3D Mapping of indoor environments with a time of flight camera

Rodrigo Baravalle, Amaury Nègre

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

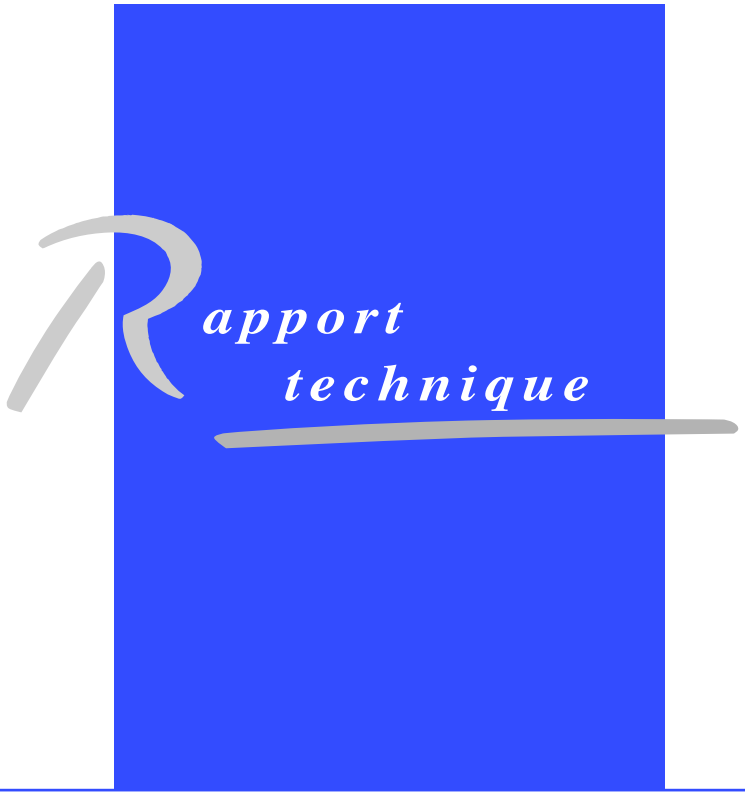
3D Mapping of indoor environments with a time of flight camera

Rodrigo Baravalle — Amaury Nègre

N° 0406

February 2010

Domaine 1



ISSN 0249-0803 ISRN INRIA/RT--0406--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
GRENOBLE - RHÔNE-ALPES

3D Mapping of indoor environments with a time of flight camera

Rodrigo Baravalle, Amaury Nègre

Domaine : Mathématiques appliquées, calcul et simulation
Équipe-Projet e-motion

Rapport technique n° 0406 — February 2010 — 12 pages

Abstract: In this work we make a 3D representation of an indoor environment surrounding a robot. So we assume an static environment. We used a Swissranger SR4000 camera to get 3D point clouds of the environment, and ROS for implementing the software. With these information and the position of the robot, we are able to construct a 3D map in real time.

Key-words: 3d mapping, slam, octree

Centre de recherche INRIA Grenoble – Rhône-Alpes
655, avenue de l'Europe, 38334 Montbonnot Saint Ismier
Téléphone : +33 4 76 61 52 00 — Télécopie +33 4 76 61 52 52

3D Mapping of indoor environments with a time of flight camera

Résumé : Dans cet document, nous présentons nos travaux sur la cartographie 3D d'un environnement intérieur statique dans lequel se déplace un robot. Nous utilisons une caméra 3D Swissranger SR4000 pour obtenir un nuage de points 3D de l'environnement. Avec ces mesures plus la position du robot (obtenu avec un SLAM 2D), nous somme capables de reconstruire l'environnement 3D en temps réel.

Mots-clés : reconstruction 3D, octree

Contents

1 Introduction

3D mapping has been receiving great attention over the past years. One of the main reasons for this is the evolution of the hardware and the possibility of managing great amounts of data. 3D maps have applications in robot navigation, manipulation, telepresence, games, etc. The past year the Kinect[1], a controller developed for the Xbox 360 was used for this purpose, as can be seen in [3].

Many 3D mapping algorithms build 3D maps taking into account actual data coming from sensors, without the help of any other information. Here, we propose to use 2D maps of the same environment, in order to use previously calculated information. More specifically, first we build a 2D map of the environment, in which we take into account the problem of loop closure, and then we simply add data coming from the 3D camera. In this way, we could simplify the mapping process.

The aim of this work is to obtain a 3D map of the environment of the robotic wheelchair we have. For that, we have a swissranger SR4000 camera which is mounted on it. This camera can retrieve 3D point clouds of the environment in real time. We implemented the code in the ROS robotic platform. Next sections give a brief description of the hardware and software used for that purpose.

2 Hardware

In this section we explain the main characteristics of the main components of the robotic system. We used a swissranger camera, a MTi xsens sensor and a Blue Botics wheelchair.

2.1 swissranger SR4000

This camera is developed by Mesa Imaging[2]. It works with the Time of Flight (TOF) principle.

Its manual explains in detail the way it works:

"In Time of Flight systems, the time taken for light to travel from an active illumination source to the objects in the field of view and back to the sensor is measured. Given the speed of light c, the distance can be determined directly from this round trip time. To achieve the time of flight measurement the SR4000 modulates its illumination LEDS, and the CCD/CMOS imaging sensor measures the phase of the returned modulated signal at each pixel. The distance at each pixel is determined as a fraction of the one full cycle of the modulated signal, where the distance correponding to one full cycle is given by

$D = \frac{c}{2f}$

where c is the speed of light and f is the modulation frequency. If f is equal to 30MHz, this distance is 5m".

This camera has a resolution of 176x144 pixels, which is good enough to have a detailed representation of objects. It has several modulation frequencies, which can be configured, and each one has a different range of depth measurement. Mainly, two of them are used: 0-5m and 0-10m. When an object is outside this "non-ambiguity range", but can be detected due to its intensity, it is folded into

[1] http://en.wikipedia.org/wiki/Kinect
[2] http://www.mesa-imaging.ch/

the range. For instance, if using a range of 0-10m, and a shiny object is 7 meters away from the camera, it will be folded back, and so it will appear as if it was placed at 2 meters of distance. When the measurement is too noisy, it appears as a cone of points (with low intensity) that begins in the origin of coordinates of the camera.

Because of that, we used the range 0-10m, and we filter points (by hardware) which has not enough intensity, using the parameter "amplitude threshold" of the camera. It is less probable that objects with more than 10m of distance from the camera can be captured enough intensity. Also, the environment in which the wheelchair moves has distances greater than 5 meters.

Apart from the raw point cloud, the camera provides information about confidence and intensity of each measurement. This could be used to filter points which are provably the result of bad measurements.

2.2 MTi xsens

This product is developed by *xsens*[3]. We used it with the idea of having accurate information about the orientation of the camera. The best way to understand the capabilities of the sensor is by its manual: "The MTi is a complete miniature inertial measurement unit with integrated 3D magnetometers (3D compass), with an embedded processor capable of calculating roll, pitch and yaw in real time, as well as outputting calibrated 3D linear acceleration, rate of turn (gyro) and (earth) magnetic field data.". Its main applications are robotics, aerospace, and autonomous vehicles.

With this sensor, we tried to get the 3D orientation of the camera in space and then use these values to transform the data accordingly. Mixing the values of orientation of the camera, retrieved by this sensor, and the position of the robot in the map, we were able to transform point clouds to the map reference.

2.3 Blue Botics

The wheelchair platform we used was developed for the MOVEMENT[4] project. It has one SICK-LMS 200 laser along with wheels and a drive motor to move it. This robot can be operated manually through a joystick (this is the mode we utilized) or automatically through velocity commands.

In order to use the wheelchair, it is necessary to pass the firmware to it through an Ethernet connection. After that, it is possible to communicate with it using a CGI interface.

The platform is capable of retrieving odometry information (which is an estimated position and orientation). This information in turn is used by 2D mapping algorithms to get a position in the 2D map.

3 Software

In this section we talk about *ROS*, the software platform used for implementing the algorithm, and then we present the algorithm of mapping we developed.

[3] http://www.xsens.com/en/general/mti

[4] Modular Versatile Mobility Enhancement Technology, Specific Targeted Research Project nr. 511670

3.1 Robot Operating System (ROS)

ROS is an open source operating system for robots. It provides facilities like message-passing between processes, and hardware abstraction. One of its main characteristics is that it is a distributed system, in which different processes can communicate each other by a standard way of passing messages between them. Each separate process is called a *node*. It exists a central node which is called *roscore*, which is responsible for all the communication.

In a tipical robot system usually exists a lot of nodes, for instance, one node can be responsible of sending the actual position of the robot while other is constructing a map of the environment. It could also be one node which calculates the best path to reach a goal, other that communicates with the hardware of a laser attached to the robot, etc. One important advantage of having a distributed system is that you could have each process running in different processors, or even in different computers; this is particularly interesting for performance.

A collection of nodes is called a *package*. In a package it usually exists a set of nodes, together with libraries, configuration files, datasets, etc. A collection of packages is called a *stack*. Usually a stack denotes some specific subset of a system, like for instance the navigation subsystem.

For now, the only operating system fully supported for using ROS is Ubuntu. It is possible to use C++ or Python to integrate ROS to an existing application. For that, the ROS packages *roscpp* and *rospy* are provided. Simply by including it with an include directive in a .cpp we can start using ROS in our application.

ROS has many libraries for common use when building such a system. Here we will briefly describe the ones which we have used.

3.1.1 Utilities

- *sensor_msgs*[5] a collection of messages used by sensors, like point clouds, or images. These messages are used for communication between different nodes.
- *tf*[6] is a ROS library which gives facilities for using different coordinate systems that a robotic platform has. It provides an interface for knowing where are the different frames over time, and to reason about them. In our case, it is useful for transforming point clouds, which comes from the camera, to the map reference system.
- *pcl* (point cloud library[7]) is a collection of utilities for managing point clouds. Among them, we could find filtering, surface reconstruction, registration, segmentation, etc.

In the next section we will show some bunches of code that show how we used ROS in the implementation.

[5] http://www.ros.org/wiki/sensor_msgs
[6] http://www.ros.org/wiki/tf
[7] http://www.ros.org/wiki/pcl

4 3D Mapping

In this section we will explain the main idea of the algorithm and the way we implemented it over ROS.

4.1 Main Idea

If, for each incoming 3D point cloud, we have the position of the robot in a 2D environment, we could just sum up the different range images coming from the camera to obtain a 3D map of the environment. The advantage of doing this is that we save computational time required to perform a 3D positioning algorithm, but on the other hand we lose precision and also the 3D map depends on the results of the 2D mapping algorithm. Anyway, we found it useful, because our robot can only move in 2D.

4.2 Localization

To obtain the localization of the robot we must find it's position in a map. We compared two different ways of doing this. One of them is to construct the map in 2D and then use the node *amcl* from ROS, which, based on actual laser information, gives us the actual position on that map. Another way is to use the *slam* 2d algorithm that is present in ROS to have the position in a map which is being constructed while the robot is moving.

In the second approach, it is not necessary to have a previously constructed map, but due to actualizations that the algorithm does, the map is sometimes adjusted and so we got a distorted 3D map. On the other hand, amcl requires to set up an initial position in that map. We obtained similar results using both approaches. The principal problem of both occurs when they construct distorted maps, due to inaccuracy on this maps. One solution is to construct the map first, and then use the map only if it does not have inaccuracies, with amcl. We can always construct a new map if the first is not as good as we want.

As a future work, we propose here to implement ICP algorithms in order to match the different point clouds. Doing it in this way will bring us more accurate 3D maps.

4.3 Implementation

First of all, we should construct the transformation between the swissranger camera and the wheelchair. We placed it 1.4 m above the base of the robot and 0.77m back in the x coordinate. So we write:

```
swissranger.transform.translation.x = x_scan_offset_ -0.77;
swissranger.transform.translation.z = 1.4;
```

After that we only have to send the transformation over ROS, and so it will be available for making the necessary transformations of the point clouds coming from the camera to the map we are constructing.

ROS allow us to track different coordinate systems using the *tf* library. In order to use the transformation we created, we have to call this library as follows:

```
sensor_msgs::PointCloud pc_current;
...
```

```
tlistener.lookupTransform("/map", pc_current.header.frame_id,
                          pc_current.header.stamp,
                          bswisstransform);

pcl::transformAsMatrix(bswisstransform, mat);
```

A way to obtain a more accurate transformation is to use the output of an MTi xsens sensor, particularly, we obtained the rotation transformation through the values of the gyroscopes in the three euclidean edges. This values in turn, were used to make a transformation which is sent over ROS. The code results:

```
sensor_msgs::PointCloud pc_current;
...
tlistener.lookupTransform("/map", pc_current.header.frame_id,
                          pc_current.header.stamp,
                          swisstransform);


...
// obtain the values of the gyroscopes
// and save it in stransform

tlistener.lookupTransform("rot_swissranger", "map",
                          pc_current.header.stamp,
                          stransform);
...
// final transformation
transf=tf::Transform(tf::Quaternion(stransform.getRotation()),
                     tf::Vector3(swisstransform.getOrigin()));
```

In this code, *rot_ swissranger* represents the transformation of the gyroscopes with respect to the map, which is what we actually want. So we put together these values with the values of the position (x, y) of the camera in the map.

The results obtained were similar, except in the case when the wheelchair stops suddenly and the camera changes its orientation vertically. But even in this case, the error was acceptable (because the movement is small). We decided so not to use the MTi sensor for the sake of simplicity. Anyway, the code could be changed easily in the future.

Once we have the transformation between the camera and the map, we use the pcl ROS library to pass it to a matrix representation:

```
pcl::transformAsMatrix(swisstransform, mat);
```

After that, we perform two filtering steps in order to have a better and more useful representation of each point cloud. Both of them are provided by pcl. First, an *outliers removal*. In each input, there usually exists points which do not represent real data, and when there exists many inputs, the error could be incremented and difficult to filtrate. They usually appear as isolated points. With this filter we are able to eliminate most of them.

The code is implemented as follows (simplified for better understanding):

```
pcl::StatisticalOutlierRemoval<pcl::PointXYZ> sor;

sor.setInputCloud (point_cloud_in);
sor.setMeanK (50);
```

```
sor.setStddevMulThresh (1.0);
sor.filter(point_cloud_out);
```

In the code, we pass *point_cloud_in* and we get the filtered version in *point_cloud_out*. The second filter is useful for storage and computational resources. Because the map could be huge, we perform a *downsampling* in every input to get a simplified version. There exists a parameter to control the resolution of the final pointcloud.

We pass the result of the previous filtering to this step (transformed to a sensor_msgs::PointCloud2 datatype).

```
pcl::VoxelGrid< sensor_msgs::PointCloud2 > vgrid;

vgrid.setInputCloud (pc_downsampled);
double res = 0.01;
vgrid.setLeafSize (res, res, res);
vgrid.filter(pc_out);
```

In *pc_out* the filtered point cloud. Here the resolution is lowered 10 times. With that resolution the quality of the map remains good and the map could be efficiently used and saved. In either case the resolution could be a parameter of the algorithm, depending on each necessity. The algorithm uses a centroid-based[8] downsampling.

The next step is simply to add the new pointcloud. Because adding point clouds is computational expensive, we made a vector of pointclouds. So when a new pointcloud arrives we calculate the number of points we already summed and when it reaches a threshold, we take a new element in the vector.

```
std::vector<pcl::PointCloud<pcl::PointXYZ> > pcls;

if(!firstPointCloud
   && pcls[pcl_actual].points.size() > threshold) {
  ROS_INFO("Point Cloud reached threshold");
  pcl_actual++;
  firstPointCloud = true;
}

if(firstPointCloud) {
  pcls.resize(pcl_actual+1);
  pcls.at(pcl_actual) = pointcloud_filtered;
  firstPointCloud = false;
}
else {
  pcls.at(pcl_actual) += pointcloud_filtered;
}
```

Finally, we save the map in pcd format to store and visualize it. The pcd format is defined in pcl and it is just an ascii file with the points of the point cloud along with metadata about the points. The library also provides a visualizer (*pcd_viewer*) to see the points saved in that format. We do outliers removal again with the final map to remove some spurious points that results from the merging step.

[8] http://en.wikipedia.org/wiki/Centroid

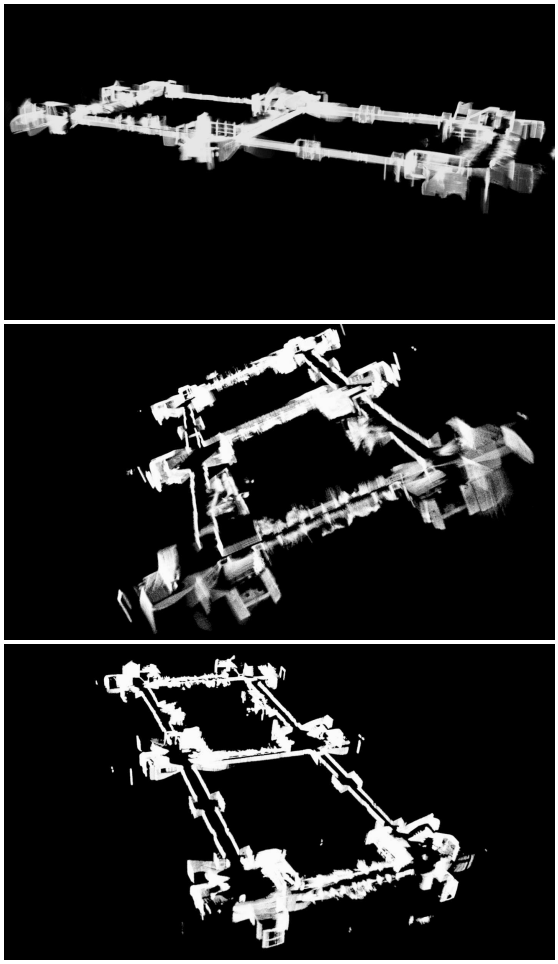


Figure 1: 3D map of the first floor at INRIA Grenoble.

5 Results

The Figure 1 show screenshots with the resulting 3D map of the first floor of INRIA Grenoble. Even doing the mentioned downsampling, the resulting map has great detail of the environment of the robot, as can be seen in Figure 2. Nevertheless, due to limitations of the camera, it has problems with glasses. The figure 3 demonstrates these problems. The basic problem is that because it works with light, when it reaches a glass, light interacts in a different way with the light coming from the camera, and so measures are not accurate.

6 Conclusions

We have made a simple 3D mapping algorithm using an open source software developing platform. Using a 2D based slam localization algorithm we are able to construct a 3D map of the surrounding environment of a moving robot in an unknown environment. The advantages of this approach relies mostly in its

Figure 2: Detailled views of the map.

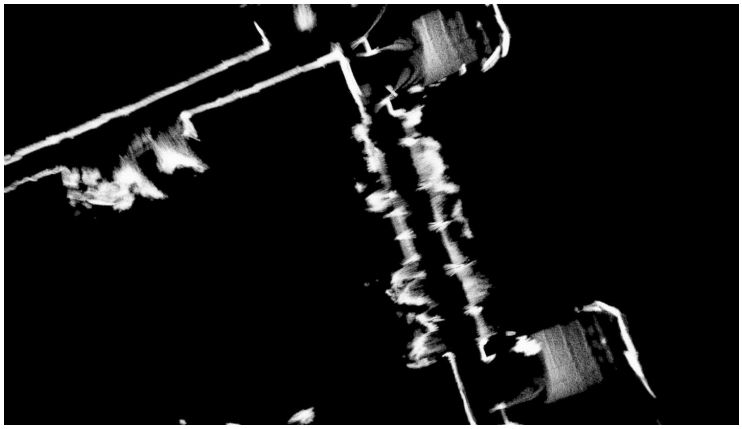


Figure 3: Problem due to the limitation of the camera when detecting glasses.

simplicity and power of representation. Maps obtained have good quality and the algorithm can easily generate them.

The disadvantages of this approach relies in the dependence on the 2D mapping algorithm. This problem could be solved by using a 3D matching algorithm like ICP that could try to assemble them correctly.

7 Future Work

It is possible to continue this work to get a better representation of the environment. One of the most interesting works will be to use some algorithm to join the different point clouds. This will give more independence with the 2D mapping algorithm and less errors.

Another possible future work is to make an online algorithm, for instance using parallel programming and implementing it using CUDA or OpenCL. With that it will be possible to see in real time the construction of the 3D map.

In addition, it will be interesting to implement *people filtering*[2]. So the mapping process will not depend on the presence of people in the environment.

Ir order to help rendering, it will be important to save the map in a different data structure. One of the possibilities is represented by surfaces[1], because they are better for saving and rendering, or even *surfels*, as in [3]. In addition, occupancy grids[4] are a good possibility to make the map more useful.

References

[1] Brian L Curless. New methods for surface reconstruction from range images. Technical report, Stanford, CA, USA, 1997.

[2] Dirk Hähnel, Dirk Schulz, and Wolfram Burgard. Map building with mobile robots in populated environments, 2002.

[3] Peter Henry, Michael Krainin, Evan Herbst, Xiaofeng Ren, and Dieter Fox. Rgb-d mapping: Using depth cameras for dense 3d modeling of indoor environments. *International Symposium on Experimental Robotics*, 2010.

[4] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic Robotics (Intelligent Robotics and Autonomous Agents)*. The MIT Press, 2005.

Centre de recherche INRIA Grenoble – Rhône-Alpes
655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-0803