MASTER'S THESIS | LUND UNIVERSITY 2016

Geometric real-time modeling and scanning using distributed depth sensors

Anton Klarén, Valdemar Roxling

Department of Computer Science
Faculty of Engineering LTH

ISSN 1650-2884
LU-CS-EX 2016-16

Geometric real-time modeling and scanning using distributed depth sensors

(Detection of deviations in the surroundings for robotic applications)

Anton Klarén
anton@klaren.it

Valdemar Roxling
valdemar.roxling@gmail.com

June 5, 2016

Master's thesis work carried out at Cognibotics AB.

Supervisors: Klas Nilsson, klas@cognibotics.com
Jacek Malec, jacek.malec@cs.lth.se

Examiner: Elin Anna Topp, elin_anna.topp@cs.lth.se

Abstract

Technological breakthroughs have in recent years heavily increased the availability of different types of sensors and the amount of computational power, allowing for much more advanced robotic applications. By mounting multiple depth sensors on the robot and designing a fast and robust system with state-of-the-art algorithms we have made the robot aware of its surroundings. First, the surrounding is scanned into a static geometrical model. Then that model is used to detect and extract deviations in new scannings, for further analysis, to make suitable decisions on how the robot should act. We have demonstrated the system in a safety-critical setup for industrial robots, slowing down and stopping the robot when a human is too close. The system is designed for modularity, allowing for many other completely different applications, such as complex object tracking and motion planning.

Keywords: Robotics, Computer vision, SLAM, Depth sensors, System design, Safety

Acknowledgements

We would like to thank:

- Cognibotics AB and involved personnel for their help, guidance and providing us with necessary tools while working with this thesis and especially CEO Klas Nilsson as supervisor and mentor.
- Cognimatics AB for lending us an Intel RealSense sensor and for providing useful knowledge in the area of computer vision.
- Department of automatic control, LTH, for providing access to the robotics lab, a Kinect One sensor and practical robotics knowledge.
- Professor Jacek Malec as supervisor.
- Associate Professor Elin Anna Topp as examiner.

Contents

Chapter 1
Introduction

In this chapter we will present relevant background information, state the problem and motivate our solution. We will also mention related work and discuss how our work contributes to current research.

1.1 Robotics & Computer Vision

> **Robot** – "Any automated machine programmed to perform specific mechanical functions in the manner of a man" [1].

Robots today come in all shapes and sizes with a large amount of different tasks and duties ranging all the way from industrial manufacturing, transportation and medical assistance, to simple entertainment. In the beginning of the robotic evolution most tasks were very simple, and in most cases they still are. This is mainly because robots in general lack any awareness of their surroundings and cognitive features, but with the massive amount of computational power and the wide variety of sensors available today we see more and more "aware" and smart robots.

For a robot to be aware of its surroundings it needs, just like a person who uses his eyes and ears, to collect data and interpret it into something useful.

Robots are often large, heavy and strong which means that they can potentially harm humans crossing their path. Because of this, many robots are under heavy safety restrictions while operating so no human can be harmed. To satisfy these restrictions industrial robots are typically behind a fence which limits the possibility of human–robot interaction (HRI). Another way to meet these restrictions is to use specially designed robots with limitations on the maximum force they may output. Both of these approaches limit the range of possible applications [2].

1.2 Problem Description

Traditionally robots lack any kind of spatial awareness, i.e., they are not aware of objects and humans in its surroundings. This prevents many applications that require this kind of feature to become reality. Solving this problem by building a three-dimensional model of the robot's surrounding, many new areas, such as dynamic motion planning, object tracking and safer HRI, become possible.

1.2.1 Solution proposal

To solve the problem described above we propose a system that is able to process and merge data from multiple sensors that deliver depth images. The data provided by those sensors will be used to build a geometric model of the nearby surroundings. This model can be used to identify changes, i.e., unknown objects, in a later state. The sensors can be arbitrarily positioned on the robot as long as the position is known and good visual coverage is ensured. When changes are detected, the information will be analyzed further to let the system act according to predefined actions, such as slowing or stopping operation of an industrial robot, or plan the current path of movement for a mobile robot.

We think that it is very important for the proposed system to handle multiple different sensors, with data integrated to a common internal representation, in an efficient way to support short response time from the occurrence of an event to a reaction from the system. The system should also be robust and resource-efficient. By designing the system to be a modular platform many different types of applications, originating from different research areas withing robotics, become possible.

We also propose that the sensors should be mounted on the robot, instead of having a static position in the environment. This will make the sensors mobile as the robot moves and increase the total coverage. It will also allow the sensors to be integrated and shipped with the robot which leads to easier setup for the customers.

Is this solution even possible? Can the system be fast enough for real applications? If so, can the system be built in a modular way that can be used as a platform for future development and research? We will try to prove this by developing a platform with a fully functional safety application for an industrial robot, that will slow down and stop the robot's movements when a person gets too close for safe operation.

1.2.2 Related Work

In recent years there has been a lot of research in the robotics and computer vision areas, as the available computational power has reached a level where ideas from the early 90's and forward become reality, and many new robotic applications arrive with increasing demands. Armin Hornung et al. created a framework for probabilistic 3D mapping [3], with similar goal as ours, but using a different approach. Their focus was on minimizing memory usage whereas ours will be on minimizing the response time and maximizing the frame rate, for industrial usage. Their system has been used by Daniel Maier et al. to create dynamic path planning for a mobile robot [4].

Andreas Nüchter explores 3D mapping using SLAM algorithms to build a static geometric model, a similar approach as we use, but using LiDAR based sensors for detailed

static models [5]. Javier Minguez and Luis Montano propose other possible solutions to dynamic robot path planning [6]. Johann Prankl et al. [7] examines the possibility of tracking and identifying objects from point clouds, algorithms that can be used in an extension of our system.

The demand for safe operator-robot interaction in an industrial environment together with its challenges is discussed by Anna Kochan. She states many problems and the commercial solutions available at the time, many of which are still relevant for our demonstration. Some are: the need for close integration of the system with the robot controller, limited and safe robot-operation-speed, and vision based safety systems [8].

Esther Horbert et al. [9] use depth and color cameras to extract interesting parts of images using saliency maps that could be used for object tracking, and Xavi Gratal et al. [10] use point clouds to determine how to grasp objects. Both would be useful extensions to our system.

1.2.3 Compatibility

Much of current robotics research software is developed on a platform called Robot Operating System (ROS). This is a collection of frameworks that allows different applications to connect and communicate using a common interface to help design more advanced systems. Our solution does not implement this due to driver compatibility issues with the Linux OS, but would only need minor changes when required drivers get supported.

1.3 Contributions

Our system can be used as a platform to build features and applications for robotics that require spatial awareness. With minor changes it will be compatible with ROS and be connected to other ongoing research in an easy way. The modularity allows specific parts to be easily extracted from the system, e.g., the part that retrieve point clouds from depth sensors or the geometrical model builder.

Development of the system has been done in an agile pair-programming setup with Anton more focused on implementation and Valdemar on high level design. The work presented in this thesis has been carried out by the authors together and no contribution can be assigned to one person alone. The authors take shared responsibility for this thesis.

Chapter 2
Depth sensors

This chapter will present some techniques on how to acquire depth images, suggest some available products using these different techniques and give a comparison.

2.1 Depth Image

An image in computers can be seen as a table filled with values. This table is known as a raster and the values are referred to as *pixels*. A pixel can then be interpreted in various ways depending on the type of the values. A regular color camera will typically store Red-Green-Blue (RGB) pixels in the table. A depth image is an image that instead of the color of the objects stores the distance to them.

A *depth image* represents the distance to the visible surfaces from the sensor's point of view and is therefore sometimes called a 2.5D image. It can not be seen as a complete 3D model since the image will contain a lot of occluded regions where the structures of the objects are unknown. In order to get a full 3D view of the object some kind of prior shape knowledge needs to be applied to the data, alternatively data from multiple viewpoints can be merged to fill in the occluded areas. This is, however, not a part of the sensor hardware and therefore not part of this chapter.

2.2 Acquisition

There are a couple of ways to acquire depth images, most notably are *stereo triangulation systems* and *time-of-flight systems*. Both classes of depth sensors have a number of branches that will be described in detail. This is not a complete collection of depth image acquisition methods and the selection is based on currently commercially available products.

2.3 Stereo Triangulation

Stereo triangulation is a way to get an approximated depth reading by comparing two images with known origin. The basic principle is that objects closer to the camera will "move" more than objects far away, also known as motion parallax. Stereo triangulation can be divided into two groups: passive and active. However, they all rely on the same concept of determining the height of a completely defined triangle.

2.3.1 Passive Systems

A passive triangulation system uses two cameras to capture and compare two images from different angles. It is called a passive system since it only observes the world. Figure 2.1 shows an example of this.

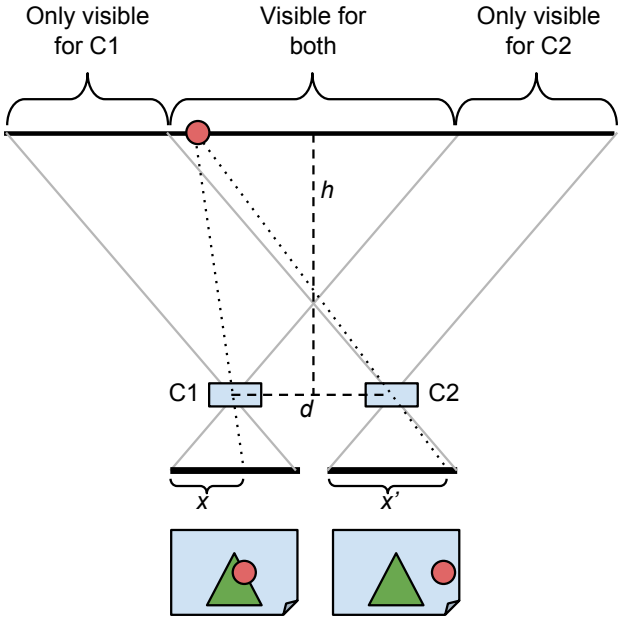


Figure 2.1: Illustration of how triangulation can be done with two cameras. Here camera C1 and camera C2 are assumed to be aligned in a horizontal setup. The height, h, can be derived by knowing the distance between the cameras, d, and the angles to the circle from each camera. The angles can be derived from x and x'. The triangle in the pictures represents a distant object with no noticeable movement.

If the distance, d, is known between the cameras, the height, h, can be calculated with

$$\begin{aligned} \alpha &= \angle \text{C1} \\ \beta &= \angle \text{C2} \\ h &= \frac{d \sin(\alpha) \sin(\beta)}{\sin(\alpha + \beta)} \end{aligned} \tag{2.1}$$

where α is the angle between the line from camera C1 to the circle and the baseline denoted by d. This angle can be derived from the x position in the picture and the camera's field of view. It is analogous to β and x' for camera C2.

Passive stereo triangulation systems require solving of the *correspondence problem*, i.e. determining that an object at x and x' is actually the same, a task that is very complex and error-prone. Solving this is computationally expensive, but there exist many different algorithms that perform this matching with varying execution time and accuracy [11]. According to [12], Semi-Global Block Matching (SGBM) is currently the fastest algorithm and is therefore a good candidate for real-time applications. However, SGBM typically contains more error in the resulting depth image.

The camera type used will also affect the accuracy of the sensor. Using color cameras with high level of detail may provide information useful in solving the correspondence problem. This can also be a disadvantage since they are very susceptible to shadows and changes in colors when perceived from different angles. Infrared cameras do not suffer from this, but they usually have a shorter range and less details. A setup with color cameras is shown in Figure 2.2.

Figure 2.2: A setup of a stereo vision system using regular color cameras and a Raspberry Pi as computational unit

2.3.2 Active Systems

An active system replaces one camera with an emitter to project a known pattern onto the world that can be used to derive the triangularization parameters. Active systems often work in the infrared spectrum in order to appear invisible to the human eye. One major drawback is that the emitted pattern might be "erased" by stronger light sources, e.g., the sun, which often limits them to indoor environments.

If the pattern is generated in a way that will allow unambiguous localization, the correspondence problem does not need to be solved and such a system is thus less computationally expensive than passive systems. However, the range is more limited compared to passive systems since the pattern will become blurred when projected at objects far away due to attenuation. This can partly be reduced by using a laser as emitter, that has a tight focus over long distances. It also emits a single wavelength that is easy to isolate. A drawback is that the rays emitted by the lasers can have high power output and could potentially damage the human eye; this safety aspect needs to be considered as well. Active systems that are used in human-computer interaction use infrared lasers to avoid blinding the users of the systems.

Active systems can determine range on flat and uniform surfaces that passive systems struggle with. Those contain very few points that can be used for correspondence point selection and the passive system will calculate the surface to be very far away. This is where the active systems excel since the surface will not destroy the emitted pattern. Active systems, for the opposite reason, struggle with edges that passive systems excels at. This is the reason that pure passive and pure active systems are rare and that most of the commercial products use a hybrid solution combining the strengths of both approaches.

2.3.3 Hybrid Systems

A hybrid system can be constructed using any of the components from passive and active systems. One example of this is the Intel®RealSense™Camera (show in Figure 2.3) that successfully combines passive infrared cameras with an emitted infrared grid. This grid is used to give details on surfaces that would otherwise be completely uniform, resulting in more accurate corresponding point selection.

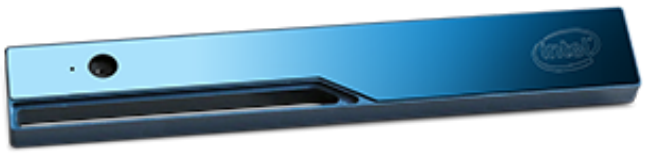


Figure 2.3: A commercial depth sensor with two IR cameras, one color camera and an IR emitter.

Source: Intel®RealSense™Camera R200.

2.3.4 Calibration

The triangulation method described in the previous sections assumes a perfect world where the cameras are perfectly aligned and capture correct images. Since this is impossible, we need some form of calibration that will transform the received data into a perfect world scenario. This calibration needs to know the camera intrinsic parameters, namely focal

length, principal point and a scale factor from focal length to pixel size. It also needs the extrinsic parameters of the camera which describe a relative position and rotation to some fixed arbitrary point. There are a lot of ways to determine these parameters, one of which is described by Andrea Fusiello et al. [13].

When all parameters are known a linear transformation can be created that will make all epipolar lines parallel to the horizontal axis and make all corresponding points have the same vertical coordinates; this transformation is called rectification. Since all corresponding points will have the same vertical alignment the matching algorithm only needs to search in one dimension to find the most probable corresponding point, speeding up the algorithms immensely. Commercial products usually perform calibration during assembling of the device, and no further calibration is needed.

2.3.5 Disparity Maps

It turns out that it is enough to save the difference in horizontal position of the corresponding points. If this is done for all pixels in the images the resulting picture is referred to as a disparity map. The depth image can then be calculated from the disparity map since the values in the disparity map are inversely proportional to the depth images.

The calculations from Equation 2.1 can thus be simplified to

$$h = \frac{c}{x - x'}$$

where c is a constant that can be derived from the camera intrinsic and extrinsic parameters. This observation saves enormous amount of calculations since c can be calculated beforehand, allowing passive and active systems to run at very high framerate on limited hardware.

2.4 Time-of-Flight Sensors

Time-of-flight (ToF) sensors utilize the fact that the speed of light is constant and that light reflects off surfaces. By measuring the time it takes for the light to perform a full round trip, from an emitter to a receiver, the distance, d, can be calculated with

$$d = \frac{c_{atm} \cdot t}{2}$$

where c_{atm} is the velocity of light in atmosphere and t is the time of a full round trip.

2.4.1 Laser Sensor

A laser based depth sensor works by sending out a laser pulse in a known direction and measuring the time of the round trip. This process is then repeated in a grid pattern until a complete depth image is filled. This yields very accurate and precise readings at the cost of acquiring time since all pixels in a depth image need separate round trips. The time it takes to capture a complete image therefore grows with image size and maximum range.

Since the image is captured during a longer time frame, moving objects will suffer from discontinuities because they can move significantly between the first and last reading, this

Figure 2.4: A commercial LiDAR system

Source: Sick™S3000 Professional.

is known as tearing. This makes them hard to use in dynamic environments but it still has a market for the static counterpart where accuracy is favored over speed.

This technology is often refereed to as "Light Detection And Ranging" (LiDAR) and Figure 2.4 shows a commercial product implementing this.

2.4.2 Phase-shift sensors

A phase-shift (PS) sensor acquires a complete depth image in a single shot and is therefor very good for capturing dynamic environments. This is accomplished by using a radio frequency-modulated (RF-modulated) light source together with a phase detector as presented by [14]. Such systems have an illumination unit that emits modulated light and an image sensor that captures the phase shift of it. They usually work in the infrared spectrum, same as with active stereo triangulation systems, to avoid disturbing humans.

The system works by sending out a signal

$$s(t) = \sin(2\pi f_m t)$$

where f_m is the RF-modulated frequency. The signal will be reflected back from targets and received as

$$r(t) = R\sin(2\pi f_m t - \phi) = R\sin\left(2\pi f_m (t - \frac{2d}{c_{atm}})\right)$$

with a phase shift, ϕ, that will be proportional to the distance traveled, d, at the speed of light in atmosphere, c_{atm}. The distance d can thus be calculated from

$$d = \frac{c_{atm}\phi}{4\pi f_m}$$

Since the emitted light is modulated we will get aliasing for all distances further away than $c_{atm}/2f_m$, the maximum unambiguous range is thus determined by the modulation

frequency. However, since there are a finite number of detectable phase shifts, there will always be a finite number of depth levels that can be detected. Lowering the frequency will yield greater range but less details.

An implementation of this technology can be found in Kinect for Xbox One, Figure 2.5, released by Microsoft in 2013. Although primarily intended for human skeleton tracking this device offers an open API where the raw depth images can be obtained.

Figure 2.5

Source: Microsoft™Kinect®for Xbox One™.

2.5 Comparison

Figure 2.6 contains an illustration of how the different methods compare against each others in terms of depth image resolution and capturing speed. As previously stated LiDAR systems are very accurate but they suffer from slow acquisition speed since they can only measure one pixel at a time. The figure also shows that depending on the algorithm used to solve the correspondence problem passive stereo triangulation systems can be made either accurate or fast.

PS systems and active stereo triangulation systems work fast and can thus deliver images with a high frame rate. However, they are weak in sunshine where active stereo triangulation systems are close to useless.

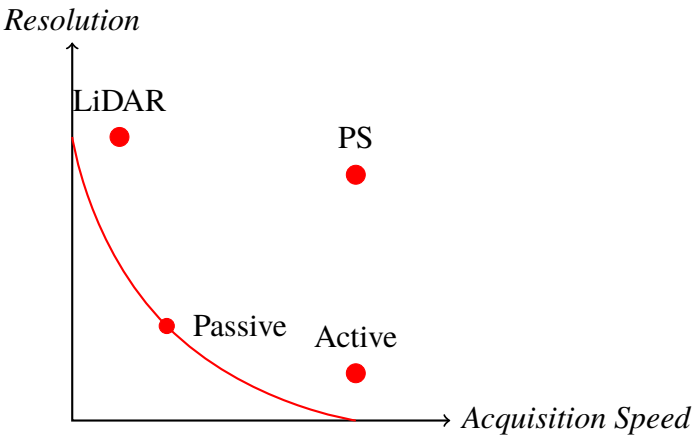


Figure 2.6: Comparison of resolution and acquisition speed between 4 different depth sensor techniques. Laser scanner (LiDAR), time-of-flight phase-shift (PS), passive stereo triangulation (Passive) and active stereo triangulation (Active). Since the speed of Passive is dependent on the chosen algorithm it is represented as a line.

A comparison of the final depth images can be seen in Figure 2.7. All images have the same number of pixels and differences are very apparent. The passive stereo image (left) is captured during perfect light conditions and gives a reasonable depth image. There are a few regions where the algorithm has failed to find correspondence points, indicated by black pixels. Another notable defect in the left image is the black border in the lower right corner that is an artifact from the rectification process; that part of the image is not visible by both cameras.

The center image is from the RealSense sensor and the limited range due to infrared cameras makes the wall "invisible", i.e., completely black. This image looks worse than the left but this can be fixed with filters that smooth out the black regions.

The last image to the right is captured by Kinect One and has a resolution typical for ToF systems. The main difference in quality stems from the fact that a ToF system delivers actual measurements of the distance for all individual pixels, where the stereo triangulation system mostly consists of interpolated values in between the estimated depth at the correspondence points.

The major drawback with current PS systems is that the sensors are large and bulky compared to the current stereo triangulation counterparts. This could be fixed with technological advancements but is not a reality yet, so if size is of concern PS systems may not be viable.

Figure 2.7: Depth image comparison. Left: Passive stereo with color cameras, Center: RealSense (Hybrid stereo with infrared cameras), Right: Kinect One (time-of-flight, phase-shifting)

2.6 Conclusions

Given the problem described in the previous chapter, the solution will require a sensor that is able to deliver images in a high frame rate. This is mostly due to the fact that it will be used in a safety setup and short response time is critical. This will exclude LiDAR and most passive triangulation systems from being viable options.

The best candidate for the proposed solution is a PS system, but the bulky size will limit the possible mounting locations on the robot. A hybrid triangulation system could be used, preferably running the SGBM algorithm, that has a smaller size and still reasonable resolution. Although the center image in Figure 2.7 has many black regions, these will eventually be filled with data as consecutive frames are merged over time, reducing this shortcoming.

Chapter 3
Theoretical background

Here we will present necessary theory to understand our solution, discuss different aspects, advantages and drawbacks to motivate our choices.

3.1 Uniform data capture and representation

As different sensors operate with different techniques they also provide various types of data, and to integrate each sensor a uniform capture interface and data representation is required. Geometrical models can be represented in a few different ways in computers, where having a set of vertices and indices form triangles, which combine into a solid mesh, is the most common way. The problem with this representation is that it requires knowledge of the structure of the objects, which a sensor normally does not provide. Another way to represent the geometry is to just have points, and ignore the structure of the objects. This is a more natural representation for the data provided by the sensors, to let every individual pixel represent one point, and the entire image will create a point cloud. A comparison between these two representations can be seen in Figure 3.1.

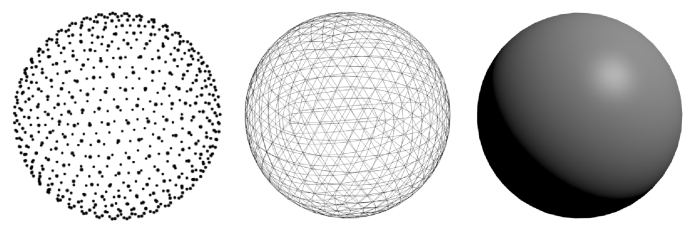


Figure 3.1: A model of a sphere can be represented in different ways. Left: Point cloud, Center: Triangles, Right: Solid mesh

3.1.1 Point Cloud

A *point cloud* can consist of millions of points, where each point is represented by its coordinate, and, in our case, a color.

Euclidean space & Cartesian coordinates

Coordinates are expressed in a defined space, and the most common is the orthonormal euclidean space, where each dimension provides one coordinate. We can represent a point in a three dimensional euclidean space with cartesian coordinates, (X, Y, Z) , where each coordinate is a distance along its corresponding axis, originating from a given base point, or origin.

Depth projection

From a combined two dimensional depth and color image provided by a sensor, a three dimensional point cloud can be acquired by multiplying each pixel and corresponding depth with an inverse camera projection matrix P^{-1}, as can be seen below, and illustrated in Figure 3.2.

$$(I_x, I_y, depth) \cdot P^{-1} = (W_x, W_y, W_z)$$

P is determined by the sensors horizontal and vertical field of view, the lens characteristics and other properties, I_x, I_y is the pixel coordinate and W_x, W_y, W_z is the world coordinate.

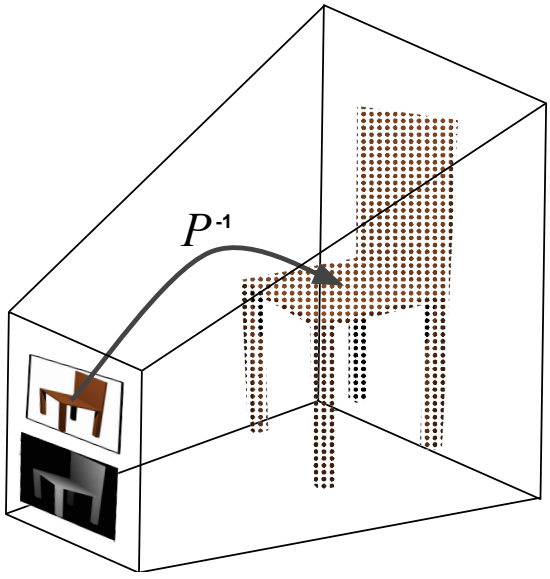


Figure 3.2: A matching color and depth image captured by a sensor can be projected to a three dimensional point cloud by applying a transformation matrix to each pixel in the image.

World space & Camera space

The coordinate of a point is determined by which base frame it is expressed in, as simply illustrated in Figure 3.3, where the point is located at the same spot, but with different coordinates, depending on the origin.

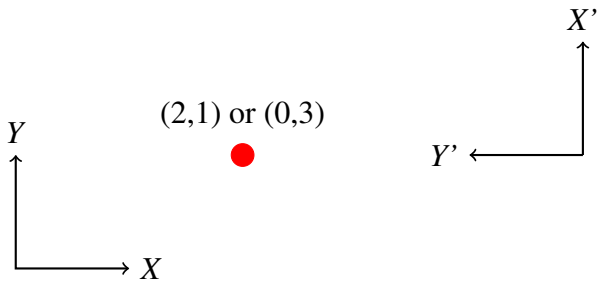


Figure 3.3: A point has different coordinates depending on which base it is represented in.

A single depth image from a sensor only provides a height model, where all the points have the same reference, called *camera space*. To get a more complete model many depth images need to be combined, with a common reference base frame, called *world space*. When a new data set is provided it has to be matched into the current model and all points need to be transformed from its own camera space into the common world space.

3.1.2 Point density

A sensor provides a certain point density at a given distance, as can be viewed in the formula below.

$$Pointdensity = \frac{resolution}{4 \cdot distance^2 \cdot \tan(\frac{hfov}{2}) \cdot \tan(\frac{vfov}{2})} \tag{3.1}$$

The numerator represents number of points in the depth image from a given sensor and the denominator is the area of the captured surface at a given distance with defined horizontal and vertical field of view.

3.1.3 Sample Size

A sensor with the resolution 500×500 pixels gives a point cloud consisting of $250,000$ points, and if operating at 30 Hz it generates 7.5 million points per second, which is much even for a computer and has to be reduced.

Noise reduction with statistical outlier removal

Since no sensor is perfect with 100% correct values in every frame the first task is to reduce the amount of noise in the cloud, by figuring out which values are incorrect, and remove these from the data set. This is a very difficult task to achieve without spending too much computational time and power.

It can, however, be achieved by applying a statistical filter that calculates the mean distance and its standard deviation between each point and its nearest neighbors, and then compares this to the entire cloud's global mean distance, and standard deviation. If it exceeds a given threshold, the point can be removed, and the cloud will be cleaned of most outliers.

Keeping features with voxel filter

When reducing the amount of points to a more manageable size it is very important that the key features of the cloud are preserved. A big flat surface can be represented with relatively few points, but a complex object might need a more dense representation. By clever selection of which points to keep, a huge amount can be discarded without losing almost any information.

A very efficient way to reduce the size is by using something called *voxel filtering*. This filter divides the cloud into a grid of equally sized boxes, called voxels. By keeping only the centroid of all points inside each box the filtered cloud size will depend on the size of the voxels.

By aligning a voxel filter in camera space, and let the voxels have a much thinner depth than width and height, the flat surfaces perpendicular to the camera, with much redundant information will end up in the same voxel whereas steep and non flat surfaces with almost no redundant information will end up in different voxels. This will efficiently reduce the size of the original cloud while retaining most of its features.

3.2 Geometrical scanning

To build a complete geometrical model we need many partly overlapping point clouds captured at different angles and locations and then figure out how to fit them together. Every cloud is captured in its own camera space, and by finding corresponding parts in other clouds they can be merged together, forming a more complete model, as illustrated in Figure 3.4. This is a process called registration.

3.2.1 Point cloud matching

Merging a stream of incoming clouds to an ever growing model is a technique called matching, and can be used when performing Simultaneous Localization And Mapping (SLAM). Each new cloud has to be registered into a current model under a limited amount of time before the next cloud can be processed.

Iterative closest point

A well known solution to the registration problem is the *Iterative Closest Point (ICP)* algorithm [15]. Given an initial guess about how the new cloud fits the model it tries to find nearby corresponding points and iteratively minimizes the root mean square distance between the correspondences until it converges. If no, or very few corresponding points are found, the algorithm can not converge and the cloud has to be discarded as it could not

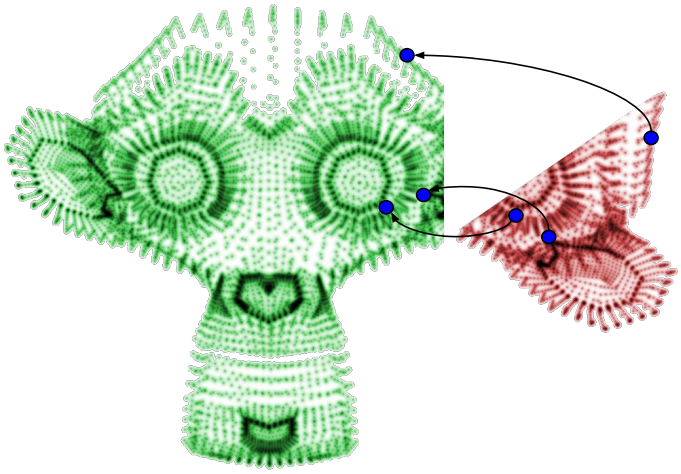


Figure 3.4: Correspondences found in two partly overlapping point clouds.

be matched to the model. The sum of the root mean square distances is also known as the fitness score.

False correspondences

If the incoming cloud is not a perfect reproduction of what it tried to capture, but contains noise and other errors there is a risk that the ICP algorithm finds false correspondences, points that look to be the same, but are not. In this case the registration will be a little erroneous and the final model will not be perfect.

In most cases every registration will create this error as there does not exist any sensors or filters good enough to fix the problem. In a big model consisting of many registrations the error will add up resulting in a phenomenon called drift, as can be seen in Figure 3.5.

These problems can be corrected by post process loop-closing algorithms, like LUM [16] and ELCH [17], which are not explained in detail here.

3.3 Deviation detection

Once a complete static model is obtained by thorough scanning of the surrounding environment it can be used to detect changes by matching new incoming clouds to the model and see where they differ.

3.3.1 Matching

A robot often has built in sensors to measure its current position and orientation. By combining this data together with the relation to the attached depth sensor, new incoming clouds can be matched to a big complete model in an efficient way.

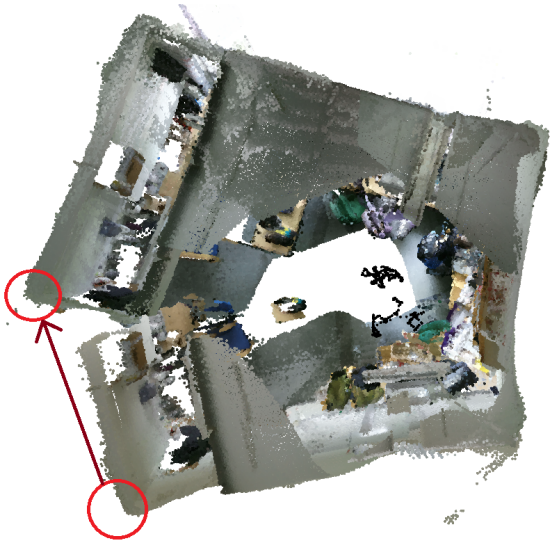


Figure 3.5: Accumulated error from around fifty registrations resulting in heavy drift from a 360° scan of a rectangular room, with drifted corners marked with circles.

Mobile Robots

A mobile robot is often equipped with many other types of sensors, like accelerometer, gyroscope and GPS that provide data for odometry algorithms, a way to estimate the robot's position and orientation. This estimation can be handed to the ICP algorithm to make the final adjustments of the position.

Industrial Robots

An industrial robot often has a precision down to a few millimeters or less and has a well known kinematic model. With this information, together with the position of the mounted sensor and the robot's current state, a very accurate value of the current position can be obtained by a simple model of linear transformations as illustrated in Figure 3.6 and described below.

$$T_{SCP} = T_1 \cdot T_2 \cdot \ldots \cdot T_{n-1} \cdot T_{SMP}$$

T_{SCP} is the sensor's current world position and orientation, T_{SMP} is the transformation of the mounted sensor, T_1 to T_{n-1} is the robot kinematic transforms and n is the number of the link where the sensor is mounted. This type of calculation is called a *kinematic chain*.

The positions of all the mounted sensors can be calculated in a fast and efficient way, and the incoming clouds can be transformed accordingly to match the complete model, and no further algorithm is needed.

3.3.2 Region of interest

When the cloud is successfully positioned we can extract each point that is at least at a certain distance from the static model, and call this the Region Of Interest (ROI). These interesting areas can then be examined in detail by advanced algorithms that would not work on a full-sized cloud within reasonable computational time.

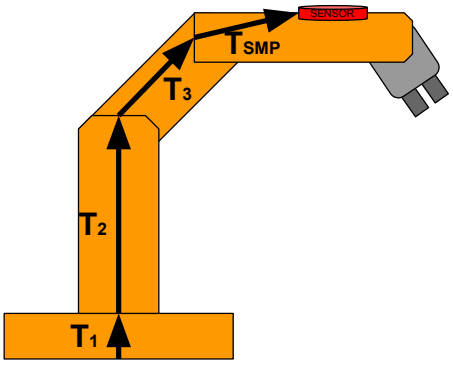


Figure 3.6: Kinematic transformations describing the current position of a mounted sensor on an industrial robot.

From the region of interest we can also inverse-project each point back to the source data and gain a full resolution color and depth image masked to contain only interesting parts. Then it is possible to use well known image analysis algorithms, as for instance object tracking and identification.

False positives & negatives

When we have a region of interest it is very important to determine whether it is an actual physical change that has to be handled, or if it is caused by noise or errors and can be ignored. Missing a physical change can be devastating in a safety application whereas giving false alerts in a production environment is very undesirable, so this has to be handled carefully.

3.3.3 Sensor positioning

A point cloud generated from a captured depth image will only contain information that is visible from the sensor's point of view. By mounting the sensors on the robot instead of having fixed positions in close proximity to the robot cell, each sensor will not only cover bigger areas as the robot moves, but is not either risking to be obstructed by poorly positioned nearby objects.

Occlusion

With a limited amount of sensors there will always be areas where no sensor is currently scanning, as illustrated in Figure 3.7, where the robot is causing self-occlusion on one sensor. A consequence of this is the possibility for objects to move closer without the system registering this which can cause trouble in some applications.

By increasing the amount of sensors and/or positioning them cleverly according to the current robot program another sensor may cover most of the self occluded areas. By having partly-overlapping fields of view the system becomes more redundant and reliable.

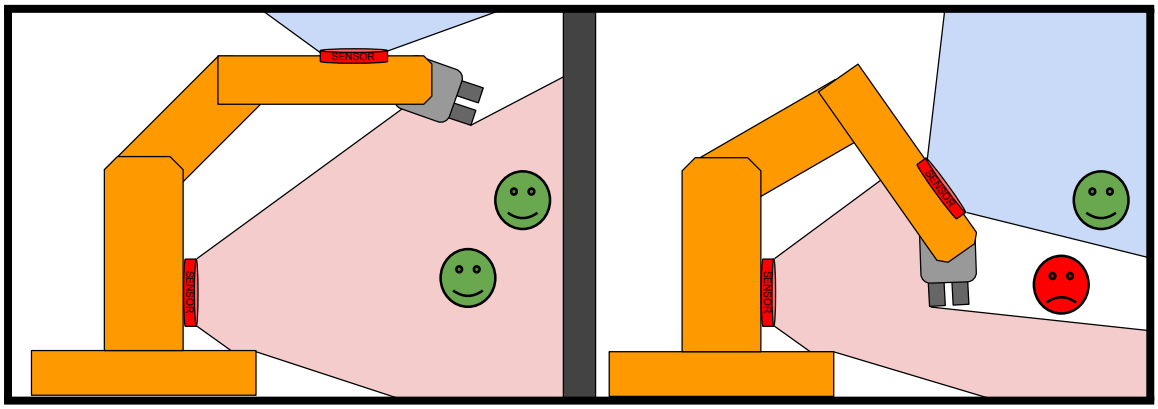


Figure 3.7: A robot with only two mounted sensors may cover all important areas (left), but may miss some due to self occlusion (right). Smart positioning of the sensors can reduce this problem.

3.3.4 Response time & Reacting

For this kind of system to be useful in a real scenario the response time, i.e., the time from when the change happened to the moment when the system has successfully identified it, has to be very short, depending on the application. However, what happens after this point is entirely up to individual applications.

3.4 Parallelization

Modern processors have the ability to execute multiple tasks simultaneously on separate physical cores. This opens up for big performance improvements if the software can utilize this feature. By extracting independent parts of the program in to separate threads these can operate much faster as they do not need to be run in sequence.

Synchronization

Sometimes separate threads need access to the same data, called shared resources. This can potentially cause disaster if not handled properly, as the program can crash or cause undefined behavior. When a thread is accessing a shared resource no other thread can be allowed access if either one, or both, intend to write data to it. This can be enforced by using a mutual exclusion semaphore, often called mutex, as a lock, or to use atomic compare exchange synchronization instructions. If a thread tries to access a shared resource already occupied, or awaits the result of a computation from another thread, it can be put to sleep and awakened when the resource no longer is occupied.

3.5 Motivation

The most convenient representation for geometrical models in the proposed solution is point clouds. They can easily be retrieved from depth sensors using projection and reduced to manageable size using statistical- and voxel-filters. There also exists algorithms to build complete geometrical models from point clouds, which can be used as static and dynamic models for deviation detection.

Chapter 4
System architecture

In this chapter we will describe our solution in more technical detail, present some major design choices and discuss the importance of these.

4.1 Introduction

The system is designed to handle multiple connected sensors simultaneously. It will retrieve incoming depth images and create corresponding point clouds of manageable size using depth projection and filtering. The point clouds are then processed in different ways depending on the current mode; building or matching. In building mode, algorithms such as ICP and LUM are used to generate a static world, as described in Section 3.2. In matching mode, algorithms from Section 3.3 are used to detect and extract data that deviates from the static world. The extracted data from each individual sensor is then merged and evaluated by the system to make a decision on how to act, e.g., slow or stop the robot in a safety application.

The system also has a graphical user interface that displays the static world along with the current ROI and the robot's position. It also handles user input during construction of the static world, such as starting/stopping the current scan as well as evaluating the result before storing it.

Since many of the ideas and algorithms we use have very high complexity and the big amount of data generated by our sensors needs to be handled within a very short time we chose to write the system in `C++`. This allows us to utilize the processing power with minimal overhead while still allowing for advanced design choices and relatively high developing speed. The system is currently bound to the Windows operating system due to sensor and USB3.0 driver compatibility issues but uses almost no further Windows-specific features and may be ported to other platforms featuring a full `C++11` compiler without much effort. A small part is written in `C#` due to available libraries and a few other system related services are written in `C` that are compiled to a real-time environment.

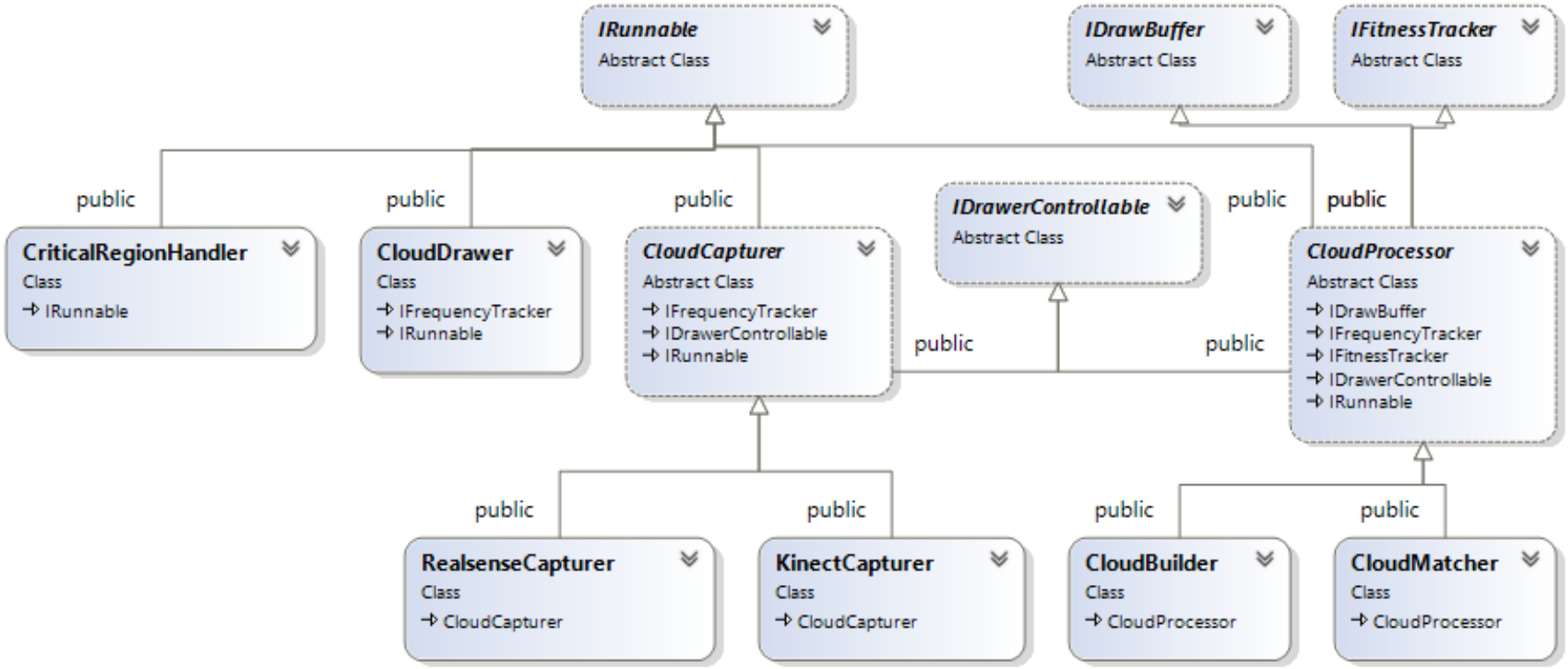


Figure 4.1: UML diagram showing the class hierarchy and abstraction levels of the `CloudCapturer`/`CloudProcessor` part of the system.

To speed up development we have made extensive use of the Point Cloud Library (PCL) [18] as it offers efficient implementations of some of the algorithms we need, also providing useful data types and tools for visualizing point clouds. Additionally, we have used **Boost**, a C++ library for efficiently implemented data structures and thread synchronization designs, as well as **Eigen**, a library for algebraic mathematics. The system has also been analyzed with **Visual Leak Detector for Visual C++** to detect bad memory management and unwanted leaks.

Totally the system consists of around 3500 lines of code. The main class and interface structure can be viewed in the UML diagram in Figure 4.1. The system consists of `CloudCapturer`:s that are classes responsible for retrieving the clouds from the corresponding depth sensor and `CloudProcessor`:s that are classes responsible for analyzing the clouds. There is one `CloudDrawer` that provides the user with a user interface and a visualization of the clouds currently processing.

There are a number of abstract helper classes that group together shared functionality. They are:

`IFrequencyTracker` adds functions and methods for tracking the frames per seconds in a thread-safe way. (Not shown in the figure)

`IRunnable` handles all thread functionality.

`IDrawerControllable` provides callbacks from the user interface so that classes implementing this can react to user input.

`IDrawBuffer` contains functions to add draw-calls to the drawing buffer. This buffer is needed since there are multiple threads that should be able to present data to the user.

Apart from the classes shown in Figure 4.1 there exist classes that handle configurations files and communication with an industrial robot, but since they are specific to the demonstration described in Section 5 they are not included as part of the base system that this chapter focuses on.

The `CloudCapturer` is coupled with a `CloudProcessor` and those always exist in a pair as illustrated by Figure 4.2. All `CloudCapturer`:s and `CloudProcessor`:s run in separate threads to support an arbitrary number of these pairs, where the limit is defined by the hardware ability to execute parallel threads. The `CriticalRegionHandler` merges signals from all the `CloudProcessor`:s to give a uniform response from the complete system.

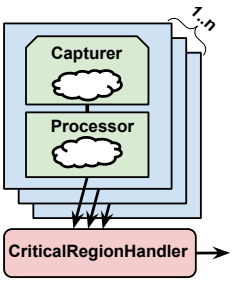


Figure 4.2: `CloudCapturer` and `CloudProcessor` pairs signals `CriticalRegionHandler` that is responsible for outputting a uniform response, e.g. removing duplicate signals.

4.2 CloudCapturer

The responsibility of the capturer interface is to convert incoming sensor data to the internal point cloud representation and make it available to the coupled `CloudProcessor`.

Data retrieval

Data is retrieved from a sensor by communicating with its hardware driver and extracting the raw depth image once it is available. A depth projection with a transformation matrix provided by the driver, as described in chapter 3.1.1, is then performed to create the cloud in camera space.

Each implementation of a capturer is different and depending on the Application Programming Interface (API) provided by the manufacturer. This abstraction layer will however make it easier to use various sensors since all sensor specific code is contained in one class and no other modifications to the system is needed. We have successfully implemented capturers for the Intel RealSense R200 and Microsoft Kinect One sensors.

Maximal throughput

Each sensor delivers new frames at a certain rate, often thirty, sixty or ninety per second. It is therefore very important to complete the depth projection before the next frame arrives, to avoid discarding of potentially important data from the sensor.

To avoid slowdowns introduced by thread-safety operations it is necessary to minimize synchronization time. This is accomplished by pre-allocating three point clouds; one cloud is used by the capturer for new data, one cloud used by the `CloudProcessor` during processing and the last cloud holds the latest completely captured cloud.

This design makes the system more predictable in terms of response time since no new memory allocations are needed during runtime. It also reduces the lock-time to fast pointer swaps, but at the cost of memory since an extra cloud is needed. The added memory overhead is negligible since the clouds are rather small compared to the whole system size. Reduced lock-time will yield better thread utilization and reduce the overall response time.

A simple illustration can be seen in the overview in Figure 4.3, and a sample pseudocode implementation can be viewed in Appendix A.1.

If the `CloudProcessor` manages to complete its computations before the capturer delivers a fresh cloud it should not work on an old frame. This problem is solved using a simple signaling semaphore, stalling the `CloudProcessor` until a new cloud has arrived

If the `CloudProcessor` is working at a much slower pace, clouds will be overwritten by the capturer. This might be seen as a waste of computational power, however, stalling the capturer would mean that the `CloudProcessor` will get an “older” cloud that what could have been available. Since the system strives to minimize response time this stall would introduce a source of delay not acceptable in this kind of system.

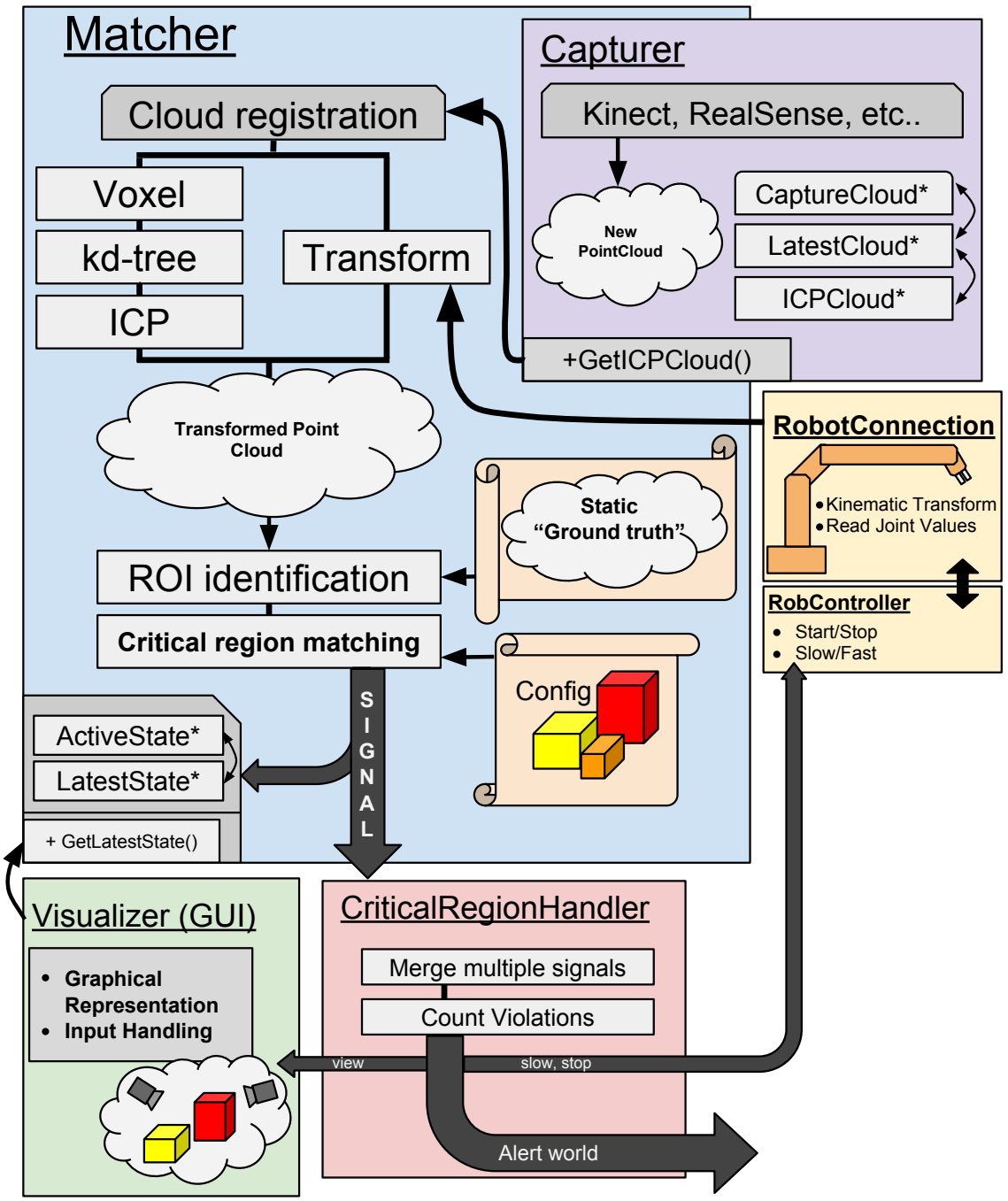


Figure 4.3: Overview of system design. A Capturer block handles an individual sensor, and converts data to internal representation. Each Matcher performs computations on incoming point clouds, the CriticalRegionHandler keeps track of interesting regions and deviations, and the Visualizer handles the graphical representation of the current state. The RobotConnection is providing the transform if the system is currently running with an industrial robot.

4.3 CloudProcessor

The `CloudProcessor`'s duty is to grab the latest cloud from the capturer, perform calculations depending on the state of the system, and make decisions on the outcome of the calculations. It can be viewed as the brain of the system because it contains most of the algorithms presented in Chapter 3. In the current state there exist two versions of the `CloudProcessor`, one handling geometrical scanning and another handling geometrical matching, as described in Section 3.2 and 3.3.1 respectively. They share much in common and work in a similar way.

4.3.1 CloudBuilder

The `CloudBuilder` is used to create the static world used during ROI extraction. This is done by estimating the location of the sensor and merging the cloud into a final cloud. The estimations are performed by running the ICP algorithm on incoming clouds and appending the new cloud to the final cloud only if the algorithm converges and the resulting fitness score is below a given threshold. To reduce the size of the final cloud heavy filters are applied during the whole process. Pseudo-code of this can be found in appendix A.2. This is a simple version of SLAM and may result in errors discussed in Section 3.2.1.

4.3.2 CloudMatcher

The `CloudMatcher`, or just matcher, is the most important part of the system and all data dependencies can be seen in Figure 4.3. The figure illustrates a complete running system and how all the parts are interconnected.

The matcher also has *critical regions* that are geometrical shapes, and the matcher will send alerts if any sensor registers data inside a shape. The matchers' responsibility is thus to take a new cloud from the capturer and match it against the static world. All sections of the new cloud that does not conform to the static world are marked as ROIs. The ROIs are then checked against a database of critical regions and potential violations are sent to the `CriticalRegionHandler`.

Figure 4.3 also shows that the matching can be done with two different pipelines; one that uses ICP to align the cloud and one that retrieves the transform based on a kinematic chain.

4.4 CriticalRegionHandler

One problem that arises from having multiple `CloudProcessor`:s performing deviation detection is to determine when an output signal should be active or not. If one sensor registers a violation of a critical region and another one do not, the signal should still be active.

This is solved by using a light switch pattern where the first sensor to register a violation activates the signal and the last sensor to unregister a violation deactivates the signal. Appendix A.4 contains pseudo-code to illustrate this. This is implemented in the `CriticalRegionHandler`.

4.5 RobotConnection

`RobotConnection` provides an abstraction level for communicating with industrial robots. It provides means of retrieving joint values, calculating kinematic transforms and issuing commands to the robot system. This is used to calculate the positions of the mounted sensors from the kinematic chain as well as controlling the operation speed of the robot.

4.6 Visualizer

The visualizer is only responsible for the graphical representation of the system and handling user input. It runs on a separate thread and collects point clouds and the other information from each matcher in a similar way as the capturers triple-pointer implementation and receives signals from the CriticalRegionHandler. This implementation allows us to visually view the state of the system without hindering the execution speed of any other part of the system.

4.7 Performance

To achieve the necessary responsiveness every part of the system needs to be designed with performance in mind. The work needs to be evenly distributed among available processor cores and no time can be wasted on synchronization or waiting. Most memory allocations should be managed during start up and the point cloud sizes should be reduced with clever filtering to avoid unnecessary computations and wasted capacity.

Threading & Processor utilization

By identifying and extracting parts to run in parallel and minimizing lock time, using lock-free data structures and other smart design patterns the overall performance of the system increased, and heavily reduced the response time to an acceptable level.

The total number of threads used by the system can be seen below.

$$2 \times NumberOfSensors + CriticalRegionHandler + Visualizer + RobotConnection$$

only one instance of the CriticalRegionHandler, the Visualizer and the RobotConnection exits

Each sensor uses one capturer and one matcher, whereas only one instance of the CriticalRegionHandler, the Visualizer and the RobotConnection exits. For optimal performance the amount of available processor cores should exceed the number of threads used by the system.

Framerate limitations

Since each part of the system is depending on data from another part, performance bottlenecks may affect big parts of the system and increase response time. The entire system can not work faster than the sensor can capture data, and the matcher is depending on new

data to make new decisions, and the CriticalRegionHandler or Visualizer has to await new states and decisions from the matcher. The entire chain can be viewed below.

$$\left.\begin{matrix} Visualizer \\ CriticalRegionHandler \end{matrix}\right\} \leq Matcher \leq Capturer \leq SensorDeliverySpeed$$

Real-Time

A real-time application or system must be able to guarantee response within a specified time constraint. This is very important especially in a safety-critical environment and robotics, but also in many other industrial cases. To achieve this the system has to fulfill two major requirements. First of all it has to run on a specific real-time operating system with certain properties. Secondly it needs to have deterministic execution time. C++ is a suitable choice for real-time systems, but the Windows platform is not, and our system would need some modifications to gain this property as previously mentioned.

Deterministic algorithms

The entire system is deterministic, with a worst case execution of fixed length, except the iterative closest point algorithm described in Section 3.2.1, as it is iterative and convergence can not be determined. This algorithm is however only used by mobile robots to calculate the position of incoming point clouds, as opposite from industrial robots where the position is determined by the current kinematic transformation chain.

Chapter 5
Evaluation

In this chapter we will present a demonstration setup featuring the most important parts, evaluate the system from a performance aspect and discuss the results.

5.1 Demonstration

To demonstrate our solution and see whether it fulfills the requirements we chose a safety critical application on an industrial robot, designed to test different aspects of the system.

5.1.1 Objective

At first the robot cell will be scanned with the sensors. Then the system shall detect nearby unknown objects and signal the robot to slow down or stop if an object gets close, and resume operation when the object is moving away. The response time of the entire system must be short enough to react on normal human behavior before any harm can be caused. The system shall also give a consistent result as objects move around and the amount of false positive signals should be low or negligible.

5.1.2 Setup

To achieve the objective we created a demonstration and captured the result with video and images, as can be viewed in Appendix B.

Robot

The most commonly used industrial robots are serial robots. We chose a medium-sized ABB IRB 2400 robot, as illustrated in Figure 5.1. It consists of six links enabling movement in six degrees of freedom, allowing it to reach any point in space with arbitrary rotation within its working range.

To connect the robot with our system we use both the provided API, called PC SDK for IRC5, to communicate operations such as movement speed control and starting/stopping the current robot program. In addition to this we use an external service to provide us with the robots current position in real-time. This is used to obtain the kinematic transforms used in calculation of the sensor's current position.

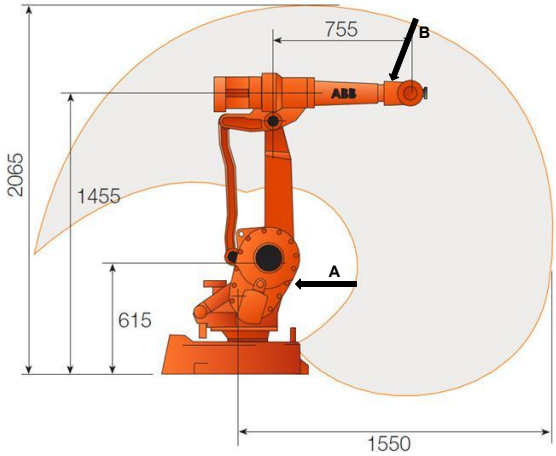


Figure 5.1: Illstration of the IRB 2400 industrial serial robot with measurements and vertical working range. The location of the mounted Kinect One sensor marked with (A) and the Intel RealSense sensor with (B).

Source: ABB IRB 2400 Industrial Robot

5.1.3 Sensors

We can only use two sensors due to a limitation in the hardware with only two available USB3-buses, so we use two depth sensors of different type. A relatively big Microsoft Kinect One yielding very good data to cover a big area with high precision, and a much smaller, but also less accurate Intel RealSense to cover areas not seen by the first sensor. The positions of the mounted sensors can be viewed in Figure 5.1. Both sensors are viable candidates, as concluded in Section 2.6.

5.2 Results

The system's overall performance is a combination of many factors. Some applications require short response time while millimeter accuracy is not important, such as our demonstration, while object tracking needs high accuracy but only moderate response time. These requirements will affect the choice and amount of sensors as well as the parameters of the system. The first working implementation of our system had sequential execution, with very low framerate and response time far above acceptable levels. However, the final parallel implementation together with other major optimizations reduced the response time to levels acceptable for almost any "real-time" application.

The demonstration successfully fulfilled the objective described in Section 5.1.1 and the result can be viewed in detail in Appendix B. The system is yet only a proof of concept, and leaves room for more improvements such as better response time and higher accuracy.

An image from the graphical visualizer, showing the state of the system running the demonstration setup can be viewed in Figure 5.2.

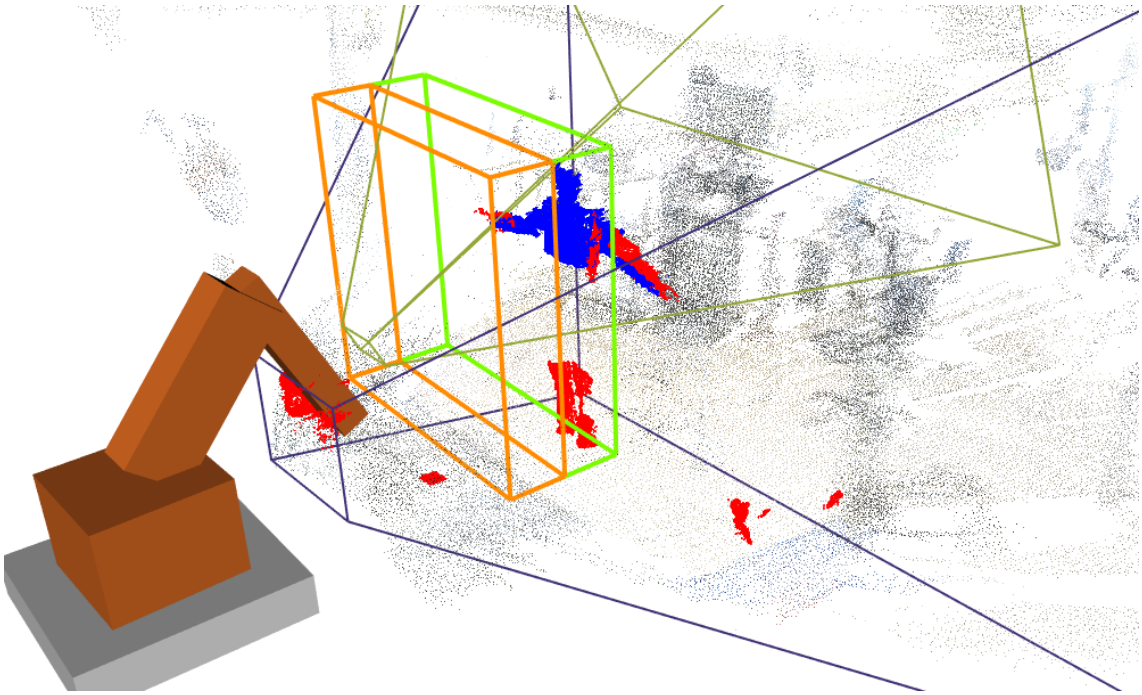


Figure 5.2: Image of the visualizer while running. The static world (sparse points) and the robot are present, along with the current viewpoints of the mounted sensors, critical regions (boxes) and region of interest (dense points).

5.2.1 Response time

The responsiveness of our system differs depending on the current sensors and operation mode.

Demonstration

In the case of our demonstration the response time can be split into four parts; the sensor's delay and delivery speed, the time from incoming cloud to deviation detection, the delay from robot API calls to robot acknowledgment and break time.

With a high-speed camera recording at 600 frames per second we captured five videos of the robot operating at a speed of 1 m/s, when we caused a region of interest violation. We measured the average total response time to be 509 ms with a standard deviation of 44 ms until the robot was completely still.

Our system can operate faster than new frames are delivered by the Microsoft Kinect One sensor of 30 images per second, resulting in a worst case scenario of 34 ms. From

the same recordings we measured the response time from violation to system signal to be an average of 150 ms with a standard deviation of 21 ms. The measurements include the graphical visualizer and delay of displaying device. The complete numbers can be viewed in Table 5.1.

Table 5.1: Response time measured by counting frames from recordings by a Casio EX-F1 camera captured at 600 frames per second. Each frame corresponds to 1.67 ms.

[ms]	Measurements $(n = 5)$	Average	Standard deviation
Robot stop	550, 485, 490, 450, 570	509.0	44.3
System registration	160, 115, 165, 173, 135	149.6	21.4

This leaves 350 ms on the robot, from ABB PC SDK API call to robot acknowledgment and complete stop. This response time is heavily dependent on the current operation speed, size and payload of the robot. We estimate half of this time is spent on the non real-time API call that can be almost eliminated by closer integration of the system and the robot controller, or by developing an API with real-time support.

General

If the system is in a mode where the transformation is not provided by a robot's kinematic transforms the response time is increased as the ICP algorithm requires more computational time. We run on approximately 5-15 frames per second depending on the complexity and size of the scene. The delay in the sensors and the responsiveness of the robot also affects the performance and response time of the system in the general case. By using ICP instead of the kinematic transform in our safety application, at most add 200 ms will be added to the total response time listed in Table 5.1. We have not tested ICP in our demonstration due to safety concerns, as the ICP algorithm may not converge and cause undesired behavior. Additional precautions are necessary before attempting this.

5.2.2 Accuracy

The accuracy of the system is highly dependent on the quality and technique of the sensors, but is also affected by the filters in the system. A result of the system detection and extraction of a region of interest can be seen in Figure 5.3.

False positives

Noise occurs in the clouds even after filtering, and a single violation from noise, i.e., a false positive, could potentially stop the robot. By requiring that a certain amount of violating points need to be within a cubic decimeter inside a violation box before taking action we reduce the amount of false positives drastically, but risk missing small objects.

From Equation 3.1 the Kinect One sensor gives a point density of 100 points/dm^2 at a distance of approximately 3.7 m and in our demonstration we require 70 points inside to cause a signal. This gives us a theoretical ability to detect a flat object with the size of a square decimeter up to 4 meters away with a normal sensor. This is very close to the accuracy of the Microsoft Kinect One used in the demonstration.

Figure 5.3: A high density region of interest detected from data captured by a Microsoft Kinect One.

Noise

The accuracy and amount of noise differs between different sensors and has to be taken into account in every application. An accuracy comparison between the sensors used in our demonstration can be seen in Figure 5.4. Additional filtering may improve the result but will increase computational cost.

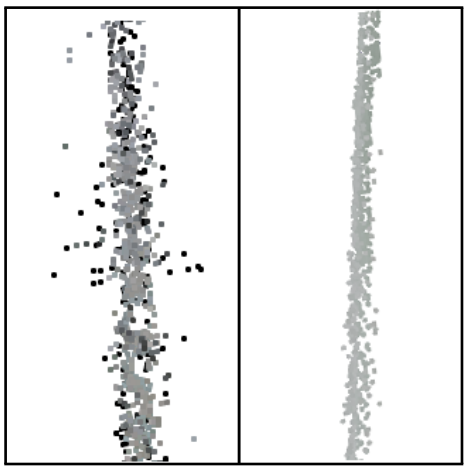


Figure 5.4: The thickness of a flat wall captured from 1.5 m over 30 s corresponds to inaccuracies in the sensor. Intel RealSense R200 (left), Microsoft Kinect One (right).

5.2.3 Multiple sensors

The system is designed to use multiple sensors and is fully functional with two physical cores as shown in the demonstration.

The evaluation is performed on a laptop with eight available physical threads and two separate USB3-buses, so due to this limitation we were unable to evaluate more than two sensors. Given more powerful hardware many more sensors can be used.

5.3 Discussion

Overall the system behaves as expected in terms of speed, flexibility and reliability. However, the high complexity together with the necessary choice of C++ make it likely to have errors causing undesired behavior. We have used the full capability of the compiler and other tools to detect most of these errors, such as memory leaks and semantic problems.

5.3.1 Demonstration

The demonstration only covers a minor part of the areas of usage for this application, so the conclusions should not be seen as a general result, but only for that specific case.

Occlusion

A major problem of safety critical applications when mounting the sensors on the robot is the guarantee of coverage, i.e., that the total view of the sensor always has clear sight on critical areas. With only two available sensors in the demonstration we can not cover our critical regions for arbitrary robot programs, as the sensors are mounted on moving parts that can be programmed to "look away". However, with a few more sensors, and the robot programmed with their position in mind, very complex tasks can be performed while still covering critical areas. In Figure 5.2 a self-occlusion is clearly visible where the person's legs are only visible by one sensor while the head and torso is visible from the other. The arms are visible by both, slightly overlapping, and the waist is fully occluded from both.

Responsiveness

The ABB PC SDK is not developed for real-time applications and is the biggest source of delay in our system. This is not the optimal way of controlling the robot, but no good alternative exists that can be implemented within the scope of this thesis. We have put much effort into minimizing the delay in the parts of the system that are within our control, and achieved very good results, but the total response time might still be too large for being viable in a safety setup.

Since the majority of the delay originates from external sources, there are apparent improvements that can be gained by using faster sensors and a closer integration with the robot system.

5.3.2 Limitations

The entire setup contains many sources of errors and limitations that affect the result. A few important ones are:

Sensor accuracy – We have not used safety-classified sensors, and therefore no guarantee can be made on the sensors' ability to create accurate images of the reality. No easily available alternatives are currently available but may be in the future.

Failing hardware – The system does not currently analyze itself and connected devices for errors such as broken hardware, cut cables or alike. This is important especially for safety-critical application, but finding a solution is not solved within the scope of this thesis.

Not real-time – The system is developed with real-time execution in mind but currently contains incompatible parts, such as the ICP algorithm and Windows OS. This introduces an uncertainty in the response time even within our system. However, real-time operation can be achieved by further development and porting to a real-time system.

Asynchronous sensor image and position – The system is currently not synchronizing timestamps on incoming sensor images and current transformation provided by the robot's kinematic model. Since the transformation updates much more frequently and is provided by a real-time system, the images may lag behind. This causes a slight difference between reality and the systems perception that is very small but still a source of error.

Sensor mounting imprecision – Mounting a sensor with duct tape is not the most optimal solution as it is not accurate and it may move during operation. This causes the exact mounting position to be unknown by the system. By integrating the sensors with the robot during manufacturing this error can be reduced significantly.

Chapter 6
Conclusions

At last we will present some suitable applications for our solution and discuss reasonable future work in the continued development of the system.

6.1 Areas of usage

Our solution is very flexible and dynamically built, so additional features can easily be added. This opens for multiple different areas of application, some already possible whereas some require additional extensions. A few will be mentioned but many more are possible.

6.1.1 Safety

Because of the nature of our system with multiple different sensors and services distributed on different computers to provide necessary information, it is close to impossible to make it safety-certified. It can however be used as a safety extension on top of a certified safety system, reducing its complexity as our system can handle situations before they become critical; avoiding unnecessary emergency stops.

It may also increase the general safety in areas where no certification is needed, such as when a robot model is considered to be safe. Even if the robot is too weak to harm a human by itself, it can be armed with a sharp object that could. In these scenarios our system can make the robot slow down or stop if anything in close proximity is detected, preventing potential accidents.

6.1.2 Object tracking & identification

The system has the potential to identify objects that are detected in the ROIs. This can be done by matching the extracted clouds to a database of known geometrical models. The

ROIs could also be back-projected into the high resolution images so traditional image analysis algorithms can be used.

The identified objects can be tracked while moving in space to provide information about speed and direction. As each frame only contains a snapshot of the state, this information has to be gathered over several frames back in time. There is currently no support for this but it does not require any major design changes other than addition of appropriate algorithms.

Having accurate object tracking and identification will not only allow for a whole new set of applications, but will also increase the accuracy of others. Using this feature in the safety application used in our demonstration could potentially reduce the number of false positives by only taking action on humans or objects of a certain size, instead of anything that appears within a critical region.

6.1.3 Path planning

Path planning determines how the robot should progress when moving from one point to another. When a robot is aware of its surroundings, this knowledge can be exploited to perform path planning dynamically. This can for example be used to make changes in the robot cell without the need for reprogramming the robot.

Additional degrees of freedom

A few industrial robots have more than the standard of 6 degrees of freedom. These can be used to move the position of some links without affecting the position or orientation of the robots tool, just as a human can move an elbow without moving the hand. By mounting sensors on these links they can be moved while working on other tasks. This would allow the robot to scan bigger areas with a small number of sensors while still maintaining productivity.

6.2 Future work & Improvements

The system is not without flaws or limitations and the following parts should be improved before adding additional features.

6.2.1 Performance

The system is already optimized for performance, but there are still parts that can be improved to reduce the response time in the system. The delay caused by using the ABB PC SDK API can be removed with a proper real-time API for robot communication, or by integrating the system with the robot controller. The depth projection used to create the point clouds can also be parallelized to reduce the input delay on the sensors, and the ICP algorithm should be investigated for either optimization or replacement with a faster or more accurate alternative.

6.2.2 Non-static world

The system currently operates in two phases. First the environment is scanned and then that data is used to identify interesting regions that change. This is a good approach as long as the environment remains static, but in the case of a big object getting moved, e.g a table, a complete re-scan has to be performed to maintain optimal functionality of the system.

A better solution would be to let the world slowly adapt to changes in the static world. The world could be represented by various groups of clouds with different properties. They could for example be divided into static, semi-static and dynamic where points can be promoted or demoted between those. The problem that arises from this is to determine which group the points should be placed in and this would require some form of artificial intelligence.

The semi-static world could slowly "decay" by removing points after some time. When an area is once again scanned by a sensor all corresponding points will be refreshed and stay in the semi-static model. This will allow objects to be moved and the scanned model will slowly be updated, as the old position will fade and the new will appear over time. A slight problem with this is that stationary objects will enter the static model, which applies to humans standing still a long enough time, determined by implementation and parameters.

6.2.3 Dynamic ROI-boxes

The critical regions are represented as static boxes in our system, which is impractical for some applications. By allowing regions of arbitrary shape and size that can move, expand or subtract depending of the current position and operating speed of the robot, or other external information, more advanced setups become possible.

This can be used to encapsulate the robot in non-critical areas that follow the robot's movement. This will allow the robot to work within critical sections without risking to trigger a violation. It can also be used to move the regions based on the robot's operating speed, allowing people to be closer when the robot is moving slowly, but require them to be further away if it is operating fast.

6.2.4 Intelligent ROI violation detection

A critical region should only be violated when a certain number of points are inside within a small volume, to avoid false positives. Even with a relatively high threshold false positives can still happen due to noise or inaccuracies of the sensor if an object is close to the region's border, and can cause the region to get activated and deactivated frequently. By requiring subsequent violations before taking action, noise or errors that are present only in one frame will not trigger a false-positive. However, this solution will introduce additional delay in the system as it requires more than one frame to take a decision.

6.3 Final words

We have successfully shown that this kind of system is not only possible to develop, but is also responsive enough for real applications, such as the safety application in our demonstration. The system is also modular enough to work as a platform for future applications and research in areas mentioned in this chapter. The system is yet only a prototype with many possible improvements and extensions.

Bibliography

[1] Collins English Dictionary - Complete & Unabridged 10th Edition. *"robot."* 2010. URL: http://www.dictionary.com/browse/robot (visited on 2016-05-18).

[2] ISO/TS 15066:2016. *Robots and robotic devices – Collaborative robots*. Standard. Geneva, CH: International Organization for Standardization, Feb. 2016.

[3] Armin Hornung et al. "OctoMap: An Efficient Probabilistic 3D Mapping Framework Based on Octrees". In: *Autonomous Robots* (2013). Software available at http://octomap.github.com. DOI: 10.1007/s10514-012-9321-0. URL: http://octomap.github.com.

[4] D. Maier, A. Hornung, and M. Bennewitz. "Real-time navigation in 3D environments based on depth camera data". In: *2012 12th IEEE-RAS International Conference on Humanoid Robots (Humanoids 2012)*. Nov. 2012, pp. 692–697. DOI: 10.1109/HUMANOIDS.2012.6651595.

[5] Andreas Nüchter. "3D robotic mapping: the simultaneous localization and mapping problem with six degrees of freedom". In: Springer Tracts in Advanced Robotics (STAR), vol. 52. Springer, 2009.

[6] Javier Minguez and Luis Montano. "Sensor-based robot motion generation in unknown, dynamic and troublesome scenarios". In: *Robotics and Autonomous Systems* 52.4 (2005), pp. 290–311. ISSN: 0921-8890. DOI: http://dx.doi.org/10.1016/j.robot.2005.06.001. URL: http://www.sciencedirect.com/science/article/pii/S092188900500093X.

[7] J. Prankl et al. "RGB-D object modelling for object recognition and tracking". In: *Intelligent Robots and Systems (IROS), 2015 IEEE/RSJ International Conference on*. Sept. 2015, pp. 96–103. DOI: 10.1109/IROS.2015.7353360.

[8] Anna Kochan. "Robots and operators work hand in hand". In: *Industrial Robot: An International Journal* 33.6 (2006), pp. 422–424. DOI: 10.1108/01439910610705572. eprint: http://dx.doi.org/10.1108/01439910610705572. URL: http://dx.doi.org/10.1108/01439910610705572.

[9] E. Horbert et al. “Sequence-level object candidates based on saliency for generic object recognition on mobile systems”. In: *2015 IEEE International Conference on Robotics and Automation (ICRA)*. May 2015, pp. 127–134. DOI: 10.1109/ICRA.2015.7138990.

[10] Xavi Gratal et al. “Scene Representation and Object Grasping Using Active Vision”. In: *IROS’10 Workshop on Defining and Solving Realistic Perception Problems in Personal Robotics*. Oct. 2010.

[11] Beau Tippetts et al. “Review of stereo vision algorithms and their suitability for resource-limited systems”. In: *Journal of Real-Time Image Processing* 11.1 (2013), pp. 5–25. ISSN: 1861-8219. DOI: 10.1007/s11554-012-0313-2.

[12] Dept. of Math and Computer Science. *Middlebury Stereo Evaluation*. Middlebury College. 2016. URL: http://vision.middlebury.edu/stereo/eval3/ (visited on 2016-04-06).

[13] Andrea Fusiello, Emanuele Trucco, and Alessandro Verri. “A compact algorithm for rectification of stereo pairs”. In: *Machine Vision and Applications* 12.1 (2000), pp. 16–22. ISSN: 1432-1769. DOI: 10.1007/s001380050120. URL: http://dx.doi.org/10.1007/s001380050120.

[14] S. B. Gokturk, H. Yalcin, and C. Bamji. “A Time-Of-Flight Depth Sensor - System Description, Issues and Solutions”. In: *Computer Vision and Pattern Recognition Workshop, 2004. CVPRW ’04. Conference on*. June 2004, pp. 35–35. DOI: 10.1109/CVPR.2004.17.

[15] P. J. Besl and H. D. McKay. “A method for registration of 3-D shapes”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 14.2 (Feb. 1992), pp. 239–256. ISSN: 0162-8828. DOI: 10.1109/34.121791.

[16] Feng Lu and Evangelos Milios. “Globally consistent range scan alignment for environment mapping”. In: *Autonomous Robots* 4.4 (1997), pp. 333–349. ISSN: 1573-7527. DOI: 10.1023/A:1008854305733. URL: http://dx.doi.org/10.1023/A:1008854305733.

[17] Jochen Sprickerhof et al. “An Explicit Loop Closing Technique for 6D SLAM.” In: *4th European Conference on Mobile Robots (ECMR)*. KoREMA, 2009, pp. 229–234. ISBN: 978-953-6037-54-4. URL: http://dblp.uni-trier.de/db/conf/ecmr/ecmr2009.html#SprickerhofNLH09.

[18] Radu Bogdan Rusu and Steve Cousins. “3D is here: Point Cloud Library (PCL)”. In: *IEEE International Conference on Robotics and Automation (ICRA)*. Shanghai, China, May 2011.

Appendices

Appendix A
Code

The full source code for the system (codename NemoSwarm) can be obtained from `https://git.cs.lth.se/Nemo/NemoSwarm` with an account authorized by the department of computer science, LTH, Lund University.

A.1 Capturer

```
PointCloud* capture_working_cloud; //Initializations omitted
PointCloud* latest_cloud;
PointCloud* matcher_working_cloud
bool has_data;
Mutex mutex;

void thread_work_function() {
    while (continue_capture) {
        acquire_cloud(capture_working_cloud); //Retrieve cloud
        mutex.take();

        swap(capture_working_cloud, latest_cloud);//Swap pointers
        has_data = true;

        mutex.give();
        mutex.notify();
    }
}
PointCloud* get_latest_cloud() {
    mutex.take();

    while (!has_data)
        mutex.wait();

    swap(matcher_working_cloud, latest_cloud);//Swap pointers
    has_data = false;

    mutex.give();
    return matcher_working_cloud;
}
```

A.2 Builder

```
Capturer capturer;
PointCloud* world_model;
VoxelFilter voxel_working, voxel_graphic;
StatisticalOutlierRemovalFilter sor;
IterativeClosestPointWithNormals icp;
Transform initial_guess;
double icp_threshold = ...

void CloudBuilder::RegisterOne()
{
    PointCloud* new_cloud = capturer.get_latest_cloud();

    PointCloud* working_cloud;
    PointCloud* graphic_cloud;

    working_cloud = voxel_working.filter(new_cloud);
    graphic_cloud = voxel_graphic.filter(new_cloud);

    sor.filter(working_cloud);
    sor.filter(graphic_cloud);

    working_cloud->calculate_normals();

    icp.set_ground_cloud(world_model);
    Transform matched_transform = icp.align(working_cloud,
       initial_guess);

    if (icp.converged() && icp.get_fitness_score() < icp_threshold){
        initial_guess = matched_transform;
        graphic_cloud->transform(matched_transform);
        world_model += graphic_cloud;
    }
}
```

A.3 Matcher

```
RobotBase* robot;
PointCloud* static_world;
IterativeClosestPoint icp;
Transform previous_transform;
BoundingBoxContainer boxes;
CriticalRegionHandler cr_handler;

void CloudMatcher::RegisterOne()
{
    PointCloud* new_cloud = capturer.get_latest_cloud();

    Transform sensor_transform;
    if (kinematic_available()) {
        sensor_transform = robot->get_sensor_transform();
    }
    else { // Estimate transform with ICP
        icp.set_ground_cloud(static_world);
        sensor_transform = icp.align(working_cloud, previous_transform)
            ;
        previous_transform = sensor_transform;
    }

    // Transform the captured cloud to world space
    new_cloud->transform(sensor_transform);

    // Remove points close to static world
    PointCloud* roi_cloud = new_cloud->get_deviation_from(static_world)
        ;

    // Check if any point is inside a bounding box
    for (Point p in roi_cloud) {
        if (boxes.point_is_inside(p)) {
            cr_handler.signal(); //start/stop/slow etc...
        }
    }
}
```

A.4 CriticalRegionHandler

```
int counter = 0;
bool out_signal = false;

void setOutSignal(bool new_state) {
    if (new_state) {
        if (++counter == 1) {
            // first sensor to detect violation
            out_signal = true;
        }
    }
    else { // new_state == false
        if (--counter == 0)  {
            // last sensor to undetect a violation
            out_signal = false;
        }
    }
}
```

Appendix B
Demo

The captured video can be found at `https://youtu.be/M7o6c2vTsoo` (2016-04-28).

B.1 Images

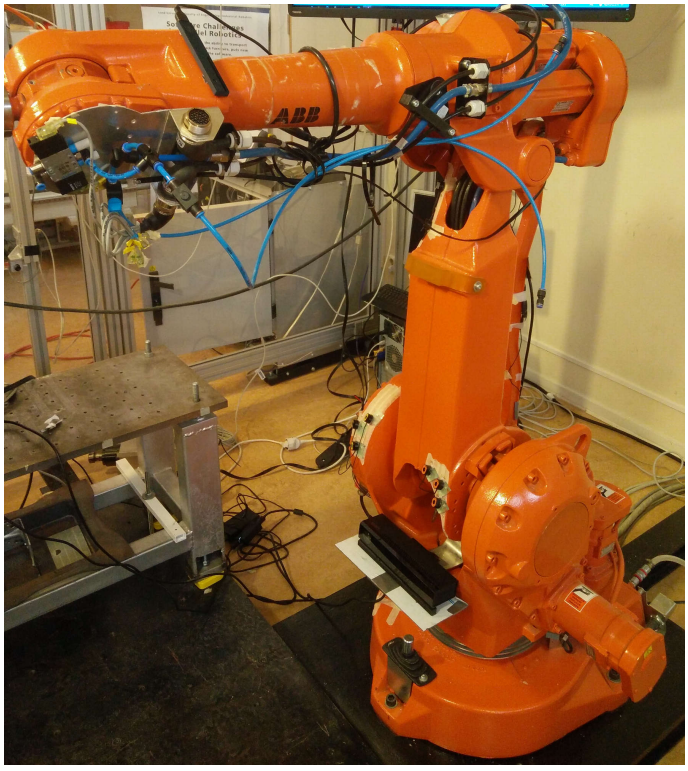


Figure B.1: The ABB IRB 2400 robot with sensors mounted.

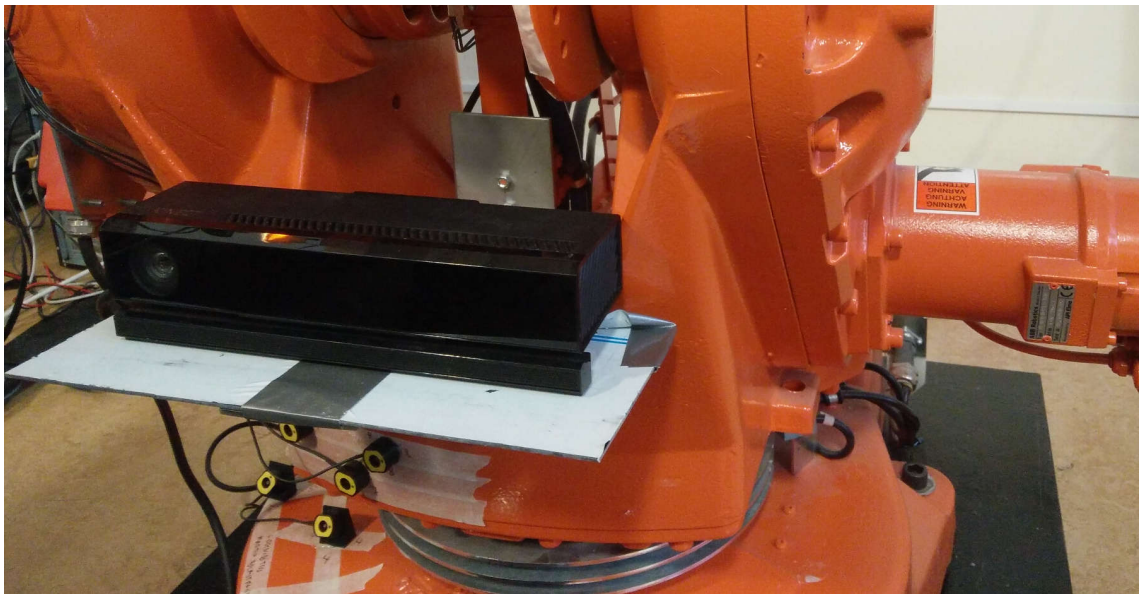


Figure B.2: The mounted Microsoft Kinect One on robot link 2.

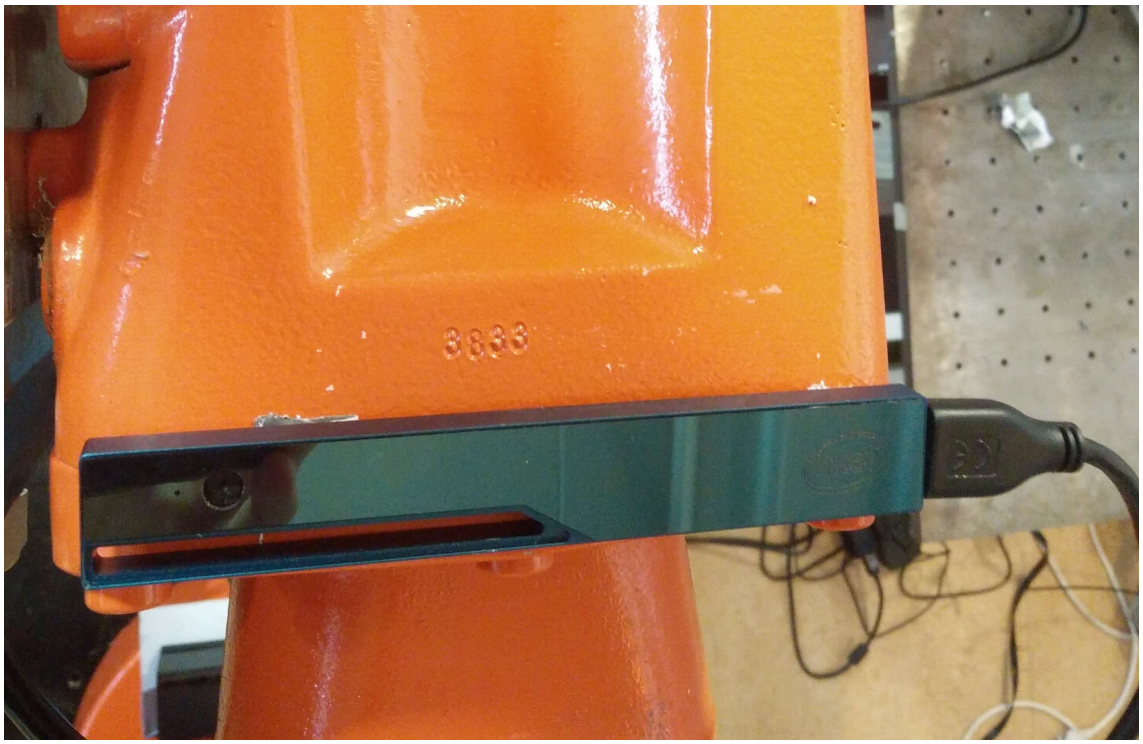


Figure B.3: The mounted Intel RealSense on robot link 4.

Appendix C
Requirements

The external dependencies of the source code:

- Microsoft Windows 7 or above
- Microsoft Visual Studio 2015 (with administrator rights)
- Visual Leak Detector (for debug only)
- Kinect For Windows SDK 2.0
- Intel RealSense SDK
- Intel RealSense R200 Camera Driver
- PCL 1.8.0 or above
- OpenNI 2.2 (included in PCL installer)

Make sure the following PATH variables are set:

- KINECTSDK20_DIR
- RSSDK_DIR
- PCL_ROOT
- OPENNI2_REDIST64

Required hardware:

- One USB3-bus (not port) per sensor (if USB3 sensor).
- A Microsoft Kinect One and/or an Intel RealSense R200.

Institutionen för Datavetenskap | Lunds Tekniska Högskola | Redovisas 3 Juni 2016
Examensarbete: Geometric real-time modeling and scanning using distributed depth sensors
Studenter: Anton Klarén, Valdemar Roxling
Handledare: Jacek Malec (LTH), Klas Nilsson (Cognibotics AB)
Examinator: Elin Anna Topp

Att öka robotars medvetande

Populärvetenskaplig Sammanfattning av ***Anton Klarén & Valdemar Roxling***

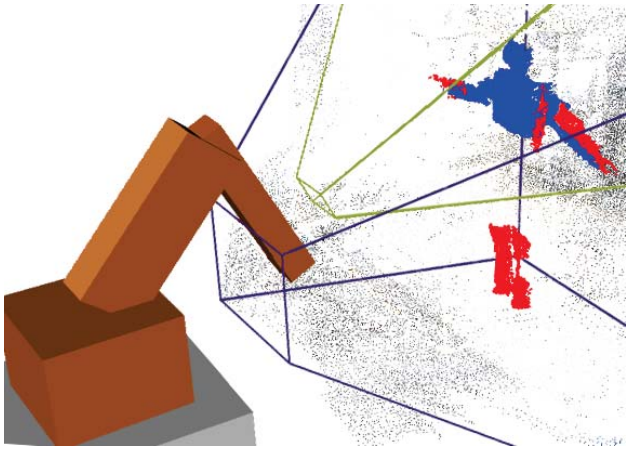
Dagens industrirobotar är helt omedvetna om sin omgivning och utför oftast enkla förprogrammerade uppgifter. Med flera moderna kameror som fångar tredimensionella bilder monterade på olika delar av roboten så får den ögon. Dessa kan sedan användas för att hålla koll och förstå vad som händer i dess närhet.

Allseende robotar

När roboten blir medveten om sin omgivning öppnas dörrarna för många nya spännande användningsområden. Med vårt system kommer människor kunna arbeta sida vid sida med robotar utan att utsättas för onödiga risker. Med kameror som sitter på roboten istället för vid sidan av så ser den alltid vad som finns i närheten oavsett var den befinner sig utan att sikten riskerar att bli helt skymd av till exempel dåligt placerade objekt. Ett system som låter oss koppla ihop flera sensorer och skapa en representation av omgivningen som roboten kan tolka möjliggör allt detta och mycket mer.

Under huven

Flera kameror som fångar omvärlden på bild i till exempel 30 gånger per sekund resulterar i gigantiska mängder information som även dagens datorer har svårt att hinna analysera. Precis som vår hjärna sållar bort irrelevant information för att kunna fokusera på det viktiga så måste roboten göra detsamma. Genom att först analysera omgivningen och identifiera fasta föremål, till exempel väggar, golv och inredning, så kan roboten lätt avfärda de som irrelevanta. Vid ett senare tillfälle används information för att urskilja mer intressanta områden, dvs. det som inte fanns där vid första analysen. Den värdefulla datorkraften kan då fokuseras på de intressanta bitarna för ytterligare analys så att roboten kan ta ett intelligent beslut beroende av situationen.



En industrirobot är ofta väldigt rörlig och kräver därför flera kameror för att kunna hålla koll på allt som händer utan att riskera missa något. På bilden har en kamera monterats på robotarmen och ser det blåa området medan den andra är monterad på magen och ser det röda området. Golv och inredning är grå i bakgrunden. Som tydligt syns så ser roboten inte hela människan för robotarmen skymmer delvis kameran på magen samtidigt som den på armen har en dålig vinkel. Dock kommer aldrig människan att bli helt skymd då hen är större än den döda vinkeln.

En värld av möjligheter

Vår lösning kan till exempel användas inom följande områden:

Säkerhet – Sakta ner eller stanna roboten om någon kommer för nära eller befinner sig i kollisionskurs. Detta kan förhindra onödiga och kostsamma produktionsstopp.

Kognition – Identifiera och spåra föremål som rör sig i närheten eller kommer in på löpande band som del i en produktionsprocess. Med hjälp av denna informationen kan roboten lättare programmeras.

Rörelseplanering – Ändra rörelsebana som anpassar sig efter hand som information blir tillgänglig. Ett alternativ till att sakta ner eller stanna vid kollisionsrisk.

Vår lösning är skapat för att lätt kunna vidareutvecklas och användningsområden är enbart begränsade av användares fantasi.

En demonstration av vårt system ur säkerhetsperspektiv kan ses på `https://youtu.be/M7o6c2vTsoo`.