2012:58

Obstacle detection and mapping for autonomous vehicles

KARL LINUS HÖSSJER

Examensarbete inom
MASKINTEKNIK
Mekatronik
Högskoleingenjör, 15 hp
Södertälje, Sverige 2012

Obstacle detection and mapping for autonomous vehicles

av

Karl Linus Hössjer

Examensarbete TMT 2012:58
KTH Industriell teknik och management
Tillämpad maskinteknik
Mariekällgatan 3, 151 81 Södertälje

KTH Industriell teknik och management

Examensarbete TMT 2012:58

Hinderdetektering och kartläggning för autonom robot

Karl Linus Hössjer

Godkänt 2012-aug-23	Examinator KTH Christer Albinsson	Handledare KTH Christer Albinsson
	Uppdragsgivare MarnaTech AB	Företagskontakt/handledare Peter Reigo

Sammanfattning

Detta projektarbete handlar om styrsystem för autonoma fordon och robotar. Huvudinnehållet har varit ett program för hinderdetektering och kartläggning.

Att hitta rätt typ av sensor har varit mycket viktigt för projektet. I början var det viktigaste att sensorn skulle ha en mycket låg kostnad. Detta ändrades senare och målet blev istället att ha en sensor kapabel av att detektera så mycket som möjligt och med en bra leverantör.

Beslutet blev slutligen att arbeta med Time of Flight kameror. De kan upptäcka det mesta, genomskinliga objekt samt svarta blanka ytor är dock problem. Föremål mindre eller tunnare än sensorns upplösningen kan naturligtvis inte detekteras, detta kan ge problem vid exempelvis hängande smala kablar.

Maskinen använder dess sensor för att bygga en karta över miljön när den rör sig i den. Kartan behövs när den planerar hur den röra sig. Detta är särskilt viktigt för golvtäckningsplannering, när den beräknar hur den ska täcka hela golvytan (för rengöring eller minsvepande).

Programmet startar genom att extrahera golv och väggar från inkommande sensordata. Den återstående informationen i datat är då hinder som antingen ska kartläggas eller bara undvikas.

Hittills har programmet endast körs i simulering eftersom den nödvändiga hårdvaran inte är färdig ännu.

Från simulering har vi sett att programmet kan kartläggning allt som anges i krav- och mål-specifikationen samt dynamiska objekt. Kartan används också av samtliga vägplanneringsfunktioner.

Nyckelord

Sensorer, Kartläggning, Robot, Hinderdetektering

KTH Industrial Engineering and Management

Bachelor of Science Thesis TMT 2012:58

Obstacle detection and mapping for autonomous vehicles

Karl Linus Hössjer

Approved	Examiner KTH	Supervisor KTH
2012-aug-23	Christer Albinsson	Christer Albinsson
	Commissioner	Contact person at company
	MarnaTech AB	Peter Reigo

Abstract

This project is about control systems for autonomous vehicles and robots. The main scope of it has been a program for obstacle detection and mapping.

Finding the right kind of sensor has been very important for the project. At first it was of most importance for the sensor to have a very low cost. The main importance later changed to having a sensor capable of detecting as much as possible and with a good supplier.

It was finally decided to work with Time of Flight cameras. They can detect everything except very transparent items and black shiny surfaces will also give it some problem. These are however problems that all vision based sensors have. Items smaller or thinner than the sensors resolution can of course not be detected.

The machine uses its sensor to build a map of the environment when it is moving in it. The map is required when it is calculating how to move. This is especially crucial for coverage planning, when it calculates how to cover a whole floor (for cleaning or mine sweeping).

The program starts by extracting floor and walls from the incoming sensor data. It can then look at the remaining data and identify different obstacles in it.

So far the program has only been running in simulation because the necessary hardware is not finished yet.

From simulation we have seen that the program is capable of mapping everything specified in requirements and goals and dynamic objects. The map is also used by path planning functions.

Key-words

Sensors, Time of Flight Camera, Mapping, Obstacle detection.

Foreword

This thesis project has been done in the field of robotics. The focus of this project was:

- Finding a sensor solution for our needs
- Using sensors for computer vision
- Making a map with the sensor data
- Calculating movement with help of the map
- Acting (stopping) on sudden data from the sensor, like someone walking in front of it.

It is assumed that the reader has basic knowledge of terms from objective programming such as *objects* and *inherence*, and *namespaces* from C.

The thesis project has been made as part of a bigger project at a Swedish company active in the field of robotics and control systems. The whole goal is to make a prototype of an autonomous machine which can demonstrate the company's positioning system. The company is also interested in control system solutions for customers together with the positioning system so this is an evaluation platform for these systems.

At first the goal was supposed to be a low cost autonomous robot for outdoor applications. We have then raised the cost limit and finally made it for indoor applications. Another change that occurred a few weeks after we decided on indoor application was that we were provided with a map from the start. We had the idea from the beginning to start with a blank map, but the positioning project would in its installation start with building a map which we might as well use. There will still be a theoretical part in this paper on how to identify and make walls from point cloud data.

We have been two persons working on this part of the project, one is me and the other one is Xuan Wang who is a Chinese exchange student studying machine learning. This paper will only discuss the parts that I have made unless clearly stated otherwise; in which case the name of the person who made it will be given.

All the program code has been made in C++. One of the reasons we chose it is that we have used PCL (Point Cloud Library) which is completely written in C++. Everything has been written in Visual Studio 2010 under Windows 7 for simplicity's sake. At a later stage in the development everything was made compliable with Linux (Ubuntu) before it was finally ported to PandaBoard.

I would like to thank my supervisor at school Christer Albinsson and my supervisor at the company Peter Reigo for giving me this important part in his project.

Table of contents

1 Introduction

1.1 Background

Autonomous machines in industries for automatic goods transportation and storeroom item management is nothing new. All these system does however rely on position indication which is very limiting and can be expensive to install. It can be wires in the floor or reflective indicators on walls/ceilings.

The company this thesis project is made for has a very accurate system for positioning indoors and outdoors; so our hope is to use this system for positioning on robots instead of physical indicators.

The advantage is flexibility and very short time required for installation.

1.2 Problem definition

In this project we are working with the assumption that the robot is a **cleaning machine** for industries, big office buildings or train stations. This control system can also be used for machines like **AGV (automated guided vehicles)** and similar.

Normally these kinds of machines have a very well defined path it should always follow, since it just follows where the markings are.

In our case we really want to make use of the flexibility of the positioning system. Therefore it must be very easy to set up new paths for AGV's, and a cleaning machine can calculate for itself how it should move.

The requirement for this is to have a very good sensor so it can map the environment. This is very essential for the cleaning machine to be able to cover the entire floor. The AGV must have a good sensor to be able to detect obstacles blocking its way (if perhaps a pallet has been left somewhere in a narrow passage). And both must detect if something (someone) enters in front of it and stop.

We need to find a sensor which will work in sunlight. It also needs to give us enough information that we can calculate/assume/estimate the shape of objects and make a model to put on a map.

The objects will usually not be seen fully from just one point so it is necessary to look at the object from more the one direction to get the whole shape. It must be possible to add more information to objects when it is seen from different positions.

When something is mapped it must be possible to use this data for navigation.

1.3 Requirements

The control system should be able to detect and map simple static (stationary) objects. A simple object is solid and do not have any concave areas.

The sensor system must have a quite low price.

1.4 Goal description

The robot should be able to make a 2D map of the environment. The map should be dynamic and can register and update changes in the environment. Hopefully it is possible to identify what the sensor sees as either wall or object and treat it respectively (there are some differences).

The robot should be able to navigate through any for this application typical indoor area (with use of the map).

1.5 Methods

First step is information gathering by researching existing papers and projects about autonomous vehicles and obstacle detection. We also need meetings and discussions with companies and hopefully experts within the optical sensor field about possible sensors.

PCL (Point Cloud Library) will provide us with most tools to work with Point Clouds which is the type of information a more advanced sensor will give us.

1.6 Limitations

There are objects which cannot be detected with certain kinds of sensors, they will not be bothered with. With optical sensors like ToF (Time of Flight)-camera is it not possible to detect completely opaque objects like glass walls (this is something we have encountered).

A great problem is when certain things change their shape in a way that we need to remove data. One such case is if we have mapped a segment of a wall, and then a door on it is opened. Then it is necessary to remove data were the closed door was and make a hole in the wall. This is something that we probably will not have time to make work.

Main chapters

2 Sensors

This is one of the main issues of this project. Finding a good sensor was vital to the success of this project.

The sensor should not only have good enough performance but must also be very cost effective. Our requirement is that the machine can be produced in higher quantities and have a low price for the end customer.

2.1 Time of Flight-, Structured Light-sensors

These sensors fall into the category of 3D image sensors (they are sometimes called 2½D sensors since they after all can only see one side of an object at the time).

This category has a very well-known sensor in it called the Microsoft Kinect which is based on structured light. Disadvantage with it is that it will not work in sunlight, even indoor near a window is a problem with it.

However the other sensors all have in common that they are extremely expensive. Most of them are made for industry applications and cost > 5 000Euro.

The manufactures of Time of Flight cameras we have found are:

- **Fotonic** – a Swedish company who specializes in customization of cameras for industries like mining industries. They do not develop their own sensor chip.
- **PMDTec** – a German company who has developed a sensor chip and a camera called **CamCube** which they use to demonstrate the chip. They otherwise only produce the chip for other companies.
- **Panasonic** – the Japanese consumer electronics company has made a device for mostly advertising purpose which they call D-Imager.
- **MESA Imaging** – a Swiss company responsible for a family of cameras called SwissRanger's.
- **Hokuyo** – a Japanese company which has a few different Time of Flight cameras.

Manufacturers of Structured light cameras seem to be even rarer:

- **PrimeSense**/Microsoft – The Kinect
- **ASUS** – a Taiwanese company who made a camera very similar to the Kinect by the name Xtion PRO

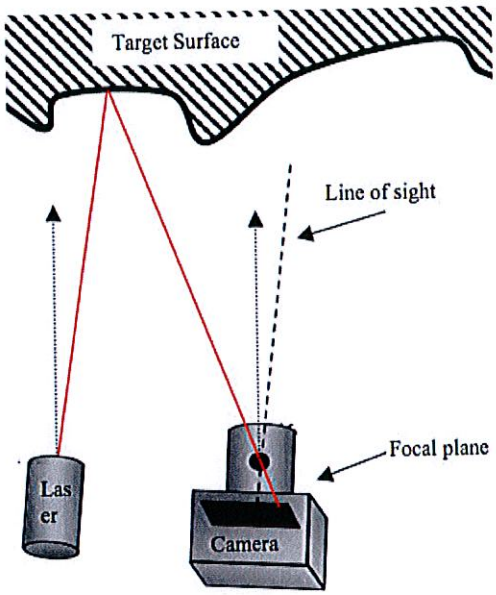
A great problem with all of these optical sensors is that they can get different readings on objects depending on its color. Some objects can even be difficult to detect at all. Since black color doesn't reflect light very well the sensors will sometimes not see it. Shiny objects are also a problem since the light will either be reflected away from the sensor or if the object is directly facing the sensor it might get too much reflection. Other colors can be seen at different distances depending on its color.

Durability: Neither of these two sensors has any moving parts in them. This is very good for lastingness. Moving parts and motors will always wear over time. Without moving parts the encapsulation can also me made water- and dirt-prof which makes it last longer.

2.1.1 Structured light

Structured light is quite simple in how it works. It basically uses triangulation to determine depth value.

A projector is used with light of a certain wavelength to emit lines over the scene. From the projectors point of view the lines will look to be straight and aligned over the scene. When looked at from somewhere else though the lines will follow the objects it hits and the lines will no longer look straight.



A sensor (CMOS usually) with a band pass filter focused on the projectors wavelength is placed to the side of the projector. A processor can then calculate depth values from how much the line has changed from the sensors point of view.

This is also possible to do with other light types than lines.

(AnandTech, 2010)

2.1.2 Time of Flight

When comparing the Time of Flight camera with other sensors it seems like its main advantage is possibility for very high speeds and real time capabilities. Its competitors like Laser Scanners and Structured Light cameras are slower and the Laser Scanner doesn't have real time capabilities (due to it being a scanning device where the scanning process takes time).

It can be based on a few different principles and all developers have found some special features. The simplest solution is the one that gave the sensor its name. The light emitters are turned on for an instance and then the time delay is measured for each pixel when it receives the reflection. Of course extremely accurate and quick timing devices are needed. At a distance of three meters the time difference is only 20 Nano seconds.

A more recent principle is using a modulated light signal and measure the phase difference when it is received. The only cameras that are using this technique all use a modulation frequency of 20MHz. The frequency sets the limit for max distance that can be seen with the camera:

$$Distance\ max = \frac{c}{2 * f} \quad ; c = speed\ of\ light\ \left(\frac{300Mm}{s}\right)$$

With a modulation frequency of 20MHz the max distance is 7.5 meters.

It is necessary to have a high frequency to get highest possible accuracy. The frequency should be adjusted to fit with desired range distance.

(Schaller, 2011)

2.1.3 ToF vs structured Light

The two sensor types are quite similar in performance in many situations. Indoor in a midrange segment they can output about the same accuracy. Structured light has the problem that all the

emitted light from it must come from a single point. To make it work in brighter environments its light output must be increased. This makes it not compliant with eye safety regulations and because of that the **Structured Light type of sensor can never be used in environments with too much ambient light (outdoors).**

ToF cameras have of course the exact same problem that its light output must be increased in order to counter ambient light. In this case however the light can be split into several light sources each with a value which is considered to be eye safe. Because of this **Time of Flight cameras can be used in environments with a high amount of ambient light.**

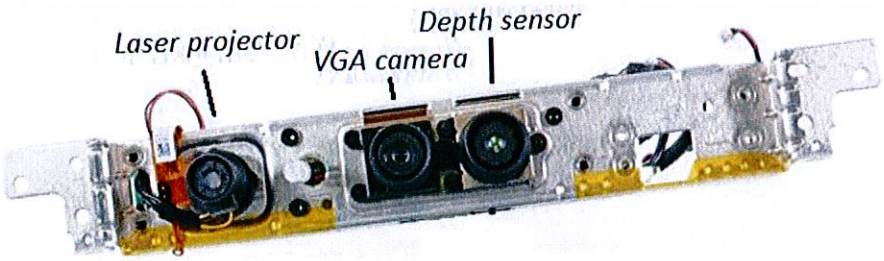
Flying points – This is a problem that only the ToF camera has. It is basically that it will get data partially from the very edge of the object and its background. The depth value will end up somewhere between the two depths depending on where the light hit most. This is not an issue with the structured light type since it will simply not get any data in at this point.

From looking at data from both sensors and comparing them we got the impression that Structured Light is more accurate and stable then Time of Flight. This also seem to be the general impression when reading about the it. We have then been discussing the matter with a manufacture of vision based cameras who dismissed our impressions. Apparently the Time of Flight sensor is just as good and can be made better then Structured Light with correct filters and processing of its data.

2.1.4 The Kinect

The Kinect belongs to the structured light family. It has an infrared light projector which illuminates the environment with a dotted grid. It uses a common CMOS sensor like most other sensors to receive the information back.

The 3D sensing part of the sensor was developed by en Israeli company called PrimeSense. Another company owned by Microsoft called Rare has made the software that runs on the Kinect which is what makes it a device usable in the entertainment industry.



Data from the Kinects depth sensor can be received at maximum 30 fps. According to the specifications the data has a resolution of 640 x 480 but this is not entirely true. The data the kinect receives from the environment has a lower resolution but is then calculated into 640 x 480 resolution. Which is quite unfortunate for most robotics projects since this resolution is simply too much to process.

The actual data from the depth sensor is an array 640 x 480 times large with 16 bit unsigned integer information. The sensor only has an accuracy of 11 bits which gives 2048 different depth values.

Data from the Kinect will look very stable and accurate (when in an environment where it works) and can look much more so than the data from a ToF camera. In the latter case the dots are moving slightly from frame to frame so the cloud looks something like the surface of an ant hill. While the kinects data stay nice and firm in while it is stationary and the environment doesn't move. The way they managed that Is because of something called "averaging placement" where the depth value in each pixel is averaged to the nearest threshold. It has a few different segments; very close, close, mid, far. The depth value within the mid-range will always be averaged to an accuracy of 1 cm and

the points in the far-range to 1 dm. This will make the data look more stable (between frames) because it will be fixed at specific values rather then moving slightly with a few millimeter difference.

The main advantage with the Kinect is of course the price. The low price might not inspire as much confidence as a Time of Flight camera with a price tag at 4 000 Euro. But the only reason the sensor is so cheap is its low cost package and especially the extreme volumes at which it is produced.

Other advantages with the Kinect is of course that is has lots of additional features:

- Built in software for "skeletal tracking" and gesture recognition.
- There is a video camera with 640x480 in resolution.
- Microphone array for possibility to locate direction of sound.
- Motor for autonomous directing of the sensor.

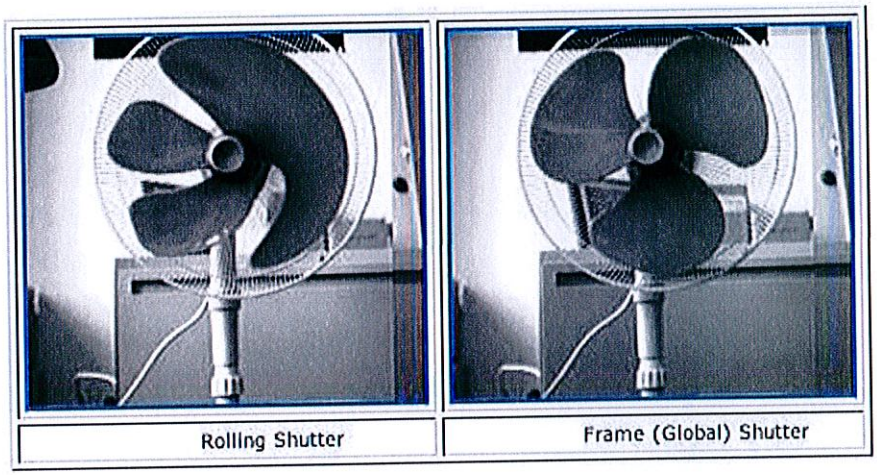
For this project however none of these features has been relevant.

2.1.5 CMOS and CCD sensors

Most Time of Flight cameras and Structured Light cameras today use a CMOS chips to receive the sensors input data.

CMOS is very common and used in most optical sensors and it is very cheap.

It has a sometimes big disadvantage however. It does not receive its input from each pixel simultaneous but rather receive it one row at the time. With either a moving sensor or en environment which moves this will cause a huge problem called "rolling shutter" effect.



(http://ieba.wordpress.com/2007/11/01/rollshut-2/, 2012)

We have found a Time of Flight camera that is based on CCD technology instead of CMOS. This sensor will not have this kind of problem because it captures the hole image at the same time.

Even if the machine we are using will be quite slow there is still the problem with objects around it moving at higher speeds.

According to the manufacturer the CCD sensor is also better with different colors and even back objects should be quite alright with it.

2.2 Ultra sonic sensors

We still had ultrasonic sensors to do at least some of the work. The only solution we could see was so called "sensor fusion" with a few sensors to get reliable data from at least one sensor at all times.

For this we had Camera (web camera) and laser ranger and IR device or laser line scanner or stereo camera vision.

2.3 Camera and Image

There seemed to be a lot of possibilities with just a simple (and most importantly cheap) camera. There is a project where they have managed to make a "Image to 3D" technique. They can actually make depth data assumptions from a single picture.

When they made the algorithm to calculate depth data they used "machine learning" where they have 10k pictures And depth data (from laser scanner) as training material.

(Ashutosh Saxena, 2007)

A more simple approach would be to only look at the color in an image and assume that obstacles always have a different color than the ground. The color of the ground that the robot has covered should be used to decide the color of the ground since we then know it was in fact ground.

(Miloslav Richter)

2.4 Laser Range finder

The hobby sensor manufacturer Parallax has a very cheap Laser Range finder, it is however only one point/measurement and max speed is 1 Hz. It is made for indoor use only.

2.5 Laser line scanner

An alternative to ToF- or structured light-camera is Laser line scanner. We had a meeting with a consultant who recommended it. He also put us in contact with a KTH professor Göran Manneberg ("Ask the physicist" from the student paper Osqledaren) who also recommended it since it is simpler and cheaper.

There is a sensor by the company Sick which is used very frequently in all sorts of academic purposes, projects like DARPA and for Autonomous Guided Vehicles in warehouses. It is however very expensive (made for industries). A Japanese company called Hokuyo has several laser line scanners and some are quite cheap. However the only one that is made for outdoor use costs 5k$ so they are also very expensive.

http://www.robotshop.com/high-end-lasers-obstacle-detectors.html

A component sales store in Sweden called Parameter AB had all the components required to make our own laser line scanner. The components they recommended were unfortunately quite expensive as well. Just the camera cost more than 500Euro, and laser 300Euro and you need filter and prism etc. We would get a lower price than the ones mentioned above but it would also take a long time to build it ourselves.

http://www.parameter.se/

The conclusion that the information regarding that LLS were a much more cost effective solution didn't seem quite accurate. The other problem is that we get so much less information from a LLS

than a Point cloud sensor. Unless the price would be much cheaper for the LLS it just didn't seem worth to choose it rather than the Point Cloud sensor.

Higher volumes: a problem with LLS is the camera; it needs a very good one and even in higher volumes the camera will still cost a lot.

Because of the nature of how these types of sensors work there will always be some mechanical wear in it due to moving parts. This also makes it slower than Time of Flight cameras.

2.6 Safety

Today this is the only sensor that is certified for use on larger autonomous vehicles and robots. There are a lot safety concerns which must be fulfilled when heavy machines and people mix. So right now it is required to use a laser device on any machine just for safety reasons.

(ANSI, 2012)

2.7 Conclusion

	Pros	Cons
ToF/Structured Light	+Very good data	-Very expensive -Much processing needed -Power consumption
Ultrasonic sensor	+Very cheap +Simple +Power consumption	-Limited data -Not very dependable
Camera and Image	+Quite cheap	-Extremely difficult -Not very dependable
Laser Range Finder	+Quite cheap	-Very limited data
Laser Line Scanner	+Good data	-Very expensive -Power consumption

One of the requirements for the machine is that it should work in sunlight. This makes use of the Kinect sensor impossible since it is very sensitive to ambient light. The Kinect was otherwise the number one candidate for sensor.

Since we initially wanted a very cheap sensor the part of choosing our sensor took a lot of time. We considered using "sensor fusion" where you combine different sensors to get better and more reliable information; the idea is to use the strength of each type and none of their weaknesses.

With sensor fusion we thought to use a normal camera together with ultrasonic sensors and laser range finder. A very basic avoid obstacle program could be made with just the ultrasonic sensors (and laser range finder for better accuracy). We then hoped to use the camera to differentiate ground from obstacles with another color. A great nicety with cameras is that image processing can always be made better. So it is always possible to improve the program.

The downside is that the ultrasonic sensor and the laser range finder are not good enough by themselves for this project. And until the image processing is good enough (which might take a very

long time to make work) the machine will have very limited obstacle detection and mapping capabilities.

2.8 The decision

With the initial goals we couldn't find any good solutions other than just ultrasonic sensors and bumpers. It was then decided that we should not focus on a very cheap sensor but rather a very good sensor that would always work which could then be used in more expensive products.

Because of this we decided on Time of Flight camera. We couldn't decide on which camera to use right away but they all have at least similar output data and functionality. So we began working with the Kinect while deciding on which Time of Flight camera to use.

The only problem is that by only using a Time of Flight camera the machine will not be safety approved. It is however always possible to add a Laser Scanner just to use for safety reason on machines that are so big that it is necessary.

3 General theory

3.1 SLAM

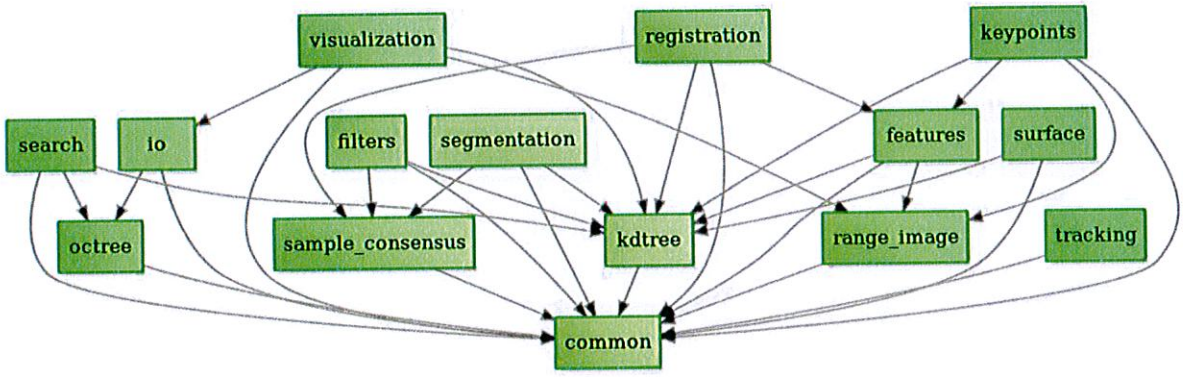
Simultaneous localization and mapping (SLAM) is when a machine with capabilities of mapping is turned on for the first time in an unknown environment. It needs a map to be able to navigate and it needs to move to make the map. It is referred to as the famous "Chicken and the egg problem" where the question is which came first.

In SLAM the machine start by mapping the area of which it can see when it is turned on. It then uses this very small mapped area to move within that area. It can then map more around it and can continue this way until everything is mapped.

SLAM is always used unless a "static map" map is used. A static map will never change so everything is added to it from the beginning and thus no mapping is taking place.

(OpenSLAM)

3.2 Introduction of the Point Cloud Library



PCL (Point Cloud Library) is a very well made library for handling n-dimensional point clouds.. Especially in the beginning of the project this library was very much relied upon. Unfortunately it is not perfect for the project, and the reason is that everything in the library is optimized for 3D while in the project 2D is usually enough. Disadvantage is decreased performance and unnecessary

memory allocation. PCL is however excellent in the beginning just to figure things out and to actually try ideas without having to spend too much time writing all the code for it.

(Cousins, 2011)

One important note is that PCL utilizes C++ extensively. Just using c which is quite common in robotics is not possible. The user also needs to have good knowledge about objective programming and templates which is used in almost every command to PCL. It will be a challenge for the user who is used to more low level hardware near programming which is usually the case in robotics.

Another issue is that PCL is dependent on several other libraries to work.

- Boost is a well-known C++ library which is OS independent. PCL uses boost for pointers and memory allocation. (Boost, 2012)
- Eigen is a mathematical library for matrix based mathematics. (Eigen, 2012)
- FLANN takes care of sorting algorithms like kdtrees. (FLANN, 2009)
- QHull for surface extractions and makings. (QHull, 2012)

3.2.1 Point types

PCL support a great deal of point types through templates. The most common one is called PointXYZ and is a coordinate in 3D space. It is also possible to have color information for each point (can be great for debugging) PointXYZRGB or PointXYZHIsomething. And Normal information PointXYZNormal, PointXYZRGBNormal etc.

Of course the more information you want to store the more memory is required. Coordinate, color and normal are 16 byte each for each point. Since the point cloud from one frame (30fps) from the sensor kinect contains 307200 points; you can see that memory is an issue if you want to store the data (420Mb/second with PointXYZRGBNormal).

3.2.2 Smart pointers

In PCL they use a feature which might be new to some C++ users. They have implemented so called "smart pointers" from the library boost. It is something similar to a garbage collector only much more limited and therefor it is quicker and don't consume as much processor power.

Basically you request a pointer from boost and at the same time create a new object which it is to point to. You can then create more smart pointers that point to the same object and boost will keep track on how many are pointing to a specific object. Once the last smart pointer is destroyed (usually when function exits) boost will delete the created object for you.

When you are new to this it is very easy to make mistakes with it. If you have a normal pointer pointing to an object created with a smart pointer; boost will not see this. If the smart pointer is destroyed your normal pointer will be pointing at invalid memory and can cause a crash if used.

Example: say you have code for visualization in your program. You make the processing of the data as you wish and then you want to draw it. Then you have to make a copy of the data using the normal memory allocation, otherwise the data will be destroyed by boost when the calculation function finishes (unless you store a smart pointer to the object).

3.2.3 Methods

PCL offers a great selection of methods for processing point clouds. The following underlying headlines provides a sort explanation of the most frequently used methods of PCL.

3.2.3.1 Plane extraction

Plane extraction with RANSAC (Random Sample And Consensus). With just a few lines of code PCL can extract planes from a specified point cloud.

It created by Fisher and Bolles in 1981 as a mean for estimating coefficients from sets of data.

In PCL this method unfortunately only allows you to extract the biggest plane in the cloud it can find. If you have a specific plane you want to extract like ground/walls; then this method will not be of any help by itself.

The reason for this is because of the very nature of how RANSAC works. It is an iterative method which works by looping though just **some of the data** in the dataset (the point cloud) and find which plane most of these points fit into. The requirement for this to work is that most points actually belong to a certain plane, otherwise the result will just be a random plane.

Parameters:
With RANSAC it is always required to specify maximum allowed distance from data (point) to the resulting plane. The more accurate data smaller distance can be used.
Number of iterations can be computed by PCL. There are otherwise equations for estimating the number of iteration to guarantee a good result that must be used.

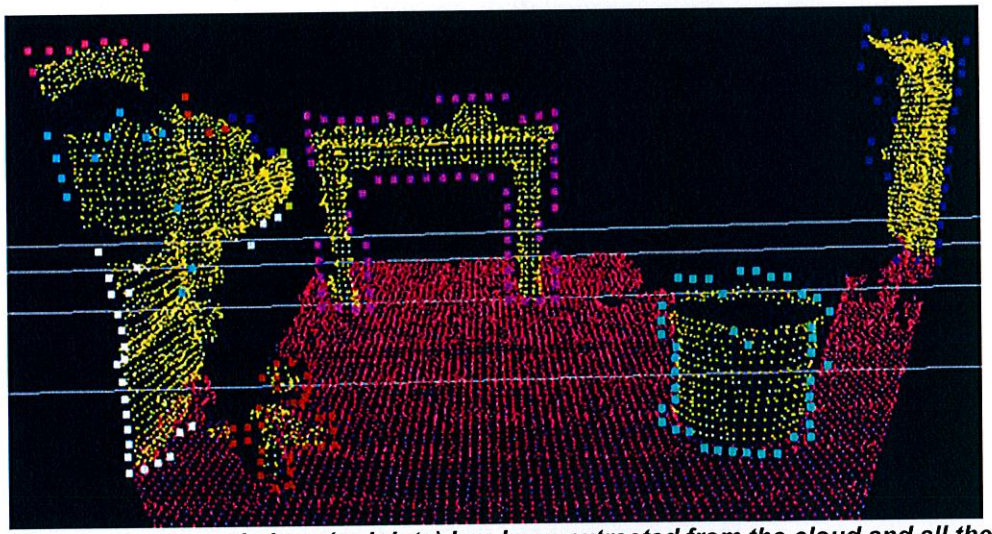
We are using a combination of RANSAC and our map when extracting planes. How it works is explained under Implementation.

For further reading about RANSAC I can recommend the Dummy version (RANSAC for Dummies).

(Zuliani, 2012)

3.2.3.2 Euclidean Cluster extraction

Clustering is used to separate all the different clusters (objects) in the point cloud. It works by looking at distances between points to see if they are close enough (parameter) to be a part of the same cluster.



Picture: the ground plane (red dots) has been extracted from the cloud and all the remaining (yellow) points are quite clustered. When the cluster extraction is run on the remaining points it will output five separate clouds.

Given parameters like max distance and max/min size; PCL can extract different "clusters" (groups of points together and not part of anything else) from the point cloud. This requires that there are

clusters in the point cloud. The original point cloud from the sensor doesn't really contain clusters. It is only after specific part of the cloud has been removed that it will be clustered. If ground/ceiling and walls are removed there will only be clusters left which are the objects (other than the planes) left in the environment.

Example: a room with a box in it. In a point cloud representation of this scene there will be a lots of points along the ground plane and the points from the box in it. The points on the lowest part of the box will be very close to the points of the ground it stands on. Therefor it is not possible to extract the box as a cluster since it is part of the points of the ground.

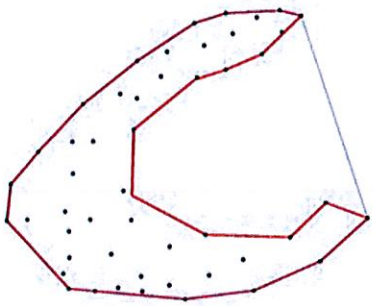
Cluster extraction is also very good for some types of noise removal. Since you can set a min cluster size, loose points without neighbors (flying points) can be ignored.

Clustering in PCL seems to work very well and I have found no reason to write a new one. If doing so however I would recommend:

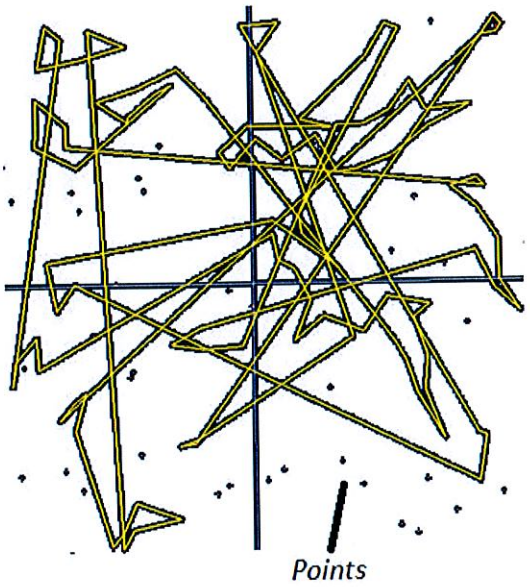
A lot of looping through the cloud is required when finding the clusters. To save processor time in the long run it is a good idea to start with sorting the points in the cloud along one of the axis (x-axis). The sorting only need to sort the points roughly, it is not necessary for them to be in perfect order. Once sorted only one more looping is required to see which points are grouped together by looking at the other axis (z-axis).

3.2.3.3 Border points calculations

PCL has methods for finding the points in a cluster that are placed outermost. This is called Convex/Concave hull.



In the picture there is a 2D point cloud on which both concave and convex hull has been calculated. The thin blue line together with the outlying dark red line show how the result from convex calculation looks. Convex hull can be thought of as a rubber band over the cloud. The two red lines show the concave calculation. There is a parameter that must be used when calculating concave which specifies how small crevices it will "go in to" sometimes called the "smoothness" value. With a high enough value the result from concave will be identical to the convex result. With a very low value the result will look very spiky.



Unfortunately PCL's implementation is not very robust on 2D clouds. The picture on the right show PCL's calculation of a concave hull on all the points seen in the picture. We can see that it has left lots of points out of the hull and the order of the points is quite horrible. This is mainly because the concaveness parameter is not optimal for this particular cloud. But this is also quite hard to calculate at runtime. In most cases though (95%) a general value seem to work.

Because of this it is necessary to loop through the lines and see that none of them are intersecting just to be sure.

Hopefully this will be improved when the developers

get the time.

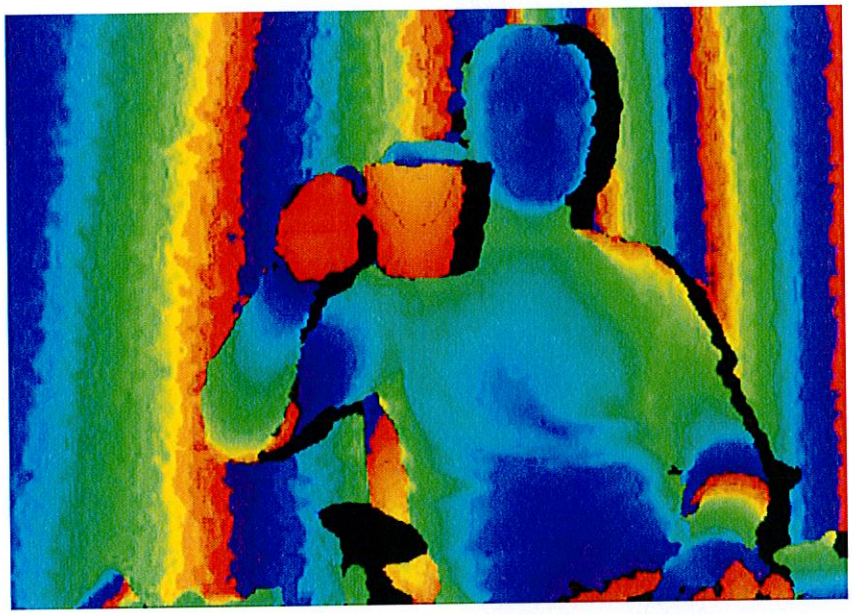
3.2.4 Support

PCL itself is just a huge library which contains information like the examples above. It has no support for communicating with sensors.

What has to be done is setting up the communication with sensor with help of a driver or by using programs like OpenNI. And then convert the input data to pcl::PointCloud<pcl::PointXYZ>. In worst case what you have to do is just copying the data into the PointXYZ format which of course takes time and additional space.

3.3 Range Image and Point Cloud

Sensors like the Kinect, Time of Flight cameras and laser scanners all get their information in form of a Range Image.



The Range Image is just a picture where every pixel represents the depth value the sensor received. This is also sometimes called "raw data" because it looks like the sensors unprocessed output. Even the Range Image is however processed by filters and threshold in the sensors processor before sent to output.

Some colors and objects are easier to see then others. In the picture I am sitting in an office chair with black shiny armrests which the sensor (Kinect) don't want to see. Most of the other black areas around the cup and me only looks to be missing because the visualization has another aspect ratio then the Kinect and also because the sensors position and the camera in the visualization is different. Some points around the edges are however missing.

3.3.1 Conversion from Range Image to Point Cloud

When the data from the sensor should be used in a 3D application (for a robot or just to see where something is) it usually required to have the depth data converted into a 3D point cloud.

The picture above is the same as the range image but as point cloud and from an angle slightly looking down. It is a little difficult to see but the point cloud is mirrored compared with the range image. In lots of cases this is a problem and the cloud therefore has to be unmirrored which takes extra time.

The information from the sensor is an array of usually 16-bit unsigned integers and a value for width and height. Every sensor has a specified angular field of view in vertical and horizontal axis. For the Kinect this is 57° horizontal and 43° vertical.

To calculate the Point Cloud it is necessary to calculate x, y, z float values for each pixel. The array is of the 1D type, example:

*unsigned uint16 __data[width * height]*

When calculating the cloud however it must be pretended that the array is of the 2D type, example:

unsigned uint16 _data[width][height]

X is calculated using the pixels position in "width" (index) and the sensors horizontal field of view (fov_h) together with basic math.

$$x_{value} = \cos\left(\frac{index_w * fov_h}{width}\right) * depth$$

Y is calculated the same way but using the sensors vertical field of view (fov_v):

$$y_{value} = \cos\left(\frac{index_h * fov_v}{height}\right) * depth$$

Z is not equal to the depth value but must also be recalculated in a similar fashion

$$z_{value} = \sin\left(\frac{index_h * fov_v}{height}\right) * depth$$

Or:

$$z_{value} = \sin\left(\frac{\text{index}_w * \text{fov}_h}{width}\right) * depth$$

3.4 OpenNI (Natural Interaction)

OpenNI is an industry led non-profit organization with the goal provide a standard for the kind of sensors that can be used for vision and audio. It is mainly founded by the company Primesense which is the company behind the sensor Kinect. (OpenNI-Foundation)

It offers a Application Programming Interface (API) for user to low level drivers from sensors and also to some middleware software.

OpenNI has the possibility to get data into the program in the direct form that PCL expects which is a great advantage. In this project we have used the OpenNI grabber interface to receive the point cloud to work with when using the Kinect sensor.

Since it is open source it already has support for lots of sensors. The sensor were using at a later stage however was not supported by OpenNI.

(OpenNI-Foundation)

3.5 Robot Operating System

ROS (Robot Operating System) is an open source project that started in 2007 which is very well used in academic projects and hobby projects. It is not quite an operating system despite its name. It currently only supports Ubuntu as operating system.

With this system installed it is possible to get a lot of functionality directly from this package. It supports PCL directly and most project that uses PCL also use ROS. Most other kinds of sensors are also supported as well as motor control. It also supports communication with the board Arduino which is very well used in prototypes and hobby projects.

In this thesis work we are not using ROS. Mostly because we already have motor control and control of communication protocols. And since we don't have unlimited processing power we never used it.

(ROS, 2012)

4 Visualization

4.1 Simulation and visualization as a tool

Most of the code has been developed with help of only simulation. It takes considerable time to load the program into the actual machine to try its performance. And in the case of path planning it would take too much time to let the machine move according to the program when testing it.

Simulation has therefore been of outmost importance. This has been done in part thanks to the debugger of Visual Studio and from visual feedback.

Because the debugger is so important it is necessary to use a good developing environment and then you must use either Windows or something like Ubuntu. This makes the hardware (the PC) very different from the board that is the end platform. Because of this it takes additional time to port the program to the board platform.

A good debugger is necessary to get the code working in the sense that it doesn't crash and make sure it acts in the way you expect. Other things in the program is too complicated and too immense for the debugger to be of much help. Example: The sensor will output thousands of points so there is no way to evaluate that kind of data with the debugger.

4.1.1 Using OpenGL for visualization

This is where visualization with OpenGL comes in.

To use OpenGL a lot of information has to be stored so it can be drawn on the screen. Everything from the unprocessed point cloud to the map with a model of the robot and the move vectors from the coverage planner are drawn with help of OpenGL.

The information is mostly points and lines which is exactly what OpenGL likes the most and they are very simple to draw. The result you get from the data processing might not be possible to draw with any existing tools since they require a specific format. This is when OpenGL shine the most since it is just to write the code for it to suit your needs.

What OpenGL doesn't handle is input from the user and GUI (Graphical User Interface). Getting it all to work (and both input and GUI are quite necessary for it to give a good impression) it not an easy task if you haven't worked with it before. Luckily there are also some premade tools for visualization.

In the appendix part there will be an example where OpenGL is setup with GLUT and creates an instance of the OpenNI-grabber to communicate with any sensor supported by OpenNI.

4.1.2 Alternatives to OpenGL

RVIZ – for ROS only. RVIZ is an easy to use tool which offers great flexibility in visualization. It works by subscribing to published messages and can draw multiple messages simultaneous with possibility to change color, scale, rotation and offset.

Cloud Viewer – included in PCL. The Cloud Viewer is probably the easiest tool to get running. It is however also the most limited one. It is only possible to draw one cloud at the time with no possibility to modify anything.

VTK – The Visualization Toolkit. This is recommended by PCL. It does seem quite difficult to get running though. It should be quite competent with lots of possibilities but if Linux is used it seems easier to go with ROS and RVIZ.

5 Mapping

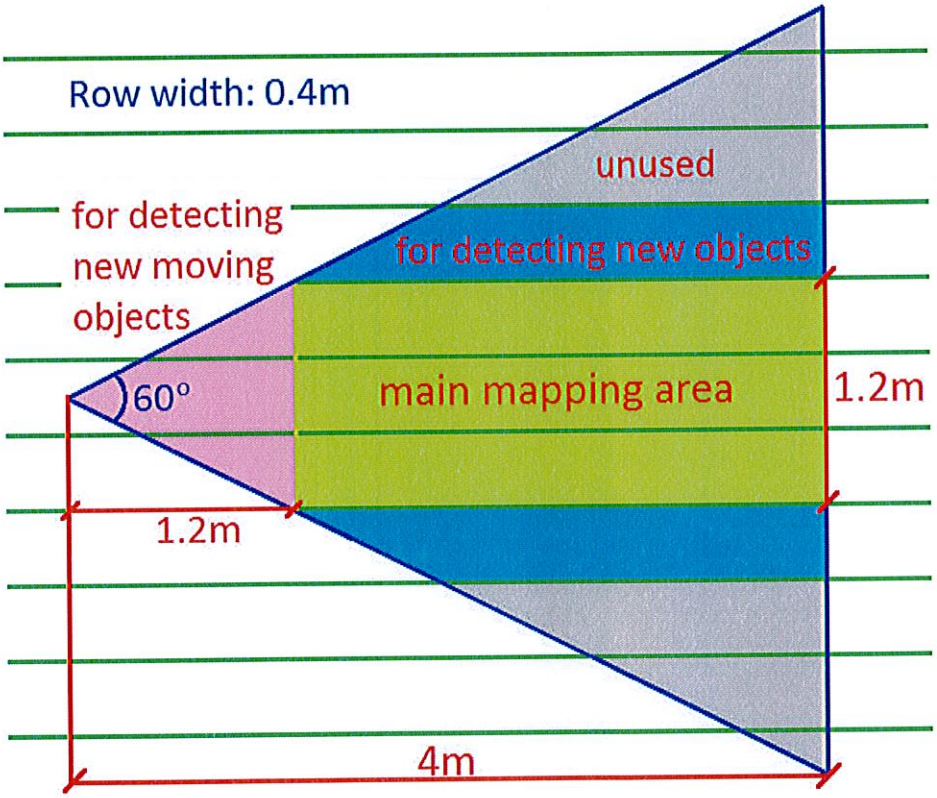
The main objective of this project has been mapping. The goal has been to find a way for an autonomous robot to be able to map its environment while it is moving. It should not be mapping only for the sake of mapping but rather to be able to use the map for better path planning and navigation. The goal has never been to make a 3D map of the room/environment.

The mappings approach we have come up with can be used with a map and with SLAM. There is a minor difference with relying on SLAM and using a premade map with wall data. With the wall data from the map (either after SLAM has already been used or from premade map) it is possible to separate objects from walls with much better accuracy then when something is seen for the first time without a map.

Dynamic environment – to be able to handle an dynamic environment it is necessary to remove objects that are no longer there. It should also be possible to realize if an object has moved. This is especially important if the mapping program identifies an object as an object that's already mapped without adding data. Because if it has been moved the maps border points are incorrect and the machine might hit it. Possible ways to counter this is discussed under "Dynamic objects".

5.1 What we see

Time of Flight cameras and Structured Light cameras have a cone shaped field of view. Almost every such camera has an angle of 60 degrees which is what the picture below is representing.



Since the field of view is shaped as a code; it is not possible to detect very near objects on the side. This our case this is only a problem if something moving comes from the side just in front of the machine on the side. This is a problem in safety aspect however. Stationary objects are never a problem (unless the machine is turned on right next to an object that's just in the way of the machine but can't be seen).

It is a rather large area that is being visible from this kind of sensor. The data from far away or far out on the side however is not as accurate as the data at the right distance in front of the sensor. Because of this we have defined a more high quality area in front of the machine within proper distances (the green area). This is the data used for mapping and making decisions of dynamic obstacles and everything else map-related.

The blue areas in the picture is only used for coverage planning in certain situations. The data from it is not used for mapping but only to detect if there is something there or not.

5.2 Mapping process

We are using SLAM in the sense that our environment is not a static map. Only the walls are put on the map, everything must be mapped.

It was decided in the beginning that we would not use a grid based map but rather a vector based map. The grid based map is still the most used way to store maps. The disadvantage is however that it takes a lot of space especially when high resolution is desired. We want to have the possibility store very large maps so a grid based solution would cause a lot of difficulties.

Vector based map seems to be considered being more complex. Operations using vector based mathematics is however very simple, and the pc is quite fast at working with it. So the preference is probably shifting towards the vector based solution.

After having worked with the map for a while it was realized that it is important to keep track on what we have mapped and where we have been. To solve this we had to implement something quite similar to a grid after all.

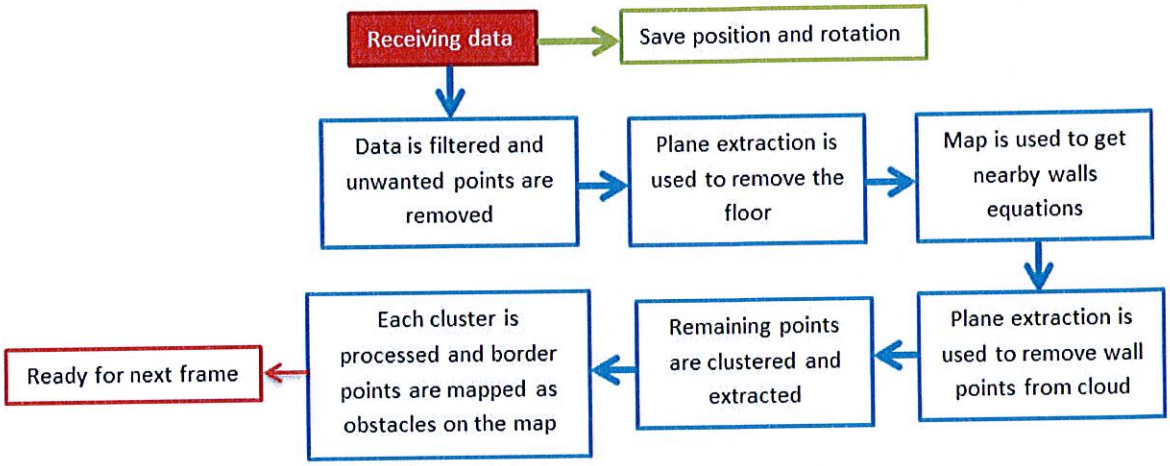
Rows: The machine will always be moving along parallel lines over the maps which looks a lot like "rows". When an object is mapped we save from which direction we have seen it (always from Left or Right) and the rows that are seen. When all rows on one side of an object has been seen then that side is "finished".

The rows can be seen in the picture above as the green lines. The distance between the lines (the row with) equals the machines working-width minus overlap amount.

It is important to note that the sensor has to see further if the rows are wider. Otherwise objects on the sides will be missed.

5.2.1 General program flow

Simplified version of the program flow:



1. Working thread receives the new frame from the sensor.
2. Position and rotation of the robot is stored as the frames translation.
3. The frame is filtered in Y and Z with limits that remove unwanted points.
4. Plane extraction is used to remove the floor from the remaining points.
5. The map is used to get the equations of nearby walls (if any).
6. Plane extraction is used to remove all points that are part of the walls.
7. Remaining point cloud is clustered and the different clusters are extracted.
8. Each cluster has its border points extracted and stored on map if new.

The position and rotation of the frame must be saved for when obstacles and walls are extracted.

5.2.2 Receiving data with OpenNI

The first step in the program is to receive data from the sensor.

Sensors of the 3D camera type does not send data continuously but rather receives the data internally and processes it and only after that tells informs its connected device that a frame is ready.

In our case we are only interested in looking at Point Cloud type of data.

5.2.2.1 Using OpenNI to receive

If the sensor is supported by OpenNI (like the Kinect) it is a very smooth experience to receive data from the kinect.

All that is needed is to setup an instance of OpenNI's grabber interface. It will then be activated when the sensor has a frame and call a callback function which has a smart pointer to the frames point cloud.

This is assuming the Grabber thread is not already occupied by the previous frame.

5.2.2.2 Using other sensors

At a later stage during the development we changed to another sensor which was not supported by the OpenNI framework.

In this situation it is required to have a driver for the sensor. Unfortunately the driver can be a problem because you need different drivers for different OS's. The company who made the sensor we use has so far only provided us with a driver for Windows.

Another problem is that it is probable that another sensor only give range data as output. Because of this we had to make our own function for calculating the point cloud from the depth data (made by Xuan) *a general idea for it is explained in the General Ignorance section above.*

There is one advantage with this though. Just receiving the range data is only a matter of copying data (which it has to do in either case) and does not really occupy processing power. Calculating a point cloud however is very heavy since it contains so many points.

It is therefore advisable to do some basic filtering on the depth data while creating the point cloud. Points that are too high up can be discovered just by using an equation involving the pixels height. Points that are too far away can be found just by looking directly at the depth value.

5.2.3 Filters

The kinect sensor has the rather unusual fault in that it gives Too much data.

When the kinect sensor is used we apply filters in the very first step that can remove all the points outside the range and height that we are interested in. The filter will also reduce the resolution of the point cloud, otherwise at a distance of two meters it will have a resolution of 3.125 mm between the points. Which is simply too many points for the processor to handle.

With the resolution reduction filter it is possible to specify desired resolution (setLeafSize method inside the filter object). We use a value of 0.02f which gives plenty of accuracy and the processor can handle it (if just barely).

5.2.4 Plane extraction

As a start we only used RANSAC plane extraction to remove floor and walls. For this to work however some quite strict requirements must be fulfilled:

- The floor must be quite visible (not blocked by too many objects/walls).
- The walls must not be blocked by too many or large obstacles.

Disadvantages are:

- The requirements above.
- No possibility to use knowledge about environment (ground and walls from map).
- Sometimes extract planes which are not actual planes.

This is clearly not good enough for most applications.

Xuan made a simple plane extraction function which went through the cloud and extracted all points that was close enough to a provided plane equation.

Unfortunately you either have to specify a very large margin (distance to the plane) or have an extremely good accuracy in position and rotation data for this to work. Otherwise the equation will be slightly off and the wall/ground points will not be found.

A solution to this was running his function with a very generous margin to guarantee that all points in the original cloud belonging to the plane are extracted to a new cloud (cloud_1). Then running the PCL RANSAC extraction on this cloud (cloud_1). The idea is that the largest plane will be the one specified in the function call; so it is this plane that will be extracted. It has been possible to use

the RANSAC extraction with a very narrow threshold on the cloud (cloud_1) which is necessary for detecting small objects like carpets and doorway-thresholds.

After having run RANSAC on (cloud_1) the points that are left in it can be put back into the original cloud since they were not part of the specified plane.

The best effect of this is that the plane we specify as the ground will usually contain a small error. From RANSAC what we get is however the very actual plane equation which can then be used together with our position and rotation to know where the machine is!

5.2.5 Clustering

Explanation under the introduction to PCL

Once plane extraction is finished all that's left in the cloud is the points that do not belong to any of the walls.

Clustering works quite perfect at this stage since the cloud is now "clustered" from the plane removal(s). The trick is to find the right limits for min-limit of points -value and also the distance -value (how far apart the points may be).

The goal is of course to set a limit which makes sure that "noise points" of "flying points" will not be regarded as objects. While making sure that very small objects will still be saved. The minimum number of points for a cluster unfortunately varies with the point resolution. With the Kinect running in highest resolution this value must be quite high, otherwise with a lower resolution camera (or after a filter) this value must be lower. We have been using a value from 10 to 100. There is also a max number of points limit value.

It seemed logical at first to use a distance value equal to the width of the machine since it can´t go between closer object then that. However; the processing time increases dramatically when the distance value is increased. It is also a problem for the "concave hull extraction" algorithm (below) when the distance between the points are within a large range.

The input cloud to this function will after clustering only contain small groups of points and loose points that we assume are some kind of nose or small objects far away.

The output clusters are saved in a list for processing.

5.2.6 Processing clusters

First step here is to run the concave hull extraction.

We are currently using the PCL version but the goal is that Xuan will write a faster and more robust one.

Each cluster that we have is a 3D point cloud. It is important when making the hull extraction that we pretend it is a 2D cloud. When we use PCL's version this is a problem since we can't tell it to make a 2D hull so we actually have to loop though the clouds and set the height value to the same value for each point (using 0.0f but the value doesn't matter as long as it is always the same). Disadvantage is that this takes extra time, and it takes a lot of extra time to do the calculations on a 3D cloud rather than a 2D cloud. This is one of the main reasons for writing your own method since it should execute a lot faster if optimized for 2D.

The reason for calculating the hull points is that we need to do some comparisons of this cloud and all the preexisting ones if they intersect with each other (if they are the same object). When doing the comparison we need to get the clouds middle and min/max positions and it goes very quick to find these on the hull since it contains so few points.

5.2.7 Putting it all together

Now we have the hull and its position data. If the position data is similar or within another objects position data then they have to be merged. If not then it is a new object. The only difference when merging is that the step above has to be done again with the points for both of the hulls.

Finally when we have the hull object (new or merged) its row data will be calculated. It simply states which rows are occupied by the object mainly which makes it easy to see when the object has been finished mapped. The row data is also used by the coverage planner.

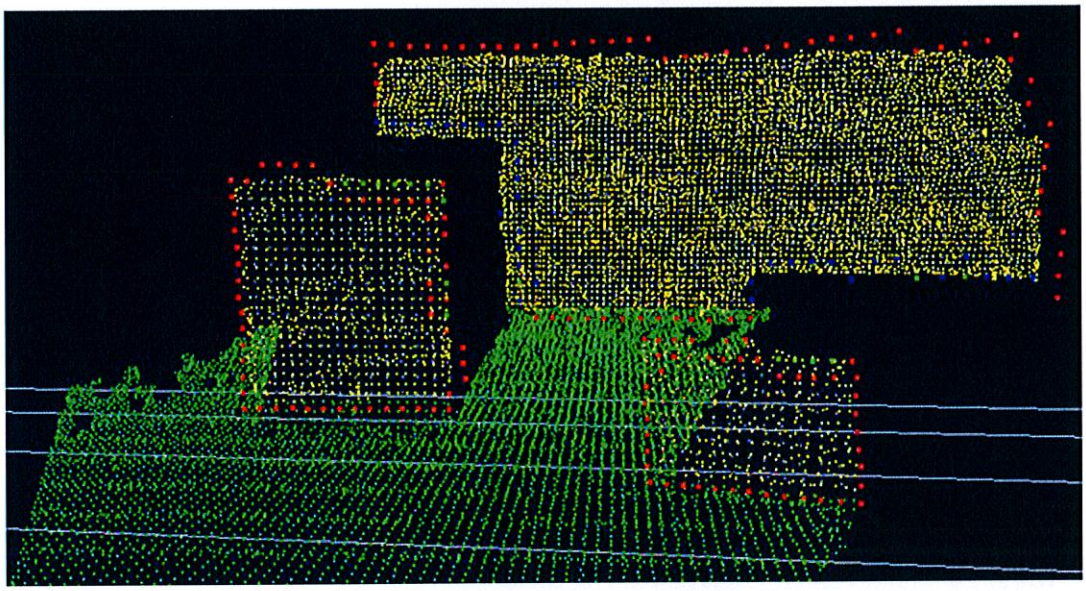
A problem when adding data to existing objects where we have already mapped is that the objects will grow with position error. If the so called dead reckoning is used then this is such a problem that it will never work. For our case since the positioning system always guarantees that the error will never grow more than +/- a few centimeters; this is not so much of a problem.

5.2.8 Simplify

The very final part is to simplify the object hull data. We have an algorithm (made by Xuan) which loops through the points in the hull and sees if there are any redundant points. If there are three or more points that are basically on a line the ones in between will be removed. The ultimate goal is that a box should only contain four points (one in each corner) after this stage.

5.3 Alternatives

We had a very different approach in the beginning. It was the depth data that we looked at and tried to get information about objects through it.



It is quite easy to understand what the object outlines are supposed to be. So from a human point of view this method looks quite nice.

However: It was soon evident that the exact height of the points was quite unimportant while within the range of the machines height. And then it was really only the outermost points that was important. After this realization the idea of looking at the range image was discarded.

5.4 Dynamic objects

There are two concerns with what is generally called dynamic objects. We don't want to put moving objects like a person on the map. We need to detect if an object has been moved just enough for the map to be incorrect. And there is also the problem that en object might have disappeared completely.

One thing that makes all of these cases problematic is that we might just have a somewhat faulty reading. This is something that need to be considered for a robust system.

5.4.1 Removed object

To detect that an object on the map is no longer there, it is necessary to check which objects we expect to see in each frame. It is then possible to realize when an object is no longer there.

It is not as simple as that of course. It is never good to either add or remove an object from just one frames information. In this case we have to not see the object a few times before it is decided that the object can be removed.

5.4.2 Slightly moved object

This is a quite difficult situation. When input data is processed we will see an object that is very close to or inside an object on the map. It is then concluded that it is the same object. The data from the frame will then be added to the object on the map.

5.4.3 Moving objects

It is important not to map anything as soon as it is seen for the first time. If this rule is followed then moving objects are not all that much of a problem.

The idea is that something has to be seen consecutively a few times before it will be mapped properly. Of course some data for the object must be stored for each frame

5.5 A quicker check

At one point we were worried about the performance issue that the mapping process takes too much time to process. This led to the making of "a quicker check".

The idea is that the mapping is **only** used while moving in an unknown environment and when what is seen by the quicker check doesn't match the map. So the Quicker Check only runs in an known environment **to ensure** that the map is still accurate.

It samples points along a few lines in the cloud (possible to use depth value instead) between 50 and 200 points on each line. These values are tested against objects and walls on the map. If something doesn't match then the normal mapping is used.

Problem is very small objects on the floor. Only solution is to run the quick check very frequently and keep one line on the floor very close to the machine and look for small changes.

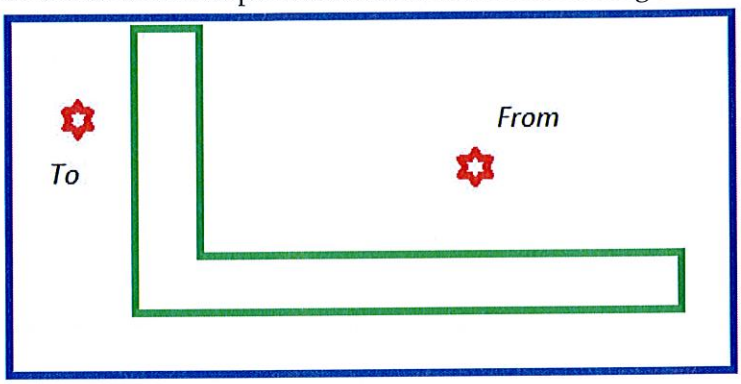
6 Path Planning (using the map)

The advantage of having a map is to be able to do more advanced path planning.

Path planning and a good map is sometimes necessary for the machine to be able to reach its destination and to do it as effectively as possible. Especially in the case of a cleaning machine or a lawn mower it is quite vital with a good path planner to ensure that all of the ground has been covered.

6.1 Navigation

There are situations where a simple robot will be unable to proceed because it is blocked along the more direct path and must "go around" the obstacle. To counter this it must have a more wholesome picture of the room.



6.2 Coverage planning

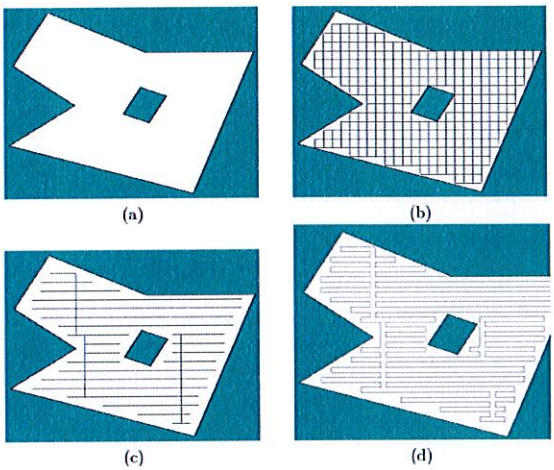
To cover the entire ground or floor with a sweeping motion has been the major part of the path planning in this project. This is because the goal of the project is a base that can be used as a cleaning machine or a lawnmower or a mine sweeper.

An existing theory for this exists and is called *boustrophedon decomposition*. It is Greek and means something like "ox-turning" that is the way a field is typically plowed in an efficient way.

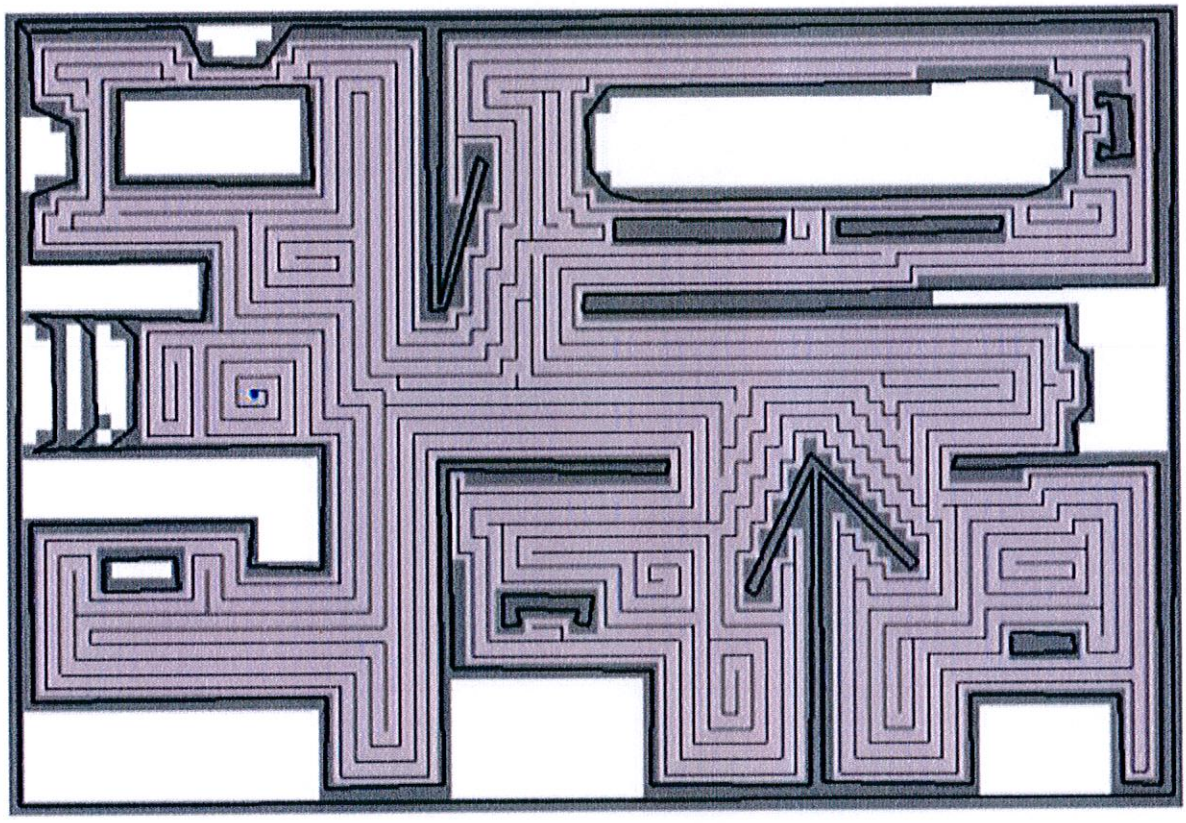
There are currently two robots which are both on a concept stadium that use a similar path planning for ground coverage. One is a lawnmower called Indego by Borsch, the other is an indoor vacuum cleaner by LG called Hom-Bot LRV5900R (discontinued) which looks quite similar to the well-known Roomba. In the advertisement videos they both seem to preform quite well, but it might not be the most unbiased demonstration. They seem quite impressive though since they don't use a real map but rather rely on SLAM.

Since we have a better map than the other two and a real positioning system this ground coverage can be made more effective in planning the path. One feature is that a return path is taken into consideration so it doesn't have to track back over already cut/cleaned ground.

There are papers that discuss optimal coverage planning for an pre-mapped area. The theory is based on making a kind of search tree over the unoccupied parts of the whole area. This will yield something like a pattern over the area that the movement can be based on. The result is fully optimal with no backtracking required. However; it only supports static maps (no change can occur) and the pattern will make the machine enter and leave a room several times before it is finished rather than staying in it until done. This probably makes it very unpractical in the case of a cleaning machine since it will cause a lot of disturbance. (LaValle, 2012)



Picture: (a) An example used for spanning tree covering. (b) The first step is to tile the interior with squares. (c) The spanning tree of a roadmap formed from grid adjacencies. (d) The resulting coverage path. - http://planning.cs.uiuc.edu/node353.html



The picture above is from an extremely sophisticated search algorithm that makes a perfect path for a cleaning machine in an office building. The line contains both how it should initially move and return path, one on each side of the line.

Requirement for this is a quite narrow machine and a completely static map.

The main disadvantage with this sort of solution is that the machine must enter and leave the same room multiple times before it is finished. This will probably annoy and disturb its occupants quite a lot. They cannot close the door when it is outside either since this will change the map.

It is not very easy to see but this is a grid map.

(Rimon, 1999)

7 Conclusions and recommendations

The final program has so far only been simulated. We do not have the final sensor as of yet and the demonstration sensor that we got to borrow does not have a driver for the board we have in the machine. However, it has been possible to validate the software by moving the sensor by hand and specify its position directly through the console. All that remains are the possible problems that may arise when more hardware comes into play. We do not believe that the vibrations will be a major problem indoors. Depending on is how much extra that the processor will be used for however it can be difficult to maintain a sufficiently high frame rate.

The requirements in the project description was to find an appropriate sensor to use for mapping. The second was to use the sensor to detect and map simple solid convex objects (e.g. box).

Sensor type (Time of Flight detector) is established and we are still discussing with the manufacturer of the sensor circuit and a manufacture.

The program is executable under Windows (development environment) and Ubuntu (verification environment) and the Panda Board (Angstrom Linux) (target environment). It can identify most types of object without any problems (aside from what has been delimited).

One of the more difficult but important goals has been to identify the process to support "dynamic objects". What this means is that some items will not necessarily always be in the same location. Items that are moved must be removed from the map or updated with new position. This was solved by simply creating a list for each frame of the object / walls that are expected to be seen in the respective frames. Has an item that should be seen not seen in a few subsequent frames, it is removed.

The objects found are also compared with the saved objects. Should the item in the respective frame to be very different from that of the map (in the sense that it takes up much more / less space) can delete the object. A margin at allowed difference is necessary when the comparison is made.

To differentiate between objects and walls in the identification is something that is no longer required. A principle for how to do this, however, that can be implemented in case of future needs.

Navigation using the map is used in the "coverage planning" part. So far, no other navigation is needed but a more general "point A to point B" principle is developed for the map type we have. However, if the map is something like a Labyrinth then this algorithm will probably not be good enough. Then a more sophisticated search tree is needed.

Recommendations:

Simulate more! It would take a terrible lot more time if the program is constantly being tested on the target hardware. Being able compile and run directly is very convenient.

Follow-up meetings as often as possible with the supervisor at the company can be very important. Even if you do not have a problem, but think they know how to proceed, the company may have other preferences.

To establish a simple and easy to overlook schedule that can be discussed each week at follow-up meeting helps both parts a lot. Partly to ensure that any deadline may be held but also to ensure that the project actually goes forward is quite nice.

The job description should be as specific as possible so both sides can see what has been finished.

8 Works Cited

http://ieba.wordpress.com/2007/11/01/rollshut-2/. (2012, 08 02).

AnandTech. (2010). *AnandTech.* Retrieved 08 01, 2012, from http://www.anandtech.com/show/4057/microsoft-kinect-the-anandtech-review/2

ANSI. (2012). *SAFETY STANDARD FOR DRIVERLESS, AUTOMATIC GUIDED INDUSTRIAL VEHICLES AND AUTOMATED FUNCTIONS OF MANNED INDUSTRIAL VEHICLES.* America: ITSDF.

Ashutosh Saxena, M. S. (2007). *Make3D: Learning 3D Scene Structure from a single still Image.*

Boost. (2012, 08 01). *Boost Documentation.* Retrieved from Boost C++ Libraries: http://www.boost.org/doc/libs/1_50_0/

Cousins, R. B. (2011). *3D is here: Point Cloud Library (PCL).* Shanghai, China.

Eigen. (2012). *Eigen main page.* Retrieved 06 01, 2012, from http://eigen.tuxfamily.org/index.php?title=Main_Page

FLANN. (2009). *Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration.*

LaValle, S. M. (2012, 04 20). *Planning algorithms.* Retrieved 07 09, 2012, from http://planning.cs.uiuc.edu/node353.html

Miloslav Richter, P. P. (n.d.). *Adapting Polynomial Mahalanobis Distance for Self-supervised Learning in an Outdoor Environment.* Czech Republic: Miloslav Richter, Petr Petyovsky, Ondrej Miksik.

OpenNI-Foundation. (n.d.). *openNI.org.* Retrieved 05 20, 2012, from OpenNI: http://www.openni.org

OpenSLAM. (n.d.). *OpenSLAM.* Retrieved 2012, from http://www.openslam.org/

QHull. (2012). *The Geometry Center Home Page.* Retrieved 07 2012, from The Geometry Center Home Page: http://www.qhull.org/

Rimon, Y. G. (1999). *Spanning-Tree Based Coverage of Continous Areas by a Mobile Robot.* Technion, Israel: Dept. of Mechanical Engineering.

ROS. (2012, 01). *Ros Operating System.* Retrieved 04 2012, from http://www.ros.org/wiki/

Schaller, C. (2011). *Time-of-Flight - A New Modality for Radiotherapy.* Germany: Erlangen.

Zuliani, M. (2012). *RANSAC for Dummies.* Udine, Italy.

9 Appendix

9.1 Visual Studio linking and C++ peculiars

How to get the PCL library working on a pc with windows and Visual Studio 2010.

Linking errors is maybe the reason why not everyone want to be a programmer. Here follows instructions on how and what to install and how to configure Visual Studio for it using prebuild libraries (there is no reason to build it yourself on windows).

All libraries are provided directly on the PCL site. Boost however is changed quite rapidly so I recommend downloading it from http://www.boost.org/users/download/.

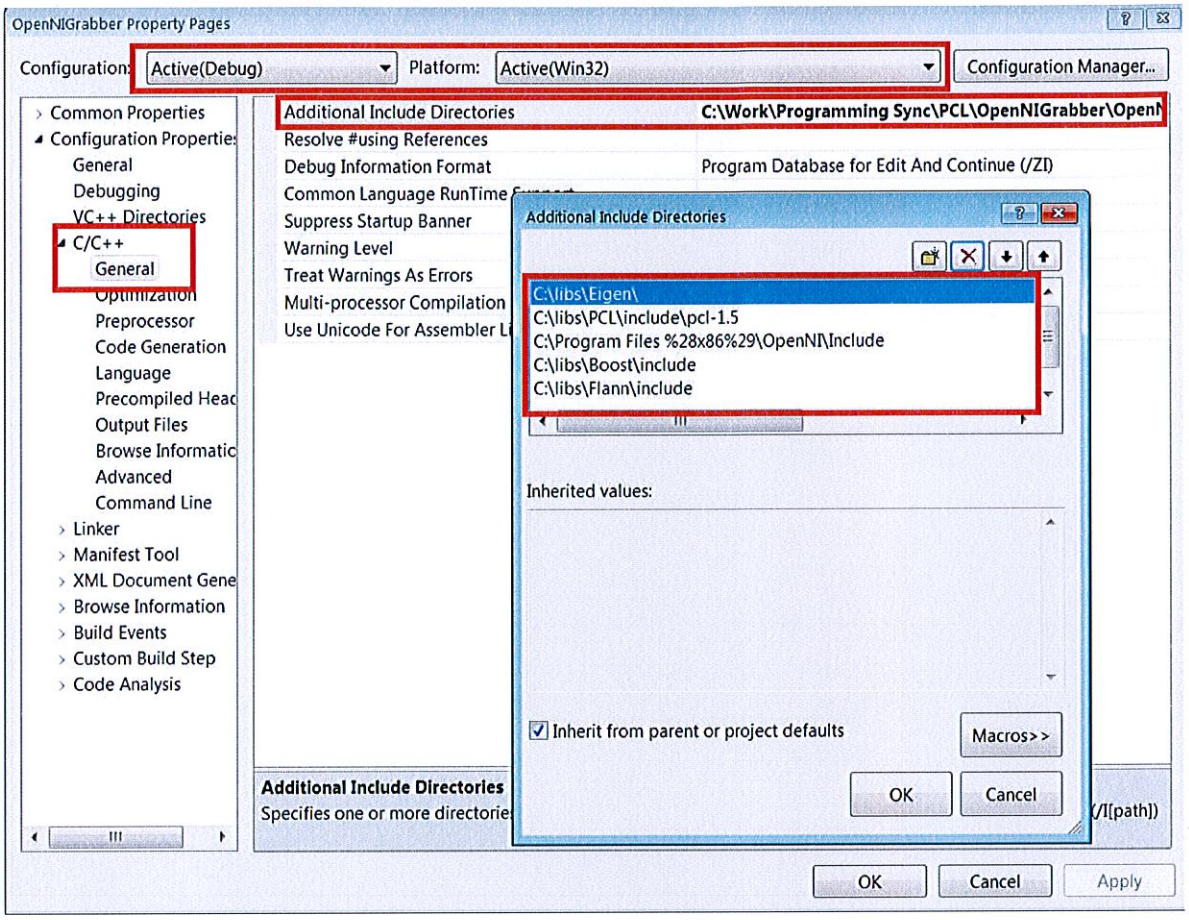
For simplicities sake I recommend using one folder where everything is put, I presume it is a folder called "libs" directly on c:\ in this tutorial.

Each of these folders (Boost-, FLANN-, Eigen-, OpenNI-, PCL-folders) will contain at least one folder called "include" and one called "lib". These are the folders that must be linked to in Visual Studio. Some of these folders are a bit stupid however, the pcl's include folder contain another folder called "pcl-*version*" and in that case this is the folder that needs to be included.

All configuration settings are reached under "Project" in the menu and then "*name* properties…" (or Alt+F7). The top most red square in picture below is very important to use the right configuration settings, I always use the debugger but if you don't then Release should be used.

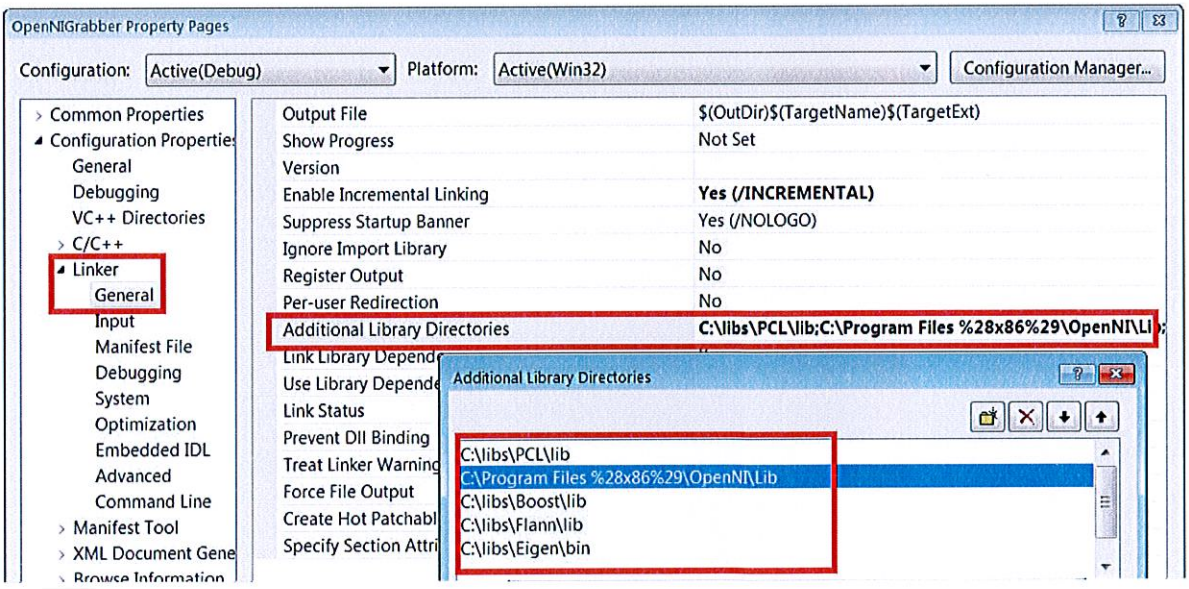
9.1.1 Linking includes

This is where include files directories are added to the project. It is only search paths that are added so no files. Include files are generally only .h files.

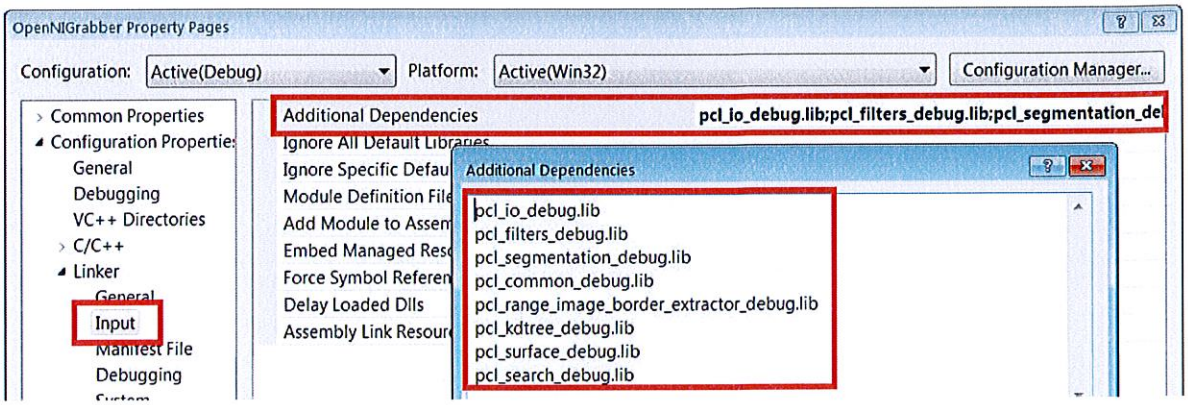


9.1.2 Linking library files

Library files has to be added at two locations. First is similar to include files:



And finally the actual library files:


OpenNIGrabber Property Pages
Configuration: Active(Debug)
Platform: Active(Win32)
Configuration Manager...
Common Properties
Configuration Properties
General
Debugging
VC++ Directories
C/C++
Linker
General
Input
Manifest File
Debugging
Additional Dependencies
pcl_io_debug.lib;pcl_filters_debug.lib;pcl_segmentation_de
Ignore All Default Libraries
Ignore Specific Defau
Module Definition File
Add Module to Assem
Embed Managed Res
Force Symbol Referen
Delay Loaded Dlls
Assembly Link Resour
Additional Dependencies
pcl_io_debug.lib
pcl_filters_debug.lib
pcl_segmentation_debug.lib
pcl_common_debug.lib
pcl_range_image_border_extractor_debug.lib
pcl_kdtree_debug.lib
pcl_surface_debug.lib
pcl_search_debug.lib

9.2 Example

Example where glut is used to create a simple OpenGL environment. OpenNI is used to setup a connection with any sensor supported by OpenNI (like the Kinect).

```
void update_cb(const pcl::PointCloud<pcl::PointXYZ>::ConstPtr &cloud_XYZ_);
```

This is the callback function used by the OpenNI grabber. Every time is has a new frame ready this function will be called. This is assuming it is not already occupied by an older frame.

```
#include <pcl/point_types.h>
#include <pcl/io/openni_grabber.h>
#include <boost/thread.hpp>

#include <GL/glut.h>

#define UPDATE_FREQUENCY 40
#define WINDOW_SX 640
#define WINDOW_SY 480
#define WINDOW_PX 10
#define WINDOW_PY 0

pcl::Grabber*myGrabber = 0;

void update_cb(const pcl::PointCloud<pcl::PointXYZ>::ConstPtr &cloud_XYZ_)
{
            // called each frame with const pointer to Point Cloud
}
void loop(void)

            //MAIN LOOP
{
            // Clear Screen And Depth Buffer
            glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
            glLoadIdentity();

            /*
                        Grid with 1 meter spacing
            */
            glColor3f     (0.5f,0.2f,0.3f);
            glLineWidth   (2.0f);
            glBegin       (GL_LINES);
            {
                        for (float inc = 0.0f; inc < 20.0f; inc+= 1.0f)
                        {
                                    glVertex3f( 0.0f, 0.01f, inc);
                                    glVertex3f(20.0f, 0.01f, inc);
                                    glVertex3f( inc, 0.01f, 0.0f);
                                    glVertex3f( inc, 0.01f,20.0f);
                        }
            }
            glEnd();

            // Swap The Buffers To Not Be Left With A Clear Screen
            glutSwapBuffers();
}
void callback_timer       (int delay)
{
            glutTimerFunc(UPDATE_FREQUENCY, callback_timer, UPDATE_FREQUENCY);
            glutPostRedisplay();
}
void init ()
{
            glShadeModel              (GL_SMOOTH);
            // Enable Smooth Shading
            glClearColor              (0.02f, 0.1f, 0.2f, 0.0f);
            // Black Background
```

```
		glClearDepth		(1.0f);

		// Depth Buffer Setup
		glEnable		(GL_DEPTH_TEST);

		// Enables Depth Testing
		glDepthFunc		(GL_LEQUAL);

		// The Type Of Depth Testing To Do
		glEnable		(GL_COLOR_MATERIAL );
		glHint			(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);
}
void reshape (int w, int h)
{
		glMatrixMode		(GL_PROJECTION );  // Select The Projection Matrix
		glLoadIdentity		();                // Reset The Projection Matrix
		glViewport		(0, 0, w, h );

		gluPerspective		(52.0f, ( float ) w / ( float ) h, 0.1, 1000.0 );
		glMatrixMode		(GL_MODELVIEW );  // Select The Model View Matrix
		glLoadIdentity		();    // Reset The Model View Matrix
}
int main (int argc, char** argv)
{
		glutInit		(&argc, argv);
		glutInitDisplayMode	(GLUT_RGB | GLUT_DOUBLE );
		glutInitWindowPosition	(WINDOW_PX, WINDOW_PY);
		glutInitWindowSize	(WINDOW_SX, WINDOW_SY );
		glutCreateWindow	("Point Cloud Vizulaicer" );
//		glutFullScreen		();          // Put Into Full Screen
		init			();
		glutDisplayFunc		(loop );
		glutTimerFunc		(UPDATE_FREQUENCY, callback_timer, UPDATE_FREQUENCY);
		glutReshapeFunc		(reshape );

		try {
			myGrabber = new pcl::OpenNIGrabber();
		}
		catch(pcl::PCLIOException error) {
			printf("No device is connected according to OpenNI!\n");
			printf("Error message is: %s\n", error.what());
			myGrabber = 0;
			return 0;
		}

		boost::function<void (const pcl::PointCloud<pcl::PointXYZ>::ConstPtr&)> f =
			boost::bind(&update_cb, _1);

		myGrabber->registerCallback(f);
		myGrabber->start();

		glutMainLoop(); // Start Gluts event-processing

		if (myGrabber!= 0)
			myGrabber->stop();

		return 0;
}
```