CHALMERS

3D head scanner

Master of Science Thesis

MAGNUS JEDVERT

Chalmers University of Technology
Department of Computer Science and Engineering
Göteborg, Sweden, August 2013

3D head scanner

MAGNUS JEDVERT

Examiner: GRAHAM KEMP

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden, August 2013

Abstract

The advent of cheap depth cameras such as Microsoft Kinect together with modern reconstruction algorithms implemented for the Graphics Processing Unit (GPU) offers the potential of many new exciting applications. In this thesis, a scanning booth equipped with three Kinect cameras is built, where a user can scan their head and upper body into a high-quality textured 3D model. This is done using a variant of the KinectFusion algorithm, adapted to work with multiple cameras. The system operates in real-time and the reconstructed model is presented within seconds.

Acknowledgements

Most of this project was done at Cybercom, a company located at Lindholmen, Göteborg. The company is interested in applications and development of new techniques in 3D perception. The idea was that potential results from this thesis may be used as a booth in the Universeum Science Discovery Center.

I would like to thank my supervisor Hans Forsberg and my other colleagues at Cybercom for giving me the opportunity to do this thesis. I have enjoyed many lively discussions in which I learned a lot, especially with Hans Forsberg and Jonas Rangsjö.

I would also like to thank my examiner Graham Kemp for his support and guidance during this work.

Finally, I would like to thank Lina for her patience and support during this period.

Contents

1

Introduction

Capturing a high quality model of a human head is of interest in a variety of domains, primarily for the entertainment industry in the creation of 3D models for movies and games, but also for applications in medicine, and for archival purposes. Commercial solutions for scanning human bodies and heads already exists, but these systems are often very expensive and often requires an expert to operate. These systems may also spend a lot of time computing before completing the final 3D model.

1.1 Objective

The objective in this thesis was to build a system using commodity hardware, where a user could scan their head and upper body in 3D. The goal was a system that was robust, simple to use, and in real-time, so that it could be used without an operator as a booth in the Universeum Science Discovery Center.

1.2 Motivating technologies

Several technologies have come together recently to inspire the direction of this work.

The biggest is the recent availability of cheap 3D depth cameras, particularly the Kinect camera from Microsoft, originally intended as a motion sensing input device for the Xbox 360 game console. It was instantly recognized by researchers and the robotics community as an affordable camera that could capture 3D images in real-time.

Another motivation is the recent progress in algorithms for simultaneously tracking camera pose and incrementally reconstructing the underlying 3D model, using data acquired from different viewpoints. In particular, the novel fusion algorithm KinectFusion [1] inspired this thesis.

Highly related to these algorithms is the emergence of the Graphic Processor Unit (GPU) as a general purpose computing device, found in nearly any PC, and with mas-

sively more computational power than the CPU. This enables many algorithms to be run in real-time.

1.3 Related work

The introduction of the Microsoft Kinect opened up a new field for 3D camera tracking and reconstruction. One of the most prominent approaches is the KinectFusion algorithm [1] that demonstrated a real-time dense 3D scanning system for static indoor scenes. Several extensions of this algorithm has since been made, for example expanding the scanning area using a moving volume [2], [3], and changing the representation to an octree [4]. Also, at least one commercial solution using this algorithm has appeared [5].

Some other approaches using the Kinect, specifically targeted at scanning human bodies have appeared. In [6], they use a Kinect to generate a body shape by fitting a number of parameters from a body model to the Kinect depth and color data. The 3D model is then generated from this template body model using the estimated parameters. This approach is not sufficient for this work since the goal is to reconstruct a detailed personalized head and upper body.

In [7], they attempt to reconstruct the full human body using three Kinects and a turntable. They solve the problem of the person not standing still during the capture process by a global non-rigid registration where each body part is aligned separately. Although this technique allows to capture the full body, the details and resolution is not as good as from KinectFusion. Also, even if this reconstruction algorithm only takes about 6 minutes, it is still too slow for a booth at Universeum.

Therefore, the approach in this work is based on the KinectFusion algorithm [1], modified to work with multiple cameras. Also, the speed and accuracy of the original algorithm is improved by tailoring it to the specific purpose as a 3D version of a photo booth. To mitigate the problem of non-rigid motion, only the head and upper body is reconstructed. This paper provides a detailed description of the implementation, including hardware setup and how the model is textured and post-processed. Finally, the system is evaluated and a large number of scanning results are presented.

1.4 Outline of thesis

The next chapter 'Preliminaries' describes some basic theory needed. It also describes the Kinect hardware and how it works, and how to calibrate it.

Chapter 3, 'Method', describes the software implemented in this thesis, and the hardware built.

Chapter 4, 'Result', presents some scanning results, and a discussion about the robustness of the system.

Finally, Chapter 5, 'Conclusion', gives a summary of the project and some concluding remarks.

2

Preliminaries

2.1 Homogeneous coordinates

Homogeneous coordinates are common in computer graphics because they allow operations such as translation, rotation, scaling and perspective projection to be implemented as matrix operations.

A vector in three-dimensional Cartesian space is extended homogeneously using the mapping

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} \rightarrow \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

and the reverse operation back to Cartesian space is called the *homogeneous divide*

$$\begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} \rightarrow \begin{pmatrix} x/w \\ y/w \\ z/w \end{pmatrix}.$$

Transformations of these homogeneous points are represented as 4×4 matrices. Translation, which is a non-linear transform in $\mathbb{R}^3$, can now be represented linearly with

$$\begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x + t_x \\ y + t_t \\ z + t_z \\ 1 \end{pmatrix}$$

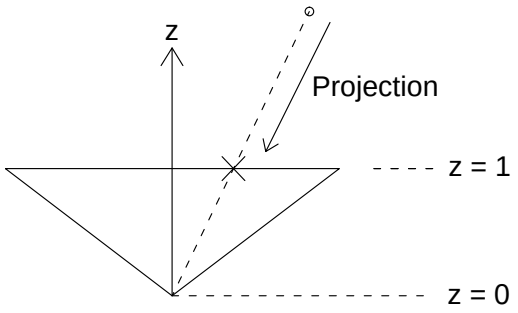


Figure 2.1: Perspective projection.

Another type of transformation that becomes linear is the *perspective projection*. When the human eye views a scene, objects in the distance appear smaller than objects close by, and this is known as perspective. The simplest perspective projection uses the origin as the center of projection and $z = 1$ as the image plane, as in figure 2.1. This transformation can be represented as

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \\ z \end{pmatrix} \rightarrow \begin{pmatrix} x/z \\ y/z \\ 1 \end{pmatrix}.$$

Rotation is a linear transform in $\mathbb{R}^3$ already, and is represented with the matrices

$$\boldsymbol{R_x}(\theta) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \quad \boldsymbol{R_y}(\theta) = \begin{pmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix},$$

$$\boldsymbol{R_z}(\theta) = \begin{pmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Transformations are composed together with matrix multiplication, usually with any projection as the last operation.

2.2 Iterated closest point

Later on, we will face the problem of geometrically aligning two 3D models together. This will be solved as an optimization problem where the goal is to minimize the sum of

squared distances between the two models. One of the models will be a surface $\mathcal{S}$ in 3D space, and the other model will be a set $\mathcal{P}$ of 3D points (a point cloud), so the objective function F to be minimized can be expressed as

$$\mathrm{F}(\boldsymbol{T}) = \sum_{p \in \mathcal{P}} \boldsymbol{dist}^2(\boldsymbol{T}p, \mathcal{S})$$

where $\boldsymbol{T}$ is a transformation, and $\boldsymbol{dist}$ is the *closest* distance between the point $\boldsymbol{T}p$ and the surface $\mathcal{S}$.

This minimization problem is solved for every depth frame captured, and is one of the most computationally intensive task in the reconstruction process. It is solved using a local search, and this is adequate because the incremental transformation between two consecutive frames is relatively small.

2.2.1 Gauss–Newton algorithm

Starting with the previous transformation as an initial guess, steps are taken toward the minimum by iteratively approximating the location of the optimal transformation. This is done by linearizing the distance function for each point — calculating the derivative with respect to each transformation parameter, and solving the resulting matrix equation with linear least squares. The global minimum will be found provided the initial estimation is within the convex basin. This algorithm is known as the Gauss–Newton algorithm.

Thus, at each step, the transformation parameters $t_1, \ldots, t_n$ are calculated by solving

$$\min_{t \in \mathbb{R}^n} \left\| \begin{pmatrix} \boldsymbol{dist}_{p_1} \\ \vdots \\ \boldsymbol{dist}_{p_m} \end{pmatrix} - \begin{pmatrix} \partial\, \boldsymbol{dist}_{p_1}/\partial t_1 & \cdots & \partial\, \boldsymbol{dist}_{p_1}/\partial t_n \\ \vdots & \ddots & \vdots \\ \partial\, \boldsymbol{dist}_{p_m}/\partial t_1 & \cdots & \partial\, \boldsymbol{dist}_{p_m}/\partial t_n \end{pmatrix} \begin{pmatrix} t_1 \\ \vdots \\ t_n \end{pmatrix} \right\|^2$$

and the current estimate is updated with $\boldsymbol{T} \leftarrow \boldsymbol{f}(t_1, \ldots, t_n)\boldsymbol{T}$ where $\boldsymbol{f}$ is a function that constructs a valid transformation out of the parameters $t_1, \ldots, t_n$.

2.2.2 Parametrization and Lie groups

In the previous section, we used a parametrization $t_1, \ldots, t_n$, and a function $\boldsymbol{f}$ to combine them into a valid transformation. We have seen how to represent transformations as 4×4 matrices manipulating homogeneous coordinates, but as a parametrization when optimizing, they are not so appropriate. The main problem is that they are over-parameterized, in that a *proper rigid transformation* only has 6 degrees of freedom (DOF), but a 4×4 matrix has 16. This means that most combinations of parameters do not even represent valid transformations.

Several different parametrizations exists with only 6 DOF, for example Euler angles and unit quaternions. In this thesis however, a parametrization will be used that is ideal for optimizations in which derivatives are considered, namely the Lie group associated

with the local 6-DOF space of proper rigid transformations. Lie groups in general will not be discussed, just how this specific Lie group arises.

Consider a parametrization

$$\boldsymbol{f}\left(t_1, t_2, t_3, t_4, t_5, t_6\right) = \boldsymbol{R_x}\left(t_1\right)\boldsymbol{R_y}\left(t_2\right)\boldsymbol{R_z}\left(t_3\right)\boldsymbol{T_x}\left(t_4\right)\boldsymbol{T_y}\left(t_5\right)\boldsymbol{T_z}\left(t_6\right)$$

where $\boldsymbol{R_{x,y,z}}$ and $\boldsymbol{T_{x,y,z}}$ are the rotation and translation matrices respectively. The choice of order in which they are multiplied together is arbitrary, but leads to different results. This suggest that there is some better, unambiguous, way to combine $t_1, \ldots, t_6$ into a transformation. Since each of the rotation and translation matrices fulfill the equation $\boldsymbol{H}(t) = \boldsymbol{H}\left(\frac{t}{k}\right)^k$, one idea is to divide each value t_i with a large integer k, multiply the matrices together, and then raise the product to k, thus mixing the multiplication order:

$$\boldsymbol{f}\left(t_1, t_2, t_3, t_4, t_5, t_6\right) = \left(\boldsymbol{R_x}\left(\frac{t_1}{k}\right)\boldsymbol{R_y}\left(\frac{t_2}{k}\right)\boldsymbol{R_z}\left(\frac{t_3}{k}\right)\boldsymbol{T_x}\left(\frac{t_4}{k}\right)\boldsymbol{T_y}\left(\frac{t_5}{k}\right)\boldsymbol{T_z}\left(\frac{t_6}{k}\right)\right)^k.$$

In the limit, this leads to

$$\begin{aligned}
&\boldsymbol{f}\left(t_1, t_2, t_3, t_4, t_5, t_6\right) = \\
&= \left(\boldsymbol{R_x}\left(\frac{t_1}{k}\right)\boldsymbol{R_y}\left(\frac{t_2}{k}\right)\boldsymbol{R_z}\left(\frac{t_3}{k}\right)\boldsymbol{T_x}\left(\frac{t_4}{k}\right)\boldsymbol{T_y}\left(\frac{t_5}{k}\right)\boldsymbol{T_z}\left(\frac{t_6}{k}\right)\right)^k = \\
&= \mathbf{exp}\left(k\,\mathbf{log}\left(\boldsymbol{R_x}\left(\frac{t_1}{k}\right)\boldsymbol{R_y}\left(\frac{t_2}{k}\right)\boldsymbol{R_z}\left(\frac{t_3}{k}\right)\boldsymbol{T_x}\left(\frac{t_4}{k}\right)\boldsymbol{T_y}\left(\frac{t_5}{k}\right)\boldsymbol{T_z}\left(\frac{t_6}{k}\right)\right)\right) \to \\
&\to \mathbf{exp}\left(t_1\boldsymbol{R_x}'(0) + t_2\boldsymbol{R_y}'(0) + t_3\boldsymbol{R_z}'(0) + t_4\boldsymbol{T_x}'(0) + t_5\boldsymbol{T_y}'(0) + t_6\boldsymbol{T_z}'(0)\right) = \\
&= \mathbf{exp}\begin{pmatrix} 0 & -t_3 & t_2 & t_4 \\ t_3 & 0 & -t_1 & t_5 \\ -t_2 & t_1 & 0 & t_6 \\ 0 & 0 & 0 & 0 \end{pmatrix} \text{ when } k \to \infty,
\end{aligned}$$

where **exp** and **log** is the matrix exponential and matrix logarithm respectively. The matrix exponential is defined as

$$\mathbf{exp}\left(\boldsymbol{A}\right) = \sum_{k=0}^{\infty} \frac{1}{k!}\boldsymbol{A}^k$$

and converges quickly for rigid transformations.

The matrices $\left\{\boldsymbol{f}\left(t_1, t_2, t_3, t_4, t_5, t_6\right) \middle| \, t \in \mathbb{R}^6\right\}$ constructed in this way are in fact always proper rigid transformations, and when combined with the operation matrix multiplication, they form a Lie group.

2.2.3 Iteratively reweighted least squares

Using notations from the previous sections, the optimal transformation $\boldsymbol{T}^*$ that best aligns the two models $\mathcal{P}$ and $\mathcal{S}$ is defined as

$$\boldsymbol{T}^* = \arg\min_{\boldsymbol{T}} \mathrm{F}(\boldsymbol{T}) = \arg\min_{\boldsymbol{T}} \sum_{p\in\mathcal{P}} \boldsymbol{dist}^2(\boldsymbol{T}p, \mathcal{S}).$$

Assuming the distances are normally distributed around zero, $\boldsymbol{T}^*$ represent the solution of the maximum-likelihood problem. This assumption is not correct however, and the sum in this form is dominated by outliers. To handle this problem and make the measure more robust, a function ρ is introduced that weighs the residuals:

$$\boldsymbol{T}^* = \arg\min_{\boldsymbol{T}} \sum_{p\in\mathcal{P}} \rho\left(\boldsymbol{dist}\left(\boldsymbol{T}p, \mathcal{M}\right)\right).$$

To be able to solve this optimization problem using linear least squares as described earlier, the sum must be transformed into a sum of weighted squares. This is done by approximating ρ locally around a current residual r_0:

$$\rho(r_0 + r) \approx \omega(r_0)\left(r_0 + r\right)^2.$$

Several different choices for ρ was tested in this thesis, including for example the L_1 norm, but the final norm chosen was the Tukey norm [8], where

$$\omega_{r_{max}}(r) = \max\left(0,\ 1 - \left(\frac{r}{r_{max}}\right)^2\right)^2.$$

With this norm, outliers with residual $> r_{max}$ have no influence at all. The appearance of this function can be seen in figure 2.2, this is to be compared with for example $\omega = 1$ for the L_2 norm.

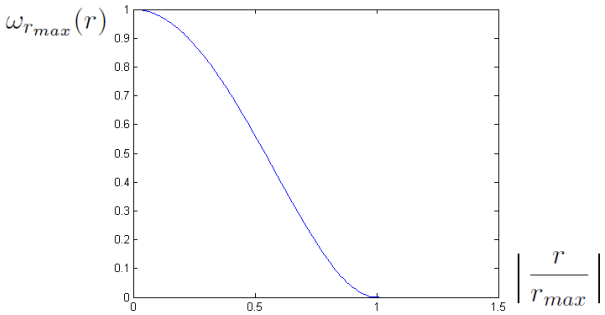


Figure 2.2: Tukey weight function.

2.3 Kinect

An overview of the Kinect hardware can be seen in figure 2.3. The Kinect sensor consists of three parts; a camera that captures color images, an IR laser emitter, and another camera that only captures IR light. The IR laser emitter and IR camera are used to create a depth map using a structured light technique. The IR laser emits a known pseudorandom dot pattern of IR light at 830 nm. Figure 2.4 shows an example of this pattern. The dot pattern is observed with the IR camera, and the detected dots are compared against the known pattern. This is like a stereo camera system, but with one of the cameras replaced with a static virtual image of the dot pattern. Since the IR camera has a certain horizontal separation from the IR emitter, the IR dots will end up at different image locations depending on the depth. The difference in image location of the IR dot, seen by the IR camera and the IR emitter respectively, is known as *disparity*. This is illustrated in figure 2.5. The disparity values from each IR dot are used to create a *disparity map* of size 640×480, where each disparity value is stored as a 11-bit integer. This is the actual raw data that is sent from the Kinect to the computer, at a frame rate

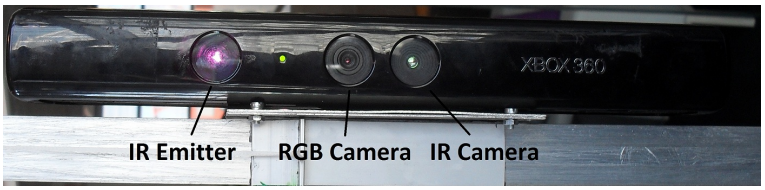


Figure 2.3: The Kinect Camera.

Figure 2.4: IR dot pattern.

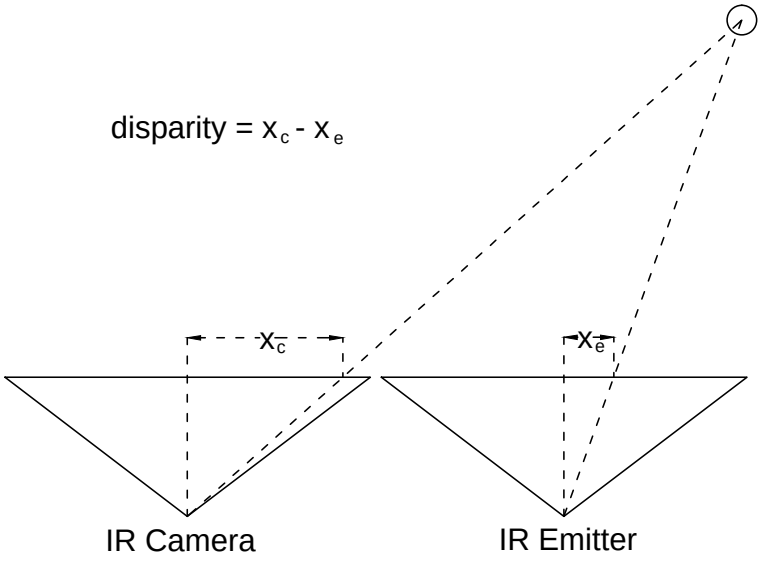


Figure 2.5: Disparity.

of 30 Hz. The disparity values have a one-to-one correspondence to a depth value.

The Kinect unit is optimized for detecting objects about 2.5 meters way, and cannot detect objects closer than ~ 0.5 meters, mainly because the IR dots become too intense to distinguish at this range, as can be seen in figure 2.6.

Figure 2.6: IR dots close up.

The RGB-camera in the Kinect unit has a frame rate of 10 Hz when used in the highest resolution mode, which is a 1280×1024 Bayer image.

Kinect calibration

Each Kinect unit is calibrated at the factory, and has built in numbers for converting the disparity values to depth values, and also for mapping 3D points to the color camera. However, the factory calibration uses a simple model where depth values are fast to compute, and it is possible to improve the accuracy using a more complex model at the expense of some computation time. The calibration model used in this thesis is from [9], and the calibration parameters were estimated using a toolbox accompanying that paper. An image from the calibration procedure can be seen in figure 2.7.

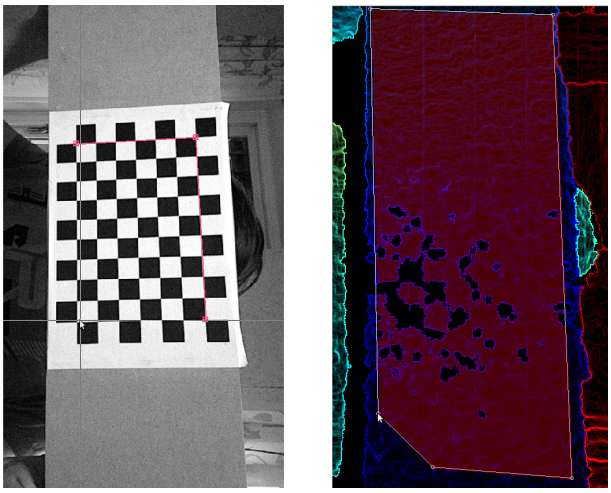


Figure 2.7: Calibrating Kinect camera using toolbox accompanying [9].

The algorithm from [9] simultaneously calibrates the color camera and the depth camera in the Kinect, as well as the relative pose between them. The color camera intrinsics is based on the following model: a 3D point p is transformed into image coordinates (u, v) through the equations

$$\begin{aligned}
\boldsymbol{x_n} &= \begin{pmatrix} x_n \\ y_n \end{pmatrix} = \begin{pmatrix} p_x/p_z \\ p_y/p_z \end{pmatrix}, \\
\boldsymbol{x_g} &= \begin{pmatrix} 2k_3 x_n y_n + k_4 \left(r^2 + 2x_n^2\right) \\ 2k_4 x_n y_n + k_3 \left(r^2 + 2y_n^2\right) \end{pmatrix}, \quad r^2 = x_n^2 + y_n^2, \\
\begin{pmatrix} x_k \\ y_k \end{pmatrix} &= \left(1 + k_1 r^2 + k_2 r^4 + k_5 r^6\right) \boldsymbol{x_n} + \boldsymbol{x_g}, \\
\begin{pmatrix} u \\ v \end{pmatrix} &= \begin{pmatrix} f_x x_k + u_0 \\ f_y y_k + v_0 \end{pmatrix}
\end{aligned}$$

where $\boldsymbol{k} = \left(k_1, \ldots, k_5\right)$ are distortion coefficients, $\boldsymbol{f}$ the focal lengths, and (u_0, v_0) the image center.

A disparity value d from the disparity map at coordinates (u, v) is transformed into

a 3D point p through the equations

$$
\begin{aligned}
d_k &= d + D_{u,v} \exp\left(\alpha_0 - \alpha_1 d\right), \\
z &= \frac{1}{c_1 d_k + c_0}, \\
\begin{pmatrix} p_x \\ p_y \\ p_z \end{pmatrix} &= \begin{pmatrix} \left(u - u_0\right) / \left(f_x z\right) \\ \left(v - v_0\right) / \left(f_y z\right) \\ z \end{pmatrix},
\end{aligned}
$$

where $\boldsymbol{f}$ is the focal lengths for the depth camera, (u_0, v_0) the image center of the depth camera, and $\alpha_0, \alpha_1, c_0, c_1, D_{u,v}$ are distortion coefficients.

Note that $D_{u,v}$ is a distortion value for each pixel in the depth map, figure 2.8 shows an example of this spatial distortion pattern.

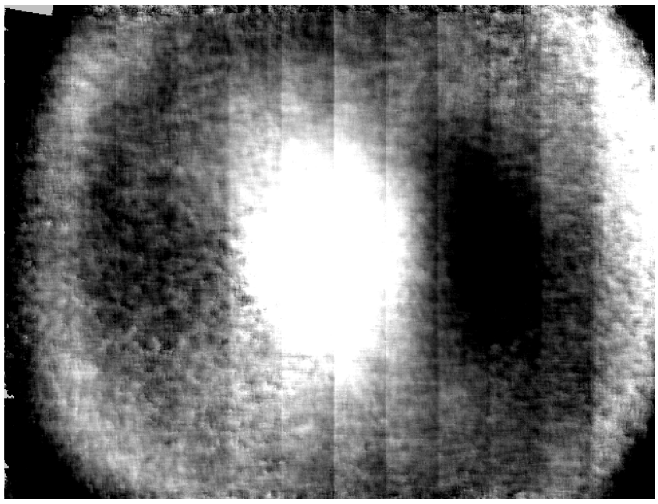


Figure 2.8: Distortion values $D_{u,v}$.

3

Method

This chapter describes the approach used to achieve the results in this thesis and discusses some implementation choices.

The main design choice stood between

- a single-camera system, where the camera either rotates around the person or the person stands on a rotating turntable,
- a multi-camera system, where the cameras are placed around the person.

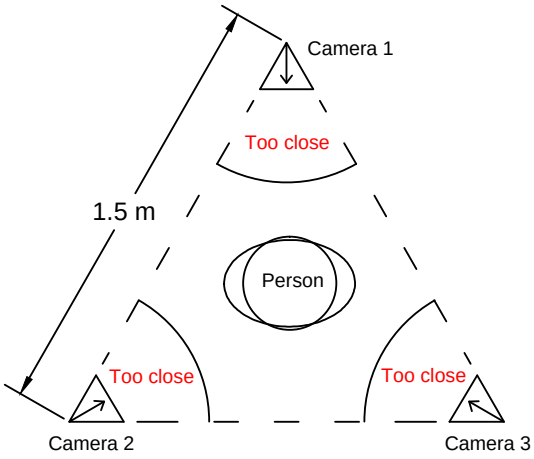
The approach with multiple cameras was considered more appealing for several reasons. First, the scanning procedure becomes faster with a multi-camera system, and it is also less intrusive than a system where the user has to stand on a rotating turntable. Secondly, it was considered mechanically simpler, and in return, more algorithmically complex and novel, to use multiple cameras.

3.1 Hardware

This section will briefly describe the hardware built for this thesis.

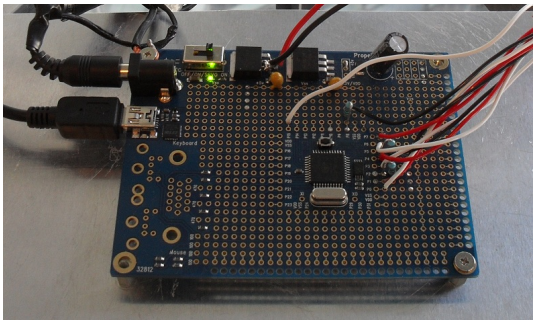
3.1.1 Scanning booth

The scanning booth has a triangular base and the frame is built with aluminium square tubes. The height of the booth is 2 meters. An overview of the booth can be seen in figure 3.1. The booth is equipped with three Kinect cameras, placed at the corners of the triangle. Each camera is connected to a trolley that can slide up and down the vertical tube. Each trolley is connected to a servo at the top of the booth that can pull it up and down, as shown in figure 3.1(d). The servos are controlled from the PC via a microcontroller (Parallax Propeller), seen in figure 3.1(c).

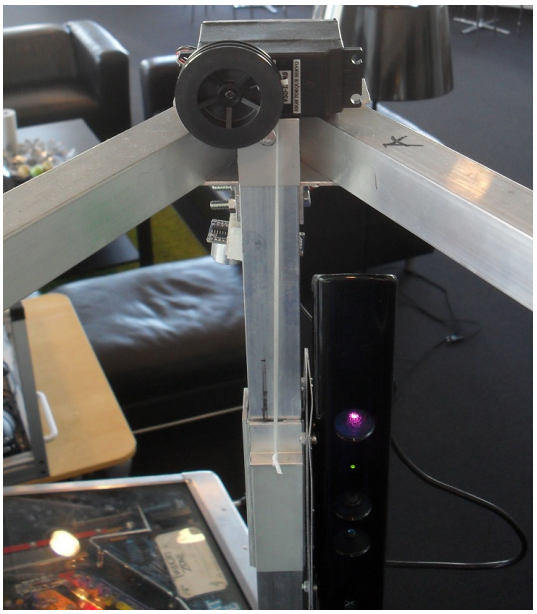


(a) Top view sketch.

(b) Scanning booth at Cybercom office.



(c) Parallax Propeller.



(d) Servo and trolley.

Figure 3.1: Scanning booth.

3.1.2 PC

The software developed in this thesis is running on a PC equipped with an AMD Phenom CPU and an Nvidia GeForce GTX 570.

3.2 Software overview

The software was originally intended to be an extension and modification of the KinectFusion algorithm [1] implemented in the Point Cloud Library (PCL) under the name KinFu. The intention was to modify the source code so that data from all cameras could be integrated concurrently into a single coherent model directly. However, it later turned out necessary to run each camera separately during the scanning procedure, and then merge all data as a post-processing step. One reason for this is that it is difficult to align the cameras in the beginning of the procedure, because each camera sees an almost independent part of the head and there is little overlap. Alignment is easier to do at the end of the procedure, when each camera has acquired data from many different point of views. Another reason for separating tracking and merging, as it turned out, is that some people turn their necks during scanning, and this breaks the assumption of model as a rigid body. To avoid this problem, tracking was restricted to only include the head, but the post-processing step reconstructs the upper body too. If the person turns his neck excessively during scanning, only the head is reconstructed properly.

3.2.1 Performance issues and simplified model

The KinFu algorithm could barely handle the full frame of a single camera, and using three cameras would cause a delay at the end of the scanning procedure, twice as long as the procedure itself. During scanning, each camera essentially sees only one part of the person from slightly different point of views, so the full generality of the KinectFusion algorithm is unnecessary. For these reasons, a simpler algorithm was implemented for the tracking phase, that was fast enough to handle all three cameras concurrently. The simplified algorithm uses a model consisting of a single high-resolution depth image, called the reference frame. Each frame is tracked against the reference frame, and then fused into it, improving tracking for subsequent frames. An example of how the initial and final reference frames may look can be seen in figure 3.2.

3.2.2 Aligning cameras and post-processing

A reference frame is created for each camera, and at the end of the scanning procedure, the relative pose between them are calculated. After aligning the reference frames, depth frames from different cameras can be integrated into a globally aligned framework. This is done using the original integration method from KinectFusion, which will be described later. This integration algorithm was extracted from the KinFu source code and rewritten slightly. This is the only part of the KinectFusion algorithm that actually ended up in the final implementation.

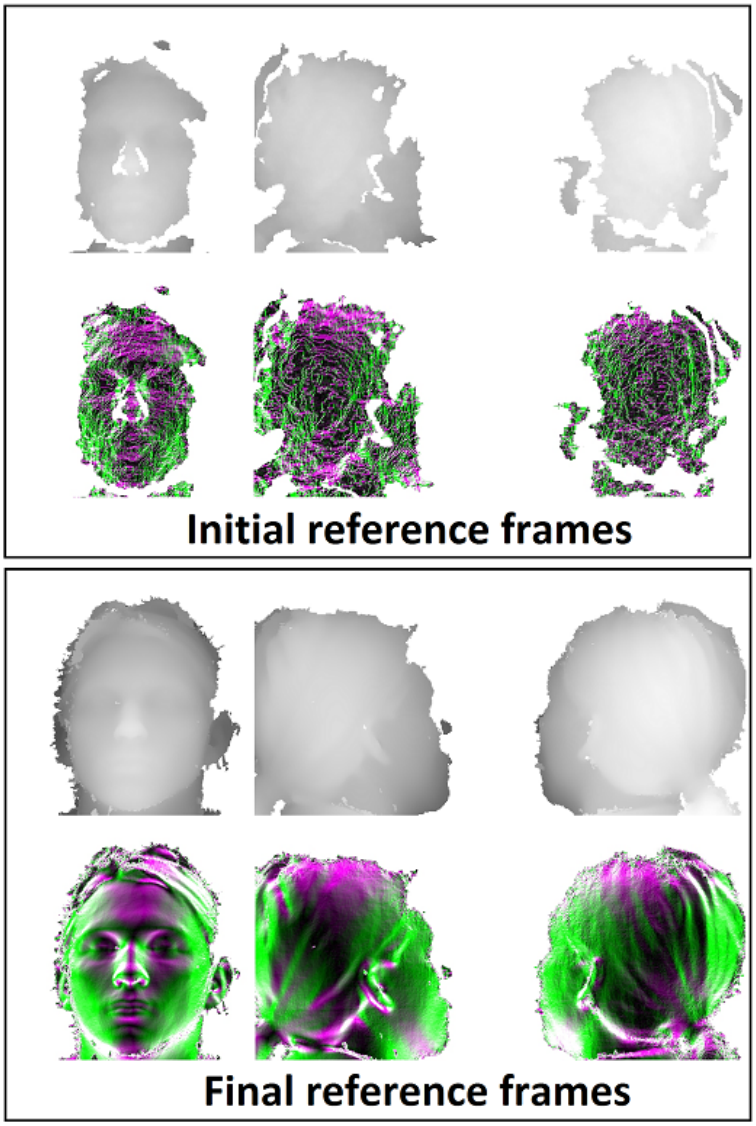


Figure 3.2: Reference frame for each camera. The top grayscale image is the actual depth frame. The bottom colored image is the spatial derivatives of the corresponding depth frame, which reveals more details.

3.2.3 Software

Besides KinFu, another module from PCL was used, called gpu::people. This module can segment out body parts from a point cloud, figure 3.3 shows an example. This module was used for detecting when a person enters the booth, and also for locating the head. The head location was calculated from the point cloud by fitting a sphere with constant radius ≈ 10 cm to the points labeled as head.

The module is designed to work with a camera placed horizontally and not vertically, and where the person is not so close to the camera, so a pre-processing step was implemented where the point-cloud is reprojected to suit the module. Also, all points outside the booth were filtered out to help the module further.

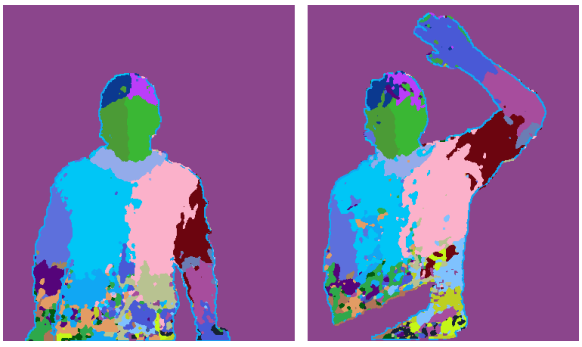


Figure 3.3: Body part segmentation by gpu::people.

Kinect driver

Initially, the OpenNI driver was used to fetch data from the Kinect cameras, as it is the standard driver for PCL. However, the OpenNI driver had problems working with multiple cameras. Also, by default, it sends the depth map generated using the factory calibration model, and it was problematic to gain fast access to the raw disparity data. The raw data is needed for the more sophisticated calibration model described in section 2.3. For these reasons, the driver was changed to libfreenect, which allows direct access to the unprocessed data stream for both the depth data and the RGB data. A fast implementation of the sophisticated calibration model was implemented, relying heavily on SSE vector instructions for the CPU, that convert the packed stream of raw 11-bit disparity values directly into a calibrated mesh that can be rendered on the GPU.

3.2.4 Step-by-step outline of scanning procedure

This is an outline of how the system works:

- A person enters the booth. This is detected with gpu::people.
- The head position is calculated, and a reference frame is created for each camera.

- The cameras start moving up and down, and the relative pose between each frame and its corresponding reference frame is calculated. The frames are then fused into the reference frames to improve further tracking.
- All raw depth frames from the scanning procedure are globally aligned against each other, and sent to a post-processing step where a high-quality model is built.
- A triangular mesh is then generated from this model, and colored using the RGB-frames captured during the scanning procedure.
- The final model is presented for the user as a shaded rotating animation, both with and without colors. It is also possible to render the model in 3D on a stereo display.

3.3 Camera tracking

Camera tracking is the most critical step in the processing pipeline. If the algorithm loses track of camera pose for any frame, it is likely that it will also fail for all subsequent frames. Tracking is also coupled with positive feedback, in that an accurate pose estimation results in a more detailed reference frame, which in turn results in more precise pose estimations for subsequent frames. The quality of the final model is also highly dependent on the accuracy of the estimated camera pose.

The tracking algorithm is a specific implementation of the iterated closest point (ICP) algorithm, using iteratively reweighted least squares with the Tukey norm where, in each step, the $\boldsymbol{dist}$ function is parametrized over the associated Lie group and solved with the Gauss–Newton algorithm. This is explained in more detail in section 2.2. The only unknown left is how the $\boldsymbol{dist}$ function is defined. The model used for tracking against, called $\mathcal{M}$ in section 2.2, is the reference frame. The function $\boldsymbol{dist}$ is a metric, that takes a 3D point as input, and returns the distance to the other model (the reference frame). To be true to ICP, this distance should be the distance to the *closest* point, but this is relaxed to be the *projective point-to-plane distance* described soon. The 3D points used as argument to the $\boldsymbol{dist}$ function, called $\mathcal{P}$ in section 2.2, are generated from the raw depth frames.

3.3.1 Reference frame representation

The reference frame, $\boldsymbol{ref}$, is a high-resolution, orthographically projected depth map. An example can be seen in figure 3.2. Projection is done orthographic instead of projective, because fewer point become occluded by each other that way, and it also simplifies further computation. The dimension of the reference frame is 512×512, where each element is a floating point value representing the depth. The reference frame represents a physical square of 26 cm $\times$ 26 cm, meaning that each pixel represents a patch of about 0.5 mm $\times$ 0.5 mm. Thus, a pixel $\boldsymbol{ref}_{u,v}$ represents the 3D point $\left(ku, kv, \boldsymbol{ref}_{u,v}\right)$, where $k = \frac{26}{512}$ cm.

These points are regarded as connected to each other spatially, so that they form a continuous surface, even though the reference frame may contain some depth irregularities.

3.3.2 Projective point-to-plane distance

Since the reference frame is regarded as a continuous surface, each pixel $\boldsymbol{ref}_{u,v}$ in the reference frame can be seen as a local plane, with surface normal

$$\boldsymbol{n}_{u,v} = \begin{pmatrix} \Delta \boldsymbol{ref}_{u,v}/\Delta ku \\ \Delta \boldsymbol{ref}_{u,v}/\Delta kv \\ -1 \end{pmatrix} = \begin{pmatrix} \frac{\boldsymbol{ref}_{u+1,v} - \boldsymbol{ref}_{u-1,v}}{2k} \\ \frac{\boldsymbol{ref}_{u,v+1} - \boldsymbol{ref}_{u,v-1}}{2k} \\ -1 \end{pmatrix}.$$

The distance for a point $\boldsymbol{p}$ to the reference frame is then calculated by *projecting* $\boldsymbol{p}$ onto the reference frame, resulting in some $(u, v, z) = \boldsymbol{proj}(\boldsymbol{p}, \boldsymbol{ref})$, and then calculating the distance between this projected point and the local plane at $\boldsymbol{ref}_{u,v}$

$$\boldsymbol{dist}(\boldsymbol{p}, \boldsymbol{ref}) = \left(\begin{pmatrix} ku \\ kv \\ \boldsymbol{ref}_{u,v} \end{pmatrix} - \begin{pmatrix} ku \\ kv \\ z \end{pmatrix} \right) \bullet \frac{\boldsymbol{n}_{u,v}}{\|\boldsymbol{n}_{u,v}\|} = \frac{z - \boldsymbol{ref}_{u,v}}{\|\boldsymbol{n}_{u,v}\|},$$

where $(u, v, z) = \boldsymbol{proj}(\boldsymbol{p}, \boldsymbol{ref})$.

This distance is not necessarily the closest, but this metric works well in practice and is fast to compute.

3.3.3 Implementation details

To make the algorithm fast enough, all parts were implemented on a GPU. The geometrical transformation of a depth frame, together with the projection $\boldsymbol{proj}$ onto the reference frame, is perfectly suited for a GPU. This is a rendering problem and can be done using standard functions in the Open Graphics Library (OpenGL), utilizing all specialized hardware in the GPU. First, the raw depth frame is converted to a triangular mesh, connecting each point to its neighbors, and then rendered using the pose estimation together with a projection that corresponds to the reference frame. This results in a new depth map $\boldsymbol{z}$, with the same format as the reference frame, see figure 3.4.

The distance function can then be simplified to work on $\boldsymbol{z}$ directly with

$$\boldsymbol{dist}\left(\boldsymbol{z}_{u,v}, \boldsymbol{ref}_{u,v}\right) = \frac{\boldsymbol{z}_{u,v} - \boldsymbol{ref}_{u,v}}{\|\boldsymbol{n}_{u,v}\|}.$$

Then, the derivatives of this function with respect to the transformation parameters are calculated, and used for estimating the location of the optimal pose as described in section 2.2. This is also implemented on the GPU, using the Compute Unified Device

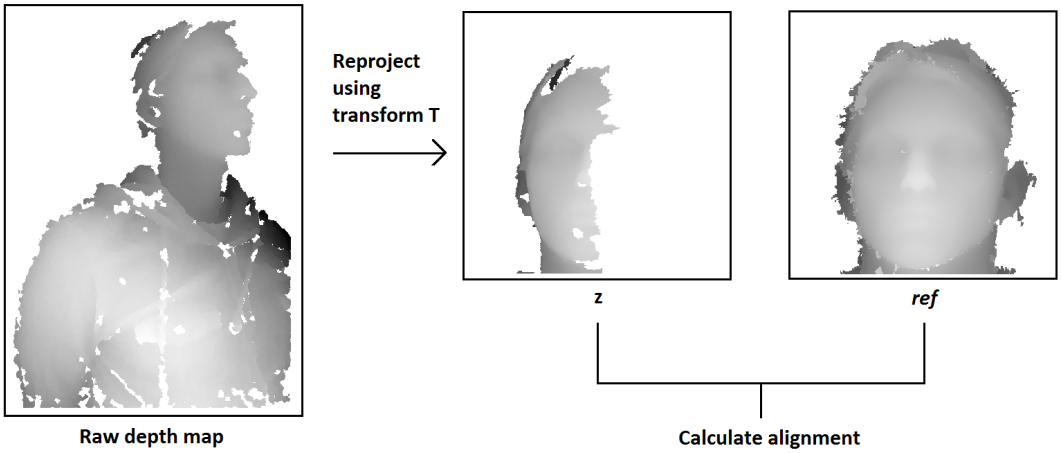


Figure 3.4: One iteration in the alignment process.

Architecture (CUDA) platform, working directly on the depth map $\boldsymbol{z}$ generated in the previous step. The derivatives and weights from the Tukey norms are calculated in parallel for each pixel $\boldsymbol{z}_{u,v}$, and the matrix products used to solve the linear least squares problem are constructed directly, without constructing any intermediate matrix. The resulting matrix products are just a 6×6 matrix and a vector with 6 elements, and they are sent back to the CPU. On the CPU, the normal equation is solved using standard Gaussian elimination, and the result is used to construct a new estimation of the optimal camera pose, using the associated Lie group described in section 2.2.2.

This process is iterated several times. The parameter used in the Tukey norm, r_{max}, is updated every iteration. The choice of r_{max} affects the cost landscape, and if chosen too small while the depth frame is not properly aligned, may cause the algorithm to get stuck in a local minimum. Therefore, r_{max} is set to a high value in the first iteration (70 mm), and slowly lowered to a small value (2 mm), over the course of 13 iterations. Also, the first 8 iterations are performed on a scaled-down version of $\boldsymbol{ref}$ with size 128×128. This reduces computation time somewhat, but the main reason is to smooth the cost landscape and improve the chance of finding the global minimum.

The computation time for one iteration using a reference frame of size 512×512 takes about 0.3 ms, including rendering and calculating the linear least squares problem. For all three cameras, the algorithm needs to do $3 \cdot 13 \cdot 30 = 1170$ iterations per second, which it can handle without problems.

3.3.4 Data integration with reference frame

After the optimal camera pose has been calculated in the tracking step, the frame is integrated into the reference frame. This is done using a running average for each depth value.

3.4 High-quality model generation

After all frames have been tracked, they are sent to a post-processing step where a single coherent high-quality model is generated. This is done using the integration algorithm from KinectFusion [1].

Generating a high-quality model is more involved than simply concatenating all points into a coherent framework. Each Kinect generates over 9 million points per second, so there is an abundance of points, and the main problem is how to fuse them together and remove outliers. Working with explicit points directly is not very convenient, and the underlying depth maps actually contain more information. A pixel in a depth map is not only a point, it also indicates that the ray between the point and the camera consists of empty space. This information is easier exploited in an implicit data format.

3.4.1 Implicit format

In an implicit method, the surface of the model is represented by a signed distance function (SDF) $f : \mathbb{R}^3 \mapsto \mathbb{R}$ that maps each point in $\mathbb{R}^3$ to a value that represents the signed distance to the nearest surface. These values are positive outside of the model, and negative inside. Distance here does not necessarily mean Euclidean distance. In fact, in this particular implementation, distance values are bounded to $[-1, 1]$. However, close to the surface, distance values vary smoothly and form a pseudo-Euclidean metric. The actual surface is implicitly defined as the isosurface (zero-crossing) of the SDF.

Using an implicit function has many advantages over explicit representations such as point clouds. It becomes easier to fuse data together, and deal with uncertainty and multiple measurements.

In this particular implementation, the signed distance value $f(\boldsymbol{p})$ for a point $\boldsymbol{p}$ is calculated in the following way: for each depth map D_i, project $\boldsymbol{p}$ onto D_i and compare the distance value d_s stored there with the actual distance d_p to $\boldsymbol{p}$. If these values are close enough according to some threshold t, $|d_s - d_p| < t$, this gives a signed distance value of $(d_s - d_p)/t$.

If instead d_s is significantly greater than d_p, this indicates that $\boldsymbol{p}$ lies on the ray of empty space between the camera and the observed point, and this gives a capped distance value of 1.

If instead d_s is significantly less than d_p, no distance value can be calculated, because $\boldsymbol{p}$ is occluded and may be close to another surface behind the observed point.

The final signed distance value for $\boldsymbol{p}$ is the average of the signed distance value from each depth map, calculated in this way. The algorithm is explained in more detail in algorithm 1 and figure 3.5.

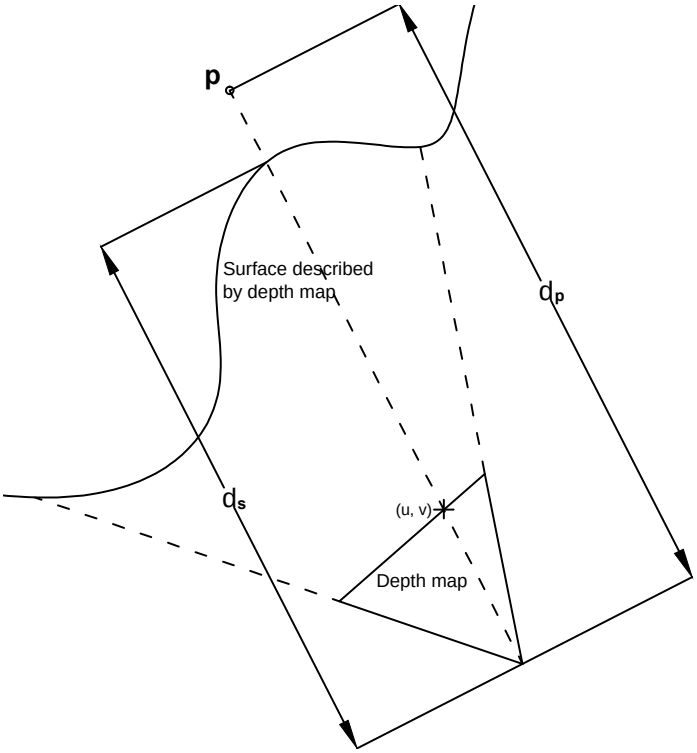


Figure 3.5: Illustration of variables used in the calculation of signed distance values.

Data: Aligned depth maps D_i, distance threshold t
Result: Signed distance value $f(\boldsymbol{p})$ for point $\boldsymbol{p}$
$s := 0$
$w := 0$
foreach *depth map* D_i **do**
  if $\boldsymbol{p}$ *in view of depth map* D_i **then**
    $(u, v) :=$ project $\boldsymbol{p}$ onto D_i
    $d_s :=$ distance stored in D_i at (u, v)
    $d_p :=$ actual distance from $\boldsymbol{p}$ to camera
    diff $:= (d_s - d_p)$ / t
    if diff ≥ -1 **then**
      sdf $:=$ min(1, diff) **comment:** cap the value so that sdf $\in [-1, 1]$
      s += sdf **comment:** running total sum
      w += 1 **comment:** running count
    end
  end
end
return s / w **comment:** average distance

Algorithm 1: Calculation of signed distance values.

Choice of threshold t

The default threshold value is 30 mm in the KinFu implementation. This is a fairly high value, where few distance values get capped, resulting in the zero-crossing becoming the 'mean' in some sense. Under the particular circumstances scanning is performed in this thesis, it was beneficial to lower the threshold value to 1 mm. With this value, most distance values get capped, so the influence from each depth map is similar to the sign function. This shifts the zero-crossing towards the 'median' rather than the 'mean', making it more robust to outliers.

3.4.2 Volumetric representation and GPU implementation

The previous section described how the implicit function f is calculated for an arbitrary point $\boldsymbol{p}$. To make it numerically feasible, f is only evaluated at a finite number of points, namely at the vertices of a a regular three dimensional grid. In this particular implementation, the grid consists of 512^3 vertices, and the side of each cell in the grid represents a physical size of about 0.75 mm.

The algorithm is implemented to be fast on a GPU, where the bottleneck often is memory access. Random access in global GPU memory is prohibitively slow. However, it is possible to store relatively small matrices as textures in the GPU, where random access bandwidth is improved by a texture cache, optimized for locality in two dimensions. To utilize this, each depth map is sent and stored on the GPU as a texture, one at a time. Then for each depth map, all values in the grid are updated in parallel, where memory can be accessed sequentially, and projection onto the depth map is implemented as texture lookups.

3.4.3 Polygonal mesh generation and coloring

The mesh generation is the final step in the model generation pipeline. It takes the grid of signed distance values, described in the previous section, and extracts the isosurface from this scalar field, as a triangular mesh. This is done using the Marching cubes [10] algorithm. This algorithm looks at the eight surrounding vertices for each cell in the grid, and checks if a zero-crossing takes place inside the cell (some vertices have different signs). If a zero-crossing takes place, it calculates where the isosurface intersects the cell edges using linear interpolation, and then creates a polygon from these intersections, representing the surface passing through the cell.

This algorithm was not implemented as part of the thesis, since there is a suitable implementation of this algorithm for the GPU in the Point Cloud Library.

Data cleansing

The mesh generation may generate small fragments of polygons floating around the actual head, due to noise in the input data. To remove these, a disjoint-set data structure was implemented to calculate the size of each connected mesh component. Then only the biggest component is kept, which hopefully is the head.

Coloring

A color is computed for each vertex in the final mesh in the following way. The RGB-frames acquired during scanning get associated with the depth frame acquired closest in time (from the same Kinect unit). Then these depth maps are converted to point clouds, and colored, by projecting each point to the associated RGB-frame and looking up the color. Then all these colored points are inserted into a global grid. The purpose of the grid is to be able to find the closest colored points for an arbitrary query point fast.

This grid is not constructed explicitly, instead, the cell index for each colored point is calculated, and then sorted upon. Looking up the colored points in a particular cell is then implemented as a binary search of the cell index.

This is faster than constructing an explicit grid, because most of the cells are empty, and there are more cells in the grid than actual points.

After the grid of colored points has been created, each vertex in the mesh finds its 10 closest colored points in the grid. This is done by searching an increasing neighborhood around the vertex until 10 points have been found.

The 10 colored points are then sorted according to the green component. Green is the most reliable component because the RGB-camera uses a Bayer filter. The final color calculated for the vertex is then the average of the 4 colors in the middle. The reason to not take the average of all 10 colored points is that they often contains outliers.

4

Result

The scanning booth has been standing in the Cybercom office for two months at the time of writing. Each scanning generates a model that is presented for the user as rotating animation, both with and without colors, as in figure 4.1. It is also possible to present the model in 3D, as in figure 4.2, but this requires the user to wear special 3D glasses.

Figure 4.1: Rotating animation of model.

Figure 4.2: Stereo display.

The delay between the end of the scanning procedure and presentation of the model is typically only a few seconds, since most of the work is done during scanning. The animation is stored as a movie and uploaded to a web-based file storage (Google Drive). This folder has accumulated a number of movies during the past two months.

4.1 Evaluation

Some people have received instructions on how to use the booth, and some people have just stepped inside the booth on their own. The instructions given was to turn slightly back and forth during scanning, while keeping the neck stiff. If the person stands still, some areas will not be seen by any camera during scanning. It also makes it more difficult to align data between cameras. A typical, but still successful, result of a person standing still during scanning can be seen in figure 4.3.

Another source of error is non-rigid motion, the most common by far is turning the neck. A typical result of a person turning his neck can be seen in figure 4.4.

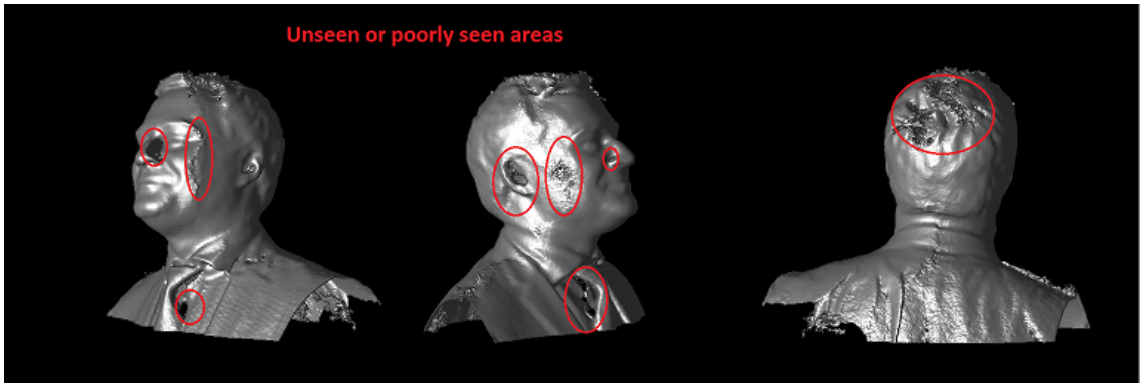


Figure 4.3: Result of person standing still.

Figure 4.4: Only head is reconstructed properly after non-rigid motion.

A more major cause of error is tracking failure. This can happen for a number of

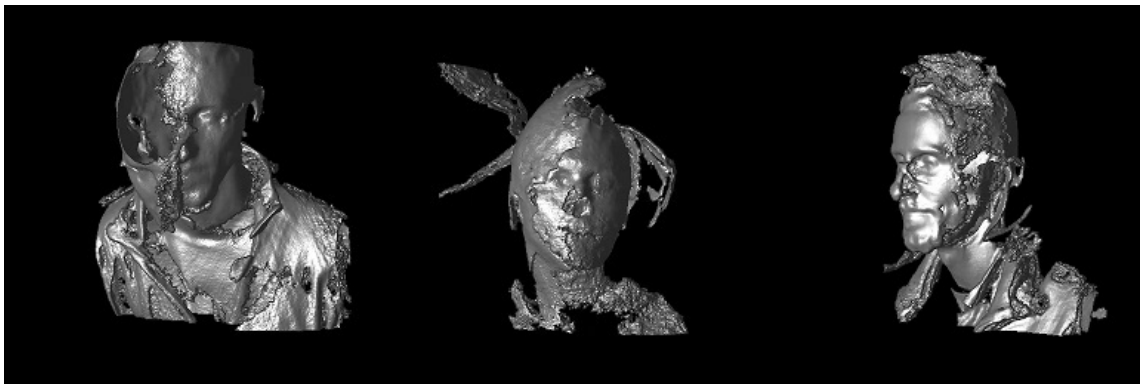


Figure 4.5: Example results after tracking failure.

reasons. The most common is that the part of the head that is tracked goes out of sight. This can happen for example if the person comes too close to the camera, since the Kinect can not see objects closer than ~ 0.5 meters, as illustrated in figure 3.1(a). The current implementation does not do any sanity checks of the tracking and integrates all frames regardless of tracking quality. Integrating a misaligned frame usually corrupts the reference frame and the final model, as can be seen in figure 4.5.

However, in most cases when the person has received instructions, the scanning works well. Some examples can be seen in figure 4.6.

Figure 4.6: Successful scannings.

5

Conclusion

This project shows that it is possible to use Kinect cameras to reconstruct 3D models with a quality comparable to models generated from commercial solutions. The use of modern algorithms and data structures for reconstruction overcomes the low-quality output from the Kinect cameras. Also, the reconstruction can be done in real-time using a commodity GPU.

Given well-formed data, the system implemented in this thesis generates very encouraging results. However, the system is not as reliable as initially intended, and users need to receive instructions to get good results. In hindsight, more time should have been spent on observing how people behave during scanning, and how data looks under these circumstances. The biggest source of error is non-rigid motion from the user, and tracking failure when a user gets to close to a camera. Instead, most of the effort was spent calibrating and tweaking the algorithm to produce good results under near-ideal circumstances.

Bibliography

[1] S. Izadi, D. Kim, O. Hilliges, D. Molyneaux, R. Newcombe, P. Kohli, J. Shotton, S. Hodges, D. Freeman, A. Davison, A. Fitzgibbon, Kinectfusion: real-time 3d reconstruction and interaction using a moving depth camera, in: Proceedings of the 24th annual ACM symposium on User interface software and technology, UIST '11, ACM, New York, NY, USA, 2011, pp. 559–568.
URL http://doi.acm.org/10.1145/2047196.2047270

[2] T. Whelan, M. Kaess, M. Fallon, H. Johannsson, J. Leonard, J. McDonald, Kintinuous: Spatially extended kinectfusion.

[3] H. Roth, M. Vona, Moving volume kinectfusion., in: BMVC, 2012, pp. 1–11.

[4] M. Zeng, F. Zhao, J. Zheng, X. Liu, Octree-based fusion for realtime 3d reconstruction, Graphical Models.

[5] Reconstructme (August 2013).
URL http://reconstructme.net

[6] A. Weiss, D. Hirshberg, M. J. Black, Home 3d body scans from noisy image and range data, in: Computer Vision (ICCV), 2011 IEEE International Conference on, IEEE, 2011, pp. 1951–1958.

[7] J. Tong, J. Zhou, L. Liu, Z. Pan, H. Yan, Scanning 3d full human bodies using kinects, IEEE Transactions on Visualization and Computer Graphics 18 (4) (2012) 643–650.

[8] Z. Zhang, Z. Zhang, P. Robotique, P. Robotvis, Parameter estimation techniques: A tutorial with application to conic fitting, Image and Vision Computing 15 (1997) 59–76.

[9] D. Herrera C, J. Kannala, J. Heikkila, Joint depth and color camera calibration with distortion correction, IEEE Trans. Pattern Anal. Mach. Intell. 34 (10) (2012) 2058–2064.
URL http://dx.doi.org/10.1109/TPAMI.2012.125

[10] W. E. Lorensen, H. E. Cline, Marching cubes: A high resolution 3d surface construction algorithm, COMPUTER GRAPHICS 21 (4) (1987) 163–169.