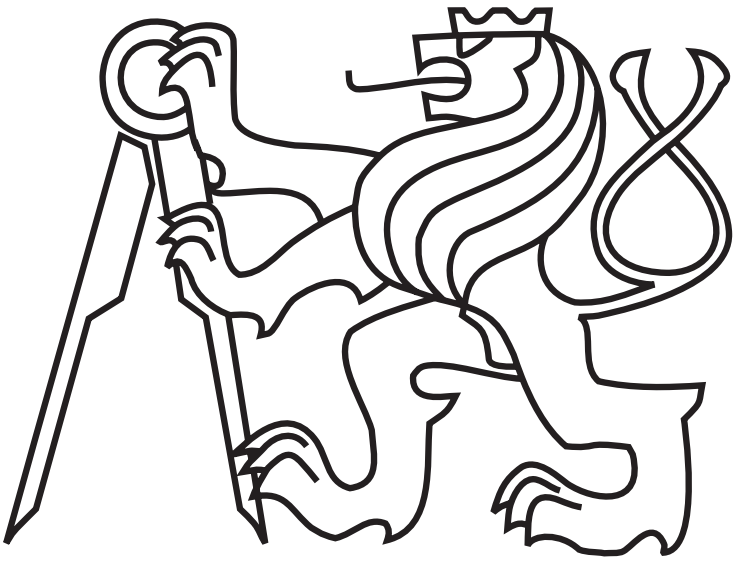
CZECH TECHNICAL UNIVERSITY IN PRAGUE

Faculty of Electrical Engineering

BACHELOR THESIS



Vladimír Petrík

Spatial Models in Mobile Robotics

Department of Cybernetics

Thesis supervisor: **Ing. Tomáš Krajník**

Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Cybernetics

BACHELOR PROJECT ASSIGNMENT

Student: Vladimír Petrík

Study programme: Cybernetics a Robotics

Specialisation: Robotics

Title of Bachelor Project: Spatial Models in Mobile Robotics

Guidelines:

1. Review methods of environment model building for mobile robotics.
2. Implement a suitable method, which would filter out false measurements from large sets of 3D point clouds acquired by a laser rangefinder.
3. Design, implement and experimentally verify a method, which would find planar areas in the filtered data and create a partial 3D polygonal environment model.
4. Create a visualization tool for the resulting model.

Bibliography/Sources: Will be provided by the supervisor.

Bachelor Project Supervisor: Ing. Tomáš Krajník

Valid until: the end of the winter semester of academic year 2012/2013

prof. Ing. Vladimír Mařík, DrSc.
Head of Department

L. S.

prof. Ing. Pavel Ripka, CSc.
Dean

Prague, January 9, 2012

České vysoké učení technické v Praze
Fakulta elektrotechnická

Katedra kybernetiky

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student: Vladimír P e t r í k

Studijní program: Kybernetika a robotika (bakalářský)

Obor: Robotika

Název tématu: Tvorba 3D modelu prostředí

Pokyny pro vypracování:

1. Seznamte se s metodami triangulace bodů v prostoru a tvorbou modelů prostředí pro mobilní robotiku.
2. Realizujte vhodnou filtraci velkého množství prostorových bodů, které byly naměřeny mobilním robotem s využitím laserového range-finderu.
3. V mračnech prostorových bodů nalezněte prostorové shluky a v každém shluku identifikujte povrchové body, které následně proložíte částečnými rovinami.
4. Výsledný model vhodným způsobem vizualizujte.

Seznam odborné literatury: Dodá vedoucí práce.

Vedoucí bakalářské práce: Ing. Tomáš Krajník

Platnost zadání: do konce zimního semestru 2012/2013

L.S.

prof. Ing. Vladimír Mařík, DrSc.
vedoucí katedry

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 9. 1. 2012

Declaration

I hereby declare that I have completed this thesis independently and that I have listed all used information sources in accordance with Methodical instruction about ethical principles in the preparation of university theses.

In Prague on 25.5.2012 .....

Acknowledgements

I would like to thank my supervisor Ing. Tomáš Krajník for his support and encouragement during this project.

Abstrakt

Cieľom tejto bakalárskej práce je vytvorenie geometrickej mapy z hrubých senzorických dát, nazbieraných mobilným robotom. Takáto geometrická mapa môže slúžiť pre algoritmy pre plánovanie trajektórie robota, prípadne pre simuláciu prostredia v ktorom sa robot pohybuje. Implementovaný algoritmus využíva vzájomné súperenie jednotlivých clustrov o namerané priestorové body. Metóda bola testovaná na reálnych dátach kde z celkového počtu pol milióna bodov vytvorila 30 polygónov. Výsledný model prostredia bol taktiež využitý v simulátore projektu Symbrion.

Abstract

The goal of this bachelor thesis is to create a geometrical map from raw sensory data collected by a mobile robot. This map can serve as an input for algorithms for planning robots trajectory or for simulation of its environment. The algorithm is based on rivalry between clusters which are competing for measured points. This method was tested by using real data producing 30 polygons from approximately 500,000 points. Resulting environmental model was tested in simulator of Large-Scale Integrated Project Symbrion.

Contents

List of Figures

List of Tables

List of Algorithms

List of Appendices

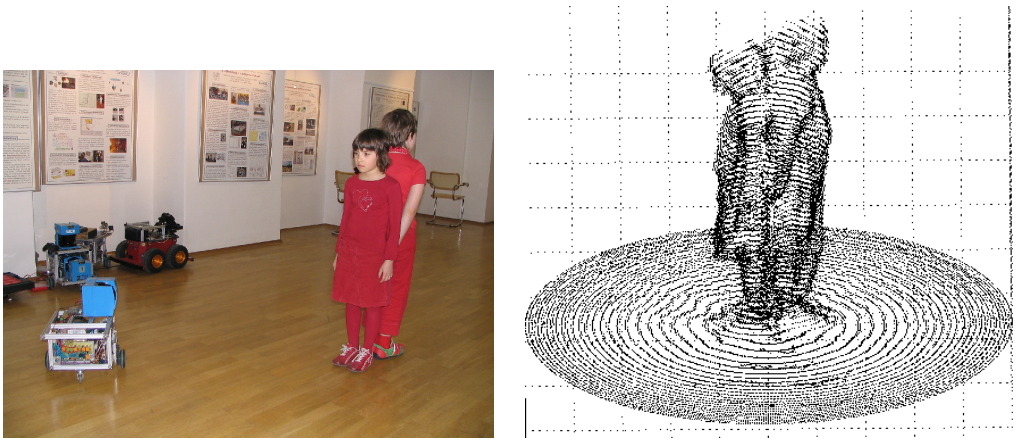
1 Introduction

From the very beginning of robot construction, there was an idea to make them fully autonomous. That means, to act on their own, independent on human supervision. These robots should be able to plan their trajectory, avoid collisions, fold clothes, etc. A fundamental issue of robot-environment interaction is the capability of the control system to model the environment. With an appropriate surrounding model robots are able to make predictions about their next states. Robots have to perform complex tasks at real time so it is important to use appropriate environment representation. There are several ways to represent environment models and some of them will be introduced in the following sections.

It is common that robots store information about the outside world in a form of a point cloud. A point cloud is a set of points in a three-dimensional coordinate system which can be measured by various types of sensors such as:

- stereo camera systems (such as Loreo),
- laser scanners (such as SICK laser),
- time of flight camera (such as SwissRanger),
- structured light systems (such as Kinect).

To construct a point cloud using a sensor mounted on a mobile robot, position of the robot has to be known. In figure 1 a point cloud is collected by a mobile robot localized by odometry.



(a) Measurement (b) Result

Figure 1: Point cloud measurement example

The data in a form of a point cloud are not suitable for simulation or motion planning because of memory requirements, and time-consumption of processing algorithms. These are the reasons why a point cloud is usually transformed to another form of environment model representation. One of these forms is a geometric map which represents environment with geometric primitives like lines, polygons, spheres, etc. The goal of this thesis is to design an algorithm which transforms a point cloud into a geometric map.

This bachelor thesis is divided into following parts. Firstly, existing methods for geometric maps construction and various types of environment model representations are explained. Next few sections are devoted to description of partial algorithms and their implementation. The explanation of the algorithm is divided into four parts: raw sensory data filtering, plane equation searching, surface based algorithms for polygon border search, final approximation and triangulation. The algorithms for equations of planes searching are based on [1]. The final section contains experimental results and time consumption analysis.

2 Related work

2.1 Environment models

In mobile robotics, environment models can be used for path planning, localization and simulation. Since environment changes with time, the model has to change as well in order to provide robot with correct information. With higher level of environment model abstraction it becomes more difficult to build the model.

2.1.1 Occupancy grid

A common representation of environment is an occupancy grid. The occupancy grid representation divides multidimensional space into cells, where each cell stores a probabilistic estimate of its state.

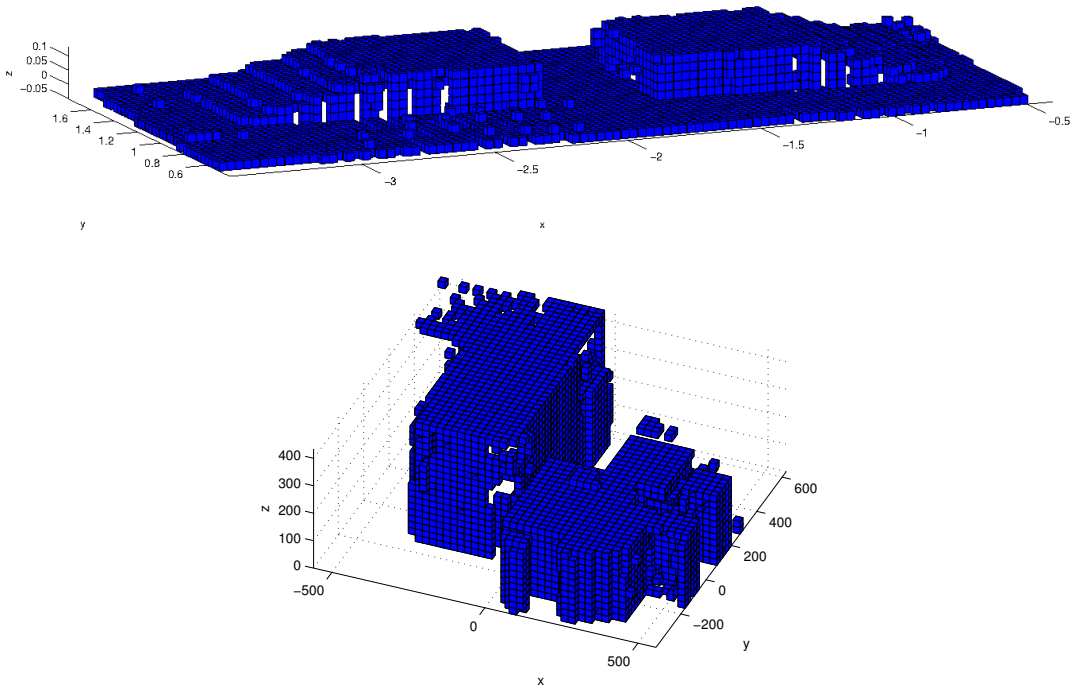


Figure 2: Examples of occupancy grid with threshold set to 10 points per cell

2.1.2 Landmark map

A landmark map is based on storing information about important places in the environment. These important places are called landmarks and are uses as reference points. A

landmark map is mostly used for mobile robot localization or navigation.

2.1.3 Geometric map

A geometric map represents an environment with geometric primitives like lines, polygons, spheres, etc. Construction of a geometric map is not as easy as construction of an occupancy grid, but it provides a compact alternative to grid-based model, requiring considerably less memory and hence less computation for path planning or simulation methods [2].

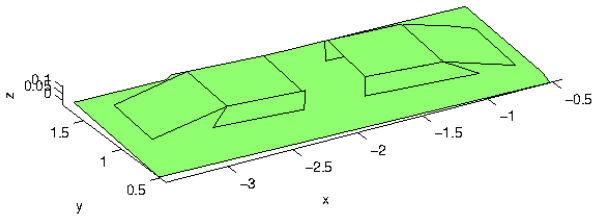


Figure 3: Geometric map

2.1.4 Topological map

A higher level of environment representation is a topological map. In contrast to geometrical representation, topological model is simplified so that unnecessary details are removed and only vital information remains. Topological models are generally described by graph structure where the graph nodes define particular locations in the environment and the graph edges define information for motion between nodes [3].

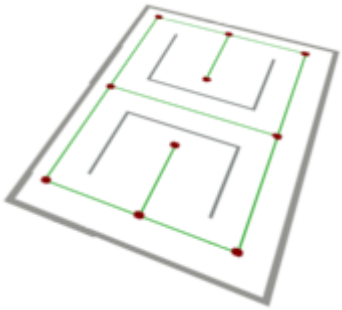


Figure 4: Example of topological model

2.2 Geometrical map construction

2.2.1 Random Samples Consensus

Random Samples Consensus (RANSAC) is an iterative method for construction of geometrical map from observed data that may contain large number of outliers. The RANSAC method is based on computing a number of points that lie within a threshold distance from a randomly selected primitive (in this case a plane). This selection is repeated several times and the primitive with the maximum value represents a fit. To generate a plane from point cloud, three points are randomly selected and then plane is constructed from these points. In [4], some methods similar to RANSAC are shown - for instance MLESAC (Maximum LikElihood SAmple Consensus) or MAPSAC (Maximum A Posteriori SAmple Consensus).

An advantage of RANSAC is that it can estimate the parameters even when a significant number of outliers is present in the data set. The disadvantage is, that number of planes or threshold has to be known before map construction. Another disadvantage is computation time. There is a possibility to limit number of iterations to reduce computation time but the obtained solution may not be optimal.

2.2.2 Hough Transform

The Hough Transform is a method for detecting parameterized objects. The simplest case of Hough Transform is the linear transform for line detection. The idea is to transform lines into space defined by the line parameters (for example slope and interception). That means a line will be represented as a point in this space which is also called a Hough space or an accumulator. For each point in the dataset all possible lines are computed (with respect to discretization) and then the score of corresponding cell in Hough space is increased. The cells with score higher then a certain threshold represent lines. An example of 2 lines with corresponding Hough space is shown in figure 5. A line in a plane is defined by an equation $y = kx + q$, but as variable k does not have an uniform distribution it is common practice to use polar coordinates instead [5].

For detection of parametrized planes a method explained in [6] will be presented. The planes are commonly represented by the equation $z = m_x x + m_y y + \rho$, however, to avoid problems bound with infinite slopes when representing vertical planes, the planes should be represented in form

$$\rho = p_x \cos\theta \sin\varphi + p_y \sin\varphi \sin\theta + p_z \cos\varphi.$$

θ, φ and ρ define the 3-dimensional Hough Space corresponding to one plane in $\mathbf{R}^3$.

The Hough Space is divided into cells and for each point in dataset all cells(θ, φ, ρ), where the point lies in plane defined by θ, φ and ρ, are voted. Voting means increasing the score by 1. The cell with the highest value represents the most prominent plane, the plane

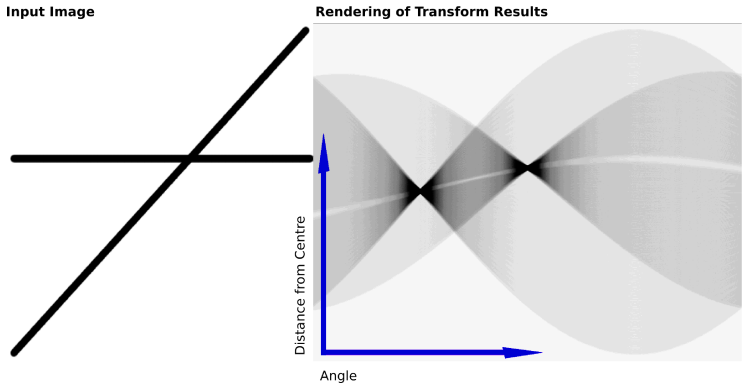


Figure 5: Example of Hough Transform (Original figure downloaded from [5])

that covers most points of the point cloud.

Disadvantage of Hough Transform as well as RANSAC is that number of planes or threshold has to be known before the map is built. Both Hough Transform and RANSAC do not consider any limits of plane span while map is under construction. That means they cannot be used for detecting walls lying in the same plane.

3 Theoretical information

As was mentioned in the Introduction, the data in a form of a point cloud take a lot of memory. It is usual to have a few hundred thousand points so it is important to store this data in an appropriate data structure. The most common structures are octree and kd-tree. In this work author had chosen to use kd-tree because it is easy to implement without insertion and deleting operations.

3.1 Kd-Tree

A kd-tree is a data structure for storing a finite set of points from k-dimensional space. It is a binary tree in which every node is a k-dimensional point which divides the space into two subspaces with equal number of points in them. This tree is a useful data structure for range search and nearest neighbor search [7].

3.1.1 Construction

There are several ways to construct a kd-tree. Recursive method described in algorithm 1 was used. The output is root node which can be used to access any point in the tree.

Input: *points* := points in subset, *depth* := actual depth in the tree
Output: *node* := contains value (point) and child nodes (left, right)

```
1  if points.size = 0 then
2  |  return NULL;
3  end
4  axis = depth mod 3;
5  if axis = 0 then
6  |  sort points by axis x;
7  else if axis = 1 then
8  |  sort points by axis y;
9  else
10 |  sort points by axis z;
11 end
12 k = points.size / 2;
13 node.value = points[k];
14 node.left = this(points before points[k], depth+1);
15 node.right = this(points after points[k], depth+1);
16 return node;
```

Algorithm 1: A construction of a 3d-tree

Because all points were known before tree construction, there was no need to have methods for adding or removing points which would lead to an unbalanced binary tree.

3.1.2 Range search

In two dimensional space, the range search returns all points inside the rectangle defined by the bottom left and top right corner. In three dimensional space, points inside a cube are returned. Range search is more efficient than Nearest Range Search because it avoids square root computing.

Input: *node* := tested node, *min* := bottom left corner, *max* := top right corner, *depth* := depth in the tree

Output: *points* := set of points in range

```
1  if node = NULL then
2  |  return;
3  end
4  axis = depth mod 3;
5  if  (axis = 0 AND node.value.x < min.x)
6  OR (axis = 1 AND node.value.y < min.y)
7  OR (axis = 2 AND node.value.z < min.z) then
8  |  this(node.right, min, max, depth + 1);
9  |  return;
10 else if  (axis = 0 AND node.value.x > max.x)
11 OR (axis = 1 AND node.value.y > max.y)
12 OR (axis = 2 AND node.value.z > max.z) then
13 |  this(node.left, min, max, depth + 1);
14 |  return;
15 else
16 |  this(node.left, min, max, depth + 1);
17 |  this(node.right, min, max, depth + 1);
18 end
19 if node.value <= max AND node.value >= min then
20 |  points.add(node.value);
21 end
```

Algorithm 2: Range search

3.2 Plane fitting

Fitting a plane is a problem where the idea is to find a plane that is as close as possible to a set of n 3-D points $(p_1, \cdots, p_n)$. The proximity is measured by a square sum of the orthogonal distances between plane and points.

Let the position of the plane be represented by a point c, element of the plane and a

normal vector n. The orthogonal distance between a point p_i and the plane is then

$$d_i = (p_i - c)^T n.$$

Thus the plane can be found by solving

$$\mathrm{e} = \min \sum_{i=1}^{n} d_i^2.$$

To solve this problem the algorithm 3 can be used. For more information, see [8].

Input: *points* := set of points
Output: c := point in the plane, n := normal vector

foreach *points* **do**
  $c = c + point$
end
$c = c/points.size$;
for $i = 0$ *to* $points.size$ **do**
  $A(0, i) = points(i).x - c.x$;
  $A(1, i) = points(i).y - c.y$;
  $A(2, i) = points(i).z - c.z$;
end
svd(U, S, V, A) ; // $USV^T = A$, Singular Value Decomposition
$n = (U(1, 3), U(2, 3), U(3, 3))$;

Algorithm 3: Plane fitting

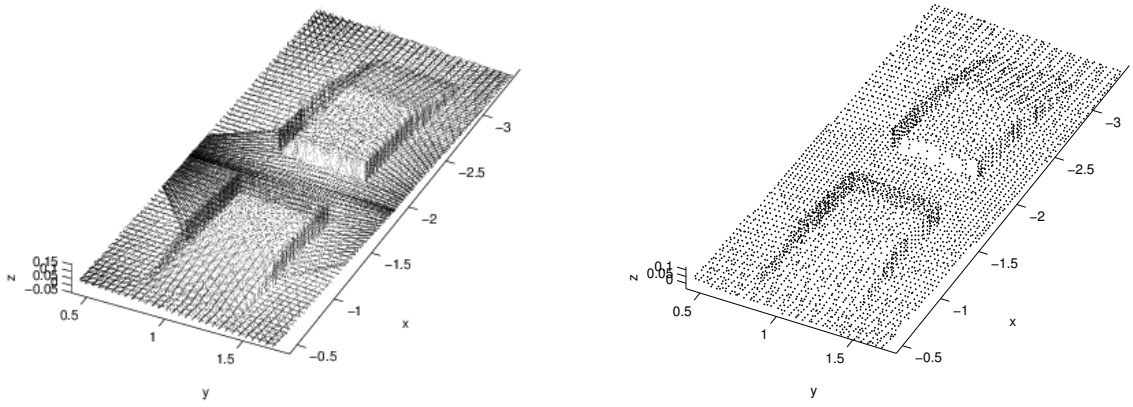
For singular value decomposition author had chosen to use a third-party library called Armadillo. Armadillo is an open-source C++ linear algebra library and can be distributed under the terms of the GNU Lesser General Public License. For detailed information about Armadillo library see [9].

4 Filtering

As considered data set contains hundreds thousands of points, which represent only a few planes, most points can be filtered out without losing too much information. Filtering is used to reduce the number of points and to remove the outliers, which speeds up the calculation and suppresses problems with measurement noise.

For filtering author designed a simple algorithm based on uniform sampling of the environment. Firstly, the whole space is divided into equal cubes ($\delta \times \delta \times \delta$). If there are not enough points inside a cube, it is classified as an outlier and all points inside the cube are removed. Otherwise all points in the cube are replaced with their median value.

To find the value of parameter δ, the largest dimension has to be found first. The largest dimension is the longest distance between minimal and maximal value in dimensions x, y or z. This distance is then divided by 100 and the result of this computation is δ. Division by 100 is optional and can be changed to any integer value. This value represents the maximum number of points in one dimension after the filtering is done. Parameter δ is one of the most important values, because it is used as a base distance unit in subsequent steps of the method, which will be explained in following text.



(a) Not filtered (b) Filtered

Figure 6: Difference between filtered and non filtered points

Input: *points* := set of points that will be filtered, *maxPointsInDimension* := maximum points in one dimension after filtering is done, *minPointsInCube* := maximum number of points in cube to classify points as outliers
Output: *filteredPoints* := filtered points

```
1  cornerL = find left corner ;          // minimal value in x, y and z dimension
2  cornerR = find right corner ;         // maximal value in x, y and z dimension
3  Δ = cornerR − cornerL;
4  δ = (the largest dimension in Δ) / maxPointsInDimension ;
5  for i = 0 to Δ.x/δ do
6      for j = 0 to Δ.y/δ do
7          for k = 0 to Δ.z/δ do
8              Point p = new Point(i ∗ δ, j ∗ δ, k ∗ δ) + cornerL;
9              cube = points.rangeSearch(p,p + δ) ;        // set of points in cube
               defined by the left(p) and the right(p + δ) corner
10             if cube.size > minPointsInCube then
11                 filteredPoints.add( median(cube) );
12             end
13         end
14     end
15 end
```

Algorithm 4: Filtering

5 Finding planes

To find the walls, plane equations have to be found first. Algorithm for planes equation finding is based on [1]. Author made changes to speed up the calculation and to improve the results. The whole algorithm can be divided into two parts:

- placing small clusters, which place a few hundred clusters to the model,
- spreading clusters, which enlarge and unite the clusters.

A cluster is considered to be a set of points lying in the same plane. After small clusters are placed, there can be more than one cluster per wall, but they can be united in the second part of the algorithm. In the end each cluster should represent one wall.

5.1 Placing small clusters

As was mentioned above, a cluster contains only points from the same plane, so author needs methods to decide whether points are from the same plane. For this there was designed a simple algorithm which returns true or false with respect to distance of points from the considered plane. If the distance between all points and a plane is smaller than parameter $minDistance$, method returns true, otherwise it returns false.

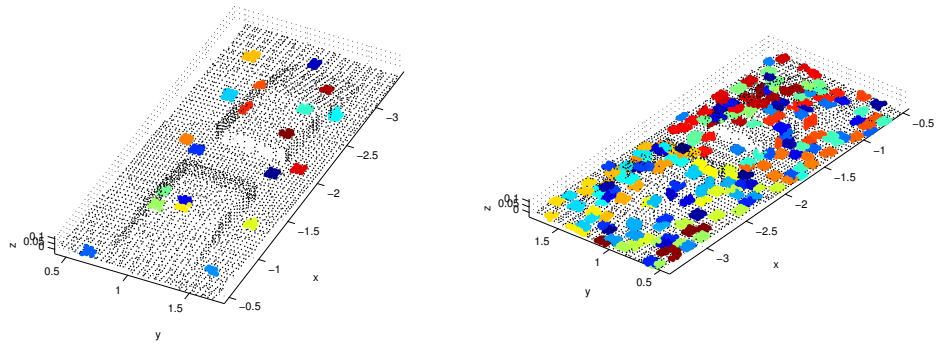
Placing clusters is an iterative process in which points are selected by breadth-first search (BFS) algorithm. In kd-trees, BFS returns points in an approximately uniform distribution. If all points in the range around the selected point are from the same plane, the cluster is placed and this is repeated few hundred times depending on parameter $numberOfClusters$. Should there be insufficient number of clusters after all points were tested, the algorithm continues with the next stage. That is not possible if the points are selected from the tree randomly because that can lead to endless cycle.

If there is not enough small clusters placed, some of the planes will not be found. This situation is shown in figure 7a.

Input: $numberOfClusters$, $minPointsInCluster$:= minimum points in cluster to place it, $range$, $minDistance$
Output: $clusters$:= set of clusters

```
1  foreach point selected by BFS algorithm from kd-tree do
2  |  cluster = points.rangeSearch(point − range, point + range);
3  |  if cluster.size < minPointsInCluster then
4  |  |  continue;
5  |  end
6  |  cluster.calculatePlane;
7  |  if cluster.arePointsFromPlane(minDistance) then
8  |  |  clusters.add(cluster);
9  |  end
10 end
```

Algorithm 5: Place small clusters



(a) 20 clusters (b) 200 clusters

Figure 7: Result of the place small clusters algorithm with a different number of clusters

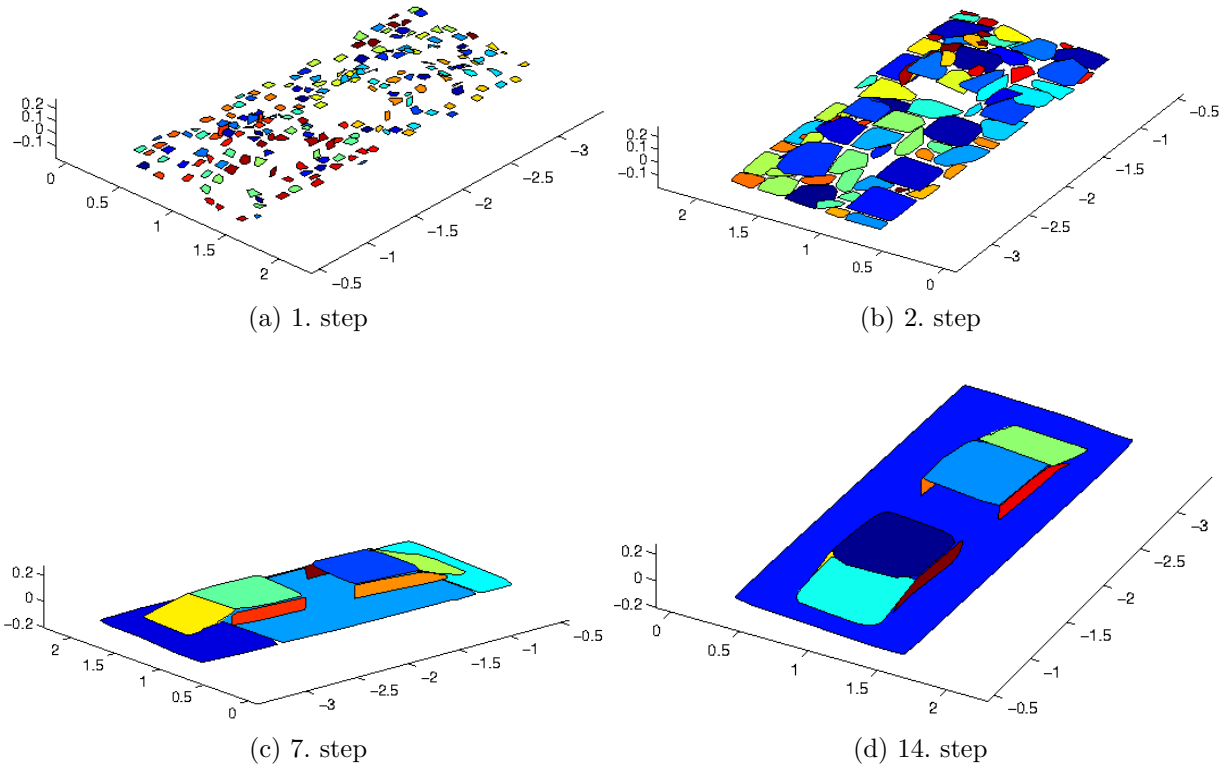
5.2 Spreading clusters

Now there are a few hundred small clusters placed and they are ready to spread. It means that they are enlarged and united. At the end of this process there should be only one cluster per wall.

Clusters are spreading gradually by adding $spreadingFactor$ to maximum distance from center in each dimension. Clusters are competing for points which means that points from a cluster can be stolen by a cluster containing more points. If $spreadingFactor$ is too small, clusters spread slowly or do not spread at all. Otherwise, if it is too large and there exists

more than one wall per plane, they are united into one wall. Empirically, we found that the optimal value is around 5δ.

The process of spreading is repeated until there are at least fifty times less changes than in the most productive run.



(a) 1. step (b) 2. step

(c) 7. step (d) 14. step

Figure 8: Process of spreading, for better illustration points are shown as polygons

Input: *spreadingFactor*, *minPointsInCluster*, *minDistance*, *outliersRange*, *outliersMinPoints*
Output: *clusters*

```
1  while changes is larger than maxChanges/50 do
2      foreach clusters do
3          if cluster.size < minPointsInCluster then
4              cluster.remove;
5          end
6          cluster.calculatePlane;
           // maxDistanceFromCenter is method, which returns point
              containing maximum distance from center in each dimension
7          Point range = cluster.maxDistanceFromCenter + spreadingFactor;
8          pointsSet = points.rangeSearch(center − range, center + range);
9          foreach point in pointsSet do
10             if point is in plane and is not inside larger cluster then
11                 cluster.add(point);
12             end
13         end
14         cluster.calculatePlane;
15         cluster.removePointsOutsidePlane(minDistance);
           // filter used to remove outliers inside the cluster
           // if there are not enough points in range around point p, p is
              removed
16         cluster.removeOutliers(outliersRange, outliersMinPoints);
17     end
18 end
```

Algorithm 6: Spreading clusters

6 Forming polygons

6.1 Points to planes fitting

Despite having points in appropriate clusters and having planes equations, points do not lie exactly in the plane. It is caused by measurement noise. To find polygon borders it is useful to fit these points to the plane.

Fitting is done by replacing points with their projection on the particular plane. A projection is the closest point lying exactly in the plane.

6.2 Finding polygon borders

Since algorithm should recognize only simple objects, author will assume that these objects are created only from convex polygons [10]. It is easier to find a convex polygon than a general polygon. There exists a solution for general polygons explained in [11].

Before explaining the algorithm, the method isSeeing(), which was used for border finding, has to be explained first. In 2-dimensional space method returns true or false depending on the position of tested point with respect to tested line. Position of a point is determined by the point to line distance. If distance is positive, point lies on the right side, otherwise on the left (if distance is zero, point lies exactly on the line) .

In 3-dimensional space it is similar, but method returns position of a point with respect to a plane. To construct this plane, the polygon edge and the plane normal vector are used. This process returns a plane perpendicular to polygon plane and polygon edge will lie on this plane. If tested point does not see any edge of the polygon, it means point is inside the polygon.

For polygon border finding the incremental algorithm is used as described in [12]. The incremental algorithm is an algorithm for computing the convex hull of a set of points. The basic idea is to add points one at a time updating the hull as algorithm proceeds.

Input: *points*
Output: *polygon*

```
// Start with simple polygon containing only two vertices
polygon.add(points[0]);
polygon.add(points[1]);
foreach point from points do
    if point is inside polygon then
        continue;
    end
    // Replace all visible edges with two new edges, which connect the
       tested point to the remainder of the old hull
    firstVisibleVertex = find first visible vertex;
    lastVisibleVertex = find last visible vertex;
    polygon.removeVerticesBetween(firstVisibleVertex,lastVisibleVertex);
    polygon.addVertexAfter(point, firstVisibleVertex);
end
```

Algorithm 7: Polygon border finding

7 Final approximation

7.1 Polygons vertices fitting

Due to the measurement noise and filtering algorithm the polygons vertices are not lying in the planes intersection. To improve the result, it is appropriate to snap them to these intersections. Firstly, snapping to the corner is tested and if there is no corner in specified distance from the vertex, the vertex is snapped to the nearest line. Snapping is done only if vertex is not too far from the planes intersection.

To decide whether planes intersection or a corner exists a simple method is used. There has to be at least one point in the range search around the corner (or around the vertex to line intersection projection) in all clusters which were used to compute the corner (or line intersection).

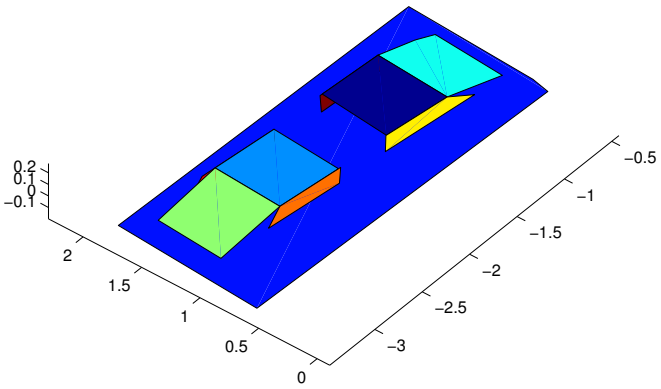


Figure 9: Result of vertices fitting process

Input: *polygons* := set of all polygons, *maxDistance* := maximum distance to fit the vertex
Output: *polygons* := fitted polygons

```
1  foreach polygon from polygons do
2      foreach vertex from polygon.vertices do
3          closerCorner = find the closer corner;
4          if closerCorner.distanceTo(vertex) < maxDistance then
5              vertex = closerCorner;
6              continue;
7          end
8          closerInter = find the closest intersection to vertex;
9          if closerInter.distanceTo(vertex) < maxDistance then
               // vertex to intersection projection
10             vertex = closerInter.projection(vertex);
11         end
12     end
13 end
```

Algorithm 8: Polygon vertices fitting

7.2 Vertices reduction

After polygons vertices fitting, there can exist more than two points per edge. To reduce this collinear points author used an algorithm 9.

Input: *maxDistance* := maximum distance from line to classify points as collinear
Output: *polygons* := polygons with reduced vertices

```
1  foreach polygon from polygons do
2      foreach vertex from polygon.vertices do
3          Line l = construct line from vertex before and after vertex;
4          if l.distanceTo(vertex) < maxDistance then
5              vertex.remove;
6          end
7      end
8  end
```

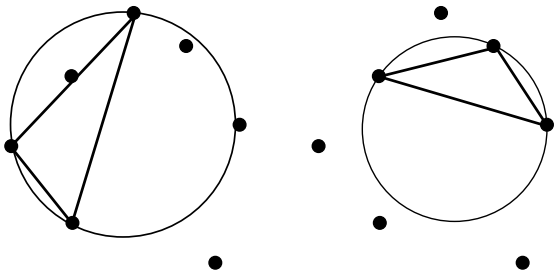
Algorithm 9: Vertices reduction

7.3 Triangulation

A lot of motion planning or simulation methods accept only a set of triangles in the input. The main reason is that only three vertices are needed to construct the plane and due to the rounding more than three points usually do not lie exactly on the plane.

In this case, triangulation is a process which divides polygons into triangles. Author used a Delaunay triangulation, which maximizes the minimum angle of all the angles of the triangles so they tend to avoid skinny triangles. The main condition is that a circle circumscribing any Delaunay triangle does not contain any other input point in its interior [13].

In algorithm 10 was used a method to decide if a triangle meets Delaunay condition. Firstly, method calculates circumscribing circle center and radius. For any point with distance to center smaller than radius, method returns false. Otherwise, it returns true.



(a) Does not satisfy Delaunay condition

(b) Satisfies Delaunay condition

Figure 10: Triangulation example

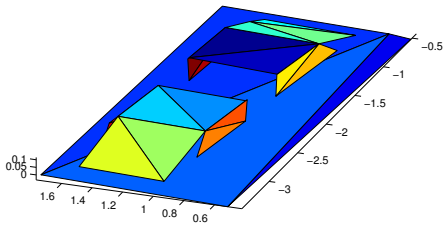


Figure 11: Result of triangulation process

Input: *vertices* := polygon vertices ordered counter clockwise
Output: *triangles* := set of triangles

```
1  while vertices.size > 2 do
2      foreach vertex v in vertices do
3          Triangle t = construct triangle from vertices v_-1, v and v_+1;
           // index -1 means vertex before v
4          if t meets delaunay condition then
5              triangles.pushBack(t);
6              remove v from vertices;
7              break;
8          end
9      end
10 end
```

Algorithm 10: Triangulation

8 Experimental results

In this section will be presented the results of algorithm described in sections above. To test the algorithm author used datasets collected by mobile robots in Gerstner laboratory on Czech Technical University in Prague. These datasets are displayed on figure 6a, figure 12 and they are also provided on enclosed CD in folder data (these datasets will be referred to as data1 and data2).

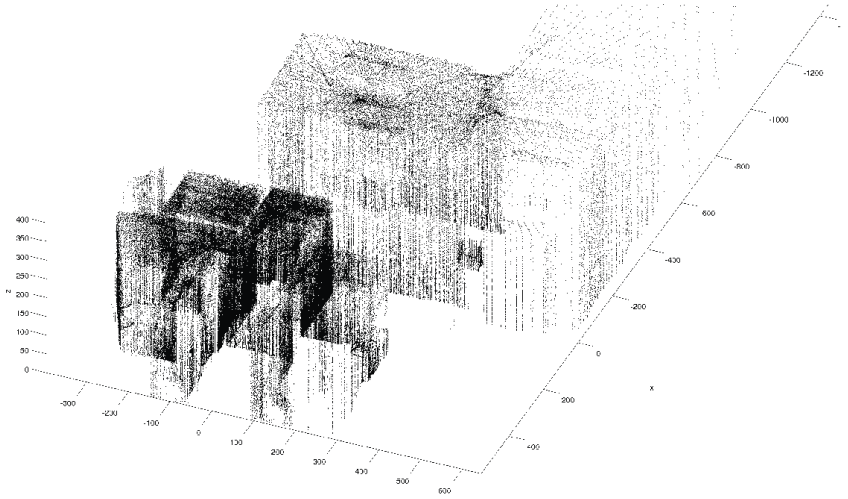


Figure 12: Dataset data2

Importance of the filtering algorithm is shown in table 1. The number of points in each dataset is reduced to approximately 5000 points after the filtering.

Dataset	Before filtering	After filtering
data1	173661	4678
data2	523664	4153

Table 1: Number of points in datasets

Dataset	Number of placed clusters	Number of founded walls
data1	200	9
data2	200	30

Table 2: Number of clusters

One of the most important values are numbers of found walls and triangles. In table 2 and 3 this information is displayed. These values can be used as approximate information

Dataset	Number of triangles
data1	20
data2	102

Table 3: Number of triangles

about time consumption in motion planning algorithm.

In figure 13 and table 4 can be seen running times of all parts of algorithm. All running times were measured on notebook with 3.9 GB RAM and processor Intel Core 2 Duo CPU P8400 @ 2.26GHz × 2. The complete computation time is not only sum of partial algorithms because of input/output operations.

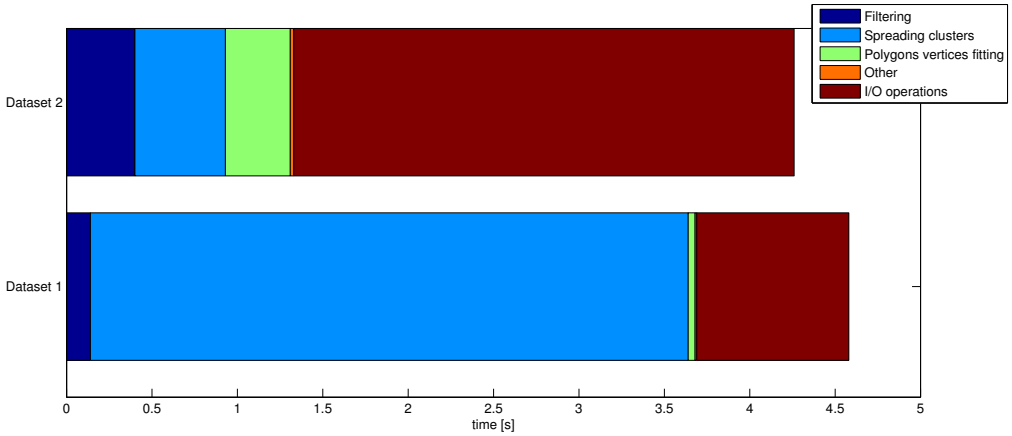


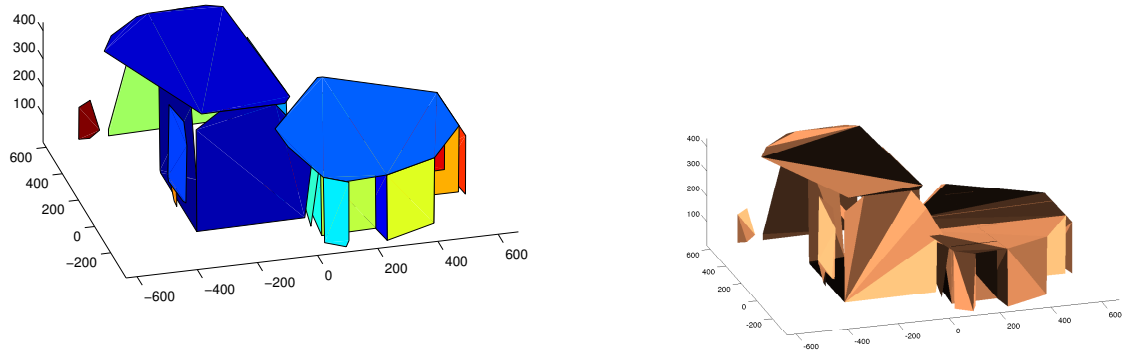
Figure 13: Algorithm time-consumption

Algorithm name	data1	data2
Filtering	140 ms	410 ms
Placing small clusters	10 ms	20 ms
Spreading clusters	3 500 ms	530 ms
Fitting to planes	0 ms	0 ms
Finding polygons border	0 ms	10 ms
Polygons vertices fitting	40 ms	380 ms
Vertices reduction	0 ms	0 ms
Triangulation	0 ms	0 ms
Complete	**4 580 ms**	**4 260 ms**

Table 4: Algorithm time-consumption

The final result for dataset data1 is shown in figure 9 (before triangulation) and in

figure 11 (after triangulation). The result for data2 is shown in figure 14a and 14b.



(a) Before triangulation

(b) After triangulation

Figure 14: Results for dataset data2

The result for dataset data1 was also tested as environment model in simulator of Large-Scale Integrated Project Symbrion. In [14] this geometrical map was compared to another two reconstructed models. The simulation on this model was significantly faster than simulation with other 3D models. Performance of the simulation is displayed in table 5. Complete results as well as simulation description and description of methods used for reconstruction is shown in [14].

Method used for reconstruction	mean [ms]	dev [ms]
GG-based	333	125
GSRM	115	43
Method described in this work	2.5	0.5

Table 5: Performance of a simulation

9 Implementation

In this section author explains the basic concept of algorithm implementation. The whole implementation can be divided into two parts computation tools and visualization tools

9.1 Computation tools

Author programmed computation tools in C++ language and followed concept of the object-oriented programming. The main intention was to create easy to use class, which can be used in many different projects. The main class is called *Reconstruction* and contains all the algorithms described in the sections above, as well as methods for loading and saving data from and into the file.
Example of the *Reconstruction* class use:

```
Reconstruction recon;
recon.loadPointsFromFile('with_floor3.txt');
float delta = recon.filterPoints(100, 10);
recon.savePointsToFile('filtered.txt');
```

Program was compiled on Linux operating system but it can also be compiled on Windows OS. For compiling it is important to have Armadillo library installed and to add *-larmadillo* as parameter for compiler. It is also strongly recommended to have optimization enabled when compiling programs, because Armadillo is a template library.

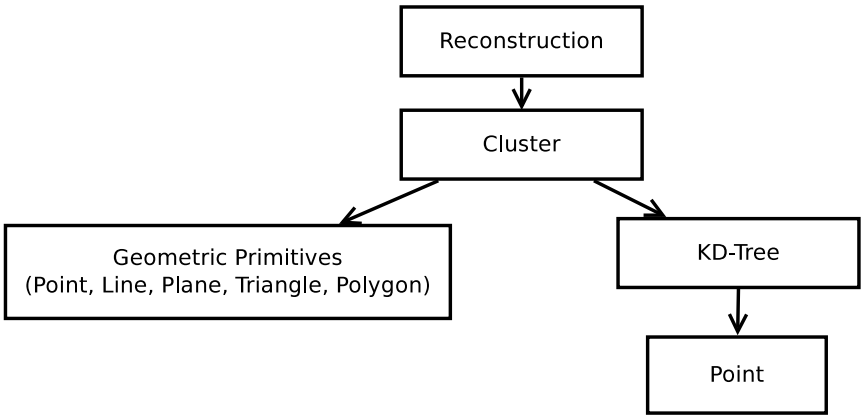


Figure 15: Class Hierarchy

9.2 Visualization tools

For visualization author has decided to use Matlab environment. The programmed scripts can be used to visualize data saved by the computation tools.

9.2.1 Display points, polygons, triangles

The only parameter for all the functions described in this section is the name of the file where data are saved.

To visualize points, function *displayPoints* can be used. It displays non clustered points as small black points and clustered points as bigger coloured points. The points from the same cluster are coloured equally. Triangles and polygons are also distinguished by colours.

9.2.2 Display process of computation

The script *displayProcess* can be used to visualize whole process of computation. The process of placing small clusters is shown up front, and then follow the processes of spreading and final approximation. There are two possibilities on how to visualize the process of spreading:

- in a form of point cloud,
- as polygons.

The script is set to display polygons by default but it can be changed by setting variable *showPolygons* to false. There are more parameters that can change process of visualization like *showPlacing, pauseSpreading*, etc. Complete list of parameters with their default values is on the top of the script.

10 Conclusion

The main goal of this bachelor thesis was to create an algorithm for detecting the walls in a point cloud measured by the mobile robot. The purpose of this algorithm was to generate a simplified model of an environment around the mobile robot for faster motion planning, faster simulation and for the memory requirements reduction. Many methods were designed for 3D object reconstruction and author explained some of them in section 2.

Because it is not efficient to operate on unfiltered data, simple filtering algorithm which provides uniformly distributed data and significantly reduces amount of points in the data sets was implemented. The filtering algorithm is simple and filtering process can fail if there are outliers far away from other points in point cloud. Due to the lack of time, the algorithm for outlier removal was not implemented. This algorithm was described in section 4.

Filtered points were clustered. The main condition for the clustering algorithm is that all the points in the cluster should lie on the same wall. The clustering algorithm is divided into two parts *Placing small clusters* and *Spreading clusters*. Both parts were explained in the section 5.

The next step was to find the borders of the walls. Author used the surface based algorithm described in 6. Now, the walls are represented by the mathematical equations but due to the measurement noise, border of the walls do not lie exactly on the planes intersections. The section 7 contains description of the simple approximation algorithm which fits border to the closest corner or to the closest planes intersection.

In computer graphics it is usual to represent object by the set of triangles so author programmed an algorithm which transforms the mathematical equations of the walls into the set of triangles.

The algorithm was tested on two datasets and the results are shown in section 8. There is also time-consumption analysis for all sub-algorithms for the both datasets. For visualization were programed a few scripts for the Matlab, which are explained in section 9. Also a few words about the implementations can be found in this section.

The result of the algorithm was used for the purpose of the Large-Scale Integrated Project Symbrion to speed up simulation of robot swarm behaviour. The algorithm was also published at Mathmod 2012 - 7th Vienna International Conference on Mathematical Modelling [14] and IEEE ICRA workshop on Reconfigurable Modular Robotics [15].

References

[1] Ing. Václav Vopěnka. Environment Model Reconstruction from a 3D Point Cloud. Bachelor Thesis, 2009.

[2] David J. Austin and Brenan J. Mccarragher. Geometric constraint identification and mapping for mobile robots, 2001.

[3] Y. Tarutoko, K. Kobayashi, and K. Watanabe. Topological map generation based on delaunay triangulation for mobile robot. In *SICE-ICASE, 2006. International Joint Conference*, pages 492 –496, oct. 2006.

[4] Jongmoo Choi and G. Medioni. Starsac: Stable random sample consensus for parameter estimation. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 675 –682, june 2009.

[5] Wikipedia. Hough transform — Wikipedia, the free encyclopedia, 2012. [Online; accessed 16-May-2012].

[6] Dorit Borrmann, Jan Elseberg, Kai Lingemann, and Andreas Nüchter. The 3d hough transform for plane detection in point clouds : A review and a new accumulator design. *3D Research*, 02003(2):32, 2011.

[7] Andrew Moore. *A tutorial on kd-trees*. University of Cambridge Computer Laboratory Technical Report No. 209, 1991.

[8] Inge Söderkvist. Using SVD for some fitting problems. University Lecture, 2009.

[9] Conrad Sanderson. Armadillo library, 2011. [Online; accessed 11-March-2012].

[10] Math Open Reference. Convex polygon definition, 1998. [Online; accessed 02-May-2012].

[11] M. Teichmann and M. Capps. Surface reconstruction with anisotropic density-scaled alpha shapes. In *Visualization '98. Proceedings*, pages 67 –72, oct. 1998.

[12] Tim Lambert. Incremental algorithm for computing the convex hull of points, 1998. [Online; accessed 11-March-2012].

[13] Wikipedia. Delaunay triangulation — Wikipedia, the free encyclopedia, 2012. [Online; accessed 11-March-2012].

[14] V. Vonásek, M. Kulich, T. Krajník, M. Saska, D. Fišer, V. Petrík, and L. Přeučil. Techniques for Modeling Simulation Environments for Modular Robotics. In *Proccedings of International Conference on Mathematical Modelling*, pages 1–6, Vienna, 2012. Vienna University of Technology.

[15] M. Saska, V. Vonásek, M. Kulich, D. Fišer, T. Krajník, and L. Přeučil. Bringing Reality to Evolution of Modular Robots: Bio-Inspired Techniques for Building a Simulation Environment in the SYMBRION Project. In *Reconfigurable Modular Robotics: Challenges of Mechatronic and Bio-Chemo-Hybrid Systems*, pages 1–6, Piscataway, 2011. IEEE.

Appendix A

Contents of the enclosed CD

bachelor_thesis.pdf	The PDF version of this document.
data/	The folder with the used datasets.
computation/	The source code used for computation.
visualization/	The scripts used for visualization.
results/	The folder with computed results.