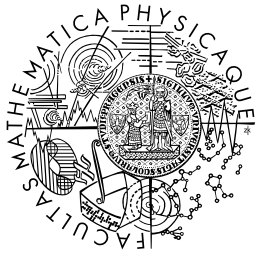

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Jan Škoda

3D Navigation for Mobile Robots

Department of Theoretical Informatics and Mathematical Logic

Supervisor of the master thesis:	prof. RNDr. Roman Barták, Ph.D.
Study programme:	Informatics
Study branch:	IUI

Prague 2017

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In ..... date ..... signature of the author

Název práce: 3D navigace pro mobilní roboty

Autor: Bc. Jan Škoda

Katedra: Katedra teoretické informatiky a matematické logiky

Vedoucí diplomové práce: prof. RNDr. Roman Barták, Ph.D., Katedra teoretické informatiky a matematické logiky

Abstrakt: V práci je navržen 3D navigační systém pro hledání cest pro autonomní vozidla. Systém zpracovává point-cloud data z rgb-d kamery a vytváří trojrozměrnou mřížku obsazenosti s adaptivní velikostí buněk. Obsazené buňky mřížky jsou popsány normálním rozdělením charakterizujícím body naměřené v prostoru buňky. Tato normální rozdělení jsou pak použita pro klasifikaci buněk, stanovení průjezdnosti a detekci kolizí. Prostor průjezdných buněk je následně použit pro plánování cest. Schopnost pracovat v trojrozměrném prostoru umožňuje použití autonomních robotů ve složitě strukturovaných prostředích s několika úrovněmi povrchu, nerovným povrchem, mosty a tunely. To je důležité pro použití robotů v reálných prostředích, městech nebo při záchranných misích.

Klíčová slova: robotika, navigace, 3d, hledání cest

Title: 3D Navigation for Mobile Robots

Author: Bc. Jan Škoda

Department: Department of Theoretical Informatics and Mathematical Logic

Supervisor: prof. RNDr. Roman Barták, Ph.D., Department of Theoretical Informatics and Mathematical Logic

Abstract: We propose a novel 3D navigation system for autonomous vehicle path-planning. The system processes a point-cloud data from an rgb-d camera and creates a 3D occupancy grid with adaptable cell size. Occupied grid cells contain normal distribution characterizing the data measured in the area of the cell. The normal distributions are then used for cell classification, traversability and collision checking. The space of traversable cells is then used for path-planning. The ability to work in three-dimensional space allows the usage of autonomous robots in highly structured environments with multiple levels, uneven surface or various elevated and underground crossings. That is important for the usage of robots in real-world scenarios, in urban areas or for disaster rescue missions.

Keywords: robotics, navigation, 3d, path finding

Contents

1. Introduction

In robotic research, the problem of navigation is among the most important. Basically all autonomous mobile robots need some kind of navigation to fulfill the *mobile* term. Even though there are countless robots employed especially in industry, the field of mobile robotics still didn't make noticeable impact on industry or our daily life.

One of the reasons is the complexity of orientation and navigation in diverse real-world environments. For a robot, just sensing a 3D environment and modelling it in an internal map is a complicated task, especially because the robot has to keep track of its location. And planning movement in such a map is another step forward, that is dependent on the results of robot's localization and mapping capabilities. The available systems are therefore not very reliable, which limits the usage of autonomous mobile robots in most use-cases.

We understand navigation as a process of planning a path of a mobile robot from its current position to a desired goal location, following the planned path, and avoiding any discovered obstacles along the way. The desired paths have to fulfill several conditions to ensure safety and feasibility of the navigation. Moreover, the paths can be also compared in terms of desirability – for example short or smooth paths are usually more desirable than long and curved ones. Such paths should therefore be preferred in the navigation process. Beyond the path planning, the navigation problem also involves reacting to changes of the environment model, that is to be stored in some kind of map.

The simplest navigation systems work with a fixed map prepared in advance and an external source of location (such as GPS) and their implementation is straightforward and reliable. However, when an unknown environment, indirectly measured location and uncertainty come into play, the navigation problem gets more difficult. The area of navigation has been thoroughly studied, but mostly for two dimensional space. Although there exists many systems capable of creating 3D maps, we haven't found any real-time ground vehicle navigation that could fully utilize a simultaneously updated 3D map. As we will show in Chapter 3, the existing systems mostly work with a simple 2D map, project the constructed 3D map into 2D for navigation or use a *2.5D* model – several 2D layers representing surface levels above each other [3].

The major issue of 3D navigation and one of the reasons, why most navigation systems work in 2D, is the computational complexity of 3D sensor processing, path-planning, and traversability estimation – the recognition of areas, where the robot can drive. We therefore utilize several techniques to improve computational efficiency of the system, especially octree [4] data structure for map representation and 3D normal distribution transformation [10] (3D-NDT) for data representation and traversability estimation.

In this thesis, we introduce a 3D navigation system for ground-vehicles equipped with an rgb-d camera. Our system performs map processing, traversability and collision checking, and path planning to a given goal location. When combined with a third-party localization and mapping libraries, sensor and robot drivers, the system is able to navigate in indoor environments without a previously prepared map, to build a map simultaneously with navigation, and to handle broad

range of problems and sensoric errors that can happen during the navigation (see Figure 1.1).

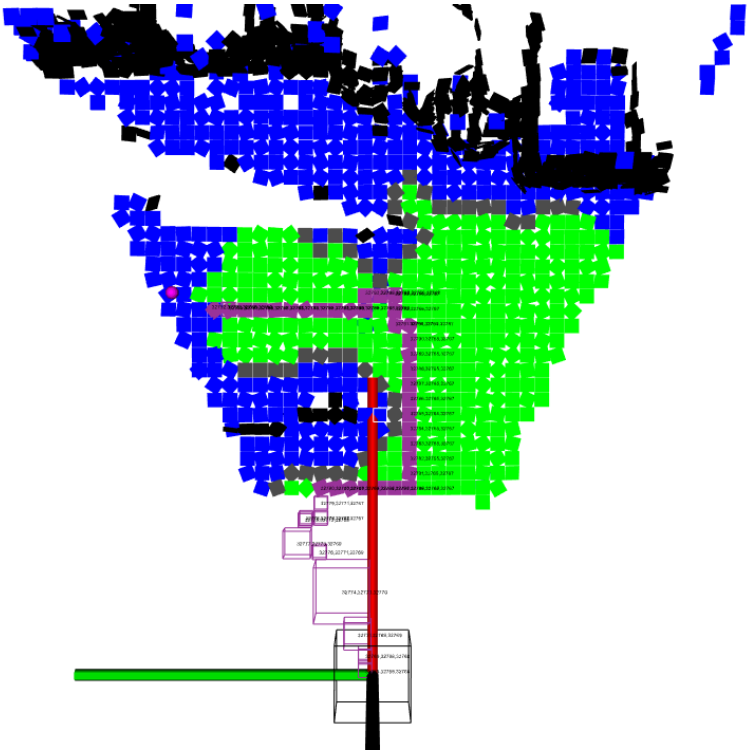

Figure 1.1: A visualization of the map and a plan. The robot pose is shown as the black cube (bottom), the goal location is the purple dot (top left). The path is denoted by purple map cells. The black cells are obstacles, blue and green cells are traversable.

The thesis is organized as follows: First, we will provide an outline of our system and the robotic platforms we used for testing and evaluation. After that, we provide a summary of related work targeted especially on comparable systems and the techniques that are used in our work. The description of our system is organized into three chapters. The Map processing chapter covers our map structure, the Path planning part covers the path-finding, traversability, and obstacle checking algorithms. Finally, the Implementation chapter documents how to use and extend our system.

1.1 Motivation

As robots operate in the 3D world, the simplification of the world into a 2D map causes a loss of information. This can prevent the robot from operating correctly in more complicated situations. We provide three examples of types of situations, that frequently happen in real-world scenarios.

In Figure 1.2, there are two surface levels and the robot can choose to ride under the bridge or on it. 2D navigation systems would be unable to use the "bridge" and mostly consider it an obstacle. In Figure 1.1, the robot has to enter a yet unmapped area. That can happen for example in rescue missions in an unknown environment or when some parts of the environment were occluded

during mapping. Finally, 2D navigation systems often cannot sense holes in the surface or ends of it, as the reader can see in the "cliff" figure 1.3.

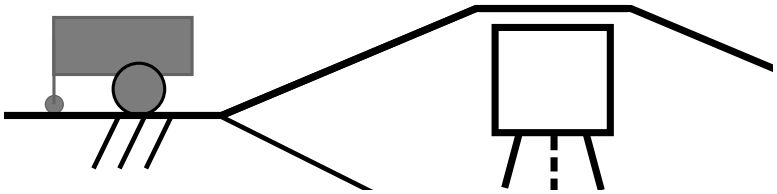

Figure 1.2: Example situation: Robot can choose to ride on or under the bridge.

In general, 3D navigation is necessary in any environment, where the robot has to traverse significantly uneven surfaces. One of the typical robotic problems, where 3D navigation would be very useful, is autonomous or assisted rescue mission in damaged buildings after various disasters. Inability of state-of-the-art navigations to work in such environments complicates the usage of robots in real-world scenarios.

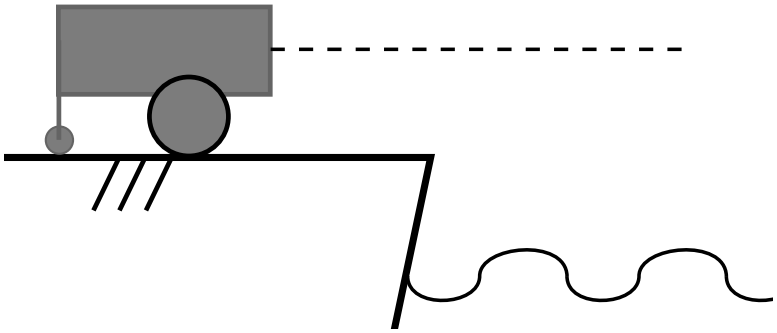

Figure 1.3: Example situation: Robot can fall off the cliff. Dashed line is used to visualize a rangefinder beam.

2. System outline

2.1 Used platforms

We have developed and tested the system on two robotic platforms of two universities in Japan and in the Czech Republic. The system was however designed with reusability in mind, so it should run on any robot matching the requirements described in Chapter 6. Moreover, we use the widespread Robotic Operating System as an interface between the robot, our system and the third-party modules. That ensures that the parts of the system can be exchanged very easily, as we checked ourself when switching the platforms. In this section, we will present the hardware we use, as it has substantial influence on the design of our navigation system.

Initial development was done on the *Yamabico* platform of the University of Tsukuba, Japan. The Yamabico robot has differential drive (two independent wheels with two engines) with precise, but two-dimensional, odometry. It is also equipped with a Hokuyo laser rangefinder, that was not used by our system. The robot is controlled over USB from a notebook placed on the top. We attached an *Asus Xtion Pro Live* rgb-d camera on top of the robot. An overview of the Yamabico robot is in Figure 2.1.

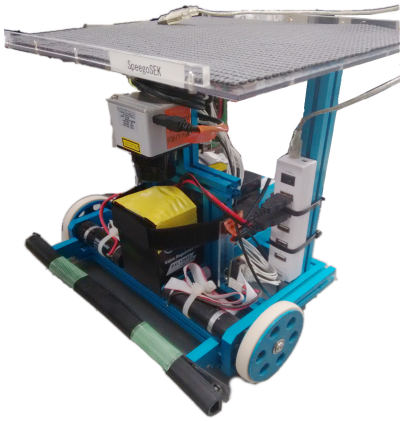

Figure 2.1: The Yamabico robot of the University of Tsukuba. Here the camera is not attached.

Figure 2.2: The robot of the Faculty of Mathematics and Physics, that was used for the evaluation.

The system development continued at the Faculty of Mathematics and Physics on another robot with similar properties. The robot also has differential drive with odometry, is controlled by the *Atmega128* board and the *Sabertooth* motor driver. It is connected to a notebook using a serial and an USB cable. We attached a *Microsoft Kinect* rgb-d camera to the mount on top of the robot.

The rgb-d camera is a sensor able to measure the depth of the observed pixels together with their color. Both mentioned cameras use infrared light for depth sensing, which makes them unusable in direct sunlight and therefore in outdoor environments. Another important downside of those cameras is that they have

a minimal sensing range around 0.4 m and cannot see any object that is closer. The *Xtion* camera is powered over USB from the notebook, the *Microsoft Kinect* camera is supposed to be powered separately, so it was customized to be powered from the robot batteries.

The communication between our navigation system, the robots and the third-party localization and mapping system is provided by the Robotic Operating System (ROS), which is offering standardized interfaces for communication between robots and their control systems. ROS aims to allow easier sharing and exchange of hardware and software packages and should help other researchers to test our system without having to adapt the code to work with their robots. More details are given in Chapter 6.

2.2 Proposed solution

The input of the system is an rgb-d measurement and a location estimate continuously received from a third-party localization and mapping system (SLAM). Our implementation currently uses the Rtabmap [6] package, but it can be exchanged for another system with a slight modification of the code. Rtabmap is able to use the rgb-d stream to track the movement of the robot and generates its odometry.

Our system receives the data from the camera through the SLAM system. The captured rgb-d frame is represented as a point-cloud [9], which is basically a list of point coordinates relative to the camera pose. Together with the point-cloud, our system receives the pose estimate of the robot at the time of capturing the respective rgb-d frame and timestamp.

We process the point-cloud input into a denser representation – a 3D octree [4] (discussed in section 4.1), which is an extension of an occupancy grid able to compactly model the map using adaptive cell size. Thanks to the octree, we can model the occupied parts of the map in detail, while the empty and unexplored parts remain coarse with a big cell size. We can then quickly check any grid cell for traversability or presence of an obstacle, which would not be possible with the original form of a list of points.

This grid approach alone would however not be suitable for traversability checking, as it would require too fine grid resolution, which would ruin the performance (see Figure 2.3). We therefore propose a solution utilizing approximation of the measured points with 3D-NDT [10] (3D Normal Distribution Transform) for traversability estimation. This is described in section 4.2.

Traversable cells of the map are be used for path-planning. We utilize the well-known A* algorithm with Manhattan distance as a heuristic and a cost, which is based mainly on the path length, as optimality criterion. As the map representation distinguishes free and unexplored areas, the plan can even traverse small unexplored areas. That is needed because maps often contain small unexplored "gaps", for example because of occlusions.

The map update has to deal with several problems caused by the SLAM system. The localization is sometimes imprecise, which means that the measured points will be added to a wrong part of the map. The system is therefore able to *rebuild* the map relocating the incorrectly localized map points, when an error is detected.

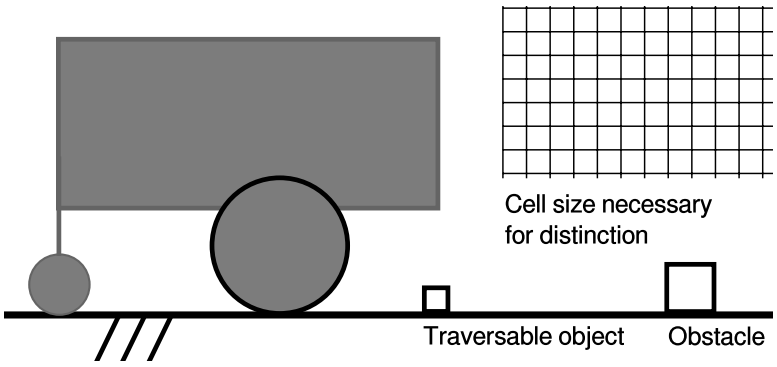

Figure 2.3: Grid size required for traversability checking would require too precise grid resolution.

3. Related work

There are many production-grade navigation systems operating in 2D [7], which are able to control the robot in dynamic environments. While mapping systems, especially visual, have been able to create 3D maps for several years, state-of-the-art navigations for ground vehicles still can't efficiently use them and instead project the 3D maps into a horizontal plane or several planes [3]. Such planes can describe a floor or multiple floors of a building, but can't represent stairs, inclined surfaces or uneven terrain. 3D maps are commonly used by navigation systems for flying robots [12], however such UAV systems don't recognize traversable surfaces, so they can't be used for ground vehicle navigation.

The field of truly 3D navigation for ground robots contains just a few results and published systems are not well suitable for usage with a real robot. For example Stoyanov et al. [11] use an approach similar to our proposal – octree structure and 3D-NDT. However, we found no navigation system that would be able to find paths in a 3D map in real time and in a dynamic environment. The mentioned system [11] cannot even replan the path with respect to obstacles discovered when following the planned path.

3.1 Our sources

If we focus on individual subproblems of the navigation – mapping, map representation and path-planning, there are interesting published results. State-of-the-art visual mapping systems for an rgb-d camera are reasonably precise and able to create 3D maps. They are however very computationally expensive and the large maps created usually need to be preprocessed in order to be efficiently usable.

Figure 3.1: An example map created by Rtabmap with the Kinect sensor. In this experiment, the sensor was carried by a person through the room to create a map [6].

We have decided to use the Rtabmap [6] localization and mapping system, that can be used for robots and handheld cameras. The system can utilize data

from various sensors such as rgb-d and stereo cameras, or laser rangefinders. Although, the created maps are not completely precise, especially when using the cameras, the system is able to correct big accumulated errors during *loop closures*, which are described in Section 4.3.1.

The created map consists of a set of colored points in space, as shown in Figure 3.1, and can be queried through the C++ library interface or using the ROS. We have chosen this system especially because of its good documentation and the support of the ROS.

For the map representation, we use a tree-structured occupancy grid with adaptable cell size called octree [4], which are commonly used for dense 3D map representation. The advantage of octree is that the empty parts of the space are represented using coarser grid cells, which results in fewer cells needed for the structure. That reduces the memory requirements, allows the map to be gradually expanded and can also save processing time when traversing the structure. We describe our usage of the octree structure in detail in Chapter 4.

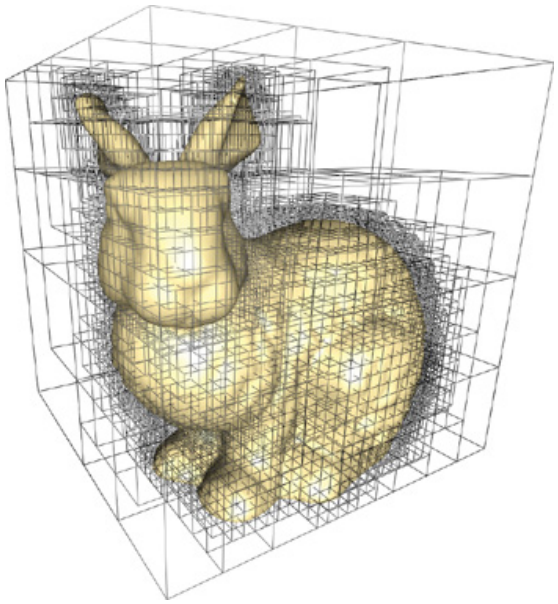

Figure 3.2: A rabbit modelled using the octree structure [4].

3.2 Comparable systems

At the beginning of the chapter, we mentioned several mostly navigation systems, with which we can compare.

Shengdong Xu et al. [12] presented a 3D navigation system for a flying drone equipped with an rgb-d camera. Moreover, they also use the octree structure for map representation to reduce the search state space in a way very similar to ours. During experiments, they showed, that the usage of the octree state space instead of a common occupancy grid leads to a nearly 24 times speedup and 3 times smaller memory usage at the cost of 11% increase of the resulting path cost. The testing of the system was done both in simulation and on a real robot.

The disadvantage of the system is that it relies on an external source of localization signal instead of using a SLAM algorithm, which restricts it to the lab

environment. It is questionable how would their system handle the problems arising from imprecise localization. As their system is targeted on flying drones, it doesn't perform any traversability checks and therefore can't be used for ground vehicles. Traversability checks also require more precise map, so the system can't be directly extended to consider traversability. For comparison, their map uses the cell resolution of 0.25 m, while our system was tested with 0.05 m cells.

In another article, Stoyanov et al. [11] presented a 3D path planner, that is based on the 3D Normal Distribution Transform. Their system is however not suited for online use on a robot, as they do not implement map-update and plan paths on a complete map of an environment. That also means that they don't deal with any localization and mapping related issues, that we describe in Chapter 4.

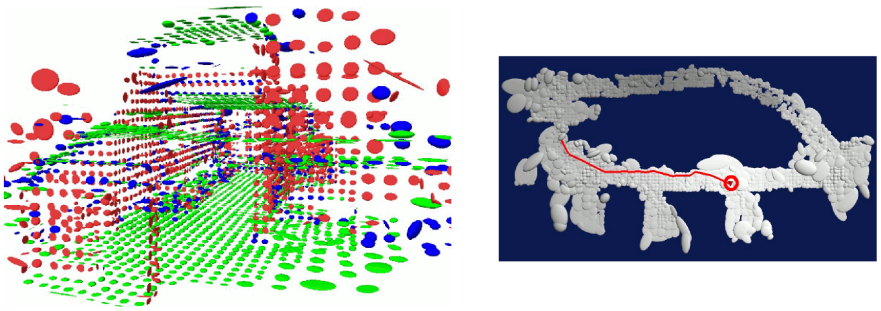

Figure 3.3: Stoyanov et al. planner. On the left side, there is a map with cells classified as traversable (green), too inclined (dark blue) and flat vertical (red). On the right side, there is an example of planned path. The images are taken from the original article [11].

Their cell classification and collision checking is very similar to ours, but the traversability checking differs. They don't distinguish between free and unexplored space, which is used in our traversability check and use the shape of the 3D-NDT to check if the surface of neighbouring cells is well aligned during the search. That is more restrictive than our system, but requires better map precision, that cannot be provided by our SLAM system and sensors.

4. Map processing

For avoiding obstacles and finding path, navigation system needs a map, an internal representation of the environment around the robot. There are many types of map representations used by mobile robots, that provide different abilities and performance characteristics. For navigation, we need a map capable of fast querying of data in given map regions, collision checking, and precise representation of surface for traversability checking. Because of those special requirements of the navigation problem, our system can't directly use the simple map built by the SLAM system and has to maintain a separate map.

In this chapter, we will present the structure and algorithms that the system uses for maintaining the map. We will describe the *octree* structure, that is used as an alternative for the common grid maps, and the 3D Normal Distribution Transform, that is stored inside of the octree cells to precisely model surface. After that, we will show the algorithms for updating map – the rgb-d measurement insertion and *loop closure* handling.

4.1 Octree structure

The internal representation of the map uses an octree structure implemented in the Octomap library [4], which is widely used in state-of-the-art robotic mapping systems. Octree is a tree structure for storing a grid with a dynamic cell size. Every node of the tree is a grid cell, which can contain none or exactly 8 children (therefore *oc*tree), which represent the 8 subcells, that have two times smaller edge size than their parent. In the following text, we will use the term *cell* instead of *node* for describing parts of octree for clarity. The root cell contains the whole environment. The cells with no children are called leafs as usual in the graph theory terminology. The structure is visualized in Figure 4.1.

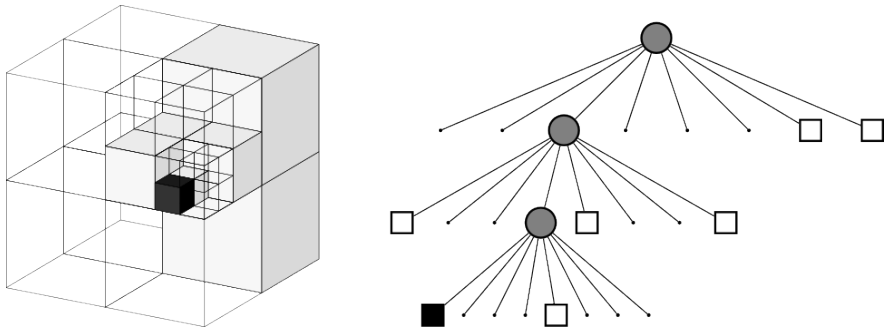

Figure 4.1: Octree data structure [4] visualization. One occupied cell is drawn in black, free cells are in non-transparent gray, unexplored cells are transparent.

The octree allows the map to describe cluttered parts of the environment with denser grid and to use coarser grid in the free space. The octree therefore saves both memory, which is an important aspect of 3D mapping, and processing time, especially during map updates. In our implementation, only unexplored and empty space is represented with coarse grid. We experimented with using coarse grid for flat surfaces and big obstacles, but it didn't lead to significantly

better performance, and on the contrary, it complicated the path-planning and reduced accuracy of the collision checking algorithm.

To maintain the grid resolution, we use the following behaviour. When a cell is updated and contains no data (is unexplored or free), we check its siblings in the tree and if all of them also contain no data, we delete them together with the updated cell, update their parent cell and perform this check recursively. On the other hand, when some data have to be inserted into the octree and we are searching for the cell that should contain them, the tree is traversed until the leaf level. If some cell along the path doesn't have a child, it is created along with all its siblings. Figure 4.1 shows an originally empty octree after insertion of some data into one cell.

Compared to occupancy grids, the disadvantage of the octree structure is the more complicated traversal. The neighbours of a cell are not adjacent in memory nor in the graph, so the routines for querying neighbours or all cells in a specified area internally traverse the tree structure. This is abstracted by the Octomap library [4].

4.2 Cell classification using 3D-NDT

Each leaf cell of the map is classified into one of those disjoint classes: *unexplored*, *free*, *obstacle* or *traversable*. All cells are initially considered *unexplored* and can later be classified as *traversable* or *obstacle* if the cell contains enough points for precise classification. On the contrary, we mark cells between the camera and observed surface as free space and therefore remove previously detected obstacles.

To distinguish traversable and obstacle cells, and precisely represent surface for traversability checking, we use the 3D Normal Distribution Transform (abbreviated as 3D-NDT) of the points measured in the cell. We basically estimate a 3D normal distribution covering points measured in every cell. If the distribution is flat and horizontal, we consider the cell to be traversable. Each normal distribution is stored in the cell as its mean value and covariance matrix:

$$\mu = \frac{1}{n} \sum_{i=1}^{n} x_i \tag{4.1}$$

$$C = \frac{1}{n-1} \sum_{i=1}^{n} (x_i - \mu)(x_i - \mu)^{\mathrm{T}} \tag{4.2}$$

where x are the vectors of 3D coordinates of points, n is the number of points, $\mu \in \boldsymbol{R}^3$ is the mean value of x and $C \in \boldsymbol{R}^{3\times3}$ is the covariance matrix. All the vector operations are element-wise. Note that not all values of the matrix have to be stored in the cell, as the matrix is symmetric.

3D-NDT can also be represented by an ellipsoid describing the confidence region of the distribution (a generalization of the confidence interval for multiple dimensions), which is the area, where certain fraction (*confidence*) of points is located (see Figure 4.2). The shape of this ellipsoid is used for traversability estimation. The center of the ellipsoid is given by the mean value μ. The directions of the principal axes of the ellipsoid are given by the eigenvectors of the distribu-

tion's covariance matrix. The squared relative lengths of the principal axes are given by the corresponding eigenvalues [5].

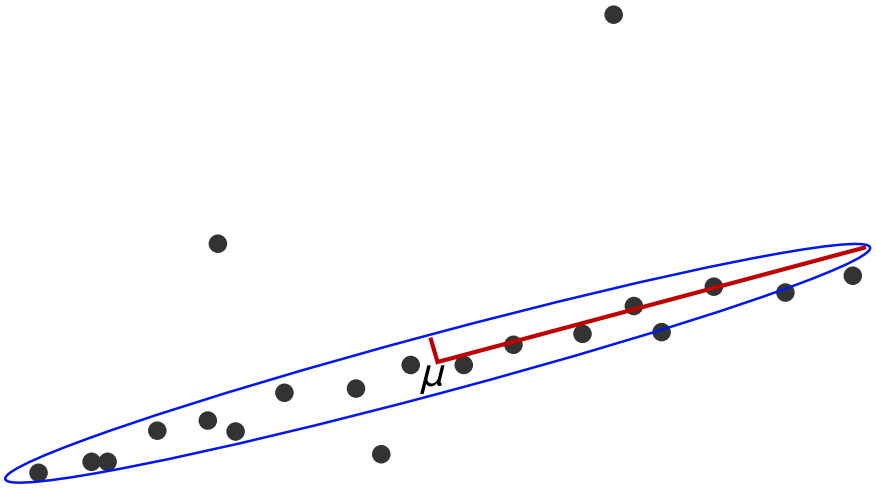

Figure 4.2: The confidence region ellipse (in blue) of a NDT representing a set of points (in black). The red segments are the axes of the ellipse. (We use 2D visualization for clarity.)

We define the normal vector of the surface measured in a cell as the shortest axis of the distribution's ellipsoid (see Algorithm 1, line 6). The axis will be inverted if it doesn't point in the direction of camera (line 8). That way, the normal vector always points in the direction of free space and its size describes the flatness of the surface – flat surface has a very small normal. Finally, we use this normal vector to estimate cell traversability: if the normal vector points upwards and is small, the surface is considered traversable (lines 14, 10). The particular values of the threshold constants were determined experimentally.

An example map with free cells properly merged (using octree) and traversable and obstacle cells classified (using 3D-NDT) can be seen in Figure 4.3. This visualization provides useful overview of the used map structure.

4.3 Map update

There are two types of map updates, that have to be handled. The first type is an rgb-d measurement, that can contain new obstacles or unexplored parts of the map. Our system receives this measurement as its sequence number, which is used as its identifier, a camera pose, and a set of point coordinates (also called *point cloud*). For performance reasons, we use only some rgb-d measurements, that might contain new information. We process a measurement when the robot's pose changes significantly or a certain time has passed since the last measurement, as you can see in Algorithm 2. The constants were chosen experimentally based on the performance results for the robot to handle the map update in real time.

Algorithm 1 Cell classification

function CLASSIFYCELL(cell,points,cameraPos)
  if $Length(points) = 0$ **then**
    return Empty
  end if
  $axes \leftarrow getConfidenceEllipsoidAxes(points, 90\%)$
  $normalVector \leftarrow \text{argmin}_{axe \in axes}\, Norm(axe)$
  if $angleBetween(normalVector, cameraVector) < \pi/2$ **then**
    $normalVector \leftarrow -normalVector$
  end if
  if $Norm(normalVector) > 0.5 \cdot cellSize$ **then**
    return Obstacle ▷ NDT is not flat
  end if
  $verticalVector \leftarrow [0, 0, -1]$;
  if $angleBetween(normalVector, verticalVector) > \pi/6$ **then**
    return Obstacle ▷ Surface is not horizontal
  end if
  return Traversable
end function

Algorithm 2 Received measurement handling

function HANDLERGBDMEASUREMENT(meas)
  if $timestamp(meas) - timestamp(prevMeas) > 10\ s$ or
    $distance(cameraCoords(meas), cameraCoords(prevMeas) > 0.1\ m$
    or $angleBetween(rotation(meas), rotation(prevMeas)$
    $> 1\ rad$ **then**
    $insertMeasurement(meas)$
    $prevMeas \leftarrow meas$
  end if
end function

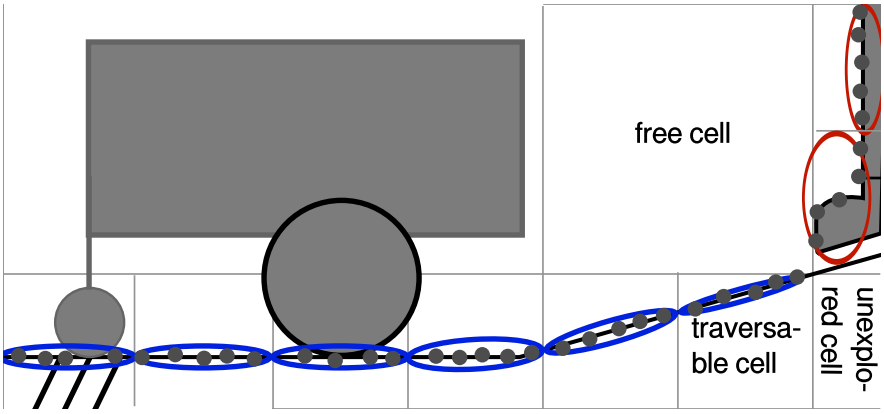

Figure 4.3: The map structure overview. The blue ellipses are traversable cells, the red ones are obstacles. Notice the effect of the octree cell merging. (We use 2D visualization for clarity.)

In the first part of the rgb-d measurement update, the measured points are divided into the cells to which they belong (see Algorithm 3, line 5). If more points than a certain threshold fell into a certain cell (line 18), we recompute its 3D-NDT and update the type of that cell (line 24) as described in Algorithm 1. We only consider points received in the latest rgb-d measurement containing this cell to ensure, that no duplicate points/surfaces caused by a localization error will be present, because they might corrupt the traversability estimation.

Except the type, we store two other values in each cell, that are needed for the loop closure handling in Section 4.3.1. The cells contain the sequence number of the measurement, based on which they were classified (line 23), except the *unexplored* cells, which have this value empty. Also, for each measurement, the system remembers how many cells were updated based on that measurement. When a cell is updated during handling of the measurement with sequence number i, we increment the $influenced\_cells_i$ and if the cell was previously not *Unexplored* and was updated based on measurement j, we decrement the $influenced\_cells_j$ (line 24).

The second part of the update marks the view area between the camera and the measured points as free space. We perform *raytracing*, drawing a ray from the camera to each observed point, and mark the cells crossing the ray as free – with the following three exceptions. The cells that are within the camera minimal range, which is about 0.4 m, are not marked as free, because we have no information about them. The cells at the edge of the camera image are also not marked as free, because the measured information is only partial. And finally the cells containing some measured points are not marked as empty, even though some raytracing line does end in those cells. This process is shown in Algorithm 3 on lines 8 and 27, and visualized in Figure 4.4.

These two parts of the map update cause the observed part of the map to be up-to-date. The first part adds new points, traversable surface and obstacles, while the second part removes old points, such as removed obstacles, from the map.

Algorithm 3 Inserting rgb-d measurement into the map

```
1: function INSERTMEASUREMENT(meas)
2:     tmpPoints ← emptyMap()                          ▷ Mapping from cell to point list
3:     seeThroughCells ← ∅
4:     cellsAtImageBorder ← ∅
5:     for all point ∈ points(meas) do                          ▷ Sort points into cells
6:         cell ← getCellAtCoords(point)      ▷ Returns cells at lowest tree level
7:         tmpPoints[cell].append(point)
8:         raytracedCells ← cellsIntersectingLineBetween(
             cameraLocation(meas), point)                ▷ Returns cells at any level
9:         seeThroughCells.extend({cell ∈ raytracedCells|
             distance(cell, cameraLocation(meas)) > 0.4 m}
10:        if pointFromRgbdImageBorder(point) then     ▷ Ignore border cells
11:            cellsAtImageBorder.append(cell)
12:        end if
13:    end for
14:    for all (cell, points) ∈ keyValuePairs(tmpPoints) do      ▷ Update cells
15:        if cell ∈ cellsAtImageBorder then
16:            continue
17:        end if
18:        if Length(points) > 5 then                          ▷ Ensure good precision
19:            Type(cell) ← classifyCell(cell, points, cameraLocation(meas))
20:            if measurementSeqNumber(cell)! = Null then
21:                influencedCells(measurementSeqNumber(cell)) − −
22:            end if
23:            measurementSeqNumber(cell) ← SeqNumber(meas)
24:            influencedCells(meas) + +
25:        end if
26:    end for
27:    for all cell ∈ seeThroughCells do                       ▷ Update free cells
28:        if cell ∈ cellsAtImageBorder or cell ∈ tmpPoints then
29:            continue
30:        end if
31:        Type(cell) ← classifyCell(cell, ∅, cameraLocation(meas))
32:        if measurementSeqNumber(cell)! = Null then
33:            influencedCells(measurementSeqNumber(cell)) − −
34:        end if
35:        measurementSeqNumber(cell) ← SeqNumber(meas)
36:        influencedCells(meas) + +
37:    end for
38: end function
```

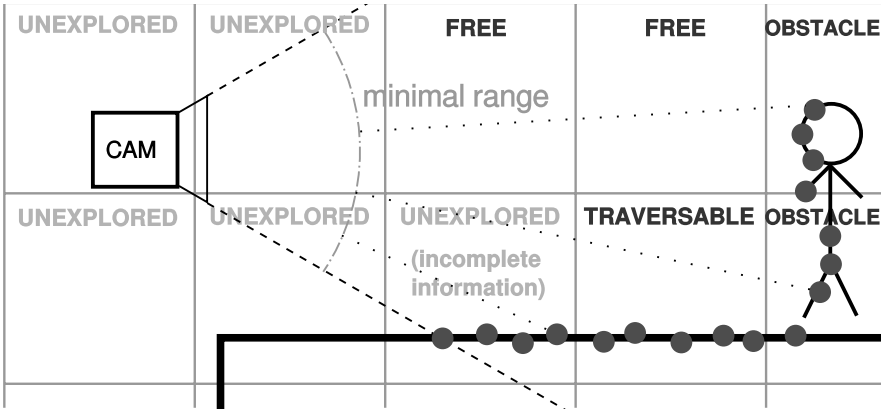

Figure 4.4: The raytracing algorithm

4.3.1 Loop closure handling

The second type of map update is the loop closure, which you can see in Figure 4.5. Loop closure is a discovery of an error in the map detected by the SLAM system. It happens when the localization provides imprecise location for some time, during which we inserted obstacles into wrong places of the map. The localization failure can be caused by odometry error when mapping unknown areas or when the robot doesn't recognize already known area.

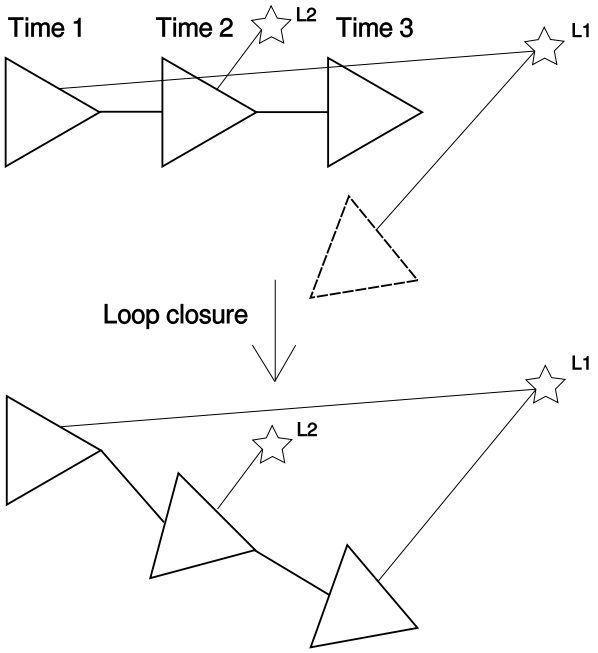

Figure 4.5: A general case of the loop closure. The robot didn't measure its pose correctly between *Time 1* and *Time 3*. When the pose was corrected after measuring *Landmark 1* again, we have to correct the previous robot poses and the measured *Landmark 2*

When a loop closure is detected, the SLAM system changes the location estimate of some rgb-d measurements and we have to *rebuild* our map accordingly to fix errors, such as obstacles inserted to wrong parts of the map. That means reinsertion of all rgb-d measurements starting with the first one with wrong lo-

cation estimate until the last measurement. That can be very time consuming, because the upper bound of computational complexity is linear with respect to the number of measurements. The loop closures in indoor environments (where we can use our rgb-d sensor) are however usually small and contain just a few latest measurements, so the loop closure handling doesn't in practice get linearly slower during time.

We have implemented several improvements to handle loop closures efficiently. The system receives loop closure measurement as a set of previously measured robot poses old_i with their new pose estimations new_i, where i is their sequence number (ordered by time). We discovered that the used SLAM system sometimes reports very small loop closures, where the changes of robot's pose coordinates were less than 0.1 mm. Those loop closures are probably a bug in the SLAM system, don't affect the overall map precision and shouldn't be handled, as it would damage the performance significantly. Therefore, if the pose error $distance(old_i, new_i)$ and 3D angular error between old_i and new_i are both less than specified thresholds, the pose change is ignored.

Otherwise, we will iterate over all the leaf cells of the map and if they were measured during or after the measurement with sequence number i, delete their data by marking them as *Unexplored*. When doing this, we use the fact that a map cell always contains data just from a single measurement as reasoned in Section 4.3.

When the incorrect data are deleted, we have to add the corrected points back into the map. Note that this implies that we have to remember all the point clouds received from the SLAM system. The system inserts the point-clouds corresponding to the deleted measurements one by one using the algorithm described in Section 4.3. During this insertion, we can skip the old measurements, that contain almost no interesting data, because the same cells were measured later. The later measurement would overwrite the old measurements anyway. This is implemented by thresholding $influenced\_cells_i$, which is described in Section 4.3.

5. Path planning

In this chapter, we will describe the path planning problem, the desired properties of a path and the algorithms used for the planning, collision checking and traversability checking. After that, we will also describe how the plan is executed on the robot.

We would like the planning algorithm to find a path between the robot position and a given goal coordinate in the map structure described in Section 4. The resulting path is a list of subsequently adjacent map cells and has to fulfill several conditions to ensure safety of the path and reliability of the system: it has to be traversable, obstacle free and mostly explored, as we will describe later in this chapter.

In addition to those conditions, we optimize the path with respect to our *cost* metric, which expresses distance and accounts for possible problems along the path such as the number of nearby obstacles, unexplored cells, and the angle of slopes. We minimize this amount of possible problems to lower the probability of discovering a problem, that would lead to replanning.

We do not limit the shape of the path for any particular robot movement constraints (such as for robots unable to turn in place) and we don't consider the robot's orientation. That is appropriate for example for robots with differential drive (vehicles with two independent wheels and two engines), but not for more complicated drives like the Ackermann steering (vehicles with one steering and one fixed axle, such as cars).

For solving this problem and planning the path, we currently use the well-known A* algorithm [2]. A* is a graph path-finding algorithm, that was created as an extension of the Dijkstra's algorithm. It is fast and finds paths, that are optimal with respect to a given cost metric.

The planning algorithm traverses the *traversable* and *unexplored* cells of the map (see Section 4.2), while preferring cells in the direction of goal using the cost metric. The map system distinguishes between free and unexplored areas, so the plan can traverse unexplored areas, while preferring the traversable ones. That is needed because maps often contain small unexplored "gaps" caused for example by occlusions. The pseudocode of our planning algorithm is shown in Algorithm 4.

5.1 Traversing unexplored areas

Ability to navigate in unexplored area is sometimes problematic, because that area might not be traversable and statistically, most of the unexplored area in the map isn't. The algorithm therefore naturally prefers to use explored, traversable areas. Also, we limit the maximal length of path in unexplored areas during the search (see Algorithm 4 line 16) and return partial plans (not ending in the goal location), if the goal is deep in the unexplored area. The path to a goal through an unexplored area leads straight to the goal without any turns caused by obstacles, so searching through it is not needed and would only hurt performance. The path through unexplored cells would be expensive according to the cost metric, so the system would try to find a shorter alternative path and continue to traverse the

explored part of the map.

In cases when the algorithm doesn't find a path to the goal, most often because of the unexplored area limit from the previous paragraph, planning returns a *partial path* (see Algorithm 4 line 27). This path ends in the cell closest to the goal, that was discovered during planning. Thanks to partial paths, the robot will get closer to the goal, explore some new areas and may discover a path leading to the goal in future planning iterations instead of failing to find a path at all.

5.2 A* implementation specifics

To show the interesting parts of our planning algorithm, we describe it in the context of the A* algorithm in this section. While general A* traverses a graph and use its nodes as states, this implementation use the octree map structure and its cells for the traversal and path description:

$$State = Cell \in Leafs(Octree)$$

The state can be directly converted to a metric $location \in \mathbb{R}^3$ of the cell center relative to the beginning of the map.

The state space is composed of a set of cells in the octree:

$$StateSpace = Leafs(Octree)$$

As the shortest path is often not the best-suited for navigation, our A* implementation utilizes the cost function $C(Cell1, Cell2)$, where $Cell2 \in Neighbourhood(Cell1)$, that expresses the suitability of a path segment for navigation. The function definition follows:

$$\begin{aligned} C(Cell1, Cell2) = {} & distance(Cell1, Cell2) \cdot obstacleCoeff(Cell2) \\ & \cdot slopeCoeff(Cell1, Cell2) \cdot cellTypeCoeff(Cell2) \end{aligned}$$

$$distance(Cell1, Cell2) = euclideanDistance(center(Cell1), center(Cell2))$$

$$cellTypeCoeff(Cell) = \begin{cases} 1 + U & \text{if } Type(Cell) = Unexplored \\ 1 & \text{if } Type(Cell) = Traversable \end{cases}$$

$$\begin{aligned} slopeCoeff(Cell1, Cell2) &= 1 + V \cdot verticalDistance(Cell1, Cell2) \\ obstacleCoeff(Cell) &= 1 + O \cdot obstaclesInRadius(Cell, RobotModelRadius) \\ verticalDistance(Cell1, Cell2) &= abs(ndtAltitude(Cell1) - ndtAltitude(Cell2)) \\ ndtAltitude(Cell) &= zCoord(ndtMeanValue(Cell)) \end{aligned}$$

where $O, U, V \in \mathbb{R}^+$ are penalty coefficients for movement near obstacles, in unexplored space and vertical movement. The $obstaclesInRadius$ values are returned by the `safetyCheck` function of Algorithm 5 on line 20. The total path cost C_p is the sum of cost functions over the individual path segments:

$$C_p(Path) = \sum_{i=1}^{n} C(Cell_{i-1}, Cell_i)$$

Heuristic function $H(Cell)$ estimates a lower bound for the cost function of a given cell. It is implemented simply as a metric distance between the cell and the goal cell:

$$H(Cell) = distance(Cell, GoalCell)$$

Because the coefficients of C were chosen to be always bigger or equal to 1, the following conditions hold:

$$C(Cell1, Cell2) \geq distance(Cell1, Cell2) \tag{5.1}$$

$$C_p(Path) \geq distance(Cell_1, Cell_n) \quad \forall Path = (Cell_1, \ldots, Cell_n) \tag{5.2}$$

$$H(Cell_1) \leq C_p(Path) \quad \forall Path = (Cell_1, \ldots, GoalCell) \tag{5.3}$$

$$H(Cell) \leq C(Cell, Neighbour) + H(Neighbour) \quad \forall Neighbour \in Neighbourhood(Cell) \tag{5.4}$$

The Statement 5.3 shows that the heuristic is admissible, i.e. it never overestimates the cost of reaching the goal. This is important for the search algorithm to find an optimal path with minimal cost. Moreover, $H(Cell)$ is also consistent by the definition in the Statement 5.4 which is important for the performance of the search algorithm. Without consistency, the cell cost of the individual cells might change, when they are found by another path, which might require some cells to be traversed several times.

The pseudocode of our A* implementation can be found in Algorithm 4.

5.3 Collision and traversability checking

The system checks traversability and possible collisions of the robot centered on a particular cell for all cells traversed by the planning algorithm. We call those two checks together a *safety* check.

The safety check can fail because of an obstacle cell, that is present within a certain distance from the checked cell, or by an absence of traversable surface somewhere within that area. The distance depends on the dimensions of the robot and in this implementation, we use a simple spherical model of a robot for safety checking. Three examples of safety check are shown in Figure 5.1.

In the collision check, we ignore obstacles that are under some traversable surface, e.g. points incorrectly measured bellow floor. This is checked in the *bellowSurface* function (Algorithm 5 line 24), where we traverse the map in an upward direction and check if any cell of that single path is traversable. Note that if any of the cells has more neighbours above itself, we select only one of them to speed up the check. Therefore, if the checked cell is partially bellow some surface, the check will return true with probability roughly proportional to the fraction of traversable surface above the cell. This check is similar to probabilistic or sampling-based methods and together with obstacle thresholding described later in this section leads to a good compromise between performance and accuracy.

During the checks, we also consider the 3D-NDT stored in the cells to increase precision of the system. A traversable cell can satisfy the check only if its surfaces altitude, the z coordinate of the 3D-NDT mean value, is close enough to

Algorithm 4 Path finding

```
1: function A*(start, goal)
2:     cameFrom ← emptyMap()
3:     pQueue ← priorityQueue(< 0, start >)  ▷ pQueue is sorted by the lower
4:     cost ← map(< start, 0 >)                ▷ bound of the total path cost =
5:     closestToGoal ← start                   ▷ C_p(Start, ..., Cell) + H(Cell)
6:     while pQueue ≠ ∅ do
7:         current ← popLowest(pQueue)
8:         if current = goal then
9:             return goal, cameFrom  ▷ Reversed path is stored in cameFrom
10:        end if
11:        for all neighbour ∈ neighbours(current) do
12:            newCost ← cost[current] + C(current, neighbour)
13:            if Type(neighbour) == Empty or
                   newCost >= cost[neighbour] or
                   safetyCheck(neighbour) == Fail  then
14:                continue
15:            end if
16:            if unexploredDistInARow() > MaxUnexploredDist then
17:                continue          ▷ Limit length of unexplored path in a row
18:            end if
19:            if H(neighbour) < H(closestToGoal) then
20:                closestToGoal ← neighbour
21:            end if
22:            cost[neighbour] ← newCost
23:            cameFrom[neighbour] ← current
24:            pQueue.add(< newCost + H(neighbour), neighbour >)
25:        end for
26:    end while
27:    return closestToGoal, cameFrom  ▷ Path was not found, only partial.
28: end function
```

that of the safety-checked traversable cell, as described in Algorithm 5 on line 8. Therefore, the system can distinguish between a small traversable asperity and a bigger stair. This check is visualized in Figure 5.1, part A.

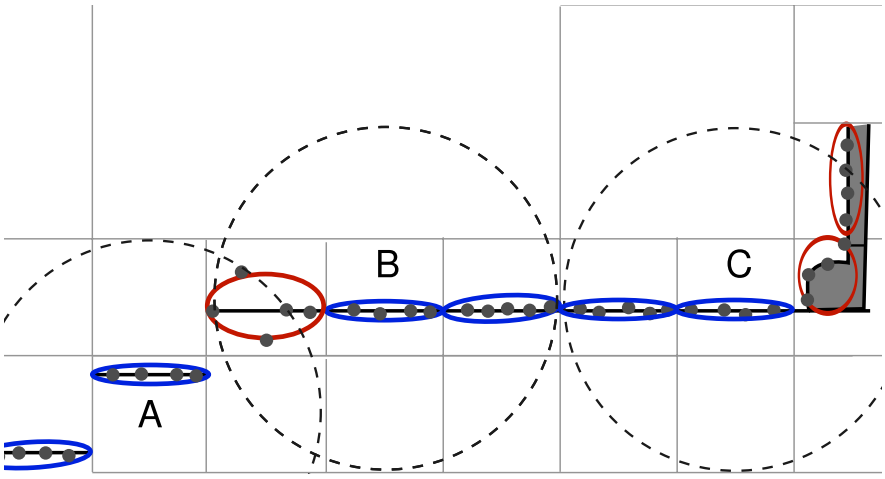

Figure 5.1: Three cases of safety check. In A, the traversability is not satisfied, as the traversable cell to the left is too low and the cell to the right is not traversable. In B, the cell is considered safe, as there is only one solitary obstacle. In C, the collision check will fail because of two obstacles.

For increased robustness against errors in the map, we threshold the number of obstacles and missing traversable area, so an error in one cell can't trigger the safety check to fail. The system requires at least two obstacle cells or at least two missing traversable surfaces for reporting the checked cell as unsafe. See the Algorithm 5, line 17, and the example in Figure 5.1. That is extremely important during the post-planning checks after map updates, that detect new obstacles along the path. Without this thresholding, one incorrect cell measurement in that check would require recomputing the whole plan.

5.4 Plan execution

We have implemented only the global path-planning and use the ROS Navigation [7] `base_local_planner` for local planning and robot control. This local planner receives the robot location and our global plan and outputs linear and angular velocity command standardized in ROS. Its algorithm generates several possible velocity commands, predicts the trajectory of the robot after 1.5 s of executing such velocity command and chooses the velocity command after which the robot gets closest to the next vertex of the global path. Illustration of local planner mechanism is in Figure 5.2.

During the execution of the plan, we check the upcoming part of the plan for traversability and collisions and trigger replanning if a problem is detected. After each new measurement inserted into the map, we iterate over the path cells and perform the safety check described in Section 5.3 on all cells, that are between 0.4 m to 3 m away from the robot and in the direction of the goal. That is the range, where the rgb-d camera can measure with good accuracy. If any cell is

Algorithm 5 Safety checking

```
1: function SAFETYCHECK(cell)
2:     obstacles ← 0
3:     nontraversable ← 0
4:     for all neighbour ← getCellsInRadius(cell, robotDiameter) do
5:         if Type(neighbour) = Obstacle and
            !bellowSurface(neighbour) then
6:             obstacles ← obstacles + 1          ▷ Colliding obstacle cell found.
7:         end if
8:         if Type(neighbour) = Traversable and
            Type(cell) = Traversable and
            abs(ndtAltitude(cell) − ndtAltitude(neighbour)) > T then
9:             nontraversable ← nontraversable + 1      ▷ Surface unreachable.
10:                                                ▷ T is a threshold constant
11:        end if
12:        if Type(neighbour) = Free and
            !bellowSurface(neighbour) and
            !aboveSurface(neighbour) then
13:            nontraversable ← nontraversable + 1
14:                                  ▷ No surface here, the robot could fall.
15:        end if
16:    end for
17:    if obstacles > 1 or nontraversable > 1 then
18:        return Fail
19:    else
20:        return Ok, obstacles, nontraversable
21:    end if
22: end function
23:
24: function BELLOWSURFACE(cell)
25:    next ← oneOf(getNeighbours(cell, Upper))   ▷ Explained in Section 5.3
26:    while distance(cell, next) ≤ robotHeight do
27:        if Type(next) = Traversable then
28:            return True
29:        end if
30:        next ← oneOf(getNeighbours(next, Upper))
31:    end while
32:    return False
33: end function                          ▷ Function aboveSurface is similar.
```

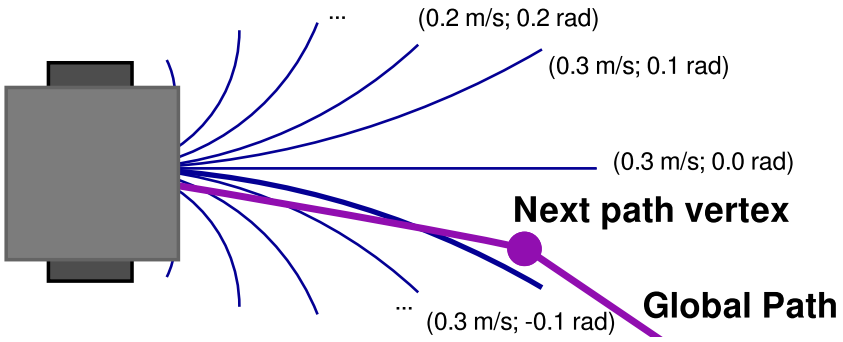

Figure 5.2: The sampling local planner. The selected path is shown as a thick blue line. The corresponding linear and angular velocity command $(0.3; -0.1)$ will be sent for execution.

considered unsafe, we stop the robot and plan the path again. The pseudocode of this check is shown in Algorithm 6.

Algorithm 6 On-update path safety check

```
1:  function checkPath(path,robotLocation)
2:      n ← len(path)
3:      i ← argmin_i distance(robotLocation, path_i)
4:      for all cell ∈ {path_i, ..., path_n} do
5:          if 0.4 m < distance(robotLocation, cell) < 3 m then
6:              if safetyCheck(cell) == Fail then
7:                  return Fail
8:              end if
9:          end if
10:     end for
11:     return Ok
12: end function
```

6. Implementation

We started with the implementation 1.5 year earlier and since then, we revised the navigation design several times based on the results of testing and came to the version we present in this thesis. The results of the implementation evaluation are presented in Chapter 7.

A key decision, which affects the implementation in many aspects, is the choice to use the ROS (Robotic Operating System) as an abstraction layer. That allows us to use ROS SLAM tools, simplifies sharing of the navigation system with robotic community and enables the system to work on different robotic platforms. We also use Octomap [4], Eigen [1] and PCL [9] libraries and Rtabmap [6] SLAM system. Detailed environment requirements are specified in Section 6.1.

The system receives rgb-d images and odometry messages from ROS. It also listens for 3D coordinates of the goal location. When an unreached goal is set, our navigation outputs `geometry_msgs/Twist` message with linear and angular velocity. That message can be then directly used by standard ROS robot controllers. The implementation is also able to visualize the processed map and the planned path, Figure 6.1 contains an example with description.

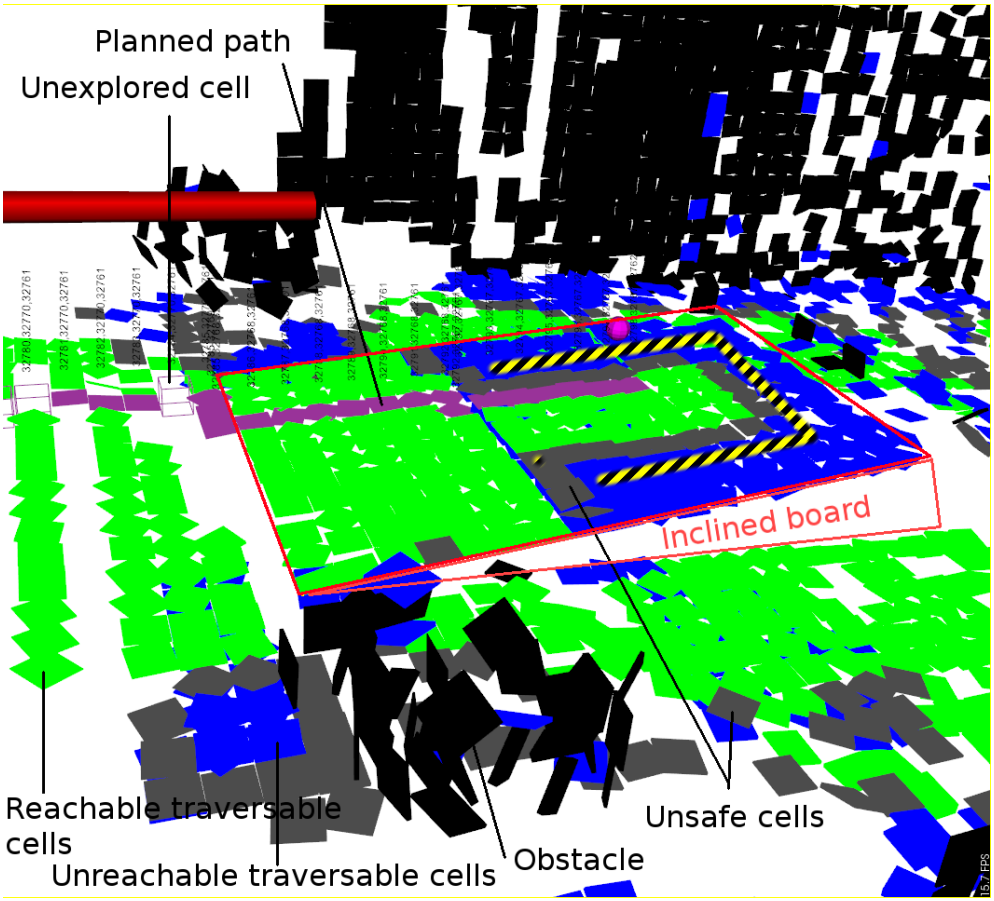

Figure 6.1: Planning on an inclined board. The robot is standing to the left of the image and the goal is denoted by the purple dot. *Unsafe* cells are the reachable cells, where the safety checking failed. The edges of the board (highlighted by stripe pattern) were detected, so the robot won't fall from it.

The following two sections of this chapter describe first the usage of the system for developers, that would want to include our navigation in their systems, and second the internals of the implementation for developers, which would want to extend or modify our system. Note that the primary goal of the implementation was to test and evaluate the proposed method of navigation, it is not ready for use in real-world scenarios nor in industry. It's interface is therefore very simple and targeted mainly on other researchers.

6.1 Usage

In this section, we would like to describe how to use our navigation system. In order to run our system on your robot, you will need the robot to match our hardware and software requirements.

As for hardware, the robot needs to be equipped with the *Microsoft Kinect* rgb-d camera. It is recommended to use differential steering, which can be realized for example by two powered wheels and a caster wheel. We also recommend 3D odometry, which can be realized by an IMU (Inertial Measurement Unit), as it enables the third-party SLAM system to provide better precision.

The robot software must be set up for use with ROS; we recommend using the *Indigo* version to ensure compatibility with our system. The set up involves providing standardized interfaces for wheel control and odometry. If you don't have such a robot at hand, you can obtain some of the commercially available robotic platforms compatible with ROS. As ROS is a very extensive system, which is widely used among the robotic community, we will not describe how to use it in this work. Both maintainers and the community provide many sources and tutorials covering its installation or the setup of your robot in detail.

Except ROS, you will need the following software packages used by our system. You can obtain compatible versions of all the following packages through ROS in version *Indigo* using the `rosdep` tool.

- *Octomap* – Octree library used by the mapping subsystem [4].
- *PCL* – Point-cloud library used for receiving the measurements [9].
- *Rtabmap* – RGB-D SLAM system [6].
- *Eigen* – A linear algebra library [1].

To tune the execution of the plan on the robot, you have to supply several `rosparam` parameters to the standard ROS plan executor `base_local_planner`, which we mentioned in Section 5.4. The most important are the velocity limits, which describe the maximal speeds, at which the robot should move, and the minimal speeds, at which the robots motors are able to move. You can find an example configuration in the `params.yaml` file on the attached CD.

For compilation of the system, we use *CMake* system, which is well suited both for ROS and C++. The supplied `CMakeLists.txt` describing the build process internally uses the ROS `catkin` tool for locating the necessary requirements in your system and building the resulting ROS package.

Once your environment is set up, you can directly launch the `grid_assembler` node. Its interface is very simple, the executable doesn't require any command line arguments and outputs extensive logs.

Even though the system is not designed to be used directly by a human operator, we implemented a debugging interface window, that can be used for setting goals and displaying the map structure. As redrawing the map in real-time would be computationally expensive, the system updates the map on demand of the user after pressing the `D` (draw) key. The position of the robot and the current plan are updated in real-time. You can input a goal for the navigation by moving mouse while holding the `Ctrl` key. The goal is represented by a green point and can be moved only in a horizontal plane. You can confirm the goal and start the navigation by pressing `Enter`. After the planning, you have to confirm the path by pressing `Space`; this confirmation is required after each replanning too. The visualization and goal selection is shown in Figure 6.2.

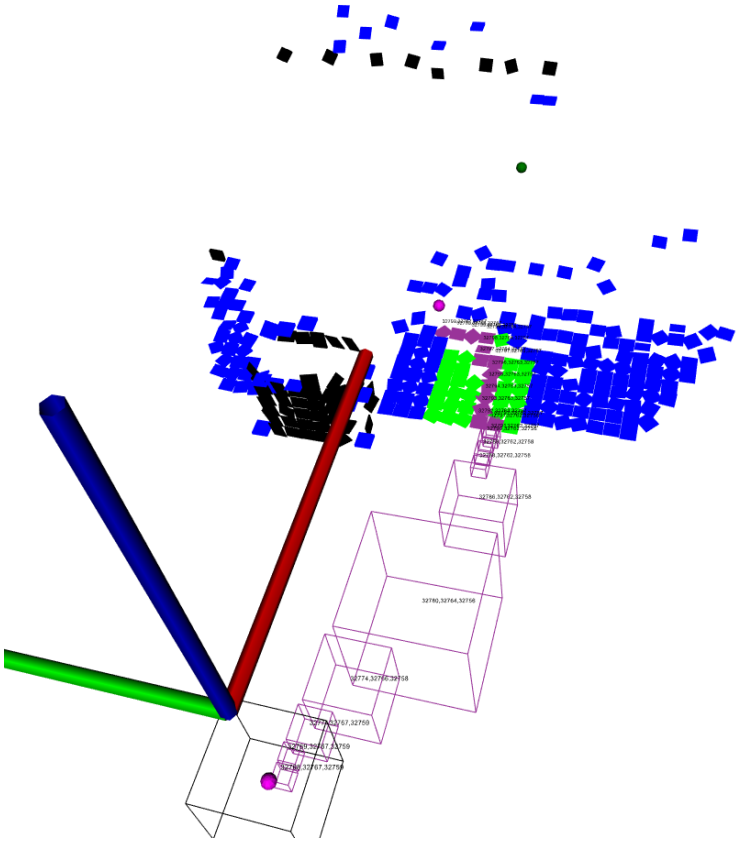

Figure 6.2: Testing user interface. The purple dots mark the start and the goal of the current plan. The black cube is the actual robot pose.

The planning can also be controlled from your ROS node or from the command line using the `rostopic` tool as in the following example:

```
rostopic pub /move_base_simple/goal geometry_msgs/PoseStamped \
        '{pose: {position: {x: 1.5, y: 2, z: 0}}}'
```

6.2 Development

We implemented the navigation system in C++11 using several development tools. For compilation of the system, we use the *CMake* system integrated with the ROS *catkin* build tool, that is able to locate the library requirements in your system and ROS installation. You can build the source code using the supplied *CMakeLists.txt* file or using the `catkin_make` tool, that reads this file internally. More details on using those build tools can be found in the ROS documentation on `http://wiki.ros.org/catkin/Tutorials`.

We also use *Doxygen* tool to generate html code documentation from the in-code comments. Those comments, which can be interpreted and shown by most modern development environments, help the programmer orient in the code and the generated documentation can serve as a reference guide (see Figure 6.3).

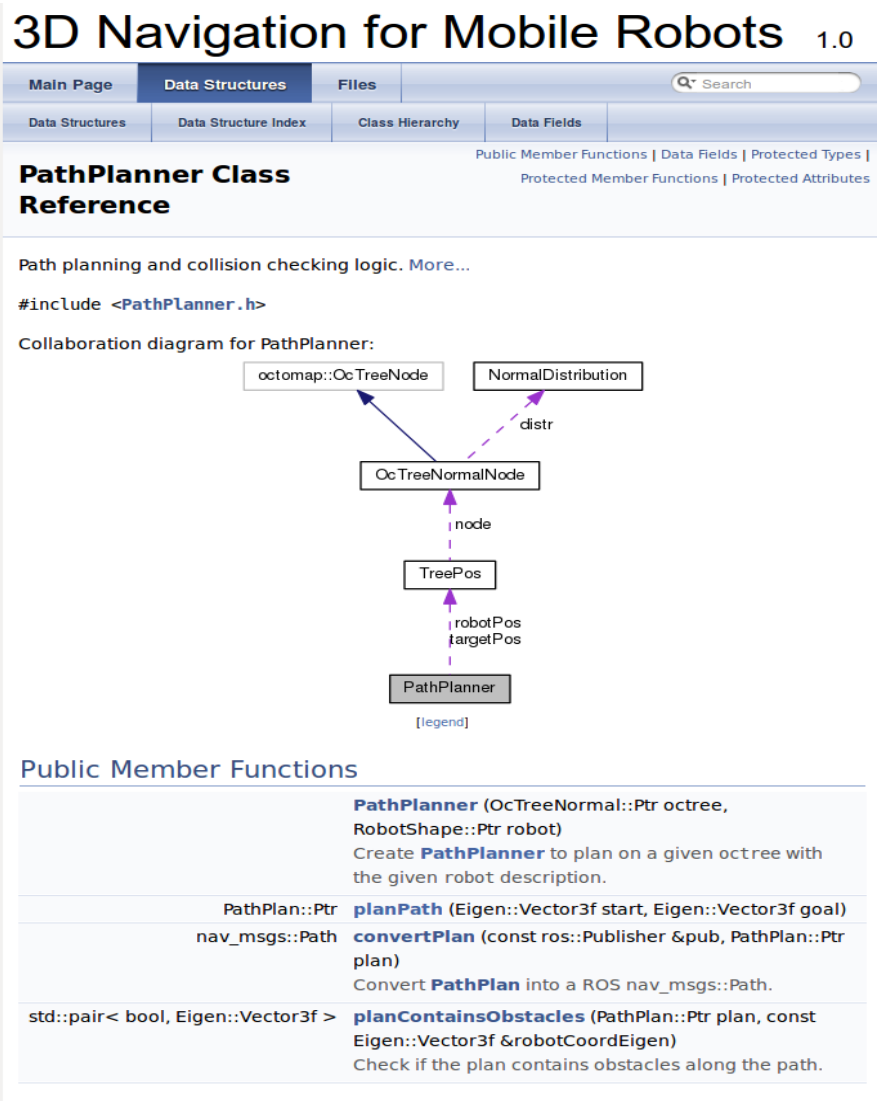

Figure 6.3: An insight into the documentation, that is available on the attached CD.

For structuring the code, we use mostly object oriented approach with a few mostly functional exceptions. We also paid attention to adhere to the good practices introduced with C++11 such as smart pointers and lock guards, that

help prevent problems common in C++ such as memory leaks or deadlocks. We also debugged the implementation in *gdb* and analyzed the memory usage in *valgrind* to assure good stability and performance.

The code is organized into several classes and files:

- *Benchmark* – We use the Benchmark class to record the duration of our algorithms and evaluate their performance. You can see the performance report when you stop the running system by pressing `Ctrl-C`.
- *MapAssembler* – MapAssembler processes data from the SLAM system and updates the data in the octree structure. In order to use another SLAM system, this is the only class that has to be changed.
- *grid_assembler_node* – This is the entry point of the system, which calls the RunNavigation described bellow.
- *NormalDistribution* – This class encapsulates the 3D NDT functionality. It is used to estimates mean and covariance matrix from a set of samples. Its implementation uses Eigen linear-algebra library.
- *OcTreeNormal* – OcTreeNormal is representing the map structure composing of an octree.
- *OcTreeNormalNode* – This class serves as a single cell of the octree structure.
- *OcTreeViz* – OcTreeViz implements the visualization of the map.
- *PathPlan* – PathPlan class represents the resulting path.
- *PathPlanner* – PathPlanner contains all the planning and safety checking logic.
- *RobotShape* – This class and its descendant SphericalRobot represent the shape of the used robot and possibly robot-specific collision checking procedure, which can be easily extended for the readers intentions.
- *RunNavigation* – RunNavigation contains the high-level navigational logic (such as when to replan).

7. Evaluation

To evaluate the results of our implementation, we have designed several experiments and observed the overall robustness and performance of the system. We paid attention especially to the reliability of our system and the special situations, when the navigation fails. Regarding the resulting paths, you can judge thir quality the attached figures. Moreover, we recorded a few short videos of the experiments, you can watch them on `http://janskoda.cz/navigation/`.

In this chapter, we will present the experiment design and several scenarios, which show the abilities of our system. We included scenarios similar to the examples provided in the motivation in Section 1.1 to show, that the system is able to work in 3D environments and therefore provides advantage over the common 2D navigation systems. At this time, we are not aware of any publicly available navigation system, that would provide similar functionality in 3D space. In the scenarios, we will also mention cases, when the system is unable to work correctly, and discuss the reasons and possible solutions.

In all the experiments, we ask the system to navigate to given coordinates relative to the robots starting position. The system has no prior knowledge of the environment nor the map and plans the path after one rgb-d measurement. As our robot unfortunately doesn't provide 3D odometry but only 2D wheel odometry, we use visual odometry tracked by the SLAM system.

The experiments are structured into the following sections, each one contains a description of the environment, a hypothesis or desired behaviour and the actual behaviour. If the system didn't work as desired, we discuss the reasons, and, in some cases, propose a possible way to fix the failure.

7.1 Basic scenario

In the first experiment, the robot had to navigate through a 80 *cm* wide corridor between several obstacles and turn behind one of the obstacles to reach the goal. One of the obstacles is an elevated desk, that might be hard to recognize, and that creates an occlusion behind the desk and near the goal location. The goal is in the observed area 2.5 *m* away from the robot (when not considering the obstacles).

The robot reached the goal successfully, which together with the following figures demonstrate the most basic functions of the implementation such as creating a correct map and planning a path avoiding static obstacles. The environment, plan, map and resulting path are drawn in Figures 7.1, 7.2 and 7.3.

Figure 7.1: The environment of the experiment.

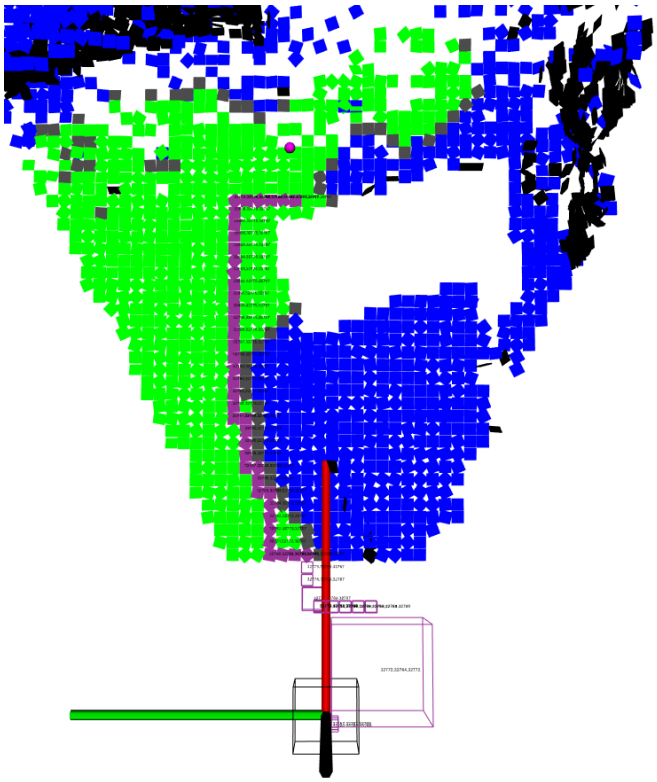

Figure 7.2: The initial map and the plan to the desired goal. The blue area in the center is an inclined wooden plane that serves as an obstacle and creates an occlusion to the left.

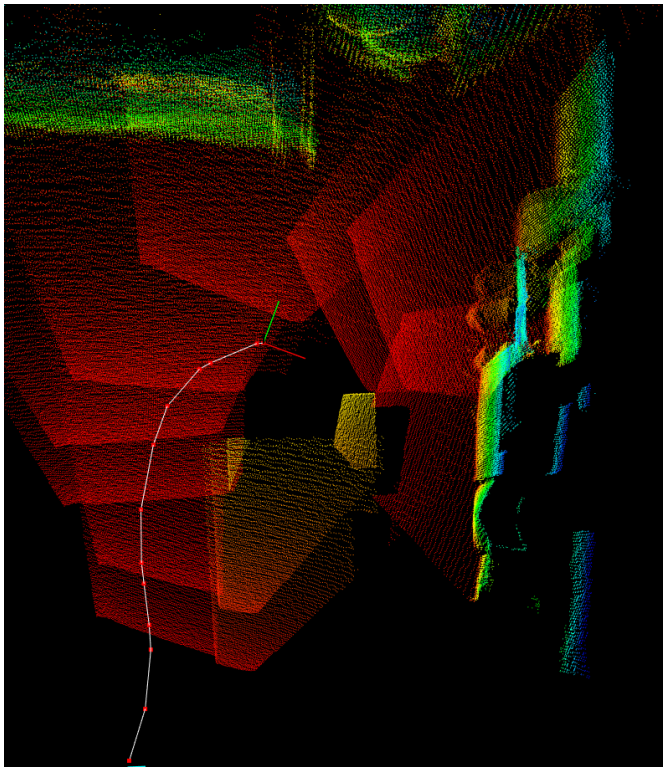

Figure 7.3: The map source point cloud and the resulting path as recorded by the SLAM system. The color spectrum represents the z coordinate of the point. The occluded area now in black still remains, as it was never observed from sufficient distance.

7.2 Elevated desk

In the second experiment, the robot had to navigate to a 50 *cm* wide elevated desk, recognize its edges and climb the slope. We had to add two artificial obstacles to ensure, that the robot wont try to climb the desk in its edge, which happened during the previous testing. The failure scenario is shown in Section 7.3

This experiment showed, that the robot is able to model the environment in a 3D map and use the additional 3D information for navigation and traversability checking.

The environment, plan, map and resulting path are drawn in Figures 7.4, 7.5 and 7.6.

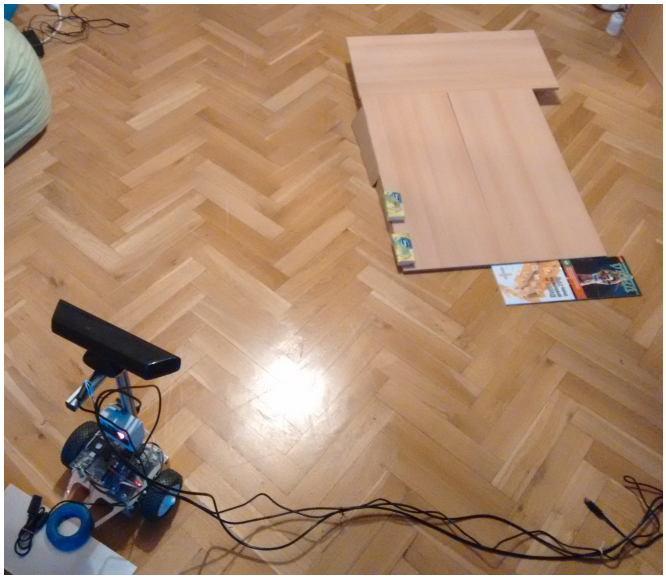

Figure 7.4: The environment of the experiment

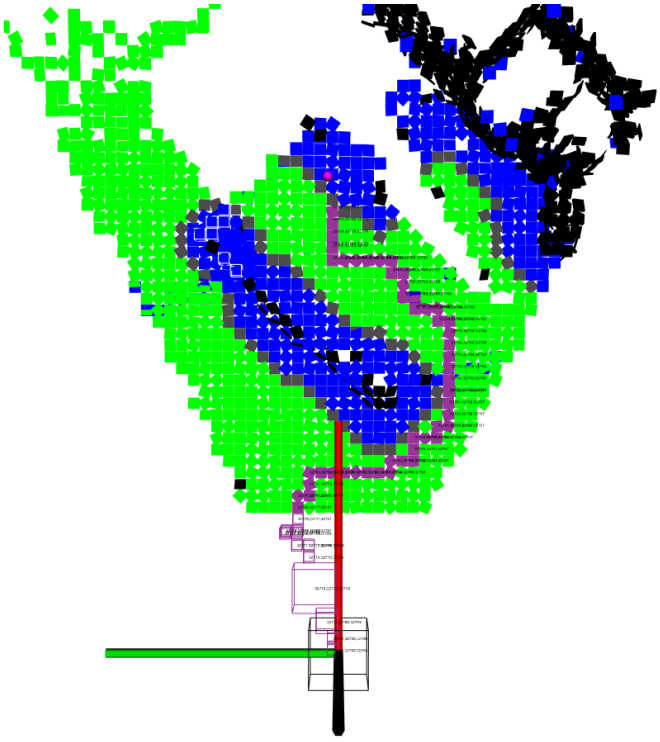

Figure 7.5: The initial map and the plan to the desired goal. Notice the few solitary obstacles, that are not considered as collisions by the planning.

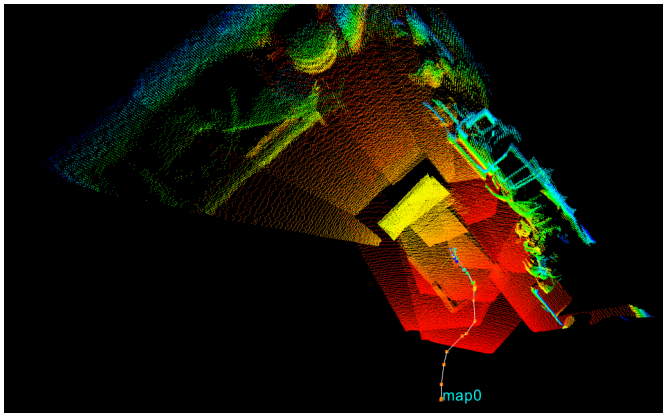

Figure 7.6: The map source point cloud and the resulting path as recorded by the SLAM system. The color spectrum represents the z coordinate of a point.

7.3 Elevated desk fail

We conducted the above experiment (see Figure 7.4) with the desk in another variant without the two artificial obstacles. In this case, we expected that the system might try to climb the desk from the edge and remain stuck there.

When following the planned path, the robot didn't keep sufficient distance from the desk edge, climbed it from its left side with only one wheel and was unable to drive further.

This happens because the system can't recognize the small difference between a traversable and not traversable stair. If we tuned the collision detection to be more aggressive and limit the traversability check further, many falsely positive detected collisions would occur because of the limited SLAM system precision.

The plan is visualized in Figure 7.7 bellow.

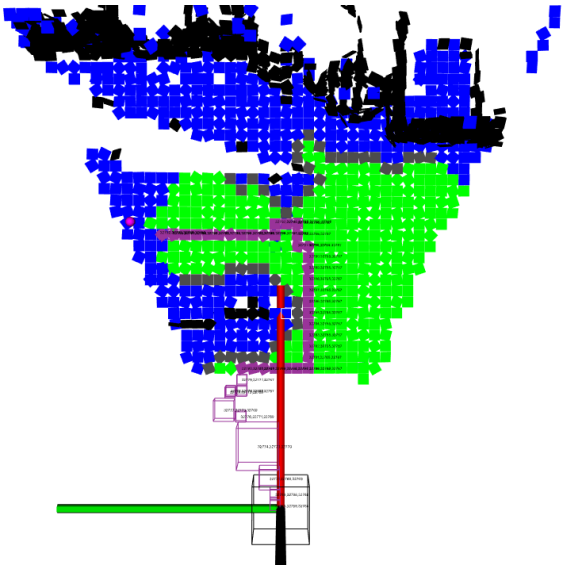

Figure 7.7: The initial map and the plan to the desired goal. Notice the few solitary obstacles, that are not considered as collisions by the planning.

7.4 Glass doors

To evaluate the abilities of the chosen sensor, the *Kinect* rgb-d camera, we included a scenario with glass and sun shining against the sensor. As you can see from the SLAM output in Figure 7.9, the rgb-d sensor doesn't recognize the glass as an obstacle, but sees through it much like regular rgb cameras. This limits the usage of our system with this sensor for example in buildings with wall-like barriers made of glass and other transparent materials. On the other hand, the sensor didn't seem to be irradiated by the infrared radiance of the sun and the rgb-d measurement remained precise.

Figure 7.8: The environment of the experiment.

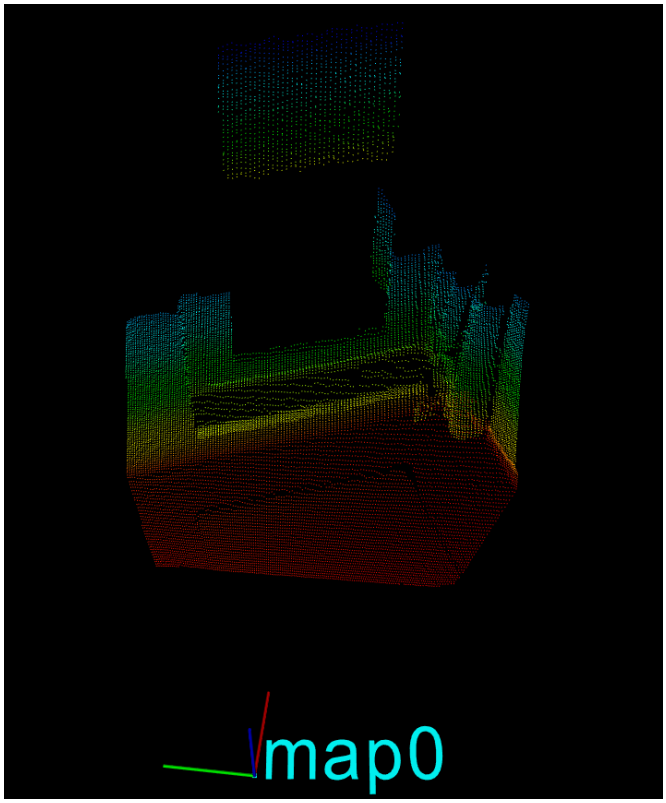

Figure 7.9: The data received from the SLAM system.

7.5 Unexplored area

In this experiment, the robot had to navigate to a goal, that was outside of the area of sight of the sensor and therefore outside of the explored part of the map. The planning system found a *partial plan* (see Section 5.1), which doesn't end in the goal location and started navigating (Figure 7.10). During the plan execution, the system replanned the path and finally reached the original goal. The final map and the path of the robot can be seen in Figure 7.11 and 7.12.

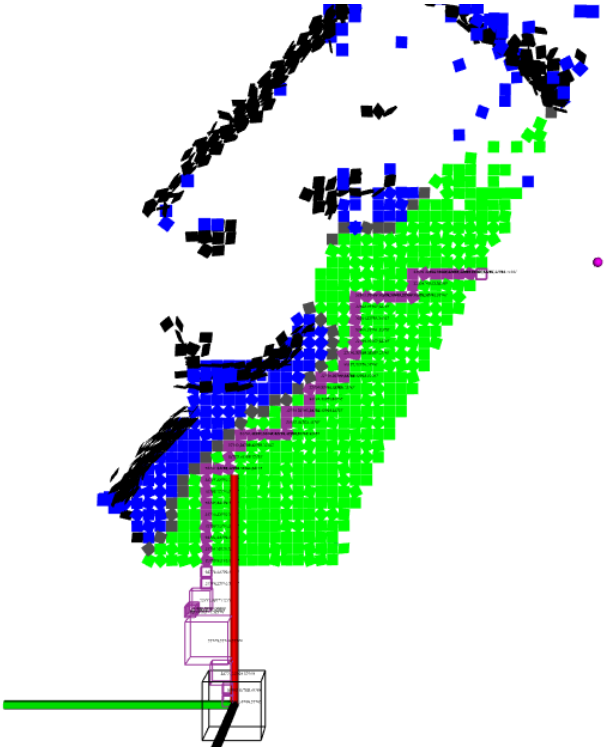

Figure 7.10: The initial map and the plan to the temporary goal.

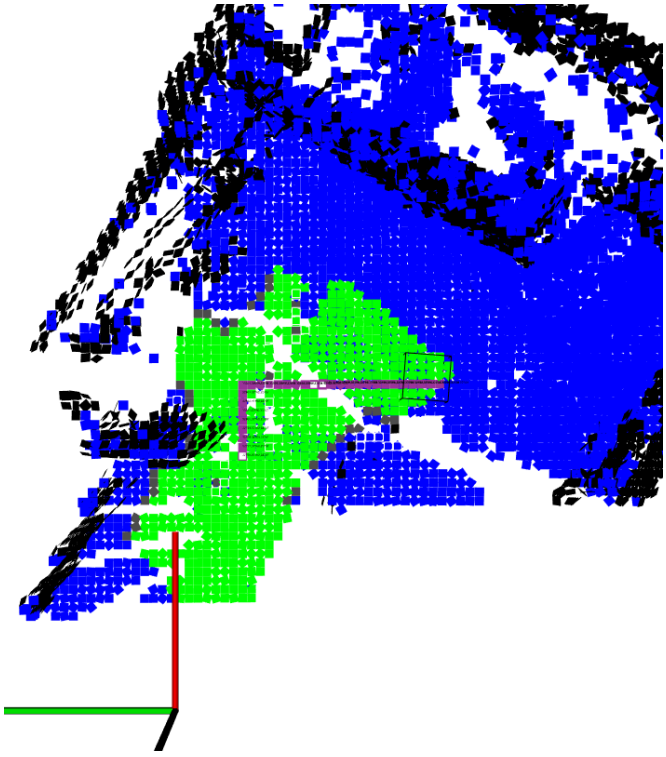

Figure 7.11: The final map and the replanned path to the desired goal.

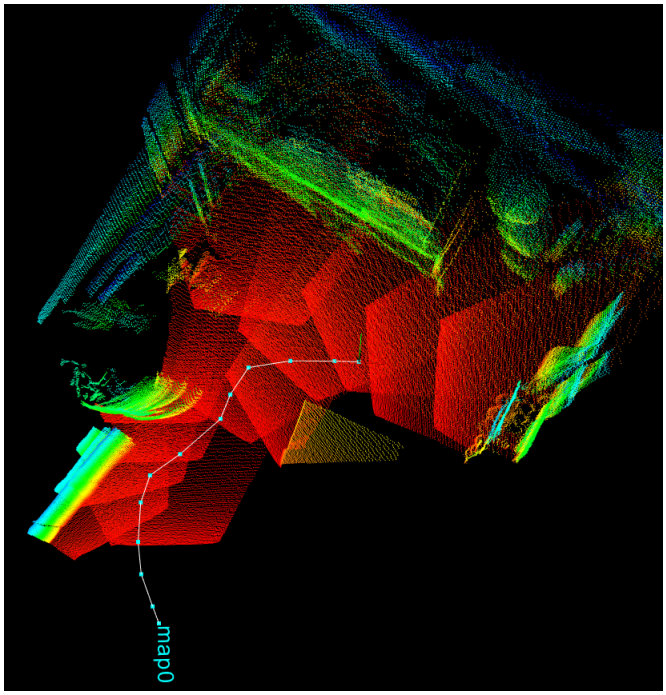

Figure 7.12: The map source point cloud and the resulting path as recorded by the SLAM system. The color spectrum represents the z coordinate of a point.

7.6 Bridge traversal

In the sixth experiment, the robot had to navigate into a part of environment, that was only accessible by a wooden "bridge". Other paths were blocked. The system was able plan such path and after replanning, the robot arrived to the goal location.

Although the navigation system worked correctly, the robot had difficulties following the path. It couldn't climb the edge of the desk because of motor controllers were unable to increase the motor torque. Also, the robot had troubles with following the straight planned path at the slope and oscillated around the path far to the sides. Those issues were however caused by the hardware platform and not the navigation system.

The environment photo, final map and the robots path can be seen in Figure 7.13, 7.14 and 7.15.

Figure 7.13: The environment of the experiment

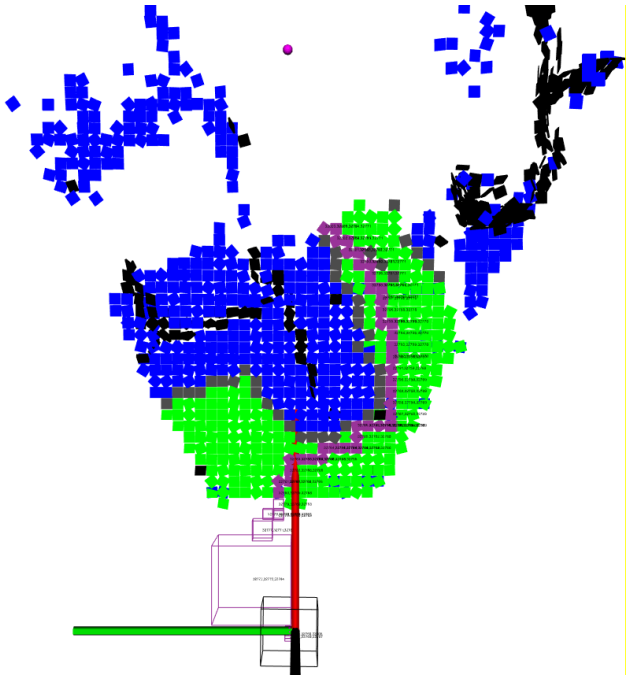

Figure 7.14: The initial map and the plan to the temporary goal.

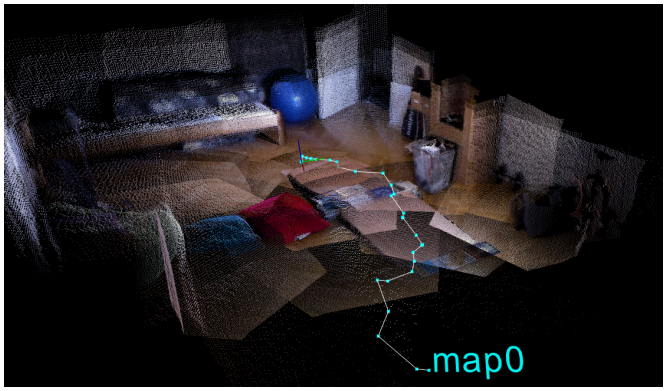

Figure 7.15: The map source point cloud and the resulting path as recorded by the SLAM system.

7.7 Longer operation

In the seventh experiment, the robot had to shuttle between its initial position and a 2 m distant location several times. It should show, that the system is able to navigate continuously for longer time periods and gradually grow the map without corrupting it even when revisiting already mapped locations. The planning was invoked 4 times and there were no obstacles between the two locations.

The environment robots path can be seen in Figure 7.16.

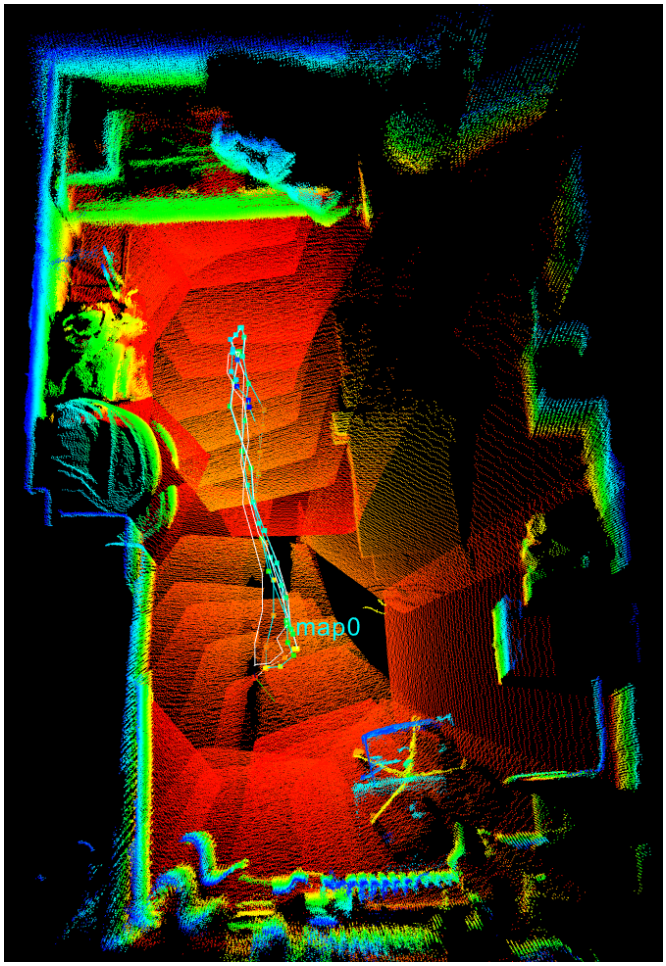

Figure 7.16: The map source point cloud and the resulting path as recorded by the SLAM system. The color spectrum represents the z coordinate of a point.

7.8 Obstacle added during execution

In this experiment, the robot had to navigate to a goal 2 m away in front of the robot. During the execution, we added a new obstacle on the planned path. We expected the robot to stop before the obstacle, replan the path and drive around the obstacle.

The system as expected and the robot reached the goal successfully. The environment, plan, map and resulting path are drawn in Figures 7.17, 7.18 and 7.19.

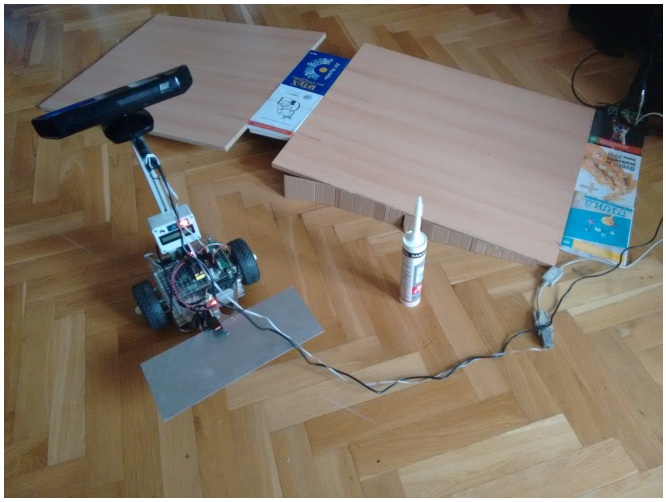

Figure 7.17: The environment of the experiment.

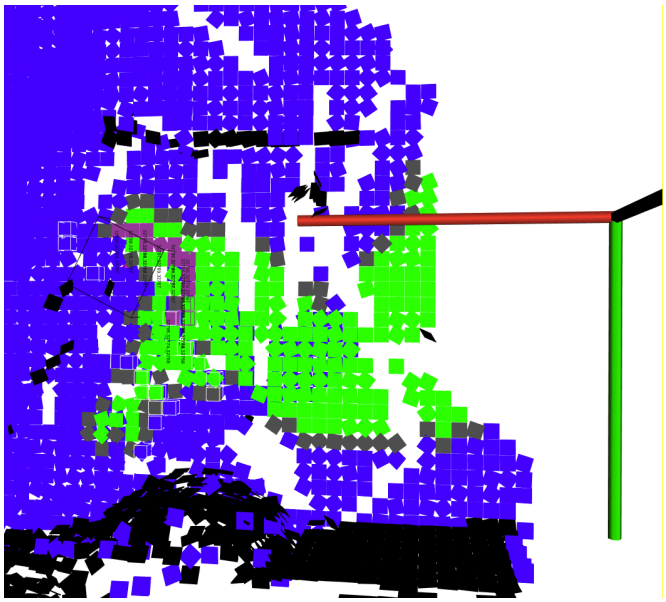

Figure 7.18: The initial map and the plan to the desired goal. The black cells forming a line to the top represent the edge of the desk. The group of black cells near the end of the red axis is the new obstacle.

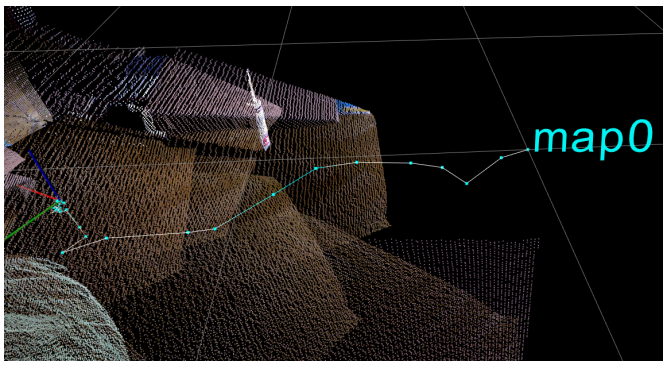

Figure 7.19: The map source point cloud and the resulting path as recorded by the SLAM system. The color spectrum represents the z coordinate of the point.

7.9 Runtime performance

During the experiments, that included around 30 planning iterations in different conditions, we collected several performance metrics from the system to evaluate the runtime performance of the system. This benchmark was performed on a Intel i7 3.0 Ghz equipped dual-core notebook. The following results are average values rounded to one decimal place. As the environment in all the scenarios has similar size, most values didn't differ much between the individual experiments; exceptions will be pointed out.

The initial planning took 2 s if the path was found, the system replanned the path several times during the navigation with mean planning time 1.5 s. If the goal was unreachable, the maximal planning time reached 4.3 s. 94.3 % of the planning runtime was spent in collision checking procedure, which traverses the nearby cells. Inserting measurements into the map took 0.9 s per measurement. The SLAM system detected small loop closures in some of the experiments and its correction took 2.7 s. Bigger loop closures can take significantly more time, as the computational complexity of loop closure handling is linear with respect to the loop size.

The performance of the system has effects on the behaviour of the robot, because the robot has to stop when replanning or correcting a loop closure. The map update is fast enough, so the robot doesn't have to stop to process it, the map update latency however means, that the robot is able to see obstacles after a second after they entered the camera view. As the sensor is able to see obstacles with sufficient precision at about 3 m distance from the robot, the map update latency limits the maximal driving speed to about 3 m/s, which we consider reasonable for most use cases.

The biggest limitation is probably the planning and replanning speed, which we shown to be mostly determined by the collision checking performance. We discuss the possible improvement in the Future work section 8.1.

8. Conclusion

We have proposed a 3D navigation system for ground vehicles, that uses an rgb-d camera and a localization and mapping system as an input and is able to navigate a robot in various indoor environments. Our system processes the rgb-d measurements into a map optimized for path planning, performs collision and traversability checking, and finds paths according to our safety and optimality criteria.

The design of our system has several advantages compared to other state-of-the-art navigations. It is well suited for uneven terrain which can even have multiple levels. 3D navigation is also safer than standard navigation systems in many situations, because it can avoid surface edges and holes in the ground, which are invisible for most available navigations. The novelty of the system lies in our map structure and map update procedures, which are able to model the environment accurately, yet efficiently with respect to map update and path planning speed. Also, we paid a lot of attention to dealing with the problems caused by imprecise localization and rgb-d input of the broadly available cameras.

On the other hand, our design results in several disadvantages as well. The used rgb-d camera can't sense any obstacles closer than approximately 0.4 m and has relatively narrow field of view, which together limits the ability of the system to avoid obstacles. Also, the computational complexity and especially the latency of the rgb-d stream processing both by the camera driver and our system slows down the reaction speed of the system. It therefore takes about 1 s to react to new or moving obstacles.

We have developed our navigation not to be dependent on any particular robotic platform or sensor, and tested it on two mobile robots of the Faculty of Mathematics and Physics in Prague and the University of Tsukuba in Japan. We also implemented the standardized interfaces of the widely-used Robotic Operating System, so that other researchers familiar with the ROS can test our solution on their robotic platforms without having to adjust the code and adapt it to work with their robots.

The results of the implementation were evaluated during a series of experiments where a robot had to navigate in environments containing a single navigational problem. In the evaluation, we have shown that the implementation of the system works as we designed it, but can also fail in some special cases. We have therefore described those problematic cases and environments and discussed why the failures happen and what would be required to fix them.

8.1 Future work

Although we have spent a lot of time with implementation and testing of our navigation system, there are several aspects, where both the design and the code could be extended and improved.

When the safety checking procedure inspects two adjacent cells, it doesn't reuse any information and traverses the whole neighbourhood of both cells, even if they share a big intersection. The procedure could therefore remember the found unsafe cells during the traversal and traverse only the new parts of the

neighbourhood, which might lead to a significant planning speedup.

As the localization precision affects the precision of the navigation system greatly, it might be beneficial to test the performance with more precise odometry. An IMU (Inertial Measurement Unit) sensor providing acceleration and orientation of a robot could be used to track the movement of the robot in 3D. Together with the third-party rgb-d SLAM system, this would lead to much more precise localization.

The paths produced by our system are optimal (with respect to our cost metric) on a grid, but this often doesn't mean they are optimal in continuous space, as is shown in Figure 8.1. A path planning better suited for robot navigation such as Theta* [8] or A* with smoothing could be implemented to improve the paths. Although both algorithms don't guarantee optimality, they can generally find smoother and shorter paths with less turns than A*.

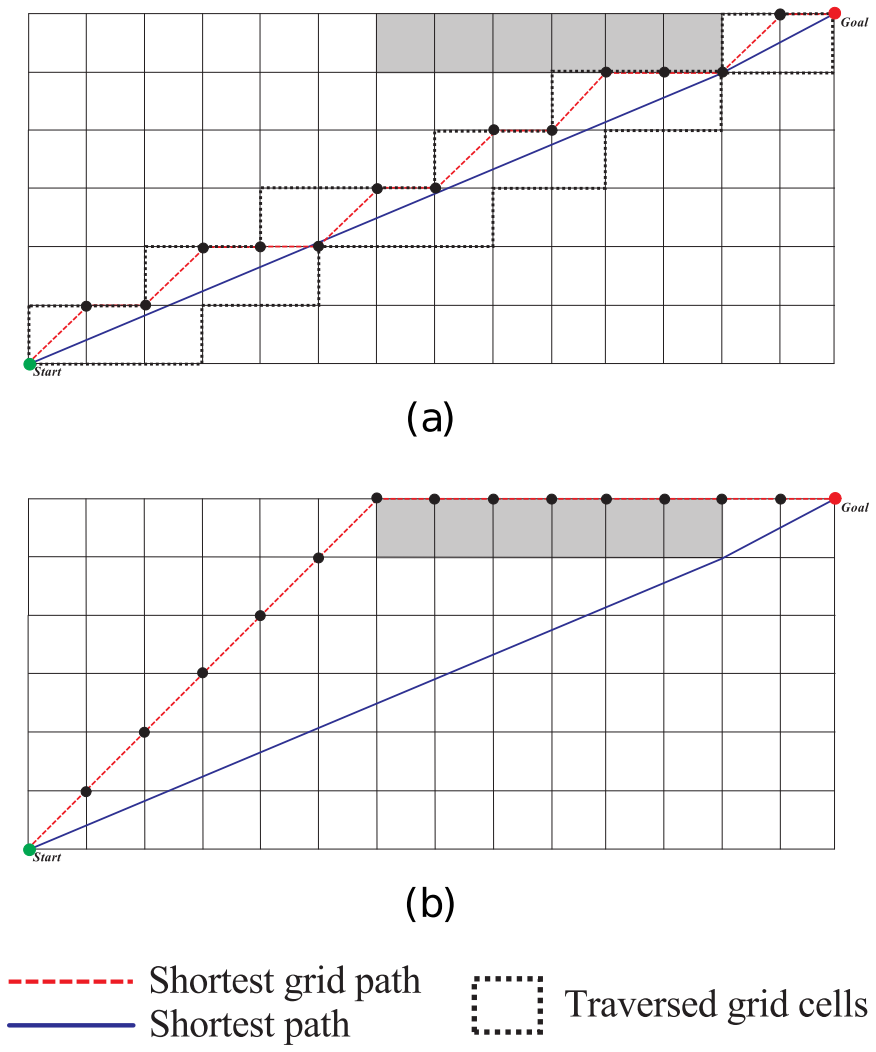

Figure 8.1: Planning on a grid doesn't yield optimal paths in continuous space. The blue path can be found by Theta* [8].

Bibliography

[1] Eigen linear algebra library. `http://eigen.tuxfamily.org/index.php?title=Main_Page`. Accessed: 2016-12-26.

[2] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[3] Armin Hornung, Mike Phillips, E Gil Jones, Maren Bennewitz, Maxim Likhachev, and Sachin Chitta. Navigation in three-dimensional cluttered environments for mobile manipulation. In *Robotics and Automation (ICRA), 2012 IEEE International Conference*, pages 423–429. IEEE, 2012.

[4] Armin Hornung, Kai M Wurm, Maren Bennewitz, Cyrill Stachniss, and Wolfram Burgard. Octomap: An efficient probabilistic 3d mapping framework based on octrees. *Autonomous Robots*, 34(3):189–206, 2013.

[5] Richard Arnold Johnson, Dean W Wichern, et al. *Applied multivariate statistical analysis*, volume 5. Prentice hall Upper Saddle River, NJ, 2002.

[6] Mathieu Labbé and François Michaud. Online global loop closure detection for large-scale multi-session graph-based slam. In *Intelligent Robots and Systems (IROS), 2014 IEEE/RSJ International Conference*, pages 2661–2666. IEEE, 2014.

[7] Eitan Marder-Eppstein, Eric Berger, Tully Foote, Brian Gerkey, and Kurt Konolige. The office marathon: Robust navigation in an indoor office environment. In *Robotics and Automation (ICRA), 2010 IEEE International Conference*, 2010.

[8] Alex Nash and Sven Koenig. Any-angle path planning. *AI Magazine*, 34(4):85–107, 2013.

[9] Radu Bogdan Rusu and Steve Cousins. 3d is here: Point cloud library (pcl). In *Robotics and Automation (ICRA), 2011 IEEE International Conference*, pages 1–4. IEEE, 2011.

[10] Todor Stoyanov, Martin Magnusson, Håkan Almqvist, and Achim J Lilienthal. On the accuracy of the 3d normal distributions transform as a tool for spatial representation. In *Robotics and Automation (ICRA), 2011 IEEE International Conference*, pages 4080–4085. IEEE, 2011.

[11] Todor Stoyanov, Martin Magnusson, Henrik Andreasson, and Achim J Lilienthal. Path planning in 3d environments using the normal distributions transform. In *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference*, pages 3263–3268. IEEE, 2010.

[12] Shengdong Xu, Dominik Honegger, Marc Pollefeys, and Lionel Heng. Real-time 3d navigation for autonomous vision-guided mavs. In *Intelligent Robots and Systems (IROS), 2015 IEEE/RSJ International Conference*, pages 53–59. IEEE, 2015.

List of Abbreviations

- 3D-NDT – Three dimensional normal distribution transform
- GPS – Global Positioning System
- IMU – Inertial measurement unit. Sensor integrating gyro and accelerometer.
- RGB-D – An RGB image enhanced with depth information of the pixels
- ROS – Robotic Operating System
- SLAM – Simultaneous Localization and Mapping
- UAV – Unmanned aerial vehicle

Attachments

A CD referred to as "Attachment 1" is attached to this work. It contains:

- ebook version of this thesis in PDF format
- source code of the implementation
- Doxygen generated html documentation of the implementation