different density thresholds 31
Figure 3.10: (a) Approximation time, (b) Total number of rectangles, (c) Number of undefined points and (d) Memory consumption with respect the threshold β for the Stanford repository dragon 31
Figure 3.11: Rectangular approximation of a tree and a pole from the first scan of the Freiburg campus data set ($\varepsilon = 98\%$, $\omega = 1mm^3$) 33
Figure 3.12: Rectangular approximation of the first (a) 10 files of the Freiburg campus dataset and (b) 4 scans of the Bremen city center dataset 33
Figure 3.13: (a) Approximation time, (b) Total number of rectangles, (c) Number of undefined points and (d) Memory consumption with respect the threshold β for the first 10 files of the Freiburg campus data set 34
Figure 3.14: Computation time, number of rectangles and memory consumption in comparison to an occupancy grid formulation for 70 files of the Freiburg campus dataset 35
Figure 4.1: Polygonal approximation pseudocode 38
Figure 4.2: Effect of ζ for fixed $\gamma = 10mm^2$ for 2D circle and triangle 40
Figure 4.3: Effect of $\gamma = 10cm^2$ for 2D circle and triangle 40
Figure 4.4: Time for splitting algorithm (Fig. 4.1) with increasing number of circles (fixed ζ and varying γ) 40
Figure 4.5: Approximation of polygons using rectangles (varying ζ for fixed $\gamma = 10\ mm^2$) 41
Figure 4.6: Example test case for 2D approximation in comparison to convex hull 42

Figure 4.7: 2D rectangle approximation of triangle shaped pointcloud using different thresholds 43

Figure 4.8: 2D rectangle approximation of disk shaped pointcloud using different thresholds 43

Figure 4.9: (a) Initial pointcloud, (b) Convex planar polygonal approximation of the pointcloud, (c) 2D rectangle approximation of the pointcloud using different visualizations 44

Figure 4.10: (a) Initial pointcloud, (b) Convex planar polygonal approximation of the pointcloud, (c) 2D rectangle approximation of the pointcloud using different visualizations 44

Figure 4.11: (a) Initial pointcloud, (b) Convex planar polygonal approximation of the pointcloud, (c) 2D rectangle approximation of the pointcloud using different visualizations 45

Figure 4.12: (a) Initial pointcloud, (b) Convex planar polygonal approximation of the pointcloud, (c) 2D rectangle approximation of the pointcloud using different visualizations 46

List of Tables

Table 2.1: Data structures theoretical comparison (big O notation) in terms of inserton, access time and memory consumption 13

Bibliography

[1] A. Elfes, “Using Occupancy grids for mobile robot perception and navigation,” *Computer,* vol. 22, no. 6, pp. 46 - 57, June 1989.

[2] S. Thrun, “Learning occupancy grid maps with forward sensor models,” *Autonomous robots,* vol. 15, no. 2, pp. 111 - 127, 2003.

[3] H. Moravec and A. Elfes, “High resolution maps from wide angle sonar,” in *Proceedings of IEEE International Conference on Robotics and Automation*, 1985.

[4] M. Herbert, C. Caillas, E. Krotkov, K. I.S and T. Kanade, “Terrain mapping for a roving planetary explorer,” in *Proceedings of the IEEE International Conference on Robotics & Automation (ICRA)*, 1989.

[5] P. Biber and W. Strasser, “The normal distributions transform: A new approach to laser scan matching,” in *Proceedings of 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems, 2003. (IROS 2003).*, 2003.

[6] V. Nguyen, S. Gaechter, A. Martinelli, N. Tomatis and R. Siegwart, “A comparison of line extraction algorithms using 2d range data for indoor mobile robotics,” *Autonomous Robots,* vol. 23, no. 2, pp. 97 - 111, 2007.

[7] H. Samet, “Spatial data structures,” *Modern Database Systemns, The Object Model, Interoperability and Beyond,* pp. 361 - 385, 1995.

[8] J. Elseberg and S. S. R. N. A. Magnenat, “Comparison of nearest-neighbor-search strategies and implementations for efficient shape registration,” *Journal of Software Engineering for Robotics,* vol. 3, no. 1, pp. 2 - 12, 2012.

[9] T. Cormen, C. Leiserson, R. Rivest and C. Stein, Introduction to Algorithms, 3rd ed., Massachusetts Institute of Techonoly.

[10] D. Richardson, The Book on Data Structures, iUniverse, 2002.

[11] N. Askitis, Fast and Compact Hash Tables for Integer Keys, 2009, pp. 113 - 122.

[12] D. Sun, A. Kleiner and C. Schindelhauer, “Decentralized hash tables for mobile robot teams solving intra-logistics tasks,” in *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems*, Toronto, Canada, 2010.

[13] D. Knuth, The art of computer programming vol 1. Fundamental Algorithms, Addison-Wesley, 1997, pp. 318 - 348.

[14] K. Berman and J. Paul, Algorithms: Parallel, Sequential and Distributed, Course Technology, 2005.

[15] I. Wald and V. Hauran, “On building fast kd-Trees for Ray Tracing, and on doing that in O(N log N),” in *IEEE Symposium on Interactive Ray Tracing*, Salt Lake City, Utah, 2006.

[16] M. Greenspan and M. Yurick, “Approximate k-d tree search for efficient ICP,” in *Proceedings of the fourth International Conference on3-D Digital Imaging and Modeling, 2003. 3DIM 2003.*, Canada, 2003.

[17] J. Kubica, J. Masiero, A. W. Moore, R. Jedicke and A. Connolly, “Variable kd-tree

algorithms for spatial pattern search and discovery," in *Robotics Institute* , 2005.

[18] J. Bentley, "Multidimensional binary search trees used for associative searching," *Communications of the ACM,* vol. 18, no. 9, pp. 509 - 517, 1975.

[19] D. Meagher, "Geometric modeling using octree encoding," *Computer Graphics and Image Processing,* vol. 19, no. 2, pp. 129 - 147, 1982.

[20] P. Payeur, P. Hebert, Laurendeau and C. Gosselin, "Probabilistic octree modeling of a 3D dynamic environment," in *1997 IEEE International Conference on Robotics and Automation, 1997.*, Canada, 1997.

[21] K. Wurm, A. Hornung, M. Bennewitz and C. B. W. Stachniss, "OctoMap: A probabilistic, flexible, and compact 3D map representation for robotic systems," in *Proc. of the ICRA 2010 workshop on best practise in 3D perception and modeling for mobile manipultation*, 2010.

[22] N. Fairfield, G. Kantor and D. Wettergreen, "Real-time SLAM with octree evidence grids for exploration in underwater tunnels," *Journal of Field Robotics, 2007,* vol. 24, no. 1, pp. 03 - 21, February 2007.

[23] D. Meagher, "Octree Encoding: A New Technique for the Representation, Manipulation and Display of Arbitrary 3D Ojects by Computer," Rensselaer Polytechnic Institute, 1980.

[24] J. Tsiombilas, "Kdtree," 2007. [Online]. Available: http://www.code.google.com/p/kdtree.

[25] S. Perreault, "Octree C++ Class Template," April 2007. [Online]. Available: http://nomis80.org/code/doc/classOctree.html.

[26] Y. Manolopoulos, A. Nanopoulos and Y. Theodoridis, R-Trees: Theory and Applications, Springer, 2006.

[27] A. Guttman, "R-Trees: A Dynamic Index Structure for Spatial Searching," in *Proceedings of the 1984 ACM SIGMOD International Conference on Management of data - SIGMOD '84*, New York, 1984.

[28] N. Beckmann, H. Kriegel, R. Schneider and B. Seeger, "The R*-tree: an efficient and robust access method for points and rectangles," in *Proceedings of the 1990 ACM SIGMOD international conference of Management of data - SIGMOD '90*, New York, 1990.

[29] T. Gutman, "R-tree source code," [Online]. Available: http://www.superliminal.com/sources/sources.htm#C%20&%20C++%20Code.

[30] Y. Tao, D. Papadias and J. Sun, "The TRP*-tree: an optimized spatio-temporal access method for predictive queries," in *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB)*, Berlin, Germany, 2003.

[31] Y. Theodoridis and T. Sellis, "Optimization issues in R-tree construction," in *IGIS'94: Geographic Information Systems, International Workshop on Advanced Research in Geographic Information Systems*, 1994.

[32] S. Berchtold, D. Keim and H. Kriegel, The X-tree: An index structure for high-dimensional data.

[33] F. Amigoni, V. Caglioti and U. Galtarossa, "A mobile robot mapping system with an information-based exploration strategy," in *Proc. ICINCO*, 2004.

[34] J. Poppinga, N. Vaskevicius, N. Birk and K. Pathak, "Fast plane detection and polygonalization in noisy 3D range images," in *IEEE/RSJ International Conference on Intelligent Robots and Systems, 2008. IROS 2008.*, Bremen, 2008.

[35] J. Biswas and M. Veloso, "Planar polygon extraction and merging from depth images," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2010.

[36] J. Xiao, J. Zhang, J. Zhang, H. Zhang and H. Hildre, "Fast plane detection for slam from noisry range images in both structured and unstructured environments," in *2011 International Conference on Mechatronics and Automation (ICMA)*, Hamburg, 2011.

[37] R. K. V. Kothuri, S. Ravada and D. Abugov, "Quadtree and r-tree indexes in oracle spatial: a comparison using gis data," in *Proceedings of the ACM SIGMOD International Conference on Management of data*, New York, 2002.

[38] A. B. J. W. H. Helmut, "Approximation of convex polygons," *Automata, Languages and Programming,* vol. 443, pp. 703 - 719, 1990.

[39] K. Pathak, N. Vaskevicius, J. Poppinga, M. Pfingsthorn, S. Schwertfeger and A. Birk, "Fast 3d mapping by matching planes extracted from range sensor point-clouds," in *IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS*, 2009.

[40] W. G, "NASA's Veteran Mars Rover Ready to Start 10th Year," 2013. [Online]. Available: http://www.nasa.gov/mission_pages/mer/news/me20130122.html.

[41] N. Vaskevicius, A. Birk, K. Pathak and S. Schwertfeger, "Efficient representation in Three-Dimensional Environment Modeling for Planetary Robotic Exploration," *Advanced Robotics,* vol. 24, no. 8-9, pp. 1169 - 1197, 2010.

[42] "Remote-controlled Robots Search World Trade Center Rubble," *JOM,* vol. 53, no. 12, pp. 4 - 7, 2001.

[43] R. R. Murphy, "Trial by fire," *IEEE Robotics & Automation Magazine,* vol. 11, no. 3, pp. 50 - 61, 2009.

[44] B. Blake, "Robots from across the globe advance on Disaster City," 2011. [Online]. Available: http://teexweb.tamu.edu/teex.cfm?pageid-USARresc&area=USAR&storyid=1103&templateid=23.

[45] K. Pathak, "Online three-dimensional SLAM by registration of large planar surface segments and closed-form pose graph relaxation," *Journal of Field Robotics,* vol. 27, no. 1, pp. 52 - 84, 2009.

[46] M. Magnusson and A. Lilienthal, "Scan Registration for Autonomous Mining Vehicles Using 3D-NDT," *Journal of Field Robotics,* vol. 24, no. 10, pp. 803 - 827, 2007.

[47] "Atlas Copco Rock Drills," 2012. [Online]. Available: http://www.mining-technology.com/contractors/drilling/atlas-copco-2.

[48] P. N. Yianilos, "Data structures and algorithms for nearest neighbor search in general metric spaces," in *Proceedings of the 4th annual ACM-SIAM Symposium on Discrete algorithms*, Philadelphia, USA, 1993.