Bachelor Thesis

# A Qualitative Analysis of Two Automated Registration Algorithms In a Real World Scenario Using Point Clouds from the Kinect

Jacob Kjær, s072421@student.dtu.dk

June 27, 2011

**Abstract**

(English) In this Bachelor Thesis in Software Engineering, a practical system for reconstructing real world objects and structures with point clouds from the Microsoft Kinect is described and tested. It uses two different registration algorithms for alignment of point cloud data, Iterative Closest Point and Feature-based registration (Optical Flow, Visual Odometry). Both algorithms are extended and implemented based on conclusions drawn in an in depth analysis of the data from the Kinect device. It is found that it is possible to make consistent reconstructions of medium-sized objects as long as they abide the limitations of the device, and that rooms reconstructed in free hand any further guidance or large-scale error correction is likely to suffer from accumulated errors.

**Abstract**

(Dansk resume) Denne bachelor thesis i Software Teknologi ved DTU beskriver et system der skal muliggøre rekonstruktion af den virkelige verden påen computer med punkt skyer fra Microsofts Kinect. Det gør brug af to forskellige registreringsalgoritmer, *Iterative Closests Point* og *Feature-based Registration*. Begge disse algoritmer bliver beskrevet i detaljer, sammen designet af deres endelige implementation med baggrund i en dybdegående analyse af Kinect'ens even til at fange punkt sky data. Det viser sig at Kinect'en udemærket kan rekonstruere visse medium-størrelse objekter sålænge de falder inde for enhedens begrænsninger, men ogsåat at rum der rekonstrueres i fri hånd har en tendens til at akkumulere fejl der kræver et fejl-håndtering pået højere niveau for at danne et virkelighedstro resultat.

# Contents

# 1 Introduction

In fall 2010 Microsoft released the Kinect. It is a piece of hardware which incorporates a structured light projector, an infrared (IR) camera, a color camera, and various other peripheals like an array of microphones and an accelerometer meant for adjustment of a built in pivot motor. In this report, the two cameras and the structured light projector are put to use.

The infrared camera, when paired with the structured light projector, is able to estimate the depth of a given point in an image relative to its viewpoint. Given that the x and y position of any point in the image plane is implicitly given, having its depth gives us its three-dimensional projective coordinates. On such coordinates one can perform a reverse perspective projection in order to extract the 3-dimensional Cartesian coordinate of the point relative to the viewpoint of the camera. Performing such a transformation on all points in the image plane will result in a set of points, called a *point cloud*. A point cloud can be reprojected and explored from all viewpoints independently of how it was generated.

Given that the Kinect also has a normal color camera, this can be used to give color to point clouds captured by the structured light depth estimation system.

## 1.1 Problem and motivation

Now, being able to capture a point cloud of a scene or an object is nice. But exploring it from different viewpoints, one will discover large holes in the objects in the scene caused by occlusion. And even more, what about that which lies behind the camera? A single point cloud captured by the Kinect can tell us nothing more about the scene than the surface which is visible from the viewpoint from which it was captured.

*Point cloud registration* is the act of matching such point-clouds captured from different viewpoints with each other so that they form a more complete representation of a given scene. The result of a registration of two point clouds is some information which can be used to align the clouds.

With good registration, surface scanners like the Kinect can be a very useful aid in re-construction of the real world on a computer. This has many uses. Computer game developers can easily make levels based on real life structures, and also very quickly generate in-game assets from real-world objects which they would otherwise have to sculpt from scratch. Educational institutions of all kinds can leverage the benefits of real world reconstruction in some of their teaching. Real estate brokers can use 3D reconstructions of homes to sell better. Geologists and archaeologists can use it as a part of their mapping processes of underground caves and ancient sites to capture details which would otherwise be very costly to describe. This list of scenarios in which this technology can be used is far from exhaustive, but the wide variety of seemingly unrelated field in the list should give an idea that it is quite interesting to look further into.

## 1.2 Goal

Point cloud alignment can be done by hand, but it is time-consuming, tedious, and possibly imprecise. However, automatic alignment (registration) algorithms already exist. Two of these are Iterative Closest Point (ICP) (plus some extensions thereof) and one based on optical flow in the Kinect color image. The goal if this project is to implement these two algorithms (plus variants) and then analyze them in a set of real world scenarios in which they are used to align point clouds captured by the Kinect. In what situations can the Kinect be used, and how well do the algorithms perform with regards to speed and precision?

Another goal, and a major motivation for this project, is to create a tool which interfaces with the Kinect for point cloud capturing and exposes these automatic registration algorithms for usage on them. It will be built mainly with the intention of testing the algorithms, but will also make it possible for end users to create and export their own aligned scans of real world objects and scenes. In the past when investigating registration algorithms, range scanners were not easily available, and therefore there has been little incentive to focus on creating a complete workbench. With the release of the Kinect, which costs only 150 USD and has sold over 10 million units world wide[1], there definitely is, and the software should be a robust milestone or prototype from which experiences can be drawn.

Registration, on a higher level, recognizes (*registers*) parts of one dataset in another. Therefore registration algorithms will not work on two datasets from completely unrelated viewpoints in a scene. They must have something in common, and the more the better. Therefore is no focus on aligning clouds taken from arbitrary positions to each other, nor is there any focus on scenes in which any kind of movement occur. Only clouds taken from similar viewpoints are aligned with the methods described within the scope of this report.

## 1.3 Structure and approach

The report is structured in this way: First the capabilities of the Kinect device is analyzed in depth. Then, the two registration algorithms are presented. They both have levels of modularity which can be changed to suit different applications, and therefore the result of the hardware analysis will be used often to rationalize their final design. Therefore the clear-cut theory on these algorithms is kept quite concise while the design aspects are introduced quickly.

This leads to the design, usage guide, and implementation notes of a software workbench which can be used to assert the quality of these algorithms on the point cloud data provided by the Kinect. The final result is discussed in a section by itself, and is then followed by a conclusion and summary of what has been done in the report.

## 1.4 Related works

Intel Research did a sophisticated reconstruction system using Primesense (Kinect) technology in 2010[2]. It features advanced large-scale reconstruction techniques

like loop-closure, a global error-minimization technique which is beyond the scope of this project.

Nicolas Burrus, who presented a lot of useful information before the release of the OpenNI drivers (to be discussed later), did his own take on such a system inspired by Intel in his RGBDemo, released in January 2011[3]. In April 2011, he released an update for this program with GPU-optimized registration.

Other similar projects include RGBDSLAM on OpenSlam.org[4], RGBD-6D-SLAM on ROS.org[5], and a design project by some fourth year students at University of Waterloo who put videos on YouTube[6].

# 2 Qualitative Analysis of Data from the Kinect

The following subsections are dedicated to analyzing the quality of the input data and aspects which can be related to this. When does the Kinect work well, and when does it not? It is important when planning a reconstruction to know how one can expect the camera to perform.

While the Kinect was initially released as a gaming peripheral for the Xbox 360 console, a set of official drivers were released around new year 2011 as part of the OpenNI project along with a very useful library for interfacing with the device and a collection of high-level computer vision functions. The OpenNI drivers are official in the sense that they are backed by the developer of the Kinect hardware, Primesense. In the end of June 2011, Microsoft also released their take on an official SDK[7].

Through OpenNI the Kinect provides both color (24-bit RGB) and depth (format description follows) images in 640x480 resolution at a speed of 30Hz. This gives a theoretical upper limit of $640 \times 480 = 307200$ points in a point cloud. In practice, a scene with good capturing conditions will result in a cloud of at most ca. 265000 points, due to how the depth map will map onto the color image, which is captured with a wider field of view. The quality of color images captured by the Kinect is about as good as a decent webcam, and bayer noise is noticeable.

## 2.1 Retrieving point clouds from the Kinect

OpenNI provides everything needed for extracting a colored point cloud from the Kinect. It can extract corresponding depth and color maps from the device, calibrate them against each other, and perform the reverse perspective projection to Cartesian coordinates as described in the introduction. Calibration of the color and depth map against each other is necessary because the source cameras for these maps are physically offset from each other on a horizontal axis. Also, the cameras have difference fields of view, and therefore different view frusta[8], so the points on the depth map must be re-projected to the viewpoint of the color map for good correspondence. OpenNI can also do that.

OpenNI also abstracts all internal scales for depth and position to metric values, where 1 meter = 1000 units. This is useful as it means that the device can exploited for measuring object sizes and lengths in a scene. It also makes it easy to measure the precision of an alignment of two clouds if the relative transformation of the camera is measure in real world beforehand.

## 2.2 Scanning surfaces with structured light

The following is based on info from the ROS Team's technical analysis of the Kinect[8].

The Kinect uses its infrared camera along with its structured light projector to estimate the surface of a scene as seen from the camera viewpoint. The projector projects an infrared pattern onto the scene and the IR camera captures it.

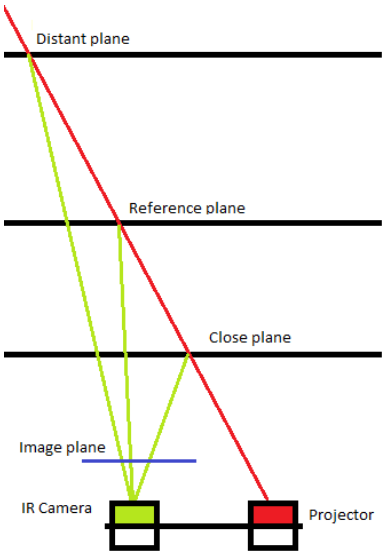


**Figure 1:** A Kinect-like setup with a single point-projector

To illustrate how surface measurement using structured light works, see figure 1. It is a Kinect-like setup in which a projector is projecting just a single point (a very simple pattern) into space along the red line. It is captured by the camera once it hits a surface. There are three planes in the illustration; a reference plane, a plane closer to the camera than the reference plane, and a more distant plane. When the point hits a surface in the close plane it will appear further right in the image than if it was hitting a surface in the reference plane. Likewise, when the point is projected onto an object on a plane which is more distant than the reference plane, the point will appear more to the left. When the origin and direction of the light is known beforehand, and the horizontal position of the dot is known for a reference depth, then it is possible to find out the depth of the surface which the point hits based on its horizontal position in the cameras image plane.

Now, the Kinect does not project just a single point, but a large amount of points in an intricate pattern. Internally, it has an image reference of what the pattern looks like from the IR cameras viewpoint when all point in the surface are at a certain, known distance. It finds correspondences between the pattern which it captures with its camera and this internal reference pattern. By comparing the horizontal position of a point in the captured image to its corresponding horizontal position in the reference image, a horizontal delta value can be extracted, which in turn can be used to calculate the depth of the pixel just like described in the above paragraph with the single-point projector. The Kinect itself actually does not calculate the depth, but returns a more abstract value for the host system to handle (see next section). While OpenNI abstracts this away for the developer, *libfreenect*, another driver and library platform, makes these 11-bit values available[9].

Because there are less pixels in the IR pattern than there are in the depth map some parts of the depth map are interpolation[8], meaning that one cannot

expect the Kinect to be pixel-precise. It works best for smooth continuous surfaces.

## 2.3 Precision Across the Depth Map

According to Nicolas Burrus, who pioneered with information about the Kinect from his own experiments, the depth of a point $z$ can be calculated in meters from the raw disparity of the point $d$ (as provided by the Kinect hardware) using the following equation[9]:

$$z = 1.0/(d \cdot -0.0030711016 + 3.3309495161)$$

I did some analysis of the equation and found it peculiar; $d$ is an 11-bit integer which ranges from 0 to 2047. $z$ will change sign from positive to negative when $d$ is around 1084, so values beyond that are useless for depth measurement. I carried out tests with the Kinect pointing straight at a wall and found that it is was to unable to reliably measure depth values below 50 cm (see figure 2). These facts mean that only $d$ values of about 434 to 1084 represent actual measurable depth values. This is 650 unique disparity values, which is not a lot. Of these, all values up to 759 represent a depth of less than 1.0 m, meaning that half of all usable disparities output by the Kinect are in the range 50 cm to 1 m. And it falls exponentially; only 16 values in the range of disparities correspond to depths between 4 and 5 meters.

This dramatic decrease of fidelity was found to be correlating with my findings in figure 3 where a large gym ball was shot at two distances, approximately 2.5m and 80 cm. The difference in precision is very noticeable - at a distance ball is hardly discernible when viewed from the side, while at close point it appears round and smooth, which it what it is.

So not only will objects which are far away from the viewer consist of less points because they take up a smaller area of the picture, they will also be more coarse. The lesson here is that for any object more distant than 2.5-3.0 m, one cannot expect a faithful representation of its surface, and that any delicate details must be captured in a range of 60 cm to 1.5 m. Whether the hardware actually is precise at higher ranges is irrelevant because of the format in which it returns its measurements is of too low fidelity.

My guess, which is not backed up by any references whatsoever, is that the reason for this depth format is that as objects come further away from the viewer, the effect of binocular disparity lessens, and thereby also the precision of the depth measurement, so the engineers behind the hardware probably did not see much benefit in leaving any useful depth information in these ranges.

## 2.4 Artifacts from Structured Light

Because the camera and projector are offset by design, this means that not all points which are visible from the camera can always be illuminated by the projector. As a result holes may appear in the depth map where the projected pattern has been occluded (see figure 4)

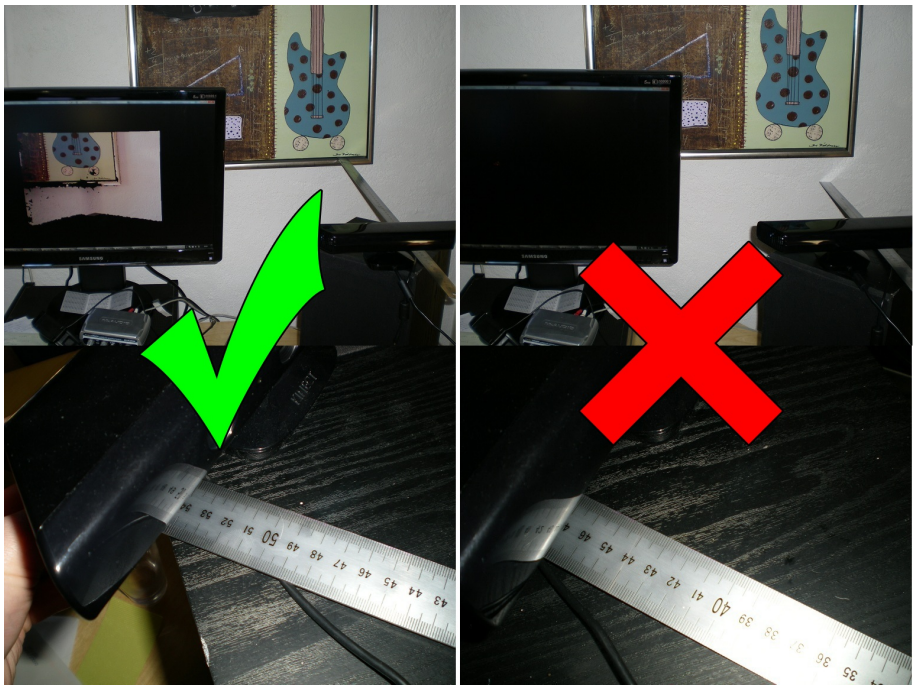

**Figure 2:** Emperical test of minimum supported depth of the hardware. Left: At over 50 cm, the wall is captured by the Kinect (as shown on the screen). Right: At less than 50 cm, it is not. While these distances are approximate, one should operate with the Kinect well beyond them for the best results.

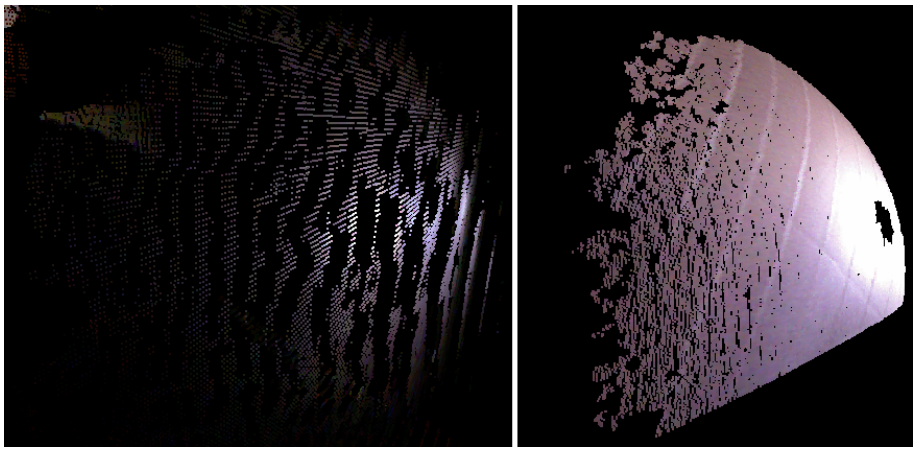

**Figure 3:** Example of how depth-map fidelity falls drastically as distances increase. A large gym ball was shot at two different distances to illustrate this. Left: the ball is over 2 m away from the camera. Right: less than 1 m away. Both clouds are shown from the side.

**Figure 4:** Example of projector-shadowing and IR interference caused by sunlight.

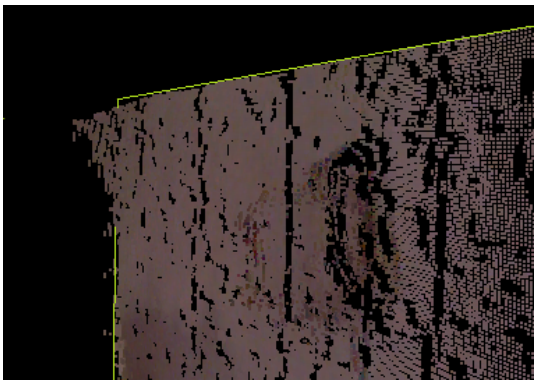

**Figure 5:** Edge of a point cloud warping towards the viewer. This is supposed to be a straight wall. The green line was added post-capture to give an indication of where the edges of the wall should be.

In general, infrared structured light will have many errors in environments consists of anything but diffuse surfaces (at IR wavelengths), or environments with other sources of infrared light. This could be the light of the sun, which created a hole in the test object (a gym ball) in figure 4 due to too much interference with the projected pattern.

## 2.5 Inconsistencies across multiple clouds

Most materials do not spread incoming light evenly in all directions. Solid materials more often than not have some amount of specular reflection. This means that one point on a surface will most likely change appearance when captured from different viewpoints. In the process of capturing a scene which is illuminated by sunlight the weather may change slightly, and if it does then so will the appearance of the shades, shadows and colors. Another source of change in lighting conditions may even be the user of the Kinect moving around the scene.

**Figure 6:** Inconsistencies across multiple pointclouds which together make up part of a room. The back wall shows signs of being constructed from several clouds with changes in apperance from cloud to cloud. The chances in shade are so large that it is hard for the eye to discern it as a single continuous wall.

So some times the precise shade of an object will be inconsistent across multiple point clouds, as it is shown in figure 6.

What this means to the registration is discussed in the individual algorithms as each of them are impacted differently.

## 2.6 OpenNI/Kinect Stock Calibrations

For the purpose of capturing point clouds, and computer vision in general, cameras provide the nicest data when they capture pictures in the way that an idealized pinhole camera would. Without going into too many formal details, it means that there should be no distortions of any kind. Most 3D computer games use such a "perfect" camera model.

The ROS team empirically measured how much their Kinect cameras diverted from the pinhole camera model and found the difference to be very small[8]. This is good.

Inspecting the point clouds provided by the Kinect through Kinect at OpenNI at close point, OpenNI's calibration of the depth and color map is very decent out of the box. For distant objects, a near-perfect calibration is hard to acquire. Part of this can be attributed to the lowered precision of the depth measurement at longer distances, and is therefore hard to do anything about client side.

At last, the corners of any point cloud may warp slightly (see figure 5). This cannot be solved with calibration, as even hand-calibrated clouds have this issue, so it is more likely due to a flaw in the lenses of the IR projector or IR camera causing the edges to be distorted. As the IR data is used in the Kinect hardware, correcting these distortions properly would require a large effort. It is easier to remove the affected areas from the dataset if need be.

For purpose of this project, it was decided that the quality of the stock OpenNI calibration is not likely to be a major source of error, and that it therefore

is considered good enough, especially when taking into account that it would add extra layers of complexity to both development and usage of the software workbench to calibrate cameras manually.

# 3 The Registration Algorithms

In these sections, the theory and design of the registration algorithms which are to be tested on the data provided by the Kinect will be presented and discussed.

## 3.1 The Iterative Closest Point Registration Algorithm

The Iterative Closest Point (ICP) algorithm is a very common registration algorithm described in 1992 in a paper by Zhengyou Zhang[10]. It goes as following, where $C_m$ denotes a point cloud which is to be registered and moved to another point cloud with a static position, $C_s$:

1. For all points $p$ in $C_m$, find the nearest neighbor $p'$ in $C_S$.
2. Find the rigid transformation, an orthogonal rotation $\mathbf{R}$ and a translation $\mathbf{t}$, which minimizes the squared distances between the neighboring pairs (enumerated with $i$):
$$\min_{\mathbf{R},\mathbf{t}} \sum_i ||(\mathbf{R}p_i + \mathbf{t}) - p'_i||^2$$
3. Apply the transformation to $C_m$.
4. Repeat until the algorithm has converged or till a desired result has been obtained.

Worded differently, the algorithm simply picks a set of temporary correspondents naively across the two input clouds based on their distance, move them closer to each other to make the best fit, and then picks a new set of temporary correspondents based on the new positions of the point clouds, and repeats the procedure until they are not really temporary any more (the algorithm has converged). It is somewhat naive compared to the feature-based algorithm which will be described in the next section, but it apparently works well, especially when tweaked [10][12][16].

A prerequisite for a good registration is that the approximate motion between the viewpoints from which the two input clouds for the algorithm is known. Zhang described this in the introduction of his paper[10].

### 3.1.1 Extendability

The ICP algorithm has been presented in its most basic form as given by Zhang, and it is also in this form which it has been described in *Computational Geometry Processing*[15]. It is, however, quite extensible if one aims to improve its speed and/or precision. *Efficient Variants of the ICP Algorithm*, categorizes these areas to[16]:

- How points are *selected* for sampling across the two input clouds.

- How points are *matched* against each other. Alternatives to nearest neighbor searches exist.
- How matches are *weighted* against each other.
- Whether some matches should be *rejected* or not, and how.
- How the sum of squared distances (error) is *minimized* in step 2 of the procedure.

Zhang proposed some additional heuristics for rejecting matches based on a dynamically picked maximum length with the intention of improving the quality of registration in his paper [10]. This strategy allows matching of partially overlapping clouds. For this project, an alternative strategy will be taken in use. It is also based on rejection and is described in section 3.1.4.

Section 3.1.5 will discuss a matching extensions for taking the color information provided by the Kinect into account.

### 3.1.2 Selecting points and nearest neighbor searches

Because each point cloud often consists of more than 200000 points, searching for nearest neighbors for every point is infeasable. [16] shows that it is is not necessary to sample all points, because a smaller subset of randomly chosen points works equally well in many cases with similar convergence rate and error. So it has been decided to use random sub-sampling in the final implementation, picking points for nearest neighbor searches by through the moving cloud with random sized steps between, aiming for 4000 correspondents per. iteration.

Naive nearest neighbor searches can be done in linear runtime by computing the distance to all points in the target cloud and taking the smallest. But it is too expensive at a larger scale. Fortunately, there exists numerous of algorithms and data structures to speed up such a search. The most efficient is likely to move it to the GPU[11], but such an optimization is beyond the scope of this project. Instead, in order to obtain acceptable run times, an off-the-shelf kD-tree implementation from the GEL library has been used in the software for this project. Zhang also used a kD tree to speed up his ICP implementation[10].

### 3.1.3 Finding the rigid transformation between corresponding points

The following instructions are a paraphrase of the instructions given in *Computational Geometry Processing*[15] for finding the rigid transformation between a set of corresponding points.

The translation part $\mathbf{t}$ of the rigid transformation is defined as the vector which will translate the center of mass of the selected points in $C_m$ to the center of mass of the correspondents in $C_s$:

$$\mathbf{t} = \frac{1}{n}\sum_{i=1}^{n}(p_i - p_i')$$

Here, $n$ denotes the amount of matched pairs found in step 1 of the algorithm, and $i$ is used to enumerate these point pair entries so that $(p_i, p'_i)$ denotes the $i$'th pair. $j$ and $k$ are also for enumeration. Notice that some points of $C_s$ may be the closest neighbor for several points in $C_m$, meaning that they will be weighted more in the center of mass calculations. This is intentional.

Now onto $\mathbf{R}$. If we have a matrix $\mathbf{H}$, defined as:

$$\mathbf{H} = \sum_{i=1}^{n} p_i {p'_i}^T - \frac{1}{n}(\sum_{j=1}^{n} p_j)(\sum_{k=1}^{n} p'_k)^T$$

Then from the singular value decomposition of $\mathbf{H} = \mathbf{U\Sigma V}^T$, it can be shown that $\mathbf{R} = \mathbf{UV}^T$. If there are some errors or some noise in the dataset, it will still work.

After I tried this procedure out in practice, I found that it did not work as intended. But after a little trial and error, it turned out that finding $\mathbf{R}$ and $\mathbf{t}$ once, applying the transformation, and then doing it again seems to give the desired result. This does, however, mean that the neighbor pairs must be stored explicitly in between those two iterations, as they are needed for calculating $\mathbf{t}$ and $\mathbf{H}$ twice. Alternatively they must be calculated again, which can be very expensive.

After spending too much time on the problem, it was decided not to dwelve further into it as it two iterations just seem to work. The software needs a rigid transformation, and this solution provides it.

J. Walther had a similar problem in his project, also on the Kinect and ICP[12].

#### 3.1.4 Partially overlapping point clouds

ICP needs to be extended to support partially overlapping point clouds. Figure 7 illustrates why. It shows two partially overlapping point sets, a blue and a red one. Attempting to register the blue onto the red with ICP will create a set of neighbors in the first step of the algorithm. These neighbor-relations are implied by the black lines connecting the points of the sets, and they illustrate a problem: All blue points in the green-tinted area have the same neighbor: the left-most point of the red point set. All but one of these neighbors are bad correspondents, and taking them into account when calculating the rigid transformation will likely result in a bad result.

In *Zippered polygon meshes from range images*, a solution for this has been presented[13]: discard all correspondents which contain such "edges" like the one which the points in the green tinted area are linked to. "Edges" is a loosely defined term, and for point clouds from arbitrary sources it might be a hard problem to define them. But with data from the Kinect, it is more gracious. I define these edges as discontinuities in the depth map from which a cloud has been generated, as it represent discontinuities in the surface which it is capturing.

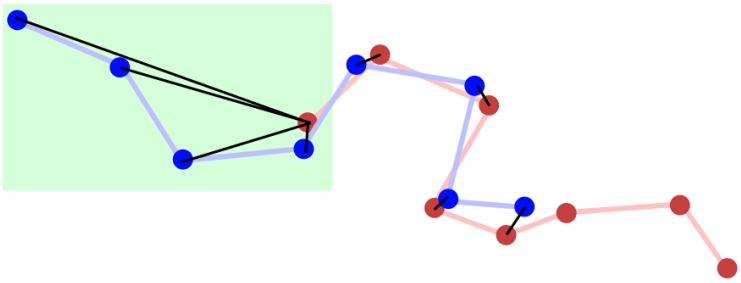

**Figure 7:** Two partially overlapping sets of points. It has been attempted to find a nearest neighbor (and proposed correspondent) from the red set for all points in the blue set. However, for all those blue points but one in the green-tinted area, the proposed correspondent is bad because the red set is discontinued. This shows why matched pairs containing edges must be discarded.

### 3.1.5 Adding color information to ICP

ICP finds correspondents based solely on their locations. But not only is one point being close to another an indicator of correspondence, so is a similar color. So by extending ICP to take similarity of color into account, better correspondences might be obtained. The paper *3d-3d registration of free formed objects using shape and texture* proposes a solution for this[14]. It requires changes to the nearest neighbor search.

Instead of measuring distances between two points based solely on their position in an $(x, y, z)$ coordinate system, distances between more abstract vectors in a $(x, y, z, [color])$ space can be used to determine correspondence. $[color]$ is one or more elements containing values related to color information. For example hue or RGB, i.e a 4-dimensional $(x, y, z, H)$ or a 6-dimensional $(x, y, z, R, G, B)$ search space. In this way, both color and spatial spaces are searched at the same time for correspondents.

One thing to take into account is the relative size of $(x, y, z)$ and $([color])$ spaces, because they run in different intervals. For a point cloud from the Kinect in a medium-size living room, the intervals of the spatial coordinates run somewhere between 1.5 and 4 meters. Scaling the color space so that it runs over larger or smaller intervals will decide how the two spaces are weighted against each other when finding correspondents.

There however is a pitfall when adding color information to the nearest neighbor search in partially overlapping clouds. Points whose closest spatial neighbor in their target cloud is an edge must be discarded when calculating the rigid transformation. But when searching the combined $(x, y, z)$ and $([color])$ space, the closest point might not be an edge when it should be. This is the case in figure 8 where the left-most blue points, those in the green-tinted area, are matched to a non-edge even they should be discarded for not have a correspondent in the other point set. Therefore, when searching combined spaces for partially overlapped point clouds, a separate search must be made in $(x, y, z)$ space only in order to determine whether a point-pair should be discarded or not.

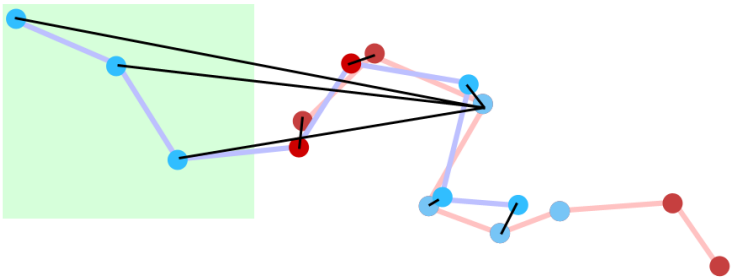

**Figure 8:** Two partially overlapping sets of colored points in which the nearest neighbor search for the points in the green-tinted area has given a set of bad correspondents. These points should be discarded, but as their common proposed correspondent is not an edge, this cannot be used to rule them out. Therefore a separate nearest neighbor search which does not take colors into account must be performed to rule them out.

### 3.1.6 Color mappings and inconsistencies

Different color mappings will have different strengths and weaknesses when adding color information to the nearest neighbor search space. When deciding which one to use, the fact that color information is likely to be somewhat inconsistent (section 2.5) has to be taken carefully in account in the decision making process.

RGB is a straightforward color mapping. In clouds with no inconsistencies, it is very precise because it provides 3 channels of independent color information for each point. But it is not invariant to changing light conditions, as figure 9 seeks to illustrate. It shows two captures a scene, in between which the direction of incoming light has changed. In image A, the spheres are lit from above. In image B, the spheres are lit from the side in a fashion so that the large green sphere is shadowing the the little red sphere, resulting in a much darker shade. As a result, attempting to find correspondents between the two images by measuring the distance between two $(r, g, b)$ vectors will result in a bad match: the light part of the red sphere in image A and the light part of the green sphere in image B. The reason for this is that the RGB values for these bright areas are are closer to each other in $(r, g, b)$ space (R: 254, G: 224, B: 244 vs. R. 208, G: 248, B: 196) than they are to the RGB values for the ball in the shade (R: 99, G: 35, B: 35).

However, in figure 9, the hue of the red ball does not change in between the two images. Therefore searching a $(x, y, z, H)$ space may be a more robust way to improve searches for correspondents, even across somewhat inconsistent datasets.

The hue of a color is represented by an angular value in a cyclic interval rather than a scalar in a linear interval like that of the elements of RGB. This means that a hue of $1°$ is closer to a hue of $359°$ than to a hue of $5°$. So distance evaluations in $(x, y, z, H)$ space should be changed to reflect this. But is it necessary? What if this cyclic property is ignored? All hue values can be fitted to a $-180°..180°$ interval, and then used them as if they are just normal scalar values. Cyan shades will end up in arbitrary ends of the interval because of

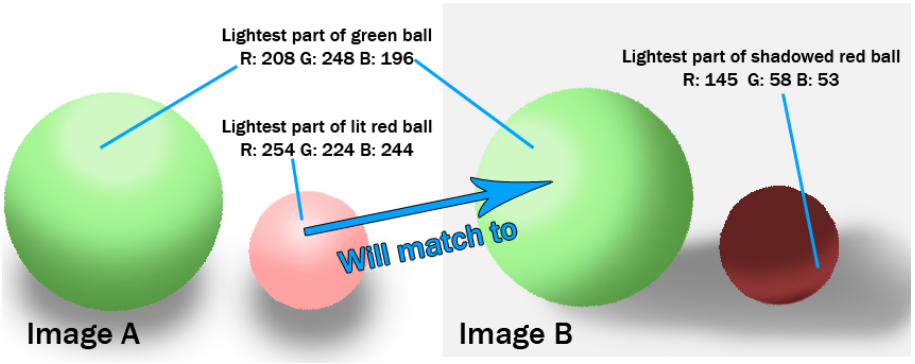


**Figure 9:** An illustration of how comparing colors based on RGB values in scenes with changing light conditions may not work well. Here, the distance from the lightest color-value of the red ball in image A is

**Figure 10:** Hue values mapped to a non-cyclic interval. Even though the cyan colors at the extremes of the interval are visually close to each other, measuring the distance between them with this mapping with not reflect that.

they lie on the discontinuity of the interval with a hue of ca. $180° = -180°$. But all other hues are going to be unaffected. Since hues are not expected to change much from image to image, unless there are a lot of cyan objects in the scene, only a minor part of the interval is going to be rendered unusable. So for the sake of simplicity, it can be chosen to ignore the cyclic property in an implementation of this mapping.

The weakness of hue is that it may not perform well in scenes with large areas with low color saturation, such as which walls, grey stones or black tiles, because the hue of non-saturated shades is undefined.

## 3.2 The Feature-based Registration Algorithm

The feature-based registration algorithm is similar to ICP in that it searches for a set of matched pairs across the two input clouds, and then finds the rigid transformation which minimizes the sum of squared distance between them. However, instead of finding a temporary set of correspondents based on some heuristics, it aims to find a set of actually correct correspondents across the two input clouds using detection and matching of invariant features. Invariant features are points of interest in an image which are likely to remain invariant as the viewpoint of the scene is changed. They are designed for recognizing objects and corresponding between across multiple images[17].

The algorithm goes as following:

1. Detect invariant features in the color-images from which $C_m$ and $C_s$ obtain their color information.

2. Extract two sets of feature-descriptors from the input images and match them against each other.

3. Extract a set of matched pairs in 3D-space by corresponding the location of each feature to the point in the point cloud which it maps to.

4. Find the rigid transformation which minimizes the squared distance between the matched pairs in 3D space. One can use the same procedure for this as described for ICP in section 3.1.3.

This algorithm has a good synergy with ICP, but different prerequisites. While it does not require points to be somewhat aligned initially like ICP does, it is dependent on the data coming from the kind of data source which the Kinect provides. The synergy is, that if it does not result in a perfect registration but just an approximation, such an approximation will be a good starting point for ICP.

### 3.2.1 Implementation considerations

Finding corresponding features between two images is a complex problem. The freely available library OpenCV (Open Computervision) offers multiple ready-made solutions which can be used to do that. The names of three of these solutions are SURF, SIFT, and FAST, and they can all detect and describe invariant features in input images which can be matched against each other with specially tailored matching algorithms which are also available in OpenCV.

The algorithm is adaptable to several of these solutions for finding correspondences in the two source images which make up the input clouds. In the product, SURF[17] was chosen because:

- It is available in OpenCV and easy to use.
- It is more than an order of magnitude faster than SIFT and performs comparably well[18].
- It performs well in images with changing light conditions[18], and such is the nature of the data on which it will be working (see section 2.5).
- I performed practical comparisons with some of the other solutions available in OpenCV (SIFT, FAST) and found that SURF was by far the best for the job in terms of precision and speed.

The descriptor matching in OpenCV is not designed specifically for rigid pose estimation in images from static scenes, or to take into account that a third dimension can be added to the location of the descriptors (if it evens takes location in an image into account when matching).

When detecting features with SURF, there are multiple input parameters will change the outcome of the process. OpenCV has made some wrappers called *adjusters*, which abstracts these parameters away in favor of an interval in which the user may signify the desired amount of features to be detected. This wrapper

may use several iterations of detection to adjust the underlying parameters to give a result which fits this interval, but in return it means that it is hopefully more robust and adaptable to different scenes. What it also means is that features and their matches are not considered to be consistent from registration to registration. This is considered a feature as it will give a few extra options to the user as for whether to re-register input data for a slightly different (and possibly better) fit.

SURF does not support color images, so all images has to be supplied to the algorithm in gray scale.

### 3.2.2 Registering to multiple clouds at the same time

When matching clouds, then the more they overlap, the more correspondents they will have. This is a trivial observation. Therefore it makes sense to capture clouds from only slightly different viewpoints when reconstructing a scene. Also, when reconstructing a room, the camera is going to be at similar viewpoints many times. This means that a newly captured cloud is likely to have correspondents in multiple clouds rather than just one.

I came up with a procedure for matching one cloud against several clouds at the same time. Something like it is also described in the Willow Garage presentation in the references[19]:

- Register the newly acquired cloud onto the previous one. Keep track of the amount of matches found, $N_{prev}$.
- Register the cloud against all clouds in the scene, optionally ignoring those which are unlikely to contain matches (based on viewpoints) for speedups. Keep track of all sets of correspondents.
- For each new set of correspondents, discard it if it has a size $N < \frac{1}{3}N_{prev}$ or if $N < 10$ in order to increase the robustness of the registration.
- Construct a rigid transformation from the remaining sets of correspondent.

The immediate benefit from registering against multiple clouds at the time is that it evens out errors across multiple point clouds, rather than accumulating them in the fashion that always matching against single clouds will. Due to the nature of these errors, it was hard to illustrate in a figure, but two files, `erroraccumsingle` and `erroraccummulti` has been included with this report as examples for to explore on a capable computer. These are the results of a reconstruction in which the viewpoint has swept from the left side of a room to the right, and then back to its initial position. Both clouds have been registered with feature-based matching only, but in `erroraccummulti` every new cloud has been registered to multiple targets, whereas `erroraccumsingle` has not. Over the course of this sweep, errors have accumulated in `erroraccumsingle` which result in the ceiling in the left side of the room appearing in two layers rather than one. This is not the case for `erroraccummulti`.

# 4 The Software

In order to properly test the registration algorithms in a real world scenario, and to actually make them usable to end users rather than developers only, a software workbench for point cloud capturing and scene/object reconstruction has been developed. Visually, it consists of a console and a graphical interface with a control panel for interaction and some space on which point clouds are rendered (see figure 11 for screen shot). Points clouds can be explored by clicking and dragging the mouse to rotate the direction of the camera. The `WASD` keys moves cameras position around in the horizontal plane, and `R` and `F` will raise and lower its height.

The work flow of the software is as follows: A new cloud is captured with the Kinect using the "Grab new cloud" button. It will be displayed, and can be explored accordingly. Then another cloud may be captured from a similar viewpoint. It can be registered onto the first with the feature-based registration algorithm or ICP at the click of a button. It is possible to use three different color spaces for ICP: none, hue, or RGB (as described in section 3.1.5). If the user is unsatisfied with the registration, then another type registration can be performed, or the most recently captured cloud can be deleted in order to make an attempt to capture a cloud which will register more easily.

After a feature-based registration has been performed, the "Show latest matches" button will show features and correspondents across the two most recently matched clouds' color images sources.

Every time the user is satisfied with a registration, a new cloud can be captured. The software assumes that this is done from a viewpoint close to the most recently registered cloud in the scene, so it will automatically move the new cloud onto that. If a registration has a bad transformation and the user wants to try another solution, the cloud can always be moved back to this position using the "Put onto previous" button, even if a transformation results in a matrix of NaN's from an unlucky feature-match with too many outliers.

Depth filtering of the most recent input cloud is also an available feature. This is useful when scanning objects, or if the user wants to match scenes based only what is close to the camera. The depth threshold can be set by the user, and it can be set automatically apply to every newly scanned cloud. It is not destructive in the sense that filtered clouds are always reconstructed from the source images of the point cloud, meaning that even if an area has been excluded from a cloud by a previous filtering action, it can be recovered by applying a filter with a deeper threshold.

Most of the functionality of the software has a keyboard mapping, except for saving and loading. The mapping is designed to be used with one hand, the left, so that users may move around the Kinect with his/her right hand. Pressing `TAB` will grab a new cloud. `Q` will match the most recent cloud against multiple clouds using feature-matching. The `1` key will match against a single cloud using the same technique, and `2` will match using ICP, for which the amount of iterations can be controlled with the `+` /`-` keys. The most recent cloud can be deleted using the `X` key, and the entire scene can be cleared with `CTRL+X`.

For simple usage, the "auto match" option will automatically match new clouds with feature-based matching against multiple targets.

## 4.1 Implementation

The two registration algorithms has been implemented in C++ on a Windows platform in the Visual Studio 2010 IDE. Libraries and technologies used in this implementation are OpenGL, GLUT, GLEW, OpenNI, OpenCV (for feature-detection and matching), AntTweakBar (a simple GUI system), and the CGLA+kD tree part of GEL (Geometry and Linear Algebra) which has been slightly customized. All of these technologies exist on multiple platforms. The source-code is OS agnostic except for some pragmas to include libraries (.libs), along with two tiny functions used to spawn simple worker threads and with a header to facilitate this. It should be very trivial to re-target for Linux or Mac OSX depending on how hard it is to set up OpenNI on these platforms (I have no clue).

The data structure *PointCloud* stores 3D points and their colors, along with additional information like: the color/depth images from which they were generated, which points are edges, and the rigid transformation of the cloud. Besides being represented in main memory, point clouds are also stored on the GPU improved rendering efficiency. Management of buffers is done by the *PointCloud* container class *CloudStore* which handles all clouds in the scene. Clouds are rendered using OpenGL's old fixed function pipeline.

Registration is offloaded to separate threads in order to keep rendering smooth and to make it possible to see ICP work as it progresses over multiple iterations, all while exploring the data. Care has been taken to ensure that no race conditions caused by multi threading will take place and that only one registration is done at the time. This is easy though, as GLUT does everything in a single-threaded, serialized fashion, making locking as simple as setting a boolean variable with no spin locks needed. Capturing maps from the Kinect and processing them into point clouds is handled by the main thread, so the UI becomes unresponsive while this is happening. But as this process only takes a fraction of a second, it is simply not noticeable, especially because the camera is likely to be static while this is happening.

The software has features to save and load point cloud data. It saves it in a simple binary format which contains multiple point clouds, and includes source images and transformations. The first 4 bytes of a point cloud file is an integer corresponding to the amount of clouds which it stores, and then the remaining data is a series of point clouds which have been serialized using *PointCloud::serialize*. All this has been implemented in a functional yet slightly archaic way using C-style file access, and while it is extremely functional it is not very tolerant to bad input data.

The file format does contain some redundant information, as it could just store the color/depth maps used to generate the clouds, their transformations, and whether they were filtered or not.

Some values, like the 640x480 resolution of the input data, is hard coded into the software, but it should be trivial to adopt to other resolutions for other

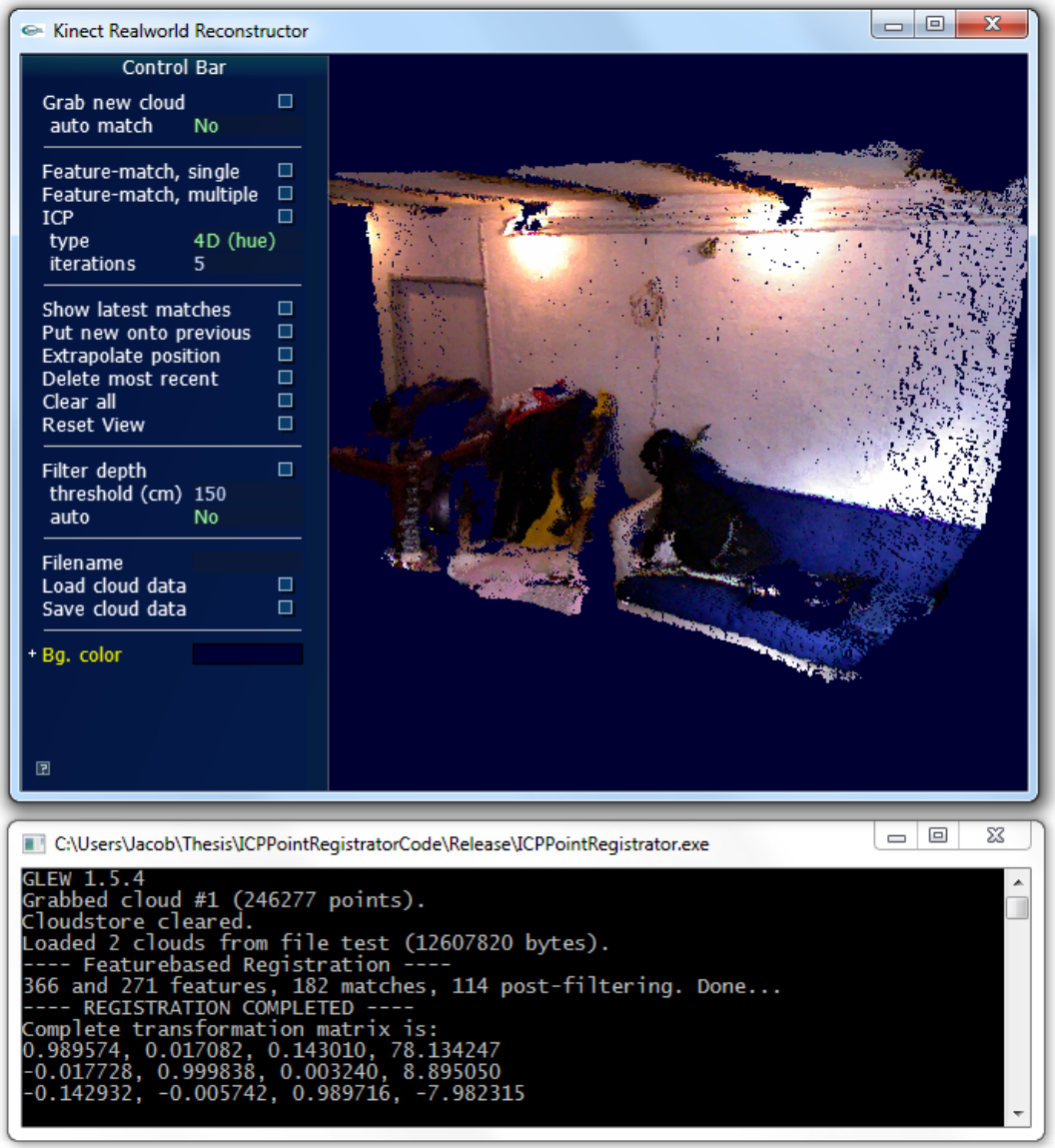


**Figure 11:** The user interface after registering two points clouds onto each other with feature-based registration. The console shows useful information about the process, like how many features were found in the images, how they matched, and what the transformation between them looks like. Not shown is the window which opens when pushing the "Show latest matches" button which will show the latest matches between the images from which the color information of the registered point clouds was obtained.

applications if need be.

The dependencies of the software are the same as described on Nicolas Burrus' RGBDemo v.0.5.0 page[3].

### 4.1.1 Notes on OpenNI and reverse perspective projections

In the development of the software for this project, it was found out with a little tinkering, trial and error, how OpenNI convertes between projective and Cartesian coordinates, given a 640x480 color and depth map which has been calibrated against each other with OpenNI's stock calibration, which itself will be discussed later. First of all, the depth value, $z$, is consistent between both coordinate systems. Given a set of image coordinates and their corresponding depth, $(x_p, y_p, z)$, their Cartesian equivalents $(x_c, y_c, z)$ can be found using the formula:

$$x_c = (x_p - 320) \cdot z/C$$

$$y_c = (y_p - 240) \cdot z/-C$$

Converting back to projective coordinates then becomes:

$$x_p = 320 + C \cdot x_p/z$$

$$y_p = 240 - C \cdot y_p/z$$

Where $C = 575.81573$. It should be noted, however, that the implementation flips Cartesian $z$ coordinates to fit with OpenGL's coordinate system.

# 5 Results

Tests were designed to test the hypothesis of this project, namely whether the Kinect is a mature solution for the problems mentioned in the motivation when paired with the ICP and feature-based registration algorithms.

From the information gathered in the qualitative analysis of the point clouds supplied by the Kinect in section 2, it can already be concluded that the device is fit for neither small objects nor overly large scenes due to its resolution and lack of fidelity at distances over 2.5 meter. But what about medium sized objects and rooms?

Tests were carried out a in basement room, all timings are from a dual core 2.8 GHz Athlon 64 x2 5600+ host with a fast GPU.

## 5.1 Object: Gnome

A 50 cm tall garden gnome was set on top of a makeshift pedestal (a stack of books) for scanning, and was then surrounded by light sources in order to try to keep it somewhat uniformly lit from capture to capture.

For each new capture, the pedestal was rotated a little around itself. A depth threshold filter of 100 cm was automatically applied to incoming data, and it efficiently removed the background, leaving the gnome by itself.

**Figure 12:** Convergence of different methods of ICP on the gnome. **Upper left**: ICP without color information. **Upper right**: ICP with hue information in a size 500 color space. **Lower left**: ICP with RGB info in a size 500 color space (best of the four). **Lower right**: ICP with with RGB in a size 3000 color space, showing the result of a unbalanced weighting between color space and location.

The feature-based matching algorithm did not do well. With only a handful of matches across the datasets, the presence of a few outliers which had made it through the filtering made it either unusable or imprecise at best. However, ICP worked quite well for this situation, especially with an added hue dimension, likely due to all the bright and fairly consistent colors. Different sizes of color spaces were experimented as shown in figure 12 where the effect of an overly large color space can also be seen.

The final result as shown in figure 15 was quite satisfactory and did not require terribly hard work to produce. Filtering off certain points would have improved the consistency of the figure immensely as the edges of the clouds contained stray pixels which added dark patches to the figure, most notably in its face.

Runtimes: ICP 3D: 50-150 ms. ICP 4D: 200-300 ms. ICP 6D: 700-1700 ms. All times are per. iteration.

File: `thegnome.cloud` (38 MB), `gnomeicptest.cloud` (7 MB)

## 5.2 Object: Leather Chair

It was attempted to scan a leather chair, with partial success (see figure 13. The chair design consists of a lot of horizontal pieces of wood which hold up the back. These horizontal pieces confused the ICP because of their pattern around the whole chair, so moving the viewpoint of the camera was a somewhat delicate process.

**Figure 13:** A partially reconstructed leather chair from different viewpoints.

For this reconstruction, being able to guide the pieces to approximate positions would have helped immensely.

Runtimes were similar to the gnome, but 6D ICP spiked at 4 seconds for some iterations.

File `chair.cloud` (110 MB)

### 5.3 Scene: Basement room

An attempt was made at scanning a room. First in free-hand. After hours of frustration over imperfect matches, the room was filled with items which could help improve the registration by adding extra texture to the scene for the feature detectors to recognize features in. This helped slightly. In the end, a decent, near-360° capture of the entire room was built from 50 carefully matched clouds. This was done after nightfall in purely artificial light. The Kinect was put on a pivot to aid stability. The result was a scan of the room which ended up suffering from accumulated errors which made the ceiling spiral slightly into the ground, and which broke the right-angled edges of the room. See figure 14.

Runtime for each feature-based match: 150-500 ms. When matching multiple clouds, up to several seconds. This could be sped up by caching the features in the target cloud from iteration to iteration.

`posterroom.cloud` (312 MB)

## 6 Conclusion

The output data of the Kinect was analyzed, and an extended version of ICP and a feature-based algorithm registration algorithm was described and implemented. The performance the several extended versions of ICP, and feature-based matching was tested in a set of real-world situations which can be linked to the motivation for doing this project. It was found that the Kinect is able

**Figure 14:** Top-down view of a rectangular room captured with the Kinect and registered using the feature-based registration algorithm described in this report. It shows how errors accumulate. The red line is the path of the walls, the green guide is how it should be.

to scan some medium-sized objects well with the right capturing strategies, and that it is quite capable of scanning medium-sized rooms with a little patience, but that errors accumulate to such an extent that explicit strategies are needed to keep the final reconstruction consistent. It was also found that ICP was more effective than feature-based matching for object reconstruction, and vice-versa for room reconstruction. It would possibly have been wise to focus on one of these aspects of reconstruction rather than both.

Suggested future work:

- A user interface which can be used for guided reconstruction.
- A system for filtering data points in reconstructions improve consistency.
- Implementing loop closure (large-scale error correction) to the system.

# References


[1] The Kinect article on Wikipedia

[2] Peter Henry, Michael Krainin, Evan Herbst, Xiaofeng Ren and Dieter Fox: *RGB-D Mapping: Using Depth Cameras for Dense 3D Modeling of Indoor Environments*, at International Symposium on Experimental Robotics, 2010.

[3] Nicolas Burrus, *RGBDemo v0.5.0* `http://nicolas.burrus.name/index.php/Research/KinectRgbDemoV5`

[4] Felix Endres, Juergen Hess, Nikolas Engelhard, Juergen Sturm, and Wolfram Burgard: *RGBDSLAM* on OpenSlam.org.

[5] Felix Endres, Juergen Hess, Nikolas Engelhard, Juergen Sturm, Daniel Kuhner, Philipp Ruchti, and Wolfram Burgard: *RGBD-6D-SLAM.* `http://www.ros.org/wiki/openni/Contests/ROS%203D/RGBD-6D-SLAM`

[6] Sean Anderson, Kirk MacTavish, Daryl Tiong, and Aditya Sharma: *Kinect Visual Odometry Mapping.* Part of a Fourth Year University of Waterloo Design Project. Videos on YouTube from user *shyxguy.*

[7] Microsoft Kinect SDK
`http://research.microsoft.com/en-us/um/redmond/projects/kinectsdk/`

[8] ROS.org: *Technical aspects of the Kinect device and its calibration.* `www.ros.org/wiki/kinect_technical/calibration`

[9] Nicolas Burrus, *Kinect Calibration*
`http://nicolas.burrus.name/index.php/Research/KinectCalibration`

[10] Zhengyou Zhang, *Iterative Point Matching for Registration of Free-Form Curves and Surfaces*, 1994.

[11] Vincent Garcia, Eric Debreuve, Michel Barlaud, *Fast k nearest neighbor search using GPU*, Computer Vision and Pattern Recognition Workshop, pp. 1-6, 2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops, 2008

[12] Jeppe U. Walther, *ICP point cloud alignment: Using Kinect depth camera*, January 2011. Report from DTU course 02507.

[13] G. Turk and M. Levoy: *Zippered polygon meshes from range images.* In *Computer Graphics Proceedings, Annual Conference Series*, pages 311 318. SIGGRAPH, 1994.

[14] F.C.M. Martins, H. Shiojiri, and J.M.F. Moura. *3d-3d registration of free formed objects using shape and texture.* In *Proceedings of the SPIE - The International Society for Optical Engineering, pages 263274*, 1997.

[15] Francois Anton, Jakob Andreas Bærentzen, Jens Gravesen, and Henrik Aanæs: *Computational Geometry Processing*, 2008.

[16] Szymon Rusinkiewicz, Marc Levoy: *Efficient Variants of the ICP Algorithm*, In Proceedings of the Third Intl. Conf. on 3D Digital Imaging and Modeling (2001), pp. 145-152.

[17] Herbert Bay, Andreas Ess, Tinne Tuytelaars, Luc Van Gool, "SURF: Speeded Up Robust Features", Computer Vision and Image Understanding (CVIU), Vol. 110, No. 3, pp. 346–359, 2008.

[18] L. Juan, O. Gwun: *A Comparison of SIFT, PCA-SIFT and SURF*, International Journal of Image Processing (IJIP), Vol. 3, No. 4. (2009), pp. 143–152.

[19] A presentation by Dirk Holz of University of Bonn for Willow Garage: *::pcl::registration: Registering point clouds using the Point Cloud Library.* January 27, 2011.

**Figure 15:** The gnome from different perspectives.

**Figure 16:** Attempting to illustrate how the ceiling has skewered into the ground compared to its original path.

**Figure 17:** A registration error in the room scene, marked with the green ring.

**Figure 18:** Another overview of the room.