UNIVERSITY OF OSLO
Department of Informatics

Three Dimensional Data Acquisition and The Registration Problem

Master thesis

Rasmus Fredrik Bugge

1st August 2007

Contents

Acknowledgements

The last eighteen months of my master studies were truly exciting and instructive! For this I am indebted to my supervisor Professor Knut Mørken. Thank you so much for insightful and motivating feedback and for taking a genuine interest in my work. I am also grateful to Professor Martin Reimers who has willingly contributed with illuminating comments on the fast sweeping method and made meaningful remarks on chapter 7.

I also address a word of thanks to Lars Mikalsen for introducing me to some of the inner workings of the C++ standard template library and for always providing professional programming advices when they are needed.

Further, I would like to express my gratitude to my aunt Birgitte Skundberg, and to Anne Mette Mevik, for trying to improve my English. Your involvement has meant a lot to me.

Last but not least, I would like to thank my dear Lizzie — for your patience and unconditional, loving support.

July 2007 Rasmus Fredrik Bugge

Introduction

The first half of this thesis is a brief survey of different scanners used for the purpose of capturing "raw data" from a three dimensional (3D) object. Such scanners are commonly referred to as 3D scanners. The data are measurements that are needed in order to generate the points that make up the foundation for all subsequent calculations. One of the main purposes of the data *acquisition* could be to reconstruct an apparently three dimensional, digital representation of the object in question.

With this knowledge, we may realize that the use of such apparatus give rise to many interesting mathematical problems, one of which we will study in depth starting at chapter 4. As the heading for that chapter suggests, the main problem we consider is known as *the registration problem*. When we have two images, i.e. visualizations of some sort, of the same object taken at different viewpoints, or at different times, these will in general not be in the same coordinate system. The problem is in essence, how to bring the images together in a *common* coordinate system.

One can imagine several ways to represent a three dimensional object, also there are several different three dimensional objects to be visualized. Thus the registration problem can be stated in many different ways depending on the context. We will consider the problem in the context of scans of rigid body objects, captured by 3D scanners. This means that we will study registration where the objects are not allowed to undergo deformations, i.e. we focus on *non-elastic* registration. Under the restrictions that the objects in question are mainly allowed to undergo only rigid body movements, i.e. rotations and translations, the problem is also referred to as the *Pose Estimation Problem* or the *Orthogonal Procrustes' Problem*[1].

Consider next a brief sketch of what is in each of the chapters that follows.

[1]According to Greek mythology, Procrustes was a bandit with a bizarre conception of hospitality. In his village, he ill-treated the bodies of unfortunate visitors to make them fit a certain bed. Paradoxically, finding the appropriate shape *preserving*, rigid body transformation is sometimes referred to as the Orthogonal Procrustes Problem [45].

A chapter overview

Chapter 1, On 3D Scanners and 3D Data Acquisition. In the first chapter we hopefully give an account of most of the different types of 3D scanners that are in use today. As the heading suggests, we will also try to give a brief account of some of the *techniques* that such scanners use when they gather 3D information. We will list some of the different areas where 3D scanners are put to use, and at the end, we take a look into how today's software handles the information, acquired from a scan.

Chapter 2, Konica Minolta VI-910. In this chapter we take a deeper look at a certain type of non-contact, 3D laser scanner currently in use at the University of Oslo. The scanner in focus is Konica Minolta's VI-910. Firstly, the technical specifications of the scanner are accounted for, and then the details are given, concerning how it operates.

A part of the standard Konica Minolta 3D scanner accessory is their proprietary software program called The Polygon Editing Tool. So at the end of the chapter, the main functions of The Polygon Editing Tool are explained in detail. Through this we will hopefully also get an intuitive feel for the registration problem.

Chapter 3, Mathematical Preliminaries. Before we proceed into Part II and III which are more mathematically involved, we need to agree upon notation and give an overview of some central mathematical topics. This is hopefully what is done in chapter 3 which concludes Part I of the text.

Chapter 4, The Non-Elastic Registration Problem. As implied in the introduction, there are many problems related to post-processing of scanner data. The main purpose of this chapter is to end up with a precise formulation of the registration problem. Since this problem in general applies to a wide variety of cases, we need to restrict it somehow. Specifically, we try to formulate the problem of aligning two sets of three dimensional point cloud data allowing for only a restricted set of affine transformations.

Chapter 5, A Closed-form Solution. Chapter 5 marks the entry point of Part III where we study several possible solutions to the registration problem. In this chapter we follow in the footsteps of Berthold K. P. Horn [26] and use unit quaternions to obtain a closed form solution to the problem.

Chapter 6, An Exhaustive Method. The usual way to approach the registration problem is to seek the help of computers to employ an iterative scheme in the search for the optimal solution parameters. Thus, we continue our study of solutions by taking a look at a *numerical* one, in chapter

6. As we will see, we leave the point cloud setting introduced in chapter 4, and try to register a special form of space curves.

Chapter 7, An Area-based Method. In this chapter we turn to look at registration of point cloud data rather than curves. As will become evident, the numerical solution we present here is very different from the previous.

Chapter 8, A Feature-based Approach. In this chapter we finish the presentation of numerical solutions with a method which in a way combine the ideas of the two former. Firstly, it is not as time consuming as the numerical solution of chapter 6. Secondly, it makes not so strong assumptions about the relative initial positions of the data as the numerical solution of chapter 7.

Chapter 9, Implementation. A C++ implementation of the algorithm outlined in chapter 7 has been made, and in this chapter we will go through the implementation specific details. Chapter 9 concludes Part III of the text.

Chapter 10, Discussion & Conclusions. In the last chapter of the text we share our thoughts on the registration algorithm outlined in chapter 7, in light of experimental results. We mention some of the experiences which we have gained in the making of the program and point out possible improvements.

Part I

Background Material

Chapter 1

On 3D Scanners and 3D Data Acquisition

As we recall from the introduction, we will start this part of the text by studying different types of 3D scanners and scanning techniques. We will give a brief description of the techniques employed by some of the most commonly used scanners and see where they find application — covering *all* 3D scanners would require more space than we have available. At the end of the chapter, we give a short overview of some of the associated software and we mention some typical software functions. We begin by considering a historical note on the development of the camera and the design of 3D scanners.

1.1 A historical note

Capturing reality, or real-world objects, in a useful way, has been a great challenge throughout the history; the first two dimensional pictures were captured on a piece of paper as late as in the beginning of the 19th century. However, the basic principles of the camera was described as early as 450 B.C. by Chinese philosophers. The need for a *three* dimensional "picture" of real-world objects was met as late as in the early 1950s, by the Ferranti company [45]. They developed one of the first successful coordinate measuring machines; a probe which had to be in contact with a workpiece in order to collect its coordinates. The probe was accurate but slow, and as it had to be in contact with the workpiece, the workpiece could possibly get damaged. For special fields with delicate objects to be scanned, this was a big concern.

The need for a quicker and safer way to collect 3D data led to the use of light. There are two considerations when dealing with light: What kind of light should be emitted, and in what way? For one, the light source can be made so that it emits one single beam, hitting the surface of the

object at one single point. When it comes to efficiency considerations, this is not considered an improvement in comparison with the contact-probe. An improvement would be to let the source emit a cone of such beams hitting a larger area of the object at a time. However, this approach has turned out to be difficult to implement, and the technique is seldom used. Finally, one could construct the source in such a way that it would emit a laser plane which would hit the object with one continuous stripe of light at a time. This last method of scanning 3D objects has proved to be the one which most manufacturers make use of today.

We have seen that there are many different approaches to the design of a 3D scanner. This implies several different techniques for the purpose of three dimensional data acquisition. We will in the following section try to cover some of the most common out these.

1.2 How does a 3D scanner work?

There is no single answer to this question that is applicable to all 3D scanners on the market. As we may have noticed, 3D scanners in general serve different purposes, and therefore they differ in both functionality, weight and size. So before clear answers can be stated, it is convenient to split the collection of 3D scanners into at least two main groups; non-contact and contact scanners. That is, they are placed into categories depending on whether or not they are in contact with the surface of the object to be scanned. Further, we note that scanners can also be classified as being active or passive, depending on whether or not they emit radiation themselves. For a more complete overview of the types of 3D scanners in use today, see figure 1.1 on the facing page due to Curless and Seitz [14].

As we can see from the overview, one of the two main groups of 3D scanners is that which includes non-contact scanners. Moreover, the group of non-contact scanners includes those scanners one can classify as optical. The group of optical scanners makes up the largest subset of shape acquisitioning systems and hence we will focus our attention on this type of scanners. Generally, optical scanners emit some kind of structured light onto an object. Then the reflected light pattern, whose shape is influenced by the surface of the object, is collected by the scanner's light receiving apparatus. Eventually, 3D information is extracted by a digital signal processor.

As there are a couple of ways to structure light, there are at least a couple of different optical 3D digitizers; the ones emitting (coded) halogen light, and the ones emitting light in a narrow beam, consisting of nearly one single wavelength (i.e., a laser). Laser light is often preferred. This is due to the facts that coherent light can be held in tight focus over a long range, it can be made insensitive to ambient[1] light, and it seldom has prob-

[1] Ambient light is light that comes from no particular light source.

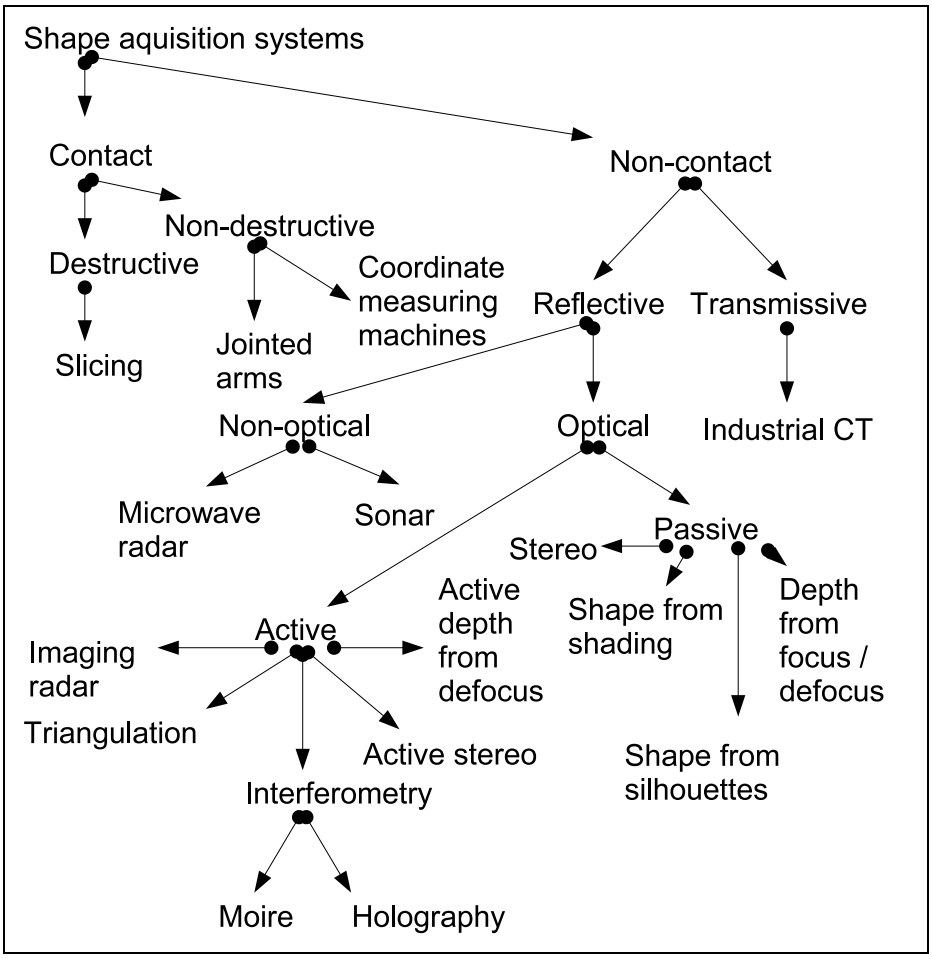


Figure 1.1: An overview, or taxonomy, of three dimensional shape acquisitioning systems.

lems with heat-dissipation (it does not lose energy to the surroundings) [13].

Also what separates the optical scanners, is how the emitter is structured and what light receiving sensor they are equipped with. Recall that scanners which emit a laser plane are the most common. Note that the emitter can project one, or a multiple of such planes. Systems that emit a single plane are often preferred, for one thing as no more steps are required to decipher light reflections [13].

The different technologies for non-contact scanners require different approaches to the task of measuring distances. We will review briefly the concepts of interferometry, triangulation, imaging radars and "shape from silhouettes".

The other main group of 3D scanners, contact scanners, includes both destructive and non-destructive scanners. A common type of contact scanners are the non-destructive mechanical scanners referred to as coordinate measuring machines (CMMs). These scanners also make use of light, however the light only reaches the tip of the inspection equipment. We will review how both the coordinate measuring machines and the destructive scanners acquire distance information.

1.2.1 Interferometry

In this context, interferometry should be understood as the use of optical interferometers which use the superposition (addition) of light waves to measure distance [45]. When two laser light patterns are superimposed on one another, an inference pattern is created. This pattern is a complex signal (with a precise mathematical representation) which can be low-passed filtered to include only the frequency[2] difference between the two initial patterns, as well as some constant terms. Among the constant terms is the phase difference[3] between the two initial patterns. When the frequencies of the two light pattern are the same, only the phase difference term remains, and from this term distance measurements can be retrieved [7].

As can be seen in figure 1.1, there are two types of interferometry techniques, moiré and holography. The technique described in the previous paragraph refers to the moiré method. Holography, however, is also based on low-pass filtering an interference pattern, and depth-values are also computed from the phase difference of two superimposed light patterns. However, it is the intensity signal of the added light patterns at the light receiving element which is low-pass filtered [7].

[2]For electromagnetic waves, like laser light, frequency is defined as the ratio of the speed of light to the wavelength.

[3]Imagine two particles, one on each of the light beams. The phase difference (measured in radians) refers to the amount of which the two particles move out of step of each other [45].

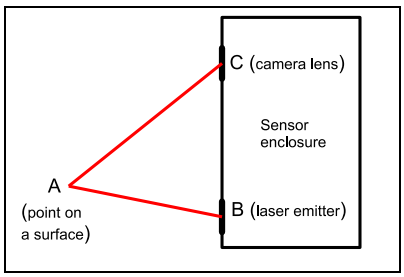


Figure 1.2: The shape of the triangle ABC is determined by the base distance (distance between the points B and C), and the angles at B and C.

1.2.2 Triangulation

Some 3D laser scanners follow a principle of measurement known as laser triangulation, or simply triangulation. Such scanners are probably the most commonly used 3D scanners.The laser-emitter, a spot on the object being scanned and the light-receiving device or camera, make up a triangle and thereby the name laser-triangulation. Three known quantities make it possible to calculate the coordinates of a point on a surface being scanned (see figure 1.2):

1. The distance between the point B where the laser is emitted and the camera lens, at C (*the base distance*),
2. the angle at the corner from where the laser is emitted,
3. the focal distance[4].

The angle at the corner from where the laser is emitted, is controlled by a signal processor. One can think of several ways to exploit trigonometric results to calculate the point's coordinates. We will review such a way in the next chapter. Like the interferometry scanners, modern 3D laser scanners can also acquire colour image data with the help of integrated digital cameras.

Two types of optical detectors are most commonly used in laser scanners that make use of triangulation: Position Sensitive Devices (PSDs) and Charge Coupled Devices (CCDs). A PSD outputs electrical current from two of its ends. When a spot is focused on the device, its output is a certain amount of electrical current, proportional to the spot's position on the device. The spot position is calculated by dividing the difference of the two amounts by the sum of them. PSDs are able to handle data at high rate and are not sensitive to the intensity distribution of the spot. A drawback with these kinds of light-receiving devices is that they determine themselves,

[4]The focal length is the distance from the center of the lens to the point where beams of parallel light (emitting from infinity), hitting the lens, are concentrated [45].

the center of the spot; if two spots are present, but overlapping, the device calculates one center which represents them both.

A CCD is a rectangular silicon piece, which can be used to receive (up to 70 per cent of) incoming light [45]. CCDs are digital in the sense that they produce discrete proportions of voltage as output, proportional to the amount of the light that hits them. A big advantage that comes with CCDs is the opportunity to post-process a signal. In chapter 2, CCDs are further discussed, since a CCD-camera is used in the 3D laser scanner we study there.

We note from [7, 14] that scanner optics should be constructed according to the *Scheimpflug* principle to ensure that the laser beam is in focus. The Scheimpflug principle describes how a camera lens, or its light receiving element, should be oriented (tilted) when the camera focuses on an object that is not parallel to the light receiving element [45]. The principle states that the plane of the light receiving element, the laser plane (the plane of focus) and the vertical plane passing through the lens should all intersect in a common line.

1.2.3 Imaging radars

Another type of active, optical scanners work by the principles of imaging radars[5]. Radars emit electromagnetic radiation towards an object, and record the round trip time t and power of the received radiation. The round trip time is defined as the time it takes for a pulse to exit a scanner, hit an object and return to the scanner. Thereby, the distance r to the object can be recovered as

$$r = \frac{ct}{2},$$

where c is the speed of light. The power of the received radiation depends among other things on r, the object's surface reflection coefficient and the transmitter hardware.

A simple type of imaging radars called time-of-flight scanners gather no other information than the distance r. They represent wide range and low accuracy and are suitable for scanning larger objects. Contrary to what is the case for 3D triangulation scanners, it can take several minutes for a time-of-flight scanner to do a high-resolution scan.

Other kinds of imaging radars include scanners which modulate either the frequency or amplitude of the emitted beam. Similar to the interferometry technique, the distance r to the object in question is calculated by combining what is known about the frequency f and the phase difference $\Delta\Phi$ of the outgoing and incoming light,

$$r = \frac{c\Delta\Phi}{4\pi f} \quad [7].$$

[5] RAdio Detection And Ranging [45].

Figure 1.3: A snap shot showing a pattern of structured halogen light on the surface of a face. Courtesy of Christian Roquefort, director of sales and marketing at InSpeck.

1.2.4 Other active scanners

As mentioned above, there are various ways to structure light. Some 3D scanners have a functionality which is based on the projection of a coded halogen light pattern onto the object to be scanned. The coded light is usually emitted by an LCD projector and is made up of parallel stripes with different intensity. When the light hits the object, a striped pattern becomes evident. Figure 1.3 is meant as an illustration of what this could look like. From the deformation of the stripes on the object's surface, depth-values are extracted. A digitizer using coded light can acquire both distance and colour image data in *one* scan. Information about colour is otherwise something one has to add and approximate using appropriate software.

1.2.5 Shape from silhouettes

Scanners that recover the shape of an object based on the object's silhouette are categorized as passive, non-contact scanners. They only detect reflected ambient radiation, like visible light. The object is initially photographed repeatedly in front of a well contrasted background and in the end the different silhouettes are merged to form the contours of the object.

As another example of passive scanners we mention stereoscopic scanners. They utilize the slight differences which occur when two cameras observe the same scene, to determine the distances to each point in the scene.

1.2.6 Non-destructive contact scanners

The other main group of 3D digitizers, contact scanners, includes coordinate measuring machines (CMMs) and hand-held, contact scanners. Coordinate measuring machines move the tip, or probe, of a mechanical arm, over the surface of the object to be scanned. Measurements are then con-

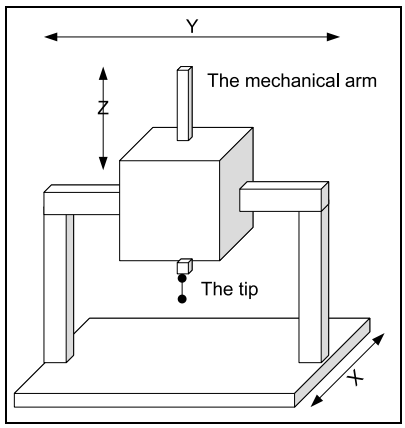


Figure 1.4: A schematic drawing of a coordinate measuring machine.

stantly transferred from the scanning equipment, back to a computer. See figure 1.4 for a schematic drawing of a CMM according to [45].

Nowadays, high quality servo motors are employed to smoothly position the probe. The motors can move the mechanical arm in any of the directions x, y, z, review figure 1.4. The tip of the mechanical arm is spring loaded to give way when it hits a surface.

The hand-held contact scanners work much the same way as the coordinate measuring machines. Their mechanical arms are jointed and thus the tip can be moved in any directions and through most angles, by hand. Each time a point is to be saved, a switch is pressed. The measurements are transferred to a host computer for further computations, e.g. to make interpolating curves for reconstructing the surfaces of the object.

1.2.7 Destructive slicing

The second group of contact scanners includes scanners that literally destroy the workpieces. The technique of destructive slicing is also referred to as cross sectional scanning. A cross sectional scanner is based upon the ideas of a 2D scanner; the apparatus cuts off thin slices of an object, e.g. from top to bottom, and returns an image of the two dimensional profile which is exposed. The set of images which is retrieved can each be represented by points and the different layers of points can be brought together to form a three dimensional point cloud.

The cross sectional scanning technique of [28] implies covering the object to be scanned, with a hardener. This way, when an image of the object's cross section is sent to a connected PC, the cross section appears as a bright area with a dark surrounding in the software's graphical user interface. By applying edge detection algorithms, the two dimensional images can be converted to points. The scanning devices of CGI [28] can cut off from an

Specs	Non-contact scanners		Contact scanners	
	From	*To*	*From*	*To*
Scan range	0.50m	300m	N/A	N/A
Capture time	0.30s	N/A	1 hour	3 hours
Accuracy (mm)	$\pm 4.00 \cdot 10^{-3}$	± 40.0	$\pm 1.30 \cdot 10^{-2}$	$\pm 4.30 \cdot 10^{-1}$
Precision (mm)	$8.00 \cdot 10^{-4}$	$3.00 \cdot 10^{2}$	$1.27 \cdot 10^{-2}$	N/A
Weight	1.00kg	N/A	5kg	N/A
Points per sec	$1.00 \cdot 10^{4}$	$1.90 \cdot 10^{6}$	N/A	$0.50 \cdot 10^{6}$

Table 1.1: Table showing technical specifications for a collection of contact and non-contact scanners. The data are collected from manufacturers like Immersion [11], Konica [25], Faro [16], 3rdtech [1], InSpeck [27], Leica [20] Breuckmann [21], nub3d [37], Steinbichler [39], Scantech [43] and CGI [28].

object, slices thinner than a $\frac{1}{500}$-fraction of a millimeter.

1.3 Technical specifications

We consider next how well 3D scanners *perform*. Hence, some technical specifications for the main types of scanners mentioned in the discussion above, are listed in table 1.1. The entry N/A indicates that the corresponding specification for a given group of scanners has not been determined. The reasons for this are discussed below.

As one can see, each specification is represented by a range of values to reflect the effect of two causes: Firstly, the performance of a high-end scanner working under optimal conditions[6] are compared to the performance of a low-end scanner working under poor conditions. Secondly, the deviations are due to the fact that some scanners are built to scan details in, say a cellular phone, whilst others are made to scan buildings. The specifications are explained below.

1. **Scan-range**: This is the depth of field. According to the table, some non-contact scanners can scan objects that are placed as near as 0.5 meters whilst others can scan objects placed as far away as 300 meters. Scanners that can handle objects placed only half a meter away, usually reach no farther than a couple of meters. An object to be scanned must be placed within the scan-range of the particular scanner to secure proper measurements.

2. **Capture time**: This is the time spent performing one scan. It depends

[6]The conditions which affects the result the most are air temperature, humidity and lighting.

for all scanners, on the size of the object and on what kind of accuracy is required.

3. **Accuracy**: This is defined as the deviation of the measured position of a point from the point's true position.

4. **Precision**: This is the smallest change of distance which can be detected.

5. **Weight**: The total weight of the scanning equipment.

We give some general notes on the entries in the table:

- A 3D optical scanner's accessory can include different lenses to be mounted on the camera in order to widen or narrow its field of view. Further, the scan range entries for contact scanners are both N/A. The nature of contact scanners implies that the object will always be within scan range. It would make more sense to talk about the maximum *size* of the object when it comes to contact scanners. Again, this vary a lot due to the many different areas of application.

- The capture time entry and the precision entry for contact scanners should not be considered representable for contact scanners in general. Few manufacturers have posted an estimate of this specification and the numbers which are given in the table, represent a small selection of cross sectional scanners [28].

- In addition to what is already mentioned, accuracy depends on the shininess of the object's surface. Shiny surfaces lead to more diffuse reflections. The accuracy of laser scanners is also limited by the phenomenon of laser speckle, i.e. the mutual inference of coherent laser beams [13]. Scanner systems will often make repeated measurements in the same view direction and return the average of these measurements in order to improve the accuracy [7].

- The precision of laser scanners is dependent on the embedded optics which dictate the beam diameter. The diameter has to be smaller than, or equal to a feature size in order to accurately measure it. The beam diameter is at its smallest in the center of the scan-range, and hence this is where we should expect the best performance with regard to precision.

In addition to the specifications which are accounted for, a scanner's "repeatability" is sometimes listed. This is the scanner's ability to reproduce a series of similar measurements of the exact same distance. However, that kind of information has been difficult to retrieve.

As we have revealed how some 3D scanners work, a natural next step is to see where they come into play.

1.4 Applications and practical concerns

From the tender start of being a device for inspection of military equipment, 3D scanners have become a widely used apparatus in many different areas[7]: Archeology, architecture, computer art, case studies, fashion and footwear, industrial design, medicine, miscellaneous industries (especially the car-, tool- and toy industry) and movies (both animated and live action).

Optical 3D scanners are perfect for scanning archaeological workpieces. The use of light means that you do not have to worry about wrecking invaluable historical artifacts. Optical digitizers also provide the high grade of accuracy which is often needed in these kinds of scans.

High grade of accuracy is also what is needed by industrial designers. Depending of course on what kind of product is invented or improved, there is often a need for a precise digital representation of a workpiece for standardizing, before mass production.

Another advantage of non-contact scanners is their ability to work across great distances. This ability is exploited when scanning within the field of architecture. One example of a large-scale scan is the scanning of the Statue of Liberty.[8] Typically within architecture, the workpieces can be as far away as one hundred meters but then the accuracy is at best ± 6 millimeters per point.

There are many examples of the use of 3D scanners in case studies. There is a consortium of six organizations which make up a European 3D project, the VIHAP[9] project [18], with the aim of conserving European art treasures for the public. Among the fruits of the project is a 3D representation of the Pisa Cathedral.

In all the above applications, the objects to be scanned are three dimensional. This means that they have to be scanned from different sides and angles. A major challenge is how to cover all the sides of an object. Aside from hand-held, mechanical scanners which of course work their way around an object with ease, different manufacturers have come up with different solutions to the multiple view problem. Some, like InSpeck [27], offer a multiple digitizer environment to scan an object from up to three different angles at the same time. Others have made rotating tables for objects (not too heavy) to be placed upon. That way, when the digitizer has scanned one side of an object, the table turns according to an angle determined by the supplied software.

The multiple view problem is however, not the only task that software needs to solve. In the following section we consider some of the other problems which needs to be solved and we list some of the market leaders on

[7] The list does not claim to be exhaustive.

[8] See `http://www.arch.ttu.edu/digital%5Fliberty/`.

[9] Virtual Heritage, (High Quality) 3D Acquisition and Presentation.

scanner software.

1.5 Related software

When the scanning phase is completed, and the raw data have been acquired, software takes care of the "reconstruction" phase. That is, software processes the initial data into geometry, often polygons. Most manufacturers of scanners have developed their own software with limited functionality, which they ship together with their scanners.

There is nonetheless a myriad of different manufacturers who develop useful software related to 3D scanning. A tiny selection of some of the manufacturers (and their software product in parenthesis) includes INUS Technology [44] (RapidForm), InnovMETRIX (PolyWorks), SensAble Technologies (FREEFORM Modeling Plus) and GeoMagic (GeoMagic Studio). According to INUS' website [44], RapidForm is the world's best selling 3D modeling software. INUS' software can handle point-clouds of up to hundreds of millions of points. The software also removes points that are generated due to bad scanner accuracy.

From the edited point-clouds, polygons (usually triangles) are computed to form workpiece "skeletons". Algorithms make sure that no polygons with bad normals are generated and also they can also make sure that holes are filled. Points are connected by curves and eventually surfaces are generated. Splines, as linear combinations of B-splines provide good tools for creating curves and surfaces. Their advantage is that they can be manipulated to form almost any shape you want, and they can be evaluated using numerically stable algorithms.

Smooth surfaces can also be produced by applying different subdivision schemes ($\sqrt{3}$, Loop's, Butterfly) which increase the number of polygons. Finally, a texture can be applied to make the digital representation "come alive". The details of exactly how all these steps are implemented are of course hidden in secret algorithms, but the software's work flow is something along these lines.

As outlined above, one often has to do several scans to capture the whole of a three dimensional object. The partial scans are brought into a common system of coordinates by the software and then they are registered or aligned to reproduce the overall shape of the object. INUS Technology, like most manufacturers of 3D modeling software, claim to have come up with algorithms, securing the tightest fit between overlapping areas. By studying the geometry of the overlapping areas, the algorithms merge them to produce a single polygonal model that includes all the surfaces seen during the digitizing process. The textures of the partials model can also be merged to form a single texture for the merged model. The result is a final, complete 3D model.

1.6 Summary

We have now accounted for many of the different types of 3D scanners on the market and looked into how they work and where they come into play. Also we have reviewed some of their technical specifications and we have mentioned some of the operations that can be performed on scanner data, by appropriate software. It is time to have a more detailed look at one particular scanner. We choose to look at a scanner from the group of non-contact, active, optical laser scanners, which are probably the most wide spread. Thus in the next chapter we review the VI-910 from Konica Minolta.

Chapter 2

Konica Minolta VI-910

In the previous chapter we had a look at 3D scanners and scanner software in general. In this chapter we will have a more detailed look at a particular 3D laser scanner named VI-910. This way, we get to see exactly how spatial coordinates can be calculated, and we get an overview of the functions offered by a particular software, The Polygon Editing Tool.

2.1 A non-contact scanner

The Konica Minolta VI[1]-910 is a 3D laser scanner. The scanner emits a class[2] 2 laser-plane, which is swept over the scanner's field of view by a galvanometer[3] driven, rotating mirror. The plane occurs as a laser-stripe on the object which is being scanned, and each stripe is reflected back to a CCD (see section 1.2.2 and section 2.2) camera. The camera is situated just above the laser emitter.

The VI-910 features a fairly light-weight body, approximately 11 kilograms, which can stand alone or on a tripod. As it is portable, a nice optional accessory is a 128MB compact flash memory card which makes it possible to carry the scanner around and save data without having to bring a computer along with it. However, it is designed for indoor use only and to be used within a temperature range of ten to forty degrees Celsius.

The scanner comes with three exchangeable light-receiving lenses. Associated with each lens, there is a *view volume*, and thus the view volume of the camera can be manipulated. The view volume has the form of a square pyramid lying on its side with its top cut off. The top points at the scanner.

[1] For countries outside Europe, VIVID-910 is the name.

[2] IEC(International Electrotechnical Commission) standard 60825-1: "It is presumed that the human blink reflex will be sufficient to prevent damaging exposure, although prolonged viewing may be dangerous" [45].

[3] An electromechanical transducer/energy-converter. It produces a limited rotational movement in response to electric current flowing through a coil of wire spun around the galvanometer which is surrounded by a magnet [45].

Figure 2.1: The Konica Minolta VI-910.

The size of the pyramid is defined by three values, i.e. the width and length of its rectangular base, and its height. We must always make sure that the object to be scanned is placed within the view volume of the camera lens in use.

An object must not be placed any nearer the scanner than 0.6 meters and not further away than 2.5 meters. We usually say that the *scan range* is from 0.6 to 2.5 meters in front of the scanner. For best results, the object should not be farther away than 1.2 meters.

Table 2.1 on the facing page shows how the view volume varies according to what lens is in use, at the two extremities (0.6 and 2.5 meters) of the scan range. The object to be scanned has to be inside the view volume defined by the three values x-range, y-range and z-range. That is, the object must be placed inside the imaginary pyramid centered straight in front of the scanner, with base length equal to the value x-range, base width equal to y-range and with height equal to z-range.

Further, it takes between 0.3 (in FAST mode) and 2.5 seconds (in FINE mode) for the scanner to complete a scan, depending on what kind of accuracy is desired. In FAST mode we will get lesser accuracy than in FINE mode and also fewer points will be sampled off the object's surface. More specifically, in FAST-mode we typically get $77,000$ pixels at output, whereas in FINE-mode we get four times as many ($308,000$). In addition, the VI-910 captures a 640×480 colour image with twenty four bit per pixel for storing colour information. It includes a SCSI II output interface for external connections and a 5.7 inch LCD viewfinder. Figure 2.1 shows a picture of the scanner.

Lens \ Extremity	at 0.6 meters:		at 2.5 meters:	
TELE	x-range:	111 mm	x-range:	83 mm
	y-range:	83 mm	y-range:	347 mm
	z-range:	40 mm	z-range:	500 mm
MIDDLE	x-range:	198 mm	x-range:	823 mm
	y-range:	148 mm	y-range:	618 mm
	z-range:	70 mm	z-range:	800 mm
			at 2.0 meters:	
WIDE	x-range:	359 mm	x-range:	1169 mm
	y-range:	269 mm	y-range:	897 mm
	z-range:	110 mm	z-range:	750 mm

Table 2.1: The range of input varies according to what lens is in use, and how far, in front the scanner, an object is placed.

2.2 The scanning pipeline

As it has already been revealed what kind of scanner the VI-910 is, much of the way it works has already been discussed. Some things are worth repeating though, and some more details worth giving. We will see this as we now consider the scanning pipeline.

The object to be scanned must be visible within the LCD viewfinder on the back of the scanner. As we recall from the previous section, we can enlarge of diminish the view volume by changing the camera lens. The laser plane will be swept over the object three consecutive times, from top to bottom. A proportion of the reflected laser is received by the CCD camera.

The CCD is made up of a silicon piece which is segmented into an array of light-sensitive cells or photosites (really capacitors), and a spot on an object is stored at a certain selection of these cells. When hit by the reflected laser, the cells charge up with an amount of electrical current proportional to the light-intensity. The charge is accumulated in an amplifier which converts it to voltage which in turn is digitized. The proprietary file format for the digitized data is encrypted.

The exact position on the CCD at which a spot on the object is stored, can be calculated down to a fraction of a pixel. One way is to sum up the products of the pixel-number and its associated intensity, e.g. stored as an eight bit word, and divide by the sum of the intensities. A high-end, 2-dimensional CCD may contain 2048×2048 (almost 4.2 million) pieces of light-intensity information.

In order for the VI-910 to retrieve surface point coordinates, the scanner makes use of Konica's *Triangulation-light-block method*. This method is a special kind of active triangulation introduced in section 1.2.1. The laser emitter, a point on the object which is being scanned and the scanner cam-

era, still make up the corners of a triangle. Also, the camera and the laser emitter are situated straight on top of each other, thus they are on the same axis. We will review in section 2.2.1 how the measurements can be carried out by considering this particular configuration.

The digitized version of the light-intensity information is eventually handled by the Polygon Editing Tool supplied by Konica Minolta. The software calculates points in three dimensions relative to a three dimensional right-handed Cartesian coordinate system. The center of the light receiving lens is set as origin.

The example below should clarify what is done by the scanner when it calculates the coordinates of a certain point in space. As Konica Minolta, we apply a three dimensional right-handed Cartesian coordinate system with its origin at the camera lens.

2.2.1 Calculating coordinates — an example

A note should be taken at this point, as we do not claim that the following represents the way Konica Minolta have made their scanners calculate coordinates. The details in their Triangulation light-block method have not been possible to retrieve. A compact expression for a point's coordinates in the case of an active laser triangulation system like the VI-910 is given in [7]. In the following calculations we give the details leading out to that expression (using only symbols). More specifically, we will compute the coordinates S_x, S_y and S_z of the point S in figure 2.2 on the next page.

Figure 2.2 illustrates a snapshot of what is typically at display both outside and inside the VI-910 when doing a scan. To enhance the visibility, the different components have been enlarged and/or diminished.

The coordinates of many points[4] on the surface of the side of the object facing the scanner, are calculated in *one* scan. In figure 2.2 we see that the laser plane hits a box, and the (Cartesian) coordinates of the point S, are what we want to compute. It turns out that the coordinates are all negative: The center of the coordinate system is located at the center of the lens. The negative z axis extends forward in front of the scanner, the negative y axis extends downwards and the negative x axis extends to the left.

We have not yet agreed upon a notation for such fundamental entities as points, lines and angles. This will be done in chapter 3 as we from that point on will be using these entities extensively. For this small example we use an intuitive notation:

- Single capital letters, as S, T and F in figure 2.3 on page 22, mark different points in space, and the point's Cartesian coordinates are denoted by indices, say S_x, S_y and S_z.

[4] Up to $308,000$ as we remember from the previous section.

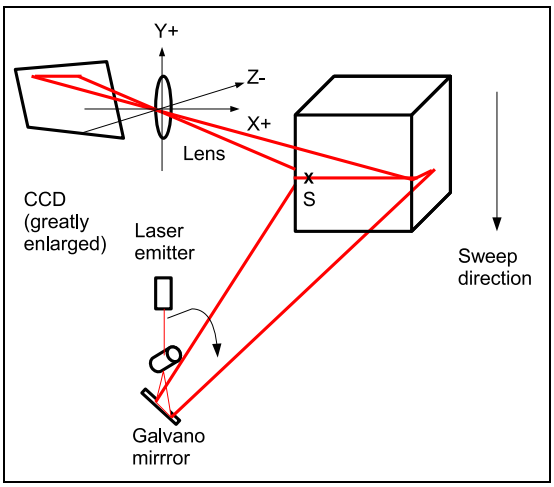


Figure 2.2: Drawing of a scan, with the laser plane as seen partly from the side.

- The sign $\angle$ followed by three capital letters denotes angles between line segments.

As stated in chapter 1, some known quantities make the distance calculations straightforward (see figure 2.3):

- The base distance (equals $-M_y$ since M_y is negative),
- the angle $\angle TMS$,
- the focal distance $-F_z$,
- the exact position of a spot on the CCD (H_x,H_y).

By similar triangles we see from figure 2.4 that

$$\frac{S_z}{S_x} = \frac{F_z}{H_x}$$

so

$$S_x = \frac{S_z H_x}{F_z}.$$

On the right side of the equal sign, only the quantity S_z is unknown. The coordinate F_z equals the (negative) focal distance and H_x is the x component of the spot on the CCD where the laser hits after being reflected off from S. Since S_z is also one of the coordinates we are after, we will return to the equation above when S_z is found.

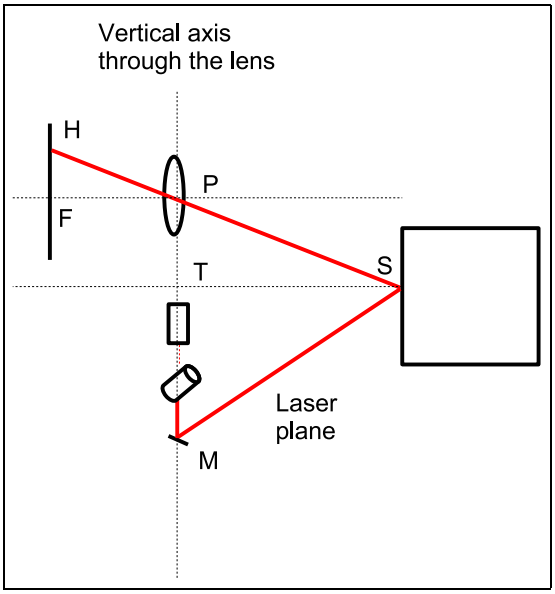


Figure 2.3: A schematic drawing of figure 2.2 as seen from the side.

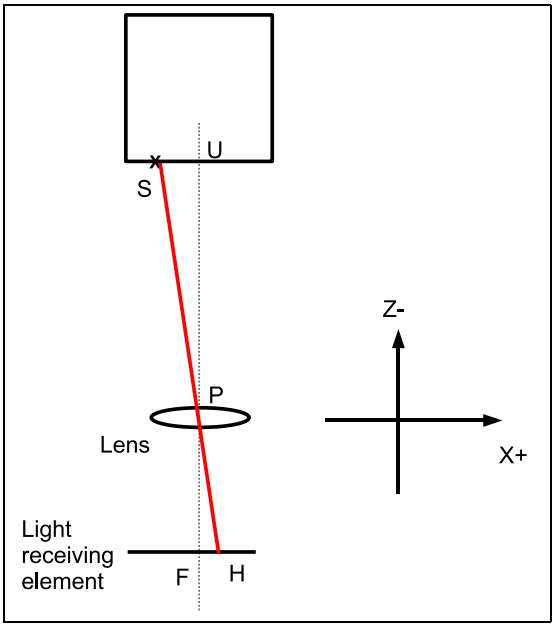


Figure 2.4: Yet a schematic drawing of figure 2.2, this time from above.

Consider next what must be done to calculate the coordinate $S_y = T_y$. We know the angle $\angle TMS$ and an expression for this angle reads

$$\tan \angle TMS = \frac{S_z}{M_y - T_y}.$$

From this we get that

$$\begin{aligned} T_y &= M_y - \frac{S_z}{\tan \angle TMS} \\ &= M_y - S_z \cot \angle TMS. \end{aligned}$$

So again we need to know the coordinate S_y. Note that M_y is known — it equals the (negative) base distance[5]. In the expression for T_y, however, we can replace the factor S_z by values that are known. By similar triangles, we see from figure 2.3 that

$$\frac{S_z}{T_y} = \frac{F_z}{H_y}$$

so

$$S_z = \frac{F_z T_y}{H_y}.$$

Inserting this in the expression for T_y above, we get

$$\begin{aligned} T_y &= M_y - \frac{F_z T_y}{H_y} \cot \angle TMS \\ T_y + \frac{F_z T_y}{H_y} \cot \angle TMS &= M_y \\ T_y \left(1 + \frac{F_z}{H_y} \cot \angle TMS \right) &= M_y \\ T_y &= \frac{M_y}{1 + \frac{F_z}{H_y} \cot \angle TMS} \\ &= \frac{M_y H_y}{H_y + F_z \cot \angle TMS}. \end{aligned}$$

All quantities on the right side of the equal sign are now known so $T_y = S_y$ is found. The quantity H_y is the vertical displacement on the light receiving element (relative to the center point F) of the laser spot "corresponding" to S.

We can now compute S_z from its expression above,

$$S_z = \frac{F_z}{H_y} S_y,$$

[5]The coordinate M_y is negative according to the position of the point M in the coordinate system which is introduced, whereas distance is thought of as a positive quantity in this context.

by inserting for S_y what we just found:

$$\begin{aligned} S_z &= \frac{F_z}{H_y}\frac{M_y H_y}{H_y + F_z \cot \angle TMS} \\ &= \frac{F_z M_y}{H_y + F_z \cot \angle TMS} \end{aligned}.$$

It remains to determine the final expression for the S_x coordinate,

$$\begin{aligned} S_x &= \frac{H_x}{F_z} S_z \\ &= \frac{H_x}{F_z}\frac{F_z M_y}{H_y + F_z \cot \angle TMS} \\ &= \frac{H_x M_y}{H_y + F_z \cot \angle TMS}. \end{aligned}$$

These results can be written more compactly,

$$[S_x, S_y, S_z] = \frac{M_y}{H_y + F_z \cot \angle TMS}[H_x, H_y, F_z].$$

which is in accordance with Besl's expression [7]. This completes the calculations.

According to Konica [25], the coordinates of a point as measured by the VI-910 deviate from the point's true position by a few tenths of a millimeter (in FINE mode). More precisely, if (x, y, z) is the point's true position, then the scanner's measurements are found within the intervals $(x \pm 0.22mm, y \pm 0.16mm, z \pm 0.10mm)$.

We should mention that the Scheimpflug principle introduced in section 1.2.2 is not satisfied. This would imply moving the CCD back and fourth along the z axis and rotating it about the x axis as the laser plane is constantly in motion. However, the scanner has built-in auto focus functionality ensuring that the object to be scanned will always be in focus, see section 2.3.1.

2.3 The Polygon Editing Tool

The Polygon Editing Tool is software developed by Konica Minolta for manipulating scanner data. It is meant for installation on a Windows machine. The Polygon Editing Tool can perform a variety of functions. Some functions set the scanner parameters, while others are related to editing. We will give a description of both types of functions.

2.3.1 Hardware settings

The software offers step scanning as well as single scanning. Step scanning implies controlling a rotating table, and making it rotate through a set of angles which we decide. The set of angles sum up to 360 so if we place our scan-object on top of the turntable[6] for a step-scan, the object will be scanned from all sides.

As mentioned briefly at the end of section 2.2.1, we can make use of the scanner's auto focus functionality to make sure that the object in question, will be in focus. Prior to a scan, we should always push the AF (auto focus) button to make the scanner determine the distance from the camera lens to the object. A picture of the object will be displayed together with the distance measurement, so that we will be able to judge whether the distance was rightfully determined. If the picture occurs blurry, we need to repeat the procedure.

2.3.2 Visualization

There are several ways to display the data collected by the scanner. By default, the mesh which is first at display is both rectangular and triangular and shaded in green. However, we can choose to

- display all the vertices,
- display the normals of all the vertices,
- display the mesh in wire frame-mode,
- display a shaded mesh or
- display the mesh in texture-mapped mode.

The functions should be quite self-explanatory. For example, displaying the mesh in texture mapped mode implies that a digital still image taken of the object by the VI-910 is mapped to the surface of the mesh. This makes it appear as a more true copy of the scanned object. We can choose to save a certain view, say when all the vertex-normals are at display (as red line segments), but the file format is again encrypted.

2.3.3 Build functions

According to the software, "building" include elaborate functionality such as

[6]We need to set which type of turntable we use and to which of the host computer's COM-ports it is connected. At the University of Oslo the turntable is connected to the COM1 port of its host computer. The turntable, or rather the control unit for the stepper motor which dictates the turntable, is of type isel(RF-1).

- Registration
- Merging
- Triangulation
- Sub sampling
- Polygon-checking
- Filling of holes
- Mesh refinement and smoothing

The top of the list reads Registration. This is something we often want to do, at least if we have scanned an object from different sides; registration brings the scans into a common frame of reference. We defer to chapter 4 for a more thorough discussion on the subject. In The Polygon Editing Tool, an initial registration can be done "manually" or automatically.

We sometimes want the software to do all the registration for us, that is, we want the registration to be done automatically. Automatic registration requires that we have selected more than five hundred vertices. Most likely, we have selected all the partial scans (at least if we want to align all of them), and this includes usually a lot more than five hundred vertices. Before the software can start the registration process, it asks us to choose one of the partial scans as a base element. This scan will be used as a "reference" and is necessary for the registration-algorithm to work properly.

For a manual registration, we also need to choose a base element. Further, the registration will be done pairwise with regard to the partial scans so we also need to choose the scan with which we want to register the base element. Hence, the two scans must at least be partially overlapping. The functionality demands of us that we pick out three pairs of points from the two scans — the two vertices which make up a pair must be at the same (by eye) spot on the scans.

When the registration is done, an error estimate is displayed, and we can choose to exit or repeat the process (to try to improve the result). The software also offers fine registration which is a refinement of the initial registration, either if it was done manually or automatically.

Another useful function which should be mentioned is merging. Merging gathers several registered scans, or elements, into one element.

The next build function in the list above is triangulation which means dividing large polygons into triangles. (Only rectangular and triangular polygons are available with this software, see figure 2.5 on the next page). A triangulation should be regular, or at least valid, to be convenient to handle. Among other things, a regular triangulation should not have holes. In figure 2.6 we see the result of triangulating the mesh from figure 2.5.

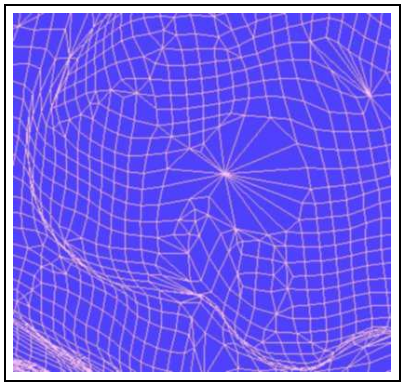


Figure 2.5: A segment of a polygon mesh representing a figure of the character Sméagol from the movie The Lord of the Rings. The segment shows the two largest toes of Sméagol's left foot as he sits on a river bank.

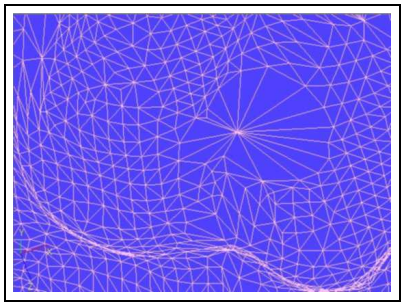


Figure 2.6: The polygon mesh representing the toes of Sméagol is now triangulated.

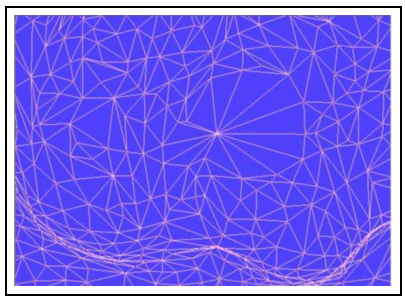


Figure 2.7: The number of points making up the polygon mesh representing the toes of Sméagol is reduced. Since we have made an adaptive subsample, points have been removed mainly from the flat areas of the foot.

At times we may wish to reduce the number of points in a mesh. With The Polygon Editing Tool, we can do so either uniformly or adaptively. Reducing the number of points uniformly, implies removing, say, every other point from the mesh. That is, points are removed without regard to surface curvature etcetera. Contrary to this, adaptive reduction implies removing points only from flat areas where many points do not contribute to the shape of the surface. In both cases we need to give as input to the software, the number of points we want to keep. (The total number of points before the subsample is taken, is displayed.) In figure 2.7 we have adaptively reduced the number of points making up the triangulation in figure 2.6.

We often want regular triangulations. For this, and for aesthetic reasons, the software offers to fill holes in the mesh representing a scan. Holes can be filled automatically or manually. If we want them to be filled automatically, the system will search through the selected mesh for holes. If a hole is found, its boundary points will be selected and highlighted in red. Before the hole is filled, a "flatness" parameter needs to be set — it reflects the curvature of the surface which will be used to fill the hole. Zero flatness implies filling the hole with a completely flat surface, while flatness set to 100 implies filling the hole with as curved a surface as possible.

Then we need to choose which algorithm we want the software to use for the process of filling the hole. There are three options:

1. We can choose to fill "by curvature", this creates meshes.
2. We can fill the hole "by curvature" and replace facets. This creates meshes and polygons on the boundary are possibly altered.
3. We can fill the hole by drawing straight lines between the nodes at the boundary.

We can fill all the holes with the current settings, or we can fill one hole at a time and possibly change the settings for each hole.

If we choose to fill holes manually, we have to locate the holes ourselves. When a hole is found, we need to click on three successive vertices on the boundary in a counter-clockwise manner, and a triangular polygon will be created with the three selected vertices as corners.

Another useful function in the Build menu is the one that checks a selected element for illegal polygons. The function goes by the same name, Check Polygons. Illegal polygons in this context can be divided into five different groups, depending on what "rule" they violate. The rules are:

- Intersection is not allowed other places than at the edges.
- Polygons must be convex. (No degeneration is allowed.)
- Polygons which share an edge must face the same way.
- Polygons must not be badly connected, i.e. they cannot share more than three vertices or two non-edge vertices.
- A boundary vertex must be shared by exactly two boundary edges.

The software offers to count how many polygons that currently violate the different rules. The vertices involved in these actions will be selected and we can choose to delete them.

From the Build menu there is also an option to smooth points or elements. This function regularizes surface point density through two attributes: Weight and repetition. Setting the weight attribute to a number between zero and one adjusts the level of regularization of the surface points density to somewhere between not regular at all, and strictly regular. The repetition attribute can similarly be set to a value between one and ten which dictates the level of smoothness of the selected points or elements.

The Build menu also includes functions such as Modify and Subdivision. When we choose to modify an element, the element is rebuilt by deleting small polygons. The Subdivision function is quite the opposite of Modify, as it rebuilds by dividing large polygons into smaller ones (triangles). Before initiating the Modify routine we have to set a minimum edge length or a minimum facet area. Edges, whose length does not exceed the minimum edge length, will be deleted, as will polygons with areas that does not exceed the minimum facet area.

2.3.4 Saving

When we are done editing, we want to save our work. The save function is found under the File menu. We have to select a folder and a name for the file in which the data will be saved. The format for the saved data is

encrypted. With that being said, the file is known to include the scan direction, parameters, classes etcetera. The file name extension for the encrypted data is CDM[7].

However the data can be exported to a known format. The software can convert the digitized data into the VRML[8] or STL[9] format, to mention just a couple. As one can see from the examples below, these formats make up appropriate starting points for the work of most developers.

Between the first and last line of an ASCII STL file,

```
solid [name]
...
endsolid [name],
```

there is allocated space which stores the unit normal and vertices of each *triangle* in a *triangular* mesh [45]:

```
triangle normal nx ny nz
    outer loop
        vertex v1x v1y v1z
        vertex v2x v2y v2z
        vertex v3x v3y v3z
    endloop
end triangle
```

The VRML format is a text format and it comes in several versions. From within The Polygon Editing Tool we can export a file from the encrypted format to both VRML version 1.0 and VRML version 2.0, see figure 2.8. We see from the example that the VRML format includes an array of indices, the coordinates of the polygon's vertices, and also additional information about the mesh.

The particular file presented in figure 2.8 represents a mesh that consists of one single *triangle* only; the `coordIndex` array contains only one single line and *three* indices. However, the VRML format also stores rectangular mesh. The three indices are used to look up in the `point` array where the triangle's vertices are stored. Thus, the vertices of the triangle are `point[0]` $= (\texttt{x1}, \texttt{y1}, \texttt{z1})$, `point[1]` $= (\texttt{x2}, \texttt{y2}, \texttt{z2})$ and `point[2]` $= (\texttt{x3}, \texttt{y3}, \texttt{z3})$.

[7] The file name extension for data produced by the VI-910i is CDK.

[8] Virtual Reality Modeling Language, a standard file format for 3D graphics.

[9] A file format native to the format produced by a STereoLitography Computer-Aided Design software created by an American company [45].

```
#VRML V2.0 utf8
# Polygon Editing Tool
# Object name : aTriangle
Collision{
    collide FALSE
    children [
        Shape{
            appearance
                Appearance{
                    material
                    DEF _DefMar Material{
                    }
                }
                geometry
                IndexedFaceSet{
                    coord
                    Coordinate{
                        point [
                            x1 y1 z1,
                            x2 y2 z2,
                            x3 y3 z3,
                        ]
                    }
                    solid FALSE
                    creaseAngle 0.5
                    coordIndex [
                        0, 1, 2, -1,
                    ]
                }
        }
    ]
}
Viewpoint{
    position x y z
    orientation 0 0 1 0
    fieldOfView float
}
```

Figure 2.8: An example file written in the VRML 2.0 file format.

2.4 Summary

Throughout this chapter, we have studied a laser scanner in some detail with a focus on its technical specifications and on the principles of distance measurement which it follows. We will not discuss these topics in any more detail. In the next chapter we will finish Part I of the text by hopefully providing the tools needed to start working on the more mathematically involved topics introduced in Part II.

Chapter 3

Mathematical Preliminaries

"The mathematics is not there till we put it there."

Sir Arthur Eddington

Hopefully we have now gained some knowledge as to how 3D scanners can gather range data, and in particular how this might be done by the VI-910. Further, the review of The Polygon Editing Tool suggested some typical operations related to scanner data. A natural next step would be to examine in depth some of the more interesting ones of those operations. But first, we need to agree upon notation and give an overview over some mathematical topics which will play a central role in the remaining text.

3.1 Scalars

The term scalar refers to a quantity without direction. In the mathematical branch of linear algebra scalars are equivalent to real numbers and we will denote them by lowercase letters, say, a, b and c. However, according to what may be argued as tradition, we will denote angles by lowercase, Greek letters, e.g. γ, β and α.

3.2 Points and vectors

Mathematically speaking, a point in $\mathbb{R}^3$ can be regarded as a vector from the origin in the frame of reference in which the point is given, up to the point. This way, there are no fundamental differences between a point and a vector. In the text, whenever we take this into account, we denote a vector or point by boldface, lowercase letters, say $\mathbf{p}$. Equivalently, we will also represent a vector or a point in terms of its components, say $[x, y, z]^T$ for a vector or point in $\mathbb{R}^3$.

When we manipulate geometry, it is however sometimes convenient to differentiate between the two; say, when we rotate a *point*, we often want to distinguish it from the *axis* (vector) of the rotation. We can easily make this distinction by using homogeneous coordinates.

3.2.1 Homogeneous coordinates

Using homogeneous coordinates, points and vectors in $\mathbb{R}^3$ are represented as four dimensional vectors. That is, they are represented as $[x, y, z, w]^T$, where $w = 1$ for points and $w = 0$ for vectors. In the following, whenever we use homogeneous representation, we will denote a point which lies in $\mathbb{R}^3$ by $\mathbf{p}^1 = [x, y, z, 1]^T$ and a vector by $\mathbf{v}^0 = [x, y, z, 0]^T$. However, as we will see in section 3.4.2 on page 36, introducing another dimension is not the only reason for choosing homogeneous representation.

3.3 Matrices

We will quickly review how we recognize a matrix: In our setting, a matrix is regarded as a collection of real numbers, ordered in a rectangular fashion. We will denote a matrix by uppercase, boldface letters, say **M**. If for example we have $\mathbf{M} \in \mathbb{R}^{p,q}$ then the matrix **M** consists of p rows and q columns of real numbers, i.e. pq real numbers.

3.4 Transformations in $\mathbb{R}^3$

If we want to be able to do computations on the data given by a 3D scanner, it is in the cards that we must be able to handle geometry in three dimensions. One of the basic tools in that respect are transformations and more precisely, matrix transformations. We will denote transformations by uppercase Greek letters, say Π. Generally, a transformation is a formula or rule that assigns to a given vector $\mathbf{p}$ in Euclidean space, a vector $\Pi(\mathbf{p})$. If Π is a matrix transformation, then $\Pi(\mathbf{p})$ would imply the computation of $\mathbf{Mp}$ for a matrix **M**.

As a matter of form we will list the four basic matrix transformations - translation, rotation, scaling and shearing. However, our focus will be on the former two, rotations and translations. They make up what is known as rigid-body transformations, as they do not affect the shape of the geometry of the given data. This is a desired property as we often want to use scanner data to reconstruct real-world rigid objects. Further, we choose to use homogeneous representation. This augments the usual 3×3 transformation matrices into size 4×4.

3.4.1 Rotation

There are many ways to represent a rotation in 3-space [45]. For a general rotation we must specify a fixed point, i.e. a point unaltered by the rotation, an axis or vector about which we want to rotate and a rotation angle [5]. If we choose the origin in our frame of reference to be the fixed point of our rotation, then any rotation can be represented in matrix form as the product $\mathbf{R}_x\mathbf{R}_y\mathbf{R}_z$. The three rotation matrices $\mathbf{R}_x$, $\mathbf{R}_y$ and $\mathbf{R}_z$ each represents a rotation about one of the coordinate axes x, y or z,

$$\mathbf{R}_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha & 0 \\ 0 & \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

$$\mathbf{R}_y = \begin{bmatrix} \cos\beta & 0 & \sin\beta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\beta & 0 & \cos\beta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

$$\mathbf{R}_z = \begin{bmatrix} \cos\gamma & -\sin\gamma & 0 & 0 \\ \sin\gamma & \cos\gamma & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

When specified this way, positive angles correspond to counter-clockwise rotations. However, it is not trivial to decompose a general rotation into the angles α, β and γ, see [5].

Our result above is in accordance with Euler's Rotation Theorem which states that any rotation in $\mathbb{R}^3$ can be specified by three angles. The three rotations represented give us the three rotation matrices and the product of these three matrices specifies the overall rotation.

A useful observation about the three rotation matrices $\mathbf{R}_x$, $\mathbf{R}_y$ and $\mathbf{R}_z$ is that they are orthogonal, so for each of them, the columns are orthogonal and they have unit length. We can extract the angle, say α of the rotation about the x axis by observing

$$\text{trace}(\mathbf{R}_x) = 2 + 2\cos\alpha \Rightarrow \alpha = \arccos\frac{\text{trace}(\mathbf{R}_x) - 2}{2}$$

A nice way to regard a general rotation in a 3 dimensional space, is given by [45]; here, rotation is regarded as an operator which fixes a 1 dimensional subspace and rotates the corresponding two dimensional subspace. The line or vector $\mathbf{v}^0$ which stays fixed, is called the fixed axis of the rotation.

3.4.2 Translation

The translation operator translates, or moves, a point in space in the direction of some vector. The "size" of the displacement is equal to the length of the vector. Thus, translation is by nature an additive operator. The point $\mathbf{p} = [x, y, z]^T$ is translated by the vector $\mathbf{v} = [x_{disp}, y_{disp}, z_{disp}]^T$, to another point $\mathbf{p}'$ by the translation transformation: $\mathbf{p}' = \mathbf{p} + \mathbf{v} = [x + x_{disp}, y + y_{disp}, z + z_{disp}]^T$. However, by using homogeneous representation we make this transformation a *multiplicative* operator. What is more, we can concatenate, say a translation, a rotation and a scale operator in any order we want, to suit a particular application ([29]). This is convenient when we try to manipulate geometry on computers which today often have graphics boards with 4×4 matrix operations embedded in their circuitry.

We denote the translation matrix by

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & x_{disp} \\ 0 & 1 & 0 & y_{disp} \\ 0 & 0 & 1 & z_{disp} \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{3.1}$$

Observe that translating a vector $\mathbf{v}^0 = [v_x, v_y, v_z, 0]^T$, has no "meaning" as a vector only has an associated length and direction. Thus, when we use our translation matrix which now is size 4×4 due to the homogeneous representation, the product $\mathbf{T}\mathbf{v}^0$ should yield the same vector as a result:

$$\begin{bmatrix} 1 & 0 & 0 & x_{disp} \\ 0 & 1 & 0 & y_{disp} \\ 0 & 0 & 1 & z_{disp} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} v_x \\ v_y \\ v_z \\ 0 \end{bmatrix} = \begin{bmatrix} v_x + 0 * x_{disp} \\ v_y + 0 * y_{disp} \\ v_z + 0 * z_{disp} \\ 1 * 0 \end{bmatrix} = \begin{bmatrix} v_x \\ v_y \\ v_z \\ 0 \end{bmatrix}.$$

In contrast, $\mathbf{T}\mathbf{p}^1$ with $\mathbf{p}^1 = [p_x, p_y, p_z, 1]^T$ now being a point, gives us a new point:

$$\begin{bmatrix} p_x + x_{disp} \\ p_y + y_{disp} \\ p_z + z_{disp} \\ 1 \end{bmatrix}.$$

3.4.3 Scaling

The scaling operator scales an object along the three axes, relative to the origin of the frame of reference which stays fixed. We can choose to either enlarge or reduce the object in each of these directions. The scaling matrix $\mathbf{S}$ has the form

$$\mathbf{S} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

$s_x, s_y, s_z \in \mathbb{R}$.

3.4.4 Shearing

For completeness we we also describe the shear matrices. We can define six shear matrices as each coordinate x, y and z in $\mathbb{R}^3$ can by sheared by either of the other two. To illustrate this, we denote the six shear matrices by $\mathbf{H}_{xy}$, $\mathbf{H}_{xz}$, $\mathbf{H}_{yx}$, $\mathbf{H}_{yz}$, $\mathbf{H}_{zx}$ and $\mathbf{H}_{zy}$, as is done in [3]. As an example we look at the matrix $\mathbf{H}_{zy}(s)$, $s \in \mathbb{R}$, which denotes a shearing of the z-coordinate by the y coordinate:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & s & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Transformation of a point $\mathbf{p}^1 = [x, y, z, 1]^T$ through this shearing, would yield a point $\mathbf{r}^1$ such that the z coordinate of $\mathbf{r}^1$ is the sum of the z coordinate of $\mathbf{p}^1$ and s multiplied by the y-value of $\mathbf{p}^1$:

$$\mathbf{r}^1 = \mathbf{H}_{zy}(s)\mathbf{p}^1 = [x, y, z + sy, 1]^T.$$

We observe that the scalar argument s to the shear matrix $\mathbf{H}_{zy}$, is the $(3, 2)$ entry in $\mathbf{H}_{zy}$. If we denote the x, y and z coordinates by $1, 2$ and 3, respectively, this observation holds in general.

3.5 Quaternions

On Wikipedia [45], one can read the following excerpt about quaternions taken from the 1909 edition of Webster's Unabridged Dictionary:

> *"The quotient of two vectors ... Such is the view of the inventor, Sir Wm. Rowan Hamilton, and his disciple, Prof. P. G. Tait; but authorities are not yet quite agreed as to what a quaternion is or ought to be."*

Today, more than one hundred years after the "invention" of quaternions, it is agreed upon that a quaternion can be defined in the following way;

Definition 1 *A quaternion, denoted $\hat{\boldsymbol{q}}$ is a four-vector resulting from adding the imaginary components i, j and k to the real numbers such that $i^2 = j^2 = k^2 = ijk = -1$ and $ij = -ji = k$, $jk = -kj = i$ and $ki = -ik = j$. In other words $\hat{\boldsymbol{q}} = r + ai + bj + ck$, with $r, a, b, c \in \mathbb{R}$*

From this it follows that the basis quaternions are 1, i, j and k. This we see, is a natural generalization of the complex numbers which can be defined as adding the component i to the reals, resulting in a two-vector.

In the same way that complex numbers can be useful to us as they give a way of describing rotations in two dimensions, we will see that quaternions give a way of describing rotations in three dimensions. Indeed, the algebra $\mathbb{H}$ of quaternions is four dimensional and therefore it can model four dimensional rotations. But then $\mathbb{H}$ also includes all three dimensional operations as a subset. Before we proceed by looking at how such an operator might look like, we list some useful quaternion properties, some of which we will make use of later.

Definition 2 *Given two quaternions* $\hat{\boldsymbol{q}}_1 = [r_1, \boldsymbol{q}_1^T]$, $\boldsymbol{q}_1 = [a_1, b_1, c_1]^T$, *and* $\hat{\boldsymbol{q}}_2 = [r_2, \boldsymbol{q}_2^T]$, $\boldsymbol{q}_2 = [a_2, b_2, c_2]^T$, *we define the following:*

1. *The quaternion conjugate* $\hat{\boldsymbol{q}}_1^* = [r_1, -\boldsymbol{q}_1^T] = r_1 - a_1 i - b_1 j - c_1 k$.
2. *The Grassman product* $\hat{\boldsymbol{q}}_1 \hat{\boldsymbol{q}}_2 = r_1 r_2 - \boldsymbol{q}_1 \cdot \boldsymbol{q}_2 + r_1 \boldsymbol{q}_2 + r_2 \boldsymbol{q}_1 + (\boldsymbol{q}_1 \times \boldsymbol{q}_2)$. *This is the (non-commutative) quaternion product.*
3. *The norm of, say* $\hat{\boldsymbol{q}}_1$, $\mathrm{N}(\hat{\boldsymbol{q}}_1) = \sqrt{\hat{\boldsymbol{q}}_1 \hat{\boldsymbol{q}}_1^*}$.
4. *The quaternion dot-product* $\hat{\boldsymbol{q}}_1 \cdot \hat{\boldsymbol{q}}_2 = r_1 r_2 + a_1 a_2 + b_1 b_2 + c_1 c_2$.
5. *The quaternion cross product* $\hat{\boldsymbol{q}}_1 \times \hat{\boldsymbol{q}}_2 = (b_1 c_2 - c_1 b_2) i + (c_1 a_2 - a_1 c_2) j + (a_1 b_2 - b_1 a_2) k$. *This is also called the Grassman outer product.*

Horn [26] gives a nice way of expressing the Grassman product in terms of a real matrix with orthogonal columns and a quaternion. Using our notation from the definition above, $\hat{\mathbf{q}}_1 \hat{\mathbf{q}}_2$ becomes

$$\begin{bmatrix} r_1 & -a_1 & -b_1 & -c_1 \\ a_1 & r_1 & -c_1 & b_1 \\ b_1 & c_1 & r_1 & -a_1 \\ c_1 & -b_1 & a_1 & r_1 \end{bmatrix} \hat{\mathbf{q}}_2^T. \tag{3.2}$$

Equivalently, if we want to keep the four vector representation of $\hat{\mathbf{q}}_1$, the product $\hat{\mathbf{q}}_1 \hat{\mathbf{q}}_2$ can be written as

$$\begin{bmatrix} r_2 & -a_2 & -b_2 & -c_2 \\ a_2 & r_2 & c_2 & -b_2 \\ b_2 & -c_2 & r_2 & a_2 \\ c_2 & b_2 & -a_2 & r_2 \end{bmatrix} \hat{\mathbf{q}}_1^T. \tag{3.3}$$

These results can be seen by expanding the Grassman product given in the definition and collecting the corresponding terms of the product. We will make use of these expressions later.

Continuing our discussion, we are interested in arriving at an expression for how this four-vector entity can be used to describe the same operation as the 4×4 rotation matrices reviewed in section 3.4.1. We follow the

outline given in [29, 45]: In two dimensions, rotation of a vector or point $\mathbf{v} = [x, y]^T$ through an angle ϕ can be expressed as a simple product of the polar representation of $\mathbf{v}$, $re^{i\theta}$, and the rotation operator $e^{i\phi}$. That is, the point $\mathbf{v}'$ which we rotate towards can be written as

$$\mathbf{v}' = re^{i\theta}e^{i\phi} = re^{i(\theta+\phi)},$$

or in matrix form

$$\begin{bmatrix} \cos\phi & -\sin\phi \\ \sin\phi & \cos\phi \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x\cos\phi - y\sin\phi \\ x\sin\phi + y\cos\phi \end{bmatrix}.$$

This suggests approaching the search for the wanted expression simply by trying to multiply a vector $\mathbf{v} \in \mathbb{R}^3$ with a quaternion $\hat{\mathbf{q}}_1$, much the same way as we do when we use rotation matrices. For multiplication between a vector in $\mathbb{R}^3$ and a quaternion in $\mathbb{R}^4$ to make sense, we treat $\mathbf{v}$ as a quaternion whose real part is 0. That is, $\mathbf{v}$ is represented as the quaternion $\hat{\mathbf{p}}_\mathbf{v} = [0, \mathbf{v}^T]$. Such an entity is referred to as a *pure* (imaginary) quaternion [29, 45].

Using the definition of the quaternion product given above, we see that $\hat{\mathbf{q}}_1\hat{\mathbf{p}}_\mathbf{v}$, however, does not yield a vector as a result. That is, the real part of the resulting quaternion is not zero, at least as long as the dot product-part is not zero. If we change the order this will not help either, as this will not alter the real part of the product.

Continuing the search, if we now introduce another quaternion $\hat{\mathbf{q}}_2$ into the equation, we have $3! = 6$ possibilities for combining the three factors $\hat{\mathbf{q}}_1$, $\hat{\mathbf{p}}_\mathbf{v}$ and $\hat{\mathbf{q}}_2$ into a product. Four of these combinations, however, simplifies to the case with one quaternion and one pure quaternion — this we know will not work. So we need to keep the two quaternions apart, or in other words, we need to keep the pure quaternion in the middle. This gives us two possibilities, $\hat{\mathbf{q}}_1\hat{\mathbf{p}}_\mathbf{v}\hat{\mathbf{q}}_2$ and $\hat{\mathbf{q}}_2\hat{\mathbf{p}}_\mathbf{v}\hat{\mathbf{q}}_1$. We make no distinction between the two quaternions $\hat{\mathbf{q}}_1$ and $\hat{\mathbf{q}}_2$, so we are left with only one possibility.

We now have two quaternions $\hat{\mathbf{q}}_1 = r_1 + \mathbf{q}_1^T$, $\hat{\mathbf{q}}_2 = r_2 + \mathbf{q}_2^T$ and the pure quaternion $\hat{\mathbf{p}}_\mathbf{v} = 0 + \mathbf{v}^T$. The real part of $\hat{\mathbf{q}}_1\hat{\mathbf{p}}_\mathbf{v}\hat{\mathbf{q}}_2$ is

$$-r_2(\mathbf{q}_1 \cdot \mathbf{v}) - r_1(\mathbf{v} \cdot \mathbf{q}_2) - (\mathbf{q}_1 \times \mathbf{v}) \cdot \mathbf{q}_2,$$

which we want to be zero. As is shown by [29], this can be achieved through two simple steps. First we let the real parts of $\hat{\mathbf{q}}_1$ and $\hat{\mathbf{q}}_2$ be equal. We may then rewrite the expression, so that it reads

$$-r_1(\mathbf{q}_1 + \mathbf{q}_2) \cdot \mathbf{v} + (\mathbf{q}_1 \times \mathbf{q}_2) \cdot \mathbf{v}.$$

Thus the second step is obviously to set $\mathbf{q}_2$ equal to $-\mathbf{q}_1$. This follows since then the sum in the first parenthesis will be zero, and $\mathbf{q}_2$ will be parallel to $\mathbf{q}_1$ so $\mathbf{q}_1 \times \mathbf{q}_2$ will be the zero vector.

This implies that $\hat{\mathbf{q}}_2 = r_2 + \mathbf{q}_2^T = r_1 - \mathbf{q}_1^T = \hat{\mathbf{q}}_1^*$. So the triple quaternion product

$$\hat{\mathbf{q}}_1 \hat{\mathbf{p}}_\mathbf{v} \hat{\mathbf{q}}_1^* \tag{3.4}$$

is an operator which takes a vector in $\mathbb{R}^3$, into a vector in $\mathbb{R}^3$.

To associate an *angle* θ with a quaternion, like we did explicitly with a rotation matrix, Kuipers [29] makes use of *unit* quaternions. The norm of such quaternions is equal to one, which implies that the Grassman product of the quaternion and its conjugate is also equal to one. In other words, if $\hat{\mathbf{q}}_u = [r_u, \mathbf{q}_u^T]$ is a unit quaternion then $\hat{\mathbf{q}}_u \hat{\mathbf{q}}_u^* = 1$, or

$$\begin{aligned}
\hat{\mathbf{q}}_u \hat{\mathbf{q}}_u^* &= r_u^2 + \mathbf{q}_u \cdot \mathbf{q}_u - r_u \mathbf{q}_u + r_u \mathbf{q}_u - \mathbf{q}_u \times \mathbf{q}_u \\
&= r_u^2 + \mathbf{q}_u \cdot \mathbf{q}_u + \mathbf{0} \\
&= r_u^2 + \mathbf{q}_u \cdot \mathbf{q}_u \\
&= 1.
\end{aligned}$$

We know from trigonometry that

$$\cos^2\theta + \sin^2\theta = 1$$

so we should have

$$\begin{aligned}
\cos^2\theta &= r_u^2 \\
\sin^2\theta &= \mathbf{q}_u \cdot \mathbf{q}_u.
\end{aligned}$$

As pointed out by [29], θ can be determined uniquely if we demand that $-\pi < \theta < \pi$. Now, for a unit vector $\mathbf{u}$,

$$\mathbf{u} = \frac{\mathbf{q}_u}{\sqrt{\mathbf{q}_u \cdot \mathbf{q}_u}},$$

we can write $\hat{\mathbf{q}}_u$ as $r_u + \mathbf{q}_u^T = \cos\theta + \mathbf{u}^T \sqrt{\mathbf{q}_u \cdot \mathbf{q}_u} = \cos\theta + \mathbf{u}^T \sin\theta$. Thus, by using unit quaternions, we can add to our knowledge about the work of the triple quaternion product $\hat{\mathbf{q}}_u \hat{\mathbf{p}}_\mathbf{v} \hat{\mathbf{q}}_u^*$, what is summarized in the following theorem. Proof can be found in Kuipers [29].

Theorem 1 *Let $\boldsymbol{u}$ be a unit vector and θ an angle such that $-\pi < \theta < \pi$. Then for any unit quaternion $\hat{\boldsymbol{q}}_u = \cos\theta + \boldsymbol{u}^T \sin\theta$ and for any vector $\boldsymbol{v} \in \mathbb{R}^3$, $\boldsymbol{v} = [v_x, v_y, v_z]^T$, the action of the operator*

$$\hat{\boldsymbol{q}}_u \hat{\boldsymbol{p}}_\boldsymbol{v} \hat{\boldsymbol{q}}_u^* \tag{3.5}$$

on $\hat{\boldsymbol{p}}_\boldsymbol{v}$ where $\hat{\boldsymbol{p}}_\boldsymbol{v} = [0, \boldsymbol{v}^T]$, may be interpreted geometrically as a rotation of the vector $\boldsymbol{v}$ through an angle 2θ about $\boldsymbol{u}$ as the axis of rotation.

Further, a sequence of rotations corresponds to the multiplication of quaternions [26]: Consider the rotation $\hat{\mathbf{q}}_{u_1} \hat{\mathbf{p}} \hat{\mathbf{q}}_{u_1}^*$. If we apply a second rotation represented by $\hat{\mathbf{q}}_{u_2}$ we get

$$\hat{\mathbf{q}}_{u_2} (\hat{\mathbf{q}}_{u_1} \hat{\mathbf{p}} \hat{\mathbf{q}}_{u_1}^*) \hat{\mathbf{q}}_{u_2}^* = (\hat{\mathbf{q}}_{u_2} \hat{\mathbf{q}}_{u_1}) \hat{\mathbf{p}} (\hat{\mathbf{q}}_{u_2} \hat{\mathbf{q}}_{u_1})^*.$$

So in the case of two rotations, the overall rotation is represented by the quaternion $\hat{\mathbf{q}}_{u_2}\hat{\mathbf{q}}_{u_1}$

Quaternions may challenge our intuition; quaternion algebra tells us that quaternion multiplication does not commute. That is, given two quaternions $\hat{\mathbf{q}}$ and $\hat{\mathbf{p}}$, $\hat{\mathbf{q}}\hat{\mathbf{p}} \neq \hat{\mathbf{p}}\hat{\mathbf{q}}$. We now know, however, that unit quaternions can be used to represent spatial rotations. So the fact that they do not commute can be seen to reflect the fact that a composition of two rotations about two distinct axes, also does not commute.

3.5.1 Representation and conversions

In numerical algorithms small errors often accumulate due to the nature of the floating point representation. An advantage that comes with quaternion representation, as pointed out by [45], is that quaternions still represent a rotation after being normalized. Contrary to this, a matrix that is slightly off needs not be orthogonal anymore and is therefore harder to convert back to a proper orthogonal matrix.

However, compact and stable as a quaternion representation may be, many computers today have built-in dedicated procedures for handling matrix computations. With this in mind, it would be convenient to have at hand a matrix representation for quaternions. Fortunately, such a representation exists.

Consider a unit quaternion $\hat{\mathbf{q}}_u = [r, a, b, c]$ and a point $\mathbf{v}$ represented by a pure quaternion $\hat{\mathbf{p}}_\mathbf{v} = 0 + \mathbf{v}^T$. Using the matrix representations (3.2) and (3.3) we can write the rotational operator $\hat{\mathbf{q}}_u\hat{\mathbf{p}}_\mathbf{v}\hat{\mathbf{q}}_u^*$ as

$$(\mathbf{Q}\hat{\mathbf{p}}_\mathbf{v})\hat{\mathbf{q}}_u^* = \mathbf{Q}^T\mathbf{Q}\hat{\mathbf{p}}_\mathbf{v},$$

where

$$\mathbf{Q} = \begin{bmatrix} r & -a & -b & -c \\ a & r & -c & b \\ b & c & r & -a \\ c & -b & a & r \end{bmatrix}$$

is a 4×4 matrix representation of $\hat{\mathbf{q}}_u$. The lower right 3×3 orthogonal submatrix of $\mathbf{Q}^T\mathbf{Q}$,

$$\begin{bmatrix} r^2 + a^2 - b^2 - c^2 & 2(ab - rc) & 2(ac + rb) \\ 2(ab + rc) & r^2 - a^2 + b^2 - c^2 & 2(bc - ra) \\ 2(ac - rb) & 2(bc + ra) & r^2 - a^2 - b^2 + c^2 \end{bmatrix},$$

is the usual rotation matrix $\mathbf{R}$ which rotates the point $\mathbf{v}$ into $\mathbf{v}'$, i.e. $\mathbf{v}' = \mathbf{R}\mathbf{v}$ [26].

Another conversion often needed, is computing a unit quaternion based on the knowledge of the unit rotation axis $\mathbf{v} = [v_x, v_y, v_z]$ and rotational

angle ϕ. The unit quaternion $\hat{\mathbf{q}}_u$ is then found as

$$\hat{\mathbf{q}}_u = \begin{bmatrix} \cos\frac{\phi}{2} \\ v_x \sin\frac{\phi}{2} \\ v_y \sin\frac{\phi}{2} \\ v_z \sin\frac{\phi}{2} \end{bmatrix}.$$

3.6 Centroid

For future computations the centroid will prove to be a useful quantity. According to [45] the centroid is defined in the following way:

Definition 3 *The centroid or barycenter of an object X in an n-dimensional space is the intersection of all hyperplanes that divide X into two parts of equal moment about the hyperplane.*

If now $X = \{\mathbf{x}_1^1, \mathbf{x}_2^1, ..., \mathbf{x}_{n_X}^1\}$ is a set of points, the informal definition of the centroid $\mathbf{c}_X^1$ of X as being the average of all points of X, will suffice for our purpose. This gives us

$$\mathbf{c}_X^1 = \frac{1}{n_X} \sum_{i=1}^{n_X} \mathbf{x}_i^1$$

where n_X denotes the cardinality of set X.

3.7 Curvature and related concepts

According to [45], the curvature κ of a point $\mathbf{p}$ on a curve in $\mathbb{R}^2$ equals in magnitude, "the multiplicative inverse of the radius of a circle which closely touches the curve at $\mathbf{p}$". For a point $\Lambda(u)$ on a parametrized space curve $\Lambda : \mathbb{R} \to \mathbb{R}^3$, this leads to the expression

$$\kappa = \frac{||\Lambda'(u) \times \Lambda''(u)||}{||\Lambda'(u)||^3},$$

where $\times$ denotes the usual vector product.

3.7.1 The Frénet frame

In terms of the same parametrized space curve Λ introduced above, we define the Frénet frame at a point $\Lambda(u)$ on the curve as the three orthonormal vectors

$$\mathbf{t}(u) = \frac{\Lambda'(u)}{||\Lambda'(u)||} \qquad \text{(the tangent vector)}$$

$$\mathbf{n}(u) = \frac{\mathbf{t}'(u)}{||\mathbf{t}'(u)||} \qquad \text{(the normal vector)}$$

$$\mathbf{b}(u) = \mathbf{t}(u) \times \mathbf{n}(u) \qquad \text{(the bi normal)}.$$

We will make use of this frame later in the text.

3.7.2 The Principal frame

At a point on a given surface, say the image of the function $\Phi : \mathbb{R}^2 \rightarrow \mathbb{R}^3$, we also have what is called the principal frame or surface frame. The principal frame is made up of the three orthonormal vectors $\mathbf{n}$, $\mathbf{e}_1$ and $\mathbf{e}_2$ which are the surface normal and the two principal directions, respectively. The principal directions are the directions of the two tangent vectors associated with the two (space) curves on the surface, with maximum and minimum curvature passing through the given point. The maximum and minimum curvature we denote κ_{max} and κ_{min}. Also, in both directions $\mathbf{e}_1$ and $\mathbf{e}_2$ we have a principal radius of curvature; r_1 and r_2, respectively.

Consider a point $\mathbf{p}$ not on the surface Φ but in proximity to Φ. The point $\mathbf{p}$ has what we will call a normal *footpoint* [33] on the surface. The normal footpoint is a point $\mathbf{y}$ on the surface Φ such that

$$d(\mathbf{p}, \mathbf{y}) = \min d(\mathbf{p}, \Phi). \tag{3.6}$$

The function $d(\cdot, \cdot)$ denotes the distance function of the surface Φ. In other words, the normal footpoint is the closest point on the surface from $\mathbf{p}$.

For the identity (3.6) to be useful, we would need to specify the form of the surface. If we knew that Φ was a triangulated surface, i.e. was made up of a set T of n triangles, then the normal footpoint $\mathbf{y}$ would satisfy

$$d(\mathbf{p}, \mathbf{y}) = d(\mathbf{p}, T) = \min_{i \in \{1,\ldots,n\}} d(\mathbf{p}, \mathbf{T}_i)$$

for a triangle $\mathbf{T}_i \in T$. Further, if a triangle $\mathbf{T}_i$ in T, $i \in [1, 2, \ldots, n]$, is thought of in terms of its three cornerpoints, $[\mathbf{v}_{i,1}, \mathbf{v}_{i,2}, \mathbf{v}_{i,3}]$, then

$$d(\mathbf{p}, \mathbf{T}_i) = ||\mathbf{v}_{i,1} + s(\mathbf{v}_{i,2} - \mathbf{v}_{i,1}) + t(\mathbf{v}_{i,3} - \mathbf{v}_{i,1}) - \mathbf{p}||$$

for weights $s, t \in [0, 1]$, $s + t \leq 1$, see [15].

In chapter 8, we will refer to the normal *footprint* of $\mathbf{p}$ on Φ. If S denotes a decomposition of the surface Φ into points, then the normal footprint is the closest point in S to $\mathbf{y}$ — the best approximation of $\mathbf{y}$. If Φ is a triangulated surface, the normal footprint is a corner point $\mathbf{v}_{i,j}$ in the mesh, $i \in [1, 2, \ldots, n]$ and $j \in [1, 2, 3]$.

If the point $\mathbf{p}$ and the centers of the osculating circles at $\mathbf{v}_{i,j}$ are on the same side of the surface, the signed distance $d(\mathbf{p}, \mathbf{v}_{i,j})$ is positive.

3.8 Summary

We summarize the notation which we will be using from this point on, in table 3.1 on the next page.

scalars	lowercase italic letters
angles	lowercase Greek letters
vectors	lowercase boldface letters
quaternions	lowercase boldface letters with a hat
matrices	uppercase boldface letters
sets	uppercase italic letters
transformations	uppercase Greek letters
vector product	$\times$
dot product	$\cdot$

Table 3.1: Summary of the notation used in this text.

In the next chapter, we will formulate a problem which is of great importance when the goal is to retrieve a full 3D representation of a scanned object; we hope to end up with a useful formulation of *the registration problem*. Preferably, the discussion will enable us to grasp more easily, the ideas behind some of the solutions which are presented in the subsequent chapters. As a wise person once said: "A problem well stated is a problem half solved".

Part II

The Problem

Chapter 4

The Non-Elastic Registration Problem

"The definition of a good mathematical problem is the mathematics it generates rather than the problem itself."

Andrew Wiles

After having settled on a notation for some fundamental entities and given an overview of some useful topics, we are ready to discuss the significant shape-acquisitioning problem known as *the registration problem*. As argued earlier, registration should be understood as the alignment of two or more sets of representations of geometric data [8]. It was also stated there that there are a menagerie of solutions, i.e. registration methods, to this problem so naturally the problem are formulated in many different ways. Consequently, we are forced to make some initial assumptions if we want to state such a formulation.

When we have carried out a scan, clouds of points are usually what is given us so we will focus on point cloud registration. Hence we will seek a precise formulation of the problem of aligning sets of point cloud data in three dimensions. Solutions to the registration problem can be generalized to solve the case of m point sets, $m > 2$, but there are also methods which are optimized for such cases where $m > 2$, see [12]. We will concentrate on registration of only two sets of point cloud data. Moreover, as we scan rigid objects, we will not allow for deformations. The transformation model will thus be affine. In our discussion we follow the lines given in [8, 19, 17, 26].

4.1 A search for a precise formulation

As stated, we assume that two sets of points in $\mathbb{R}^3$ are given, $P = \{\mathbf{p}_i\}_{i=1}^{n_P}$ and $S = \{\mathbf{s}_j\}_{j=1}^{n_S}$, (here n_P and n_S denote the cardinality of the set P and

S, respectively). Each of the sets of points is given in its own Cartesian coordinate system, and we want to find a transformation between these two systems so that the points are aligned. We will think of this transformation as an affine transformation, composed of a rotation, translation and a uniform scaling.

As pointed out by [26], there are three degrees of freedom to both translation and rotation[1] and one degree of freedom to uniform scaling. This gives us a total of seven unknows since the translational and rotational movements are independent of each other and also independent of the scaling. Consequently, we need seven equations to determine the transformation parameters — at least intuitively.

If we know the coordinates of three points in P and also their coordinates in S then we can set up nine equations; each pair of measurements (point pairs) gives us three equations, for a total of nine. We could thus disregard two of the equations and determine the seven unknowns by solving the remaining seven equations.

However, as measurements are not exact, we follow in the footsteps of [26, 33, 17] among others, and hope to determine the seven unknows more accurately by regarding more points. This way we admit that we probably will not be able to find an exact mapping — we will try to minimize the sum of the squared residuals between each pair of measurements. This procedure is a well known mathematical optimization technique [45]. We will also make up for possible outliers in the data by introducing weights, w_js. The w_js will be set to a value between zero and one reflecting the degree of reliability which we associate with the measurement in question.

Using homogeneous coordinates, we are looking for an affine transformation $\mathbf{M}$ on the form

$$\mathbf{s}_j^1 = \mathbf{M}\mathbf{p}_j^1. \tag{4.1}$$

Here, $\mathbf{M}$ is the matrix representation of the whole transformation. In our context the geometric data have come from 3D scanning of some sort, and thus the objects we consider are mostly rigid — we do not expect them to undergo deformations such as, say, shearing. Consequently, the matrix $\mathbf{M}$ can most often be decomposed into a translation $\mathbf{T}$, a uniform scaling $\mathbf{S}$ and a rotation $\mathbf{R}$.

As stated, equation (4.1) will not be exact for each j, and the residual errors ϵ_j^0 are

$$\epsilon_j^0 = \mathbf{s}_j^1 - \mathbf{M}\mathbf{p}_j^1, \quad j = 1, \ldots, n_P. \tag{4.2}$$

Thus, to sum up what we have said so far, we seek to minimize the sum

$$\sum w_j \left|\left|\epsilon_j^0\right|\right|^2$$

of squared distances between the points $\mathbf{s}_j^1$ and $\mathbf{M}\mathbf{p}_j^1$ by varying $\mathbf{M}$.

[1] If we regard a rotation about an axis through the origin of the frame of reference.

How do we know what the points $\mathbf{s}_j^1$ and $\mathbf{p}_j^1$ are? Often in point cloud registration algorithms, the points which we seek to align are referred to as corresponding points. Hence, point correspondences are something that needs to be calculated in the initial stages of most registration algorithms. A nice way to think of the point $\mathbf{s}_j^1$ corresponding to $\mathbf{p}_j^1$ is as the output of a function

$$\Gamma : P \to S,$$

applied to each $\mathbf{p}_j^1$. We have already described an easy way of finding the output of Γ in section 2.3.3; there the corresponding points were simply picked out manually using The Polygon Editing Tool. In chapter 8 we will see an example of how corresponding points can be found by applying a more complex routine.

4.1.1 Calling off the search

We want to find the minimum over $\mathbf{M}$ in the transformation applied to each $\mathbf{p}_j^1$. Our complete registration problem can thus be expressed as

$$\min_{\mathbf{M}} \sum_{j=1}^{n_P} w_j \left|\left| \Gamma(\mathbf{p}_j^1) - \mathbf{M}\mathbf{p}_j^1 \right|\right|^2. \tag{4.3}$$

The point $\Gamma(\mathbf{p}_j^1)$ in S corresponding to $\mathbf{p}_j^1$ is found using a suitable method. According to [19], the Euclidean norm is the most commonly used measure of the deviation error in registration problems.

A note should be taken at this point as we have stated the expression for point cloud data. If the point clouds come from a scan, the points are not a random collection, they are collected off an underlying surface. In this case, as pointed out by [33], a point-to-plane error metric is more suitable as it is shown to converge much faster. However, when the two data sets are far apart, the point-to-plane error metric fails to converge and the point-to-point metric is preferred. As the point-to-point error metric works reasonably well in both cases, that is the one we have used in our formulation.

The effect of registration can be seen in figure 4.1 on the following page. The left picture shows several scans prior to registration while the picture to the right displays the same scans after they have been registered in the Polygon Editing Tool as described in section 2.3. The points from each scan have been connected by line segments to generate a mesh and then the surface is shaded to enhance the visibility.

4.2 Several different approaches

The optimal transformation parameters can be found from both graphical and numerical procedures. These are *iterative* in nature [26]. That is, an

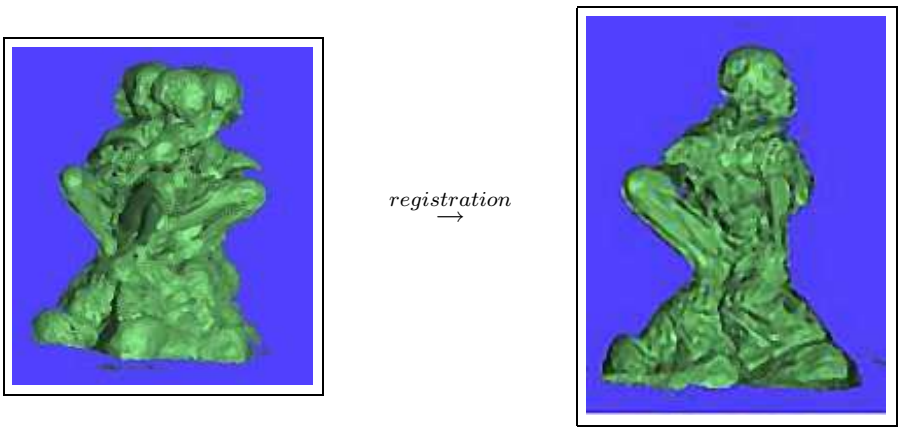


Figure 4.1: The figure to the left displays several scans as they appear in the Polygon Editing Tool immediately after scanning. The figure to the right displays the same scans after they have been subjects to both an initial manual pairwise registration and fine automatic registration.

approximate solution is given as input and then the procedures refine the approximation, step by step.

Registration methods are also often categorized as being either feature based or area based [45]. Feature based registration implies finding a subset of "characteristic" points, referred to as "features", in one image, and trying to track down for each of the features, the same type of points in another image. The transformation which is sought after is then found as the one which aligns the features. We will review a feature based method in chapter 8.

Area based registration implies alignment of shapes by some structural analysis. As an example, the method of chapter 7 performs registration by minimizing the distance between a point cloud and the *surface* from which the point cloud is sampled.

In [19], numerical registration methods are classified according to how they compute the transformation parameters: Some methods exploit the fact that there are only a finite number of parameters which have to be set, in order to define the optimal alignment. Such methods search exhaustively for these parameters and consequently they are classified as "search based" methods. We will review an example of a search based method in chapter 6. Otherwise, registration methods often compute the transformation parameters using a least squares approach and these methods are sometimes referred to as "direct" methods [45].

There have also been stated closed form solutions to the least squares problem of recovering the transformation between two distinct coordinate systems. One of these constitutes a natural follow-up of our discussion above, and we will therefore take a closer look at it in the following chapter.

Part III

Registration Methods

Chapter 5

A Closed-form Solution

To be able to retrieve a closed form solution of the non-elastic registration problem, we need to specify the transformation model: We seek the translation **T**, rotation **R** and uniform scaling **S** so that the sum

$$\sum_{j=1}^{n_P} w_j \left|\left|\Gamma(\mathbf{p}_j^1) - \mathbf{TSR}\mathbf{p}_j^1\right|\right|^2 \tag{5.1}$$

is minimized, by varying **S**, **R** and **T**. Thus, we allow for translation, uniform scaling and rotation.

According to [40], several closed form solutions of this problem exist; at least four use unit quaternions, three use the singular value decomposition, one makes use of the Polar decomposition and yet another utilizes dual quaternions. It is noted that the solutions are all mathematically identical. A study [32] of the accuracy of the solutions concludes that the difference in accuracy between the methods is insignificant compared to machine precision [40].

5.1 Preparations

Guided by the results of [26] which offers one of the four solutions using unit quaternions, we are going to see how a closed form solution of (5.1) above can be retrieved. The nature of a closed form solution implies that we must know a priori what the corresponding points are. Thus the work of a function Γ must have been done in advance, and we can assume that the point sets are ordered such that $\mathbf{p}_1^1$ is to be aligned with $\mathbf{s}_1^1$, $\mathbf{p}_2^1$ with $\mathbf{s}_2^1$ and so on. We can therefore reformulate (5.1) slightly, as

$$\min_{\mathbf{R},\mathbf{T},\mathbf{S}} \sum_{j=1}^{n_P} w_j \left|\left|\mathbf{s}_j^1 - \mathbf{TSR}\mathbf{p}_j^1\right|\right|^2.$$

It will prove useful to regard the quaternion representation of the rotation **R** (see section 3.5.1 on page 41).

For further computations we need the centroids $\mathbf{c}_P^1$ and $\mathbf{c}_S^1$ of the two point sets. As we recall, these are just the means of the two point sets. Since we have utilized weights, we need the weighted centroids,

$$\dot{\mathbf{c}}_P^1 = \frac{\sum_{i=1}^{n_P} w_i \mathbf{p}_i^1}{\sum_{i=1}^{n_P} w_i} \quad \text{and} \quad \dot{\mathbf{c}}_S^1 = \frac{\sum_{i=1}^{n_S} w_i \mathbf{s}_i^1}{\sum_{i=1}^{n_S} w_i}.$$

We now choose to move the origin in both sets to the weighted centroids of the sets:

$$\mathbf{p}_i^0 = \mathbf{p}_i^1 - \dot{\mathbf{c}}_P^1, \qquad i = 1, ..., n_P$$

and

$$\mathbf{s}_i^0 = \mathbf{s}_i^1 - \dot{\mathbf{c}}_S^1, \qquad i = 1, ..., n_S.$$

This translation requires a reconsideration of the residual error (4.2),

$$\epsilon_i^0 = \mathbf{s}_i^1 - \mathbf{TSR}\mathbf{p}_i^1, \quad i = 1, \ldots, n_P.$$

According to [26], the residuals can now be rewritten as

$$\begin{aligned} \epsilon_i(\mathbf{R}, \mathbf{T}, \mathbf{S}) &= \left(\mathbf{s}_i^0 - \mathbf{TSR}\mathbf{p}_i^0\right) + \left(\dot{\mathbf{c}}_S^1 - \mathbf{SR}\dot{\mathbf{c}}_P^1\right) \\ &= \left(\mathbf{s}_i^0 - \mathbf{SR}\mathbf{p}_i^0\right) + \left(\dot{\mathbf{c}}_S^1 - \mathbf{TSR}\dot{\mathbf{c}}_P^1\right) \\ &= \left(\mathbf{s}_i^0 - \mathbf{SR}\mathbf{p}_i^0\right) - \left(\mathbf{TSR}\dot{\mathbf{c}}_P^1 - \dot{\mathbf{c}}_S^1\right). \end{aligned}$$

We note that the index i runs from 1 to $n_P = n_S = n$. Using the usual Euclidean norm $||\cdot||_2$, the expression 5.1 on the preceding page becomes

$$\min_{\mathbf{R},\mathbf{T},\mathbf{S}} \sum_{i=1}^{n} w_i \left|\left|\epsilon_i(\mathbf{R}, \mathbf{T}, \mathbf{S})\right|\right|_2^2.$$

5.2 Determining the translation

We will first reveal what the translation must be. To simplify the expansion of the last expression, we set $\mathbf{v}_i^0 = \mathbf{s}_i^0 - \mathbf{SR}\mathbf{p}_i^0$ and $\mathbf{u}^0 = \mathbf{TSR}\dot{\mathbf{c}}_P^1 - \dot{\mathbf{c}}_S^1$. From

this we get

$$\begin{aligned}
\sum_{i=1}^{n} w_i \left|\left|\epsilon_i(\mathbf{R},\mathbf{T},\mathbf{S})\right|\right|_2^2 &= \sum_{i=1}^{n} w_i \left|\left|\mathbf{v}_i^0 - \mathbf{u}^0\right|\right|_2^2 \\
&= \sum_{i=1}^{n} w_i (\mathbf{v}_i^0 - \mathbf{u}^0)\cdot(\mathbf{v}_i^0 - \mathbf{u}^0) \\
&= \sum_{i=1}^{n} w_i \left(\mathbf{v}_i^0\cdot\mathbf{v}_i^0 - 2\mathbf{v}_i^0\cdot\mathbf{u}^0 + \mathbf{u}^0\cdot\mathbf{u}^0\right) \\
&= \sum_{i=1}^{n} w_i \left|\left|\mathbf{v}_i^0\right|\right|_2^2 - 2\sum_{i=1}^{n} w_i\mathbf{v}_i^0\cdot\mathbf{u}^0 + \sum_{i=1}^{n} w_i \left|\left|\mathbf{u}^0\right|\right| \\
&= \sum_{i=1}^{n} w_i \left|\left|\mathbf{v}_i^0\right|\right|_2^2 - 2\mathbf{u}^0\cdot\sum_{i=1}^{n} w_i\mathbf{v}_i^0 + \left|\left|\mathbf{u}^0\right|\right|_2^2 \sum_{i=1}^{n} w_i.
\end{aligned}$$

If we substitute back again, we see that what we must minimize is

$$\sum_{i=1}^{n} w_i \left|\left|\mathbf{s}_i^0 - \mathbf{SRp}_i^0\right|\right|_2^2 - 2(\mathbf{TSR}\dot{\mathbf{c}}_P^1 - \dot{\mathbf{c}}_S^1)\cdot\sum_{i=1}^{n} w_i(\mathbf{s}_i^0 - \mathbf{SRp}_i^0) + \left|\left|\mathbf{TSR}\dot{\mathbf{c}}_P^1 - \dot{\mathbf{c}}_S^1\right|\right|_2^2 \sum_{i=1}^{n} w_i.$$

Since the $\mathbf{s}_i^0$'s are related to the centroid $\dot{\mathbf{c}}_S^1$ they sum up to zero;

$$\begin{aligned}
\sum_{i=1}^{n} w_i\mathbf{s}_i^0 &= \sum_{i=1}^{n} w_i(\mathbf{s}_i^1 - \dot{\mathbf{c}}_S^1) \\
&= \sum_{i=1}^{n} w_i\mathbf{s}_i^1 - \sum_{i=1}^{n} w_i \left(\frac{\sum_{i=1}^{n_S} w_i\mathbf{s}_i^1}{\sum_{i=1}^{n_S} w_i}\right) \\
&= \sum_{i=1}^{n} w_i\mathbf{s}_i^1 - \sum_{i=1}^{n} w_i\mathbf{s}_i^1 \\
&= 0.
\end{aligned}$$

This also holds for the scaled and rotated $\mathbf{p}_i^0$'s so the whole sum in the middle is zero. Further, both of the remaining terms are nonnegative. And as we in general do not have a perfect match, we can not require that the first term be zero. We also observe that the first term does not depend on the translation $\mathbf{T}$. Hence we can minimize the total error by setting

$$\mathbf{TSR}\dot{\mathbf{c}}_P^1 - \dot{\mathbf{c}}_S^1 = 0,$$

and at the same time we reveal what the translation must be: The difference of the weighted centroid of S and the rotated and scaled, weighted centroid of P. This becomes more clear if we rearrange the terms,

$$\dot{\mathbf{c}}_S^1 = \mathbf{TSR}\dot{\mathbf{c}}_P^1.$$

5.3 Determining the uniform scale

It remains to determine the scale and rotation, and from the discussion above, our formulation of the problem can now be reduced to

$$\min \sum_{i=1}^{n} w_i \left|\left|\mathbf{s}_i^0 - \mathbf{SR}\mathbf{p}_i^0\right|\right|_2^2.$$

This expression is in accordance with the one of Horn's [26]. Again, if the error term is expanded we get

$$\begin{aligned}
\sum_{i=1}^{n} w_i \left|\left|\mathbf{s}_i^0 - \mathbf{SR}\mathbf{p}_i^0\right|\right|_2^2 &= \sum_{i=1}^{n} w_i \left|\left|\mathbf{s}_i^0\right|\right|_2^2 - 2\sum_{i=1}^{n} w_i \mathbf{s}_i^0 \cdot \mathbf{SR}\mathbf{p}_i^0 + \sum_{i=1}^{n} w_i \left|\left|\mathbf{SR}\mathbf{p}_i^0\right|\right|_2^2 \\
&= \sum_{i=1}^{n} w_i \left|\left|\mathbf{s}_i^0\right|\right|_2^2 - 2s\sum_{i=1}^{n} w_i \mathbf{s}_i^0 \cdot \mathbf{R}\mathbf{p}_i^0 + \sum_{i=1}^{n} w_i \left|\left|\mathbf{S}\mathbf{p}_i^0\right|\right|_2^2 \\
&= \sum_{i=1}^{n} w_i \left|\left|\mathbf{s}_i^0\right|\right|_2^2 - 2s\sum_{i=1}^{n} w_i \mathbf{s}_i^0 \cdot \mathbf{R}\mathbf{p}_i^0 + \sum_{i=1}^{n} w_i (s\mathbf{p}_i^0) \cdot (s\mathbf{p}_i^0) \\
&= \sum_{i=1}^{n} w_i \left|\left|\mathbf{s}_i^0\right|\right|_2^2 - 2s\sum_{i=1}^{n} w_i \mathbf{s}_i^0 \cdot \mathbf{R}\mathbf{p}_i^0 + s^2\sum_{i=1}^{n} w_i \left|\left|\mathbf{p}_i^0\right|\right|^2
\end{aligned} \tag{5.2}$$

since the scale is uniform, i.e.

$$\mathbf{S} = \begin{bmatrix} s & 0 & 0 & 0 \\ 0 & s & 0 & 0 \\ 0 & 0 & s & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad s \in \mathbb{R}$$

and since **R** is orthogonal, thus keeping the 2-norm of a vector unchanged.

The expression (5.2) which we want to minimize is easier handled if we make use of the fact that the three sums are scalars, and thus denote them as such, say a, b and c, respectively. Using a slightly different approach than [26], we must find the minimum of

$$a - 2sb + s^2 c,$$

which can be regarded as a function Θ depending on s. The minimum is found by solving the equation

$$\frac{\partial}{\partial s}\Theta(s) = 2sc - 2b = 0.$$

Thus the function Θ finds its minimum when

$$s = \frac{b}{c},$$

that is, when

$$s = \frac{\sum_{i=1}^{n} w_i \mathbf{s}_i^0 \cdot \mathbf{R}\mathbf{p}_i^0}{\sum_{i=1}^{n} w_i \left|\left|\mathbf{p}_i^0\right|\right|^2}.$$

We note that we need to know the rotation, in order to compute the scale.

5.4 Determining the rotation

For determining the rotation, consider again the total error (5.2). This is minimized if the sum in the middle is as large as possible since that sum is subtracted from the rest of the expression, which are all positive terms. In other words, the rotation must be chosen so that it maximizes

$$\sum_{i=1}^{n} w_i \mathbf{s}_i^0 \cdot \mathbf{R}\mathbf{p}_i^0.$$

We said earlier that the rotation $\mathbf{R}$ is easier handled if we regard its quaternion representation. Having a quaternion representing the rotation comes in handy as quaternion representation has several advantages, see section 3.5.1. Recall that we can represent the complete rotation of a point $\mathbf{p}_i$ in terms of a unit quaternion $\hat{\mathbf{q}}_u$ and that the rotational operator reads $\hat{\mathbf{q}}_u \hat{\mathbf{p}}_{\mathbf{p}_i} \hat{\mathbf{q}}_u^*$, where $\hat{\mathbf{p}}_{\mathbf{p}_i} = [0, p_{x,i}, p_{y,i}, p_{z,i}]$ is the pure quaternion representing $\mathbf{p}_i = [p_{x,i}, p_{y,i}, p_{z,i}]$.

Now we let $\hat{\mathbf{q}}_u$ represent the rotation of our vector $\mathbf{p}_i^0$, and let $\hat{\mathbf{p}}_{\mathbf{p}_i}$ be its pure quaternion representation. Also we obtain a pure quaternion representation $\hat{\mathbf{p}}_{\mathbf{s}_i}$ of $\mathbf{s}_i = [s_{x,i}, s_{y,i}, s_{z,i}]$, i.e. $\hat{\mathbf{p}}_{\mathbf{s}_i} = [0, s_{x,i}, s_{y,i}, s_{z,i}]$. From this we see that we must maximize

$$\sum_{i=1}^{n} w_i \left(\hat{\mathbf{q}}_u \hat{\mathbf{p}}_{\mathbf{p}_i} \hat{\mathbf{q}}_u^*\right) \cdot \hat{\mathbf{p}}_{\mathbf{s}_i}.$$

To be able to deal with this last expression, Horn [26] rewrites it:

$$\sum_{i=1}^{n} w_i \left(\hat{\mathbf{q}}_u \hat{\mathbf{p}}_{\mathbf{p}_i}\right) \cdot \left(\hat{\mathbf{p}}_{\mathbf{s}_i} \hat{\mathbf{q}}_u\right). \tag{5.3}$$

For this, the following useful result is utilized: Given three arbitrary quaternions $\hat{\mathbf{p}}$, $\hat{\mathbf{q}}$ and $\hat{\mathbf{r}}$, the identity

$$(\hat{\mathbf{p}}\hat{\mathbf{q}}) \cdot \hat{\mathbf{r}} = \hat{\mathbf{p}} \cdot (\hat{\mathbf{r}}\hat{\mathbf{q}}^*)$$

is always valid. If we set $\hat{\mathbf{p}} = \hat{\mathbf{q}}_u \hat{\mathbf{p}}_{\mathbf{p}_i}$, $\hat{\mathbf{q}} = \hat{\mathbf{q}}_u^*$ and $\hat{\mathbf{r}} = \hat{\mathbf{p}}_{\mathbf{s}_i}$ we see that the result follows.

Using (3.2) and (3.3), we can again rewrite (5.3) through a sequence of steps. We first set

$$\hat{\mathbf{q}}_u \hat{\mathbf{p}}_{\mathbf{p}_i} = \begin{bmatrix} 0 & -p_{x,i} & -p_{y,i} & -p_{z,i} \\ p_{x,i} & 0 & p_{z,i} & -p_{y,i} \\ p_{y,i} & -p_{z,i} & 0 & p_{x,i} \\ p_{z,i} & p_{y,i} & -p_{x,i} & 0 \end{bmatrix} \hat{\mathbf{q}}_u^T$$

$$= \hat{\mathbf{q}}_u \begin{bmatrix} 0 & p_{x,i} & p_{y,i} & p_{z,i} \\ -p_{x,i} & 0 & -p_{z,i} & p_{y,i} \\ -p_{y,i} & p_{z,i} & 0 & -p_{x,i} \\ -p_{z,i} & -p_{y,i} & p_{x,i} & 0 \end{bmatrix}$$

$$= \hat{\mathbf{q}}_u \mathbf{P}_i$$

and then

$$\hat{\mathbf{p}}_{\mathbf{s}_i} \hat{\mathbf{q}}_u = \begin{bmatrix} 0 & -s_{x,i} & -s_{y,i} & -s_{z,i} \\ s_{x,i} & 0 & -s_{z,i} & s_{y,i} \\ s_{y,i} & s_{z,i} & 0 & -s_{x,i} \\ s_{z,i} & -s_{y,i} & s_{x,i} & 0 \end{bmatrix} \hat{\mathbf{q}}_u^T$$

$$= \mathbf{S}_i \hat{\mathbf{q}}_u^T.$$

This gives us the expression

$$\sum_{i=1}^{n} w_i \left(\hat{\mathbf{q}}_u \mathbf{P}_i \mathbf{S}_i \hat{\mathbf{q}}_u^T \right)$$

which can be written even more compactly as

$$\hat{\mathbf{q}}_u \left(\sum_{i=1}^{n} \mathbf{N}_i \right) \hat{\mathbf{q}}_u^T,$$

where the columns one to four of $\mathbf{N}_i$ read

$$w_i \begin{bmatrix} p_{x,i}s_{x,i} + p_{y,i}s_{y,i} + p_{z,i}s_{z,i} \\ p_{y,i}s_{z,i} - p_{z,i}s_{y,i} \\ p_{z,i}s_{x,i} - p_{x,i}s_{z,i} \\ p_{x,i}s_{y,i} - p_{y,i}s_{x,i} \end{bmatrix},$$

$$w_i \begin{bmatrix} p_{y,i}s_{z,i} - p_{z,i}s_{y,i} \\ p_{x,i}s_{x,i} - p_{y,i}s_{y,i} - p_{z,i}s_{z,i} \\ p_{x,i}s_{y,i} + p_{y,i}s_{x,i} \\ p_{z,i}s_{x,i} + p_{x,i}s_{z,i} \end{bmatrix},$$

$$w_i \begin{bmatrix} p_{z,i}s_{x,i} - p_{x,i}s_{z,i} \\ p_{x,i}s_{y,i} + p_{y,i}s_{z,i} \\ -p_{x,i}s_{x,i} + p_{y,i}s_{y,i} - p_{z,i}s_{z,i} \\ p_{y,i}s_{z,i} + p_{z,i}s_{y,i} \end{bmatrix}$$

and

$$w_i \begin{bmatrix} p_{x,i}s_{y,i} - p_{y,i}s_{x,i} \\ p_{z,i}s_{x,i} + p_{x,i}s_{z,i} \\ p_{y,i}s_{z,i} + p_{z,i}s_{y,i} \\ -p_{x,i}s_{x,i} + p_{y,i}s_{y,i} - p_{z,i}s_{z,i} \end{bmatrix},$$

respectively. The sum of matrices ($\sum_{i=1}^{n} \mathbf{N}_i$) we denote $\mathbf{N}$. We thus have to maximize the quadratic form

$$\hat{\mathbf{q}}_u \mathbf{N} \hat{\mathbf{q}}_u^T,$$

where $\mathbf{N}$ is a symmetric 4×4 matrix, and $\hat{\mathbf{q}}_u$ is a four vector of unit length.

A result [30] from the theory of constrained optimization tells us that the maximum value λ_m of the quadratic form

$$\hat{\mathbf{q}}_u \mathbf{N} \hat{\mathbf{q}}_u^T$$

for $\hat{\mathbf{q}}_u^T \hat{\mathbf{q}}_u = 1$ is also the greatest eigenvalue of $\mathbf{N}$. Thus the value λ_m occurs when $\hat{\mathbf{q}}_u$ is a unit eigenvector of $\mathbf{N}$ corresponding to the eigenvalue λ_m. So the quaternion we seek, and thus the rotation, is the unit eigenvector corresponding to the greatest eigenvalue of $\mathbf{N}$.

Computing the eigenvalues of $\mathbf{N}$, and the corresponding eigenvectors should be done by using appropriate numerical tools like MATLAB. If we insisted on doing this computation by hand we would first have to find the roots of the characteristic polynomial $\pi_{\mathbf{N}}$, which corresponds to solving a fourth-order polynomial since $\mathbf{N}$ is 4×4. For this purpose, there are formulas. Then we would have to solve a linear set of equations on the form $[\mathbf{N} - \lambda_m \mathbf{I}]\mathbf{x} = \mathbf{0}$, where $\mathbf{x}$ is the (unknown) eigenvector corresponding to the largest eigenvalue of $\mathbf{N}$ and $\mathbf{I}$ is the 4×4 identity matrix.

5.5 Summary

We have settled a closed form solution to the pose estimation problem, but with the help of computers we often employ an iterative scheme instead in the search for the transformation parameters. Hence, in the following three chapters we will review three iterative algorithms.

Chapter 6

An Exhaustive Method

We will now concentrate on numerical methods for solving the *rigid* registration problem, i.e. the matrix **M** can be decomposed into a translation **T** and a rotation **R**. Numerical methods finds the six unknown parameters that determine the rigid body transformation in an iterative scheme. This way, the final solution is the result of refining several temporary solutions. This is the usual practice when one tries to solve the registration problem. However, the methods we present, differ somewhat in their construction; while some apply an exhaustive search for the six unknown parameters, others search for optimal point correspondences before they try to recover the transformation parameters.

In this chapter we will go through a method for registration of a special kind of curves, called crest lines. A rough, though somewhat intuitive description of a crest line, states that it is a characteristic curve on the most noticeable areas of a surface, such as borders of holes. A less intuitive but more precise definition is given in [23]. The authors define a crest line as "locations on a surface where the maximum surface principal curvature, in absolute value, reaches a local maximum in the principal direction of maximum curvature".

The method utilizes a technique referred to as geometric hashing which implies a brute force approach to the registration problem. We will review the ideas behind geometric hashing in the next section. We often characterize methods which search exhaustively for the few six unknown parameters as voting methods [19].

6.1 Geometric hashing

We will study the registration method of [23] which is based on the ideas of geometric hashing. The method specifically registers crest lines from CT[1]

[1]Computed Tomography is a medical imaging technique; it is a way of developing a 3D image of the inside of an object (often a part of a human body), from a series of 2D X-ray

scans.

The registration is rigid and analogous to a point cloud setting. However, the sets P and S introduced in chapter 4 should now be thought of as collections of crest lines in a CT image (of a vertebra), rather than collections of points. Given a crest line in P, we seek the crest line in S which shares the largest number of points with the crest line in P after a rigid transformation has been applied.

Geometric hashing, reviewed in [46], is a technique for recognizing objects: Though not trivial for a computer, it is possible to recognize an object when encountered in a "scene", if a sample of this object has been previously stored in a database. Hashing is based on the indexing approach and can be recognized as a two step process:

1. The first step is a *preprocessing* step where information about "models" is stored in a hash table data structure: In our setting, the crest lines in S make up the models. We will store the sample points on each crest line in S in a hash table, based on a set of parameters specific to each point.

2. The second step is the stage of *recognition*: We compute a set of parameters for each of the sample points at all the crest lines in P, as we did for the sample points in S. For each point we use its parameters as an index into the previously stored hash table. If a point from S is encountered, we compute and store a rigid transformation. Each computed transformation receives a vote.

The transformation that is ultimately applied to a crest line in P will be recovered after counting the number of votes for the stored transformations. We pick the transformation with the most votes.

We have now settled what it is we want to register and what we want to achieve with the registration. We have also sketched out the structure of the registration algorithm and it is time to dive into the details. Before we do that we advise the reader to go back and reread section 3.7 on page 42 since the terms from that section will be used repeatedly throughout the rest of this chapter.

6.1.1 Preprocessing

First of all, extracting crest lines can be difficult and for proper registration we need to compute differentiable features of higher-order, such as torsion. Thus, we approximate the crest lines, which can be noisy and not suited for the aforementioned computations, with smooth continuous spline-curves. This is treated thoroughly in [24]. In addition, the approximations are sampled and at each sample point we associate a Frénet frame,

images taken around a single axis of rotation [45].

see section 3.7 on page 42, and these frames are in essence what we align when we compute the transformations.

In the preprocessing stage we go through all the crest lines in S successively. Our goal is to store them in such a way that when we are given a crest line in P, we will be able to recognize its optimal match in S although it has a different orientation. As stated, we will store the sample points on each crest line in S based on a set of parameters that will not change under rigid transformations. For each sample point, we compute a unique set of parameters. A comparison of different invariant parameters is made in [24] and the following three have shown to be stable[2]:

1. The angle α between the tangent vectors at two sampled points,
2. the translational offset $\mathbf{t}$ between the two tangent vectors at two sampled points,
3. curvature of the curve.

In [23] five parameters are used in the registration experiments. The authors take notice of the fact that the curves are extracted off a surface and make use of the surface principal frame as well, see section 3.7 on page 42. The Frénet frame and the surface principal frame should not be related as is argued by [23], since some combinations of possible parameters are dependent on each other. The five independent parameters mentioned in [23] are: Curvature, curve torsion[3](see [23] for an explicit expression of torsion), geodesic torsion of the curve with respect to the surface, the angle between the Frénet frame-normal and surface normal and the angle between the Frénet frame tangent and the principal curvature direction $\mathbf{e}_1$.

We see that for the computation of α and $\mathbf{t}$ (see point one and two on the previous page), we need to regard two points at a time. Or rather, we regard two Frénet frames at a time. For this, we first select[4] on each curve a point referred to as a *basis* point. We will then compute the two parameters α and $\mathbf{t}$ for each of the other points on the curve by comparing their associated tangent vectors with the tangent vector at the basis point. This procedure is repeated for every possible basis point on the curve, thus resulting in storing each sample point, several times.

For each sampled point the parameters will serve as an index into a one-dimensional hash table where the point will be stored. At the same location

[2] A parameter being stable has a special meaning in this setting: Say that the variance for the parameter based on the parameter values at different points on a *single* curve is found to be σ_1. Further, we assume that the variance computed from the set of all corresponding points on *different* curves is σ_2. The parameter is then classified as stable if the ratio $\frac{\sigma_1}{\sigma_2}$ is small.

[3] "The twisting of an object due to an applied torque, or moment of force", [45].

[4] According to [23] selecting a basis point can be difficult - they suggest as a possibility to use maxima of curvature.

in the hash table where the point is stored, we store additional information; the basis point which was used to compute the point's parameter values and the "model" curve from which the point was sampled. We convert the set of parameter values into a tuple of integer values and then the integers are mapped[5] to a single integer index.

6.1.2 Recognition

During the recognition stage we are given a curve in P and the set S of "model" curves. We wish to match the curve from P with a curve in S and to retrieve the rigid transformation between them. We vote for each transformation we compute and eventually choose as the optimal transformation, the one with the most votes.

We first choose an arbitrary point on the curve in P as a basis point. For all of the remaining points on the current curve, we can compute the same parameters, three for each point, as we did in the preprocessing stage. We then map the parameters to a single integer index as before and we use the index to look up in the hash table. Each time we enter a cell in the hash table where we have stored a point from a curve in S, we have a potential match. We compute, store and cast a vote for the transformation between them.

The rotational part $\mathbf{R}$ of the transformation between a pair of Frénet frames at matching points $\mathbf{p}$ and $\mathbf{s}$ from curves in P and S, respectively, is given by the matrix product

$$\mathbf{R} = [\mathbf{t}, \mathbf{n}, \mathbf{b}][\tilde{\mathbf{t}}, \tilde{\mathbf{n}}, \tilde{\mathbf{b}}]^T.$$

The vectors $\tilde{\mathbf{t}}$, $\tilde{\mathbf{n}}$ and $\tilde{\mathbf{b}}$ are the vectors of the Frénet frame at $\mathbf{p}$. The vectors $\mathbf{t}$, $\mathbf{n}$ and $\mathbf{b}$ are the vectors of the Frénet frame at $\mathbf{s}$.

The result follows from the fact that an arbitrary point $\mathbf{u}_p$ can be written

$$\alpha\tilde{\mathbf{t}} + \beta\tilde{\mathbf{n}} + \gamma\tilde{\mathbf{b}}, \quad \alpha, \beta, \gamma \in \mathbb{R}$$

in the Frénet frame at $\mathbf{p}$. The coordinates of $\mathbf{u}_p$ relative to the Frénet frame ($[\alpha, \beta, \gamma]$) can thus be computed as $[\tilde{\mathbf{t}}, \tilde{\mathbf{n}}, \tilde{\mathbf{b}}]^T\mathbf{u}_p$ (since the Frénet frame vectors are orthonormal). We want the corresponding point of $\mathbf{u}_p$ in the Frénet frame at $\mathbf{s}$, say $\mathbf{u}_s$, to have the same relative position as $\mathbf{u}_p$ in the Frénet frame at $\mathbf{p}$. In other words, we want

$$\mathbf{u}_s = \alpha\mathbf{t} + \beta\mathbf{n} + \gamma\mathbf{b}$$

or

$$\mathbf{u}_s = [\mathbf{t}, \mathbf{n}, \mathbf{b}][\tilde{\mathbf{t}}, \tilde{\mathbf{n}}, \tilde{\mathbf{b}}]^T\mathbf{u}_p.$$

[5] By shifting bits in the binary representation of the integers and using the exclusive or operator.

This implies that the rotation **R** that relates the two frames can be written as

$$\mathbf{R} = [\mathbf{t}, \mathbf{n}, \mathbf{b}][\tilde{\mathbf{t}}, \tilde{\mathbf{n}}, \tilde{\mathbf{b}}]^T.$$

The translation **t** is computed as

$$\mathbf{t} = \mathbf{s} - \mathbf{R}\mathbf{p}$$

given the coordinates of the points **p** and **s** in a common reference frame [23].

A note should be taken from [23] regarding the computation of the transformations. For this we pass on from [23], the notion of a six-dimensional "accumulator" in which we store the transformations. The accumulator will consist of cells into which the transformations are stored. Transformations that define approximately the same motion will be stored in neighbouring cells. Further, associated with each transformation there will be an error interval. The idea is that if the error interval of a transformation touches neighbouring cells, then we cast a vote to those cells (transformations) as well. (For a maximum of 26 additional votes).

The cells with a high number of votes, correspond to rigid transformations that will be refined for the purpose of computing a final transform. For each cell with a high number of votes we retrieve the points from P and S that hashed to this cell. Based on these matching points, we compute the refined transformations. Finally we get the transformation we seek by choosing the one out of the refined transformations which registers the largest number of points.

6.2 Analysis of time complexity

For a complexity analysis of the algorithm, let

- n be the number of sampled points on a crest line approximation
- m be the number of models (curves in S)
- c represent the number of points needed to build a frame or basis; in general, we need three non-collinear points to build a frame (three orthonormal vectors), however, if we assume that there exist parameterizations of the curves in question then a (Frénet) frame can be computed using only one point (cf. section 3.7 on page 42).
- h represent the complexity of processing a hash table bin

The preprocessing stage of geometric hashing is of order $O(mn^{c+1})$, according to the complexity analysis of [46]. If we assume that the curves in

question are parametrized, then $c = 1$ which means that the preprocessing stage of the reviewed method has complexity $O(mn^2)$.

It is claimed in [23] that the recognition stage of their algorithm is worst case $O(mn^2)$. That is, it runs in polynomial time. This should correspond with the complexity analysis of [46] which says that the complexity of the recognition phase in general, is $O(hn^{c+1})$. Thus, in this context h is of order $O(m)$. In general, the factor h depends on the distribution of the indices; if say, each point has its own hash table bin, then the access could be done in constant time, $O(1)$.

A commonly used registration algorithm known as the Iterative Closest Point (ICP) algorithm (reviewed in chapter 7 on the next page) is mentioned here for comparison. As the name suggests, this procedure repeats itself, say i times. Compared to its upper bound performance $O(imn^2)$, the method of [23] performs well. Guéziec, Pennec and Ayache [23] also claim that the recognition stage of their method has better lower bound performance ($O(n)$) than the ICP algorithm ($O(mn \log n)$).

6.3 Summary

Clearly, one of the advantages of an algorithm like this is that it is efficient when one wants to perform registration of several views against a database. The process does not depend on the size of the database which is computed off-line and can be reused. However, the preprocessing stage is rather costly — this holds in general for methods based on geometric hashing. In fact, we claim that the expensive preprocessing makes methods like this unsuited for registration of only two sets of 3D scanner data.

Chapter 7

An Area-based Method

We will continue our study of iterative procedures by looking at a method which, depending on the application, can be far less expensive with regard to execution time, than the previous one. This follows from the fact that it classifies as a direct method; from the classification of registration methods at the end of chapter 4, we deduce that the method computes the transformation parameters in a least square sense. However, for the method to come up with an optimal alignment, the initial displacement of the input scans or data must be small. Thus, methods like the one we are about to study, are suited for being included at the end of a registration method, to trim the aligning.

7.1 A similar approach

Methods for pairwise registration most often use a form of, or even include, the Iterative Closest Point algorithm[1] (ICP) [8] to calculate the transformation needed to align the two geometric shapes [45]. The method of [33] which we outline in this chapter has certain similarities with the ICP. The ICP is therefore shown below.

According to its inventors, the ICP approximately registers a set of digitized data representing a rigid object, with a precise idealized geometric model (with known representation) of the same object. Also, it does so in a finite number of iterations. The algorithm will converge towards a local minimum of the sum of squared residuals, as has been proved in [8].

[1] The name Iterative Corresponding Points is suggested by [42].

The ICP Algorithm. *Let a threshold value and two sets of geometric shapes P and S with known representation be given as input. Also, assume that $P \subseteq S$. Decompose P into a point set and denote this set by P_0, set $P' = P_0$.*
do

1. for each point $\mathbf{p}_1$, $\mathbf{p}_2$, ..., $\mathbf{p}_{n_P}$ in P', find n_P distinct points in S,

$$\{\mathbf{s}_1, \mathbf{s}_2, \ldots, \mathbf{s}_{n_P}\}$$

   such that $\mathbf{s}_i$ is the closest point in S to $\mathbf{p}_i$, i=1,2,...,n_P, and denote this set by S'.
2. determine a rigid transformation Ω between P_0 and S'.
3. apply the computed transformation to the points in P_0 and let $P' := \Omega(P_0)$.
4. compute the error, e.g. as $\sum_{i=1}^{n_P} ||\mathbf{s}_i - \mathbf{TR}\mathbf{p}_i||^2$. Here **TR** is the transformation matrix of Ω.

while error > threshold

We can see from the outline of the algorithm that when the computed error falls below a preset threshold, the algorithm will terminate. The convergence relies on the fact that the sequence of values representing the (mean) squared error is non increasing and bounded below (by zero). If the sequence at some stage was to increase, then the points S' chosen at that stage would not be closer to the points in P' than the points which were chosen in the previous step of the algorithm. This will thus not be the case since at each stage we choose only the closest points S' to P'.

7.2 A general outline

The registration method of Mitra et al. [33] which we are about to study follow much the same line of thought as the ICP algorithm: They consider the point cloud setting where the points in P and S are collected off a surface. They assume in addition that P represents a subset of the data in S. Through a number of iterations, they search for the best rigid transformation that aligns the two data sets.

We recognize this as the setting in chapter 4 and we may expect that the problem will be similar to (4.3). However, Mitra et al. make use of the fact that the points are not just some random samples; they try to minimize the squared distance $d(\cdot,\cdot)^2$ between the points $\mathbf{p}_i$ in P, and the (triangulated) surface Φ_S represented by the points in S. They will achieve this by applying a rigid transformation to the point set P. In other words, they try to

minimize

$$\sum_{j=1}^{n_P} d(\mathbf{TR}\mathbf{p}_j, \Phi_S)^2. \tag{7.1}$$

Still, the matrix **T** represents translation and the matrix **R** represents rotation. The squared distance function $d(\mathbf{p}_j, \Phi_S)^2$ should be thought of as the function which assigns to a point $\mathbf{p}_j$, the shortest distance from $\mathbf{p}_j$ to Φ_S, squared.

The point cloud P is put into the squared distance field of the surface Φ_S and by following a gradient descent search the points in P approach a position corresponding to a local minimum of (7.1). The gradient descent search is a local optimization procedure which approaches a local minimum of a given function by taking steps proportional to the negative gradient of the function evaluated at certain points.

We can try to visualize the effects that the gradient descent search has on our point cloud setting: Think of the points in P as moving from their initial position, taking equally (small) steps towards a final position, at which the points are positioned so that equation (7.1) is minimized. In each intermediate position, we need to solve a linear system of equations to determine the transformation parameters which brings us one step closer to a minimum of (7.1).

7.2.1 Registration using approximants A_j

In addition to the parameters that define the rigid motion we also need to define the distance metric $d(\cdot, \cdot)$. In earlier work different distance metrics have been applied. It has been observed that a bad choice of distance metric gives poor convergence. This holds in particular for methods that require the input shapes to be close. If we look at the ICP algorithm, we may argue that it suggests a point to point distance metric. Registration methods based on such metrics have more stable performance, but tend to provide slower convergence when the point clouds are close.

Mitra et al. [33] address the issue of finding a good approximation to the squared distance function between a point and a surface. From the discussion above we may see that a *good* approximant should imply fast (e.g. quadratic) convergence when the point clouds are close and it should also result in stable convergence when the point clouds are placed far apart. The approximant provided by Mitra et al. is based on second order information about the underlying surfaces from where the points are collected. It is also valid locally in a neighbourhood around the point at which it is evaluated. These are qualities which, according to the authors, ensure both stable and fast convergence of the algorithm.

We will denote a local quadratic approximant by A which takes the

form

$$A(\mathbf{p}^1) = (\mathbf{p}^1)^T \mathbf{Q} \mathbf{p}^1, \tag{7.2}$$

where $\mathbf{p}^1 = [x, y, z, 1]^T$ is a point in $\mathbb{R}^3$ and $\mathbf{Q}$ is a 4×4 symmetric matrix containing the (ten) coefficients of A. For each point $\mathbf{p}_j$ we compute a corresponding local approximant. That is to say, the entries in $\mathbf{Q}$ depend on $\mathbf{p}_j$. We indicate this by giving the index j to the matrix as well, $\mathbf{Q}_j$. Using the approximation $A_j(\mathbf{TR}\mathbf{p}_j^1) = \left(\mathbf{TR}\mathbf{p}_j^1\right)^T \mathbf{Q}_j \left(\mathbf{TR}\mathbf{p}_j^1\right) \approx d(\mathbf{TR}\mathbf{p}_j, \Phi_S)^2$, the function given in (7.1), which we want to minimize, now becomes

$$\sum_{j=1}^{n_P} A_j.$$

7.2.2 More approximations

To determine the six unknown transformation parameters[2] which make up the rigid motion **TR**, we can set the partial derivatives of the sum (the error function) above to zero. This gives us a *non-linear* system of equations. We choose to approximate the rotation matrix **R** involved in the error function to get a *linear* set of equations instead. That is, we use an approximation to the product of the three rotation matrices $\mathbf{R}_x$, $\mathbf{R}_y$ and $\mathbf{R}_z$ from section 3.4.1:

$$\begin{aligned}
\mathbf{R}_z\mathbf{R}_y\mathbf{R}_x &= \begin{bmatrix} \cos\gamma & -\sin\gamma & 0 & 0 \\ \sin\gamma & \cos\gamma & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\beta & 0 & \sin\beta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\beta & 0 & \cos\beta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha & 0 \\ 0 & \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
&= \begin{bmatrix} \cos\gamma\cos\beta & -\sin\gamma\cos\alpha + \cos\gamma\sin\beta\sin\alpha & \sin\gamma\sin\alpha + \cos\gamma\sin\beta\cos\alpha & 0 \\ \sin\gamma\cos\beta & \cos\gamma\cos\alpha + \sin\gamma\sin\beta\sin\alpha & -\cos\gamma\sin\alpha + \sin\gamma\sin\beta\cos\gamma & 0 \\ -\sin\beta & \cos\beta\sin\alpha & \cos\beta\cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
&\approx \begin{bmatrix} 1 & -\gamma + \beta\alpha & \gamma\alpha + \beta & 0 \\ \gamma & 1 + \gamma\beta\alpha & -\alpha + \gamma\beta & 0 \\ -\beta & \alpha & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
&\approx \begin{bmatrix} 1 & -\gamma & \beta & 0 \\ \gamma & 1 & -\alpha & 0 \\ -\beta & \alpha & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.
\end{aligned}$$

That is, we assume the angles are small so that $\cos\alpha = \cos\beta = \cos\gamma \approx 1$, $\sin\alpha \approx \alpha$, $\sin\beta \approx \beta$ and $\sin\gamma \approx \gamma$, and approximate the rotation $\mathbf{R}_z\mathbf{R}_y\mathbf{R}_x$ using the matrix

$$\begin{bmatrix} 1 & -\gamma & \beta & 0 \\ \gamma & 1 & -\alpha & 0 \\ -\beta & \alpha & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

This way we get the linear system of equations stated in [33].

[2] Three angles and three translational offsets.

We see that the linearization is valid only for small rotational angles α, β and γ. Thus we need to check whether or not the computed transformation is a small step towards the local minimum. To check if the transformation is sufficiently small, Mitra et al. suggest employing the Armijo rule[3]. Generally, the Armijo rule is a way of telling if, in an iterative search for a local minimum of a function, we have a too large decrease in the function value from one step to another.

If we detect that the computed transformation is too large, it is recommended to take a fractional step. The fraction $\frac{1}{n}$ is found via line search. Typically, we start off with $n = 1$ and then decrease the fraction by multiplying it with a constant between 0 and 1. Mitra et al. suggests computing a fractional rotation $\mathbf{R}'$ by the methods of Alexa [4], and then compute the fractional translation $\mathbf{T}'$ as $(\mathbf{R}-\mathbf{I})^{-1}(\mathbf{R}'-\mathbf{I})\mathbf{T}$. Alexa suggests computing the fractional step $\mathbf{R}'$ as $e^{\frac{1}{n}\log \mathbf{R}}$, but points out that his method is better suited for general transformations (transformations including non-uniform scaling). Alexa suggests employing Rodrigues' formula when only rotations and translations are involved. (We defer to chapter 9 for a discussion on the matrix exponential and logarithm.)

If the Armijo condition is satisfied after applying a smaller transformation and in addition the computed sum is less than the given threshold value, then we are done. On the other hand, if the sum is greater than the threshold, we must repeat the procedure based on the points $\mathbf{T}'\mathbf{R}'\mathbf{p}_j$.

This completes the registration algorithm of Mitra et al. but we have yet to describe the approximant to the squared distance function. (As we recall, their concern was to develop a *good* approximant.) From its form (7.2) we see that we must find the (ten) coefficients of A_j to uniquely determine it and the authors have given two methods for finding them; an "on-demand" method and a d^2-tree method. We will study these methods in the following sections.

7.3 Computing A_j's on demand

Before we start studying the "on-demand" method, we advise the reader to go back and reread section 3.7 on page 42. We will go through the method in brief — our focus will be on the d^2-tree method which we have implemented.

The on-demand method is based on computing the second order local Taylor approximant of the squared distance function to a surface at a given point in $\mathbb{R}^3$. Finding the Taylor approximant turns out to be the biggest challenge, but as soon as it is found it can be transformed to the global coordinate system and play the role as our A_j.

[3]This is one of the two conditions that make up the Wolfe conditions common in the field of optimization [45].

Given a point $\mathbf{p}$ with coordinates $[x_p, y_p, z_p]$ in the principal frame at its normal footpoint, the Taylor approximant is found to be a weighted sum of the three squared distance functions x_p^2, y_p^2 and z_p^2; $w_1 x_p^2 + w_2 y_p^2 + z_p^2$ [41]. The three squared distance functions provide us with the squared distance to the following entities at the normal footpoint of $\mathbf{p}$:

- The principal plane containing the direction $\mathbf{r}_1$ of maximum curvature,
- the principal plane containing the direction $\mathbf{r}_2$ of minimum curvature,
- the tangent plane.

Consequently, the on-demand method requires that we for every point $\mathbf{p}_j$ in P solve the following tasks:

1. Approximate the normal footpoint of $\mathbf{p}_j$ with a point $\mathbf{s}_j \in S$.
2. Determine the principal frame $\mathbf{r}_{1,j}$, $\mathbf{r}_{2,j}$ and $\mathbf{n}_j$ at $\mathbf{s}_j$.

We also need to compute the principal radii of curvature r_1 and r_2 at $\mathbf{s}_j$ — the weights depend on them.

To approximate the normal footpoint of $\mathbf{p}_j$ we build a nearest neighbour structure for S. Arya et al. [6] show how this can be done with the help of a balanced box decomposition tree data structure. This way, neighbour queries boil down to a point location in the tree, followed by traversing certain leaf cells.

Next, we find the principal frame at the normal footprint $\mathbf{s}_j$ of $\mathbf{p}_j$. The principal frame consists of the normal $\mathbf{n}_j$ and the directions $\mathbf{r}_{1,j}, \mathbf{r}_{2,j}$ of the principal radii of curvature. For this we must first sample a set of k points, $\left\{[x_i, y_i, z_i]^T\right\}_{i=1}^k$, which are neighbours of $\mathbf{s}_j$. When Φ_S is represented by a triangulation, the neighbours of $\mathbf{s}_j$ are found as the direct neighbours, neighbours of the direct neighbours and so on. We will soon return to the issue of how to estimate the number k.

Finding the normal can then be done in several ways. One way is to fit a plane using $\mathbf{s}_j$ and its closest neighbours and to compute the normal to this plane. In [34] they fit such a plane in a *total* least square[4] sense. The sum of squares to be minimized depends on a quantity that turns out to be an eigenvalue of the matrix $\mathbf{C}$,

$$\mathbf{C} = \sum_{i=1}^{k} \left(\begin{bmatrix} x_i \\ y_i \\ z_i \end{bmatrix} [x_i, y_i, z_i] - \begin{bmatrix} x \\ y \\ z \end{bmatrix} [x, y, z] \right).$$

[4]According to Wikipedia [45], this an optimization technique which assumes that both the observed values and the predicted values contain noise. This implies that the solution to the usual least square problem $\mathbf{Ax} \approx \mathbf{b}$, $\min ||\mathbf{Ax} - \mathbf{b}||_2$, is no longer the best possible solution.

Here, $[x, y, z]^T$ is the mean of the points $\left\{[x_i, y_i, z_i]^T\right\}_{i=1}^k$. To minimize the sum of squares we need to compute the smallest eigenvalue of **C**. Moreover, the normal $\mathbf{n}_j$ we seek[5] is the eigenvector corresponding to the smallest eigenvalue. We need at least as many neighbours k as the number of unknown coefficients (four) in the plane equation.

The procedure [10, 34] for computing the directions $\mathbf{r}_{1,j}, \mathbf{r}_{2,j}$ is based on the truncated, second degree Taylor approximation $T(x, y)$ of the surface Φ_S near $\mathbf{s}_j$. We express the surface near $\mathbf{s}_j$ as a bivariate function $f(x, y) = T(x, y) + remainder$, where $T(x, y)$ is on the form $T(x, y) = ax^2 + bxy + cy^2 + dx + ey + f$. Cazals et al. [10] show how to find the six coefficients a through f from an ordinary least square fit,

$$\sum_{i=1}^{k} \left(T(x_i, y_i) - f(x_i, y_i)\right)^2, \quad i = 1, \ldots, k.$$

For this we again use the neighbouring points $\{[x_i, y_i, z_i]\}_{i=1}^k$ of $\mathbf{s}_j$, $k > 5$.

It is shown in [10] how the directions $\mathbf{r}_{1,j}$ and $\mathbf{r}_{2,j}$ of the principal radii of curvature can be found from the coefficients of $T(x, y)$: The vectors are the eigenvectors of the matrix[6] **W**,

$$\mathbf{W} = \frac{1}{1 + d^2 + e^2 + d^2e^2 - de} \begin{bmatrix} de\frac{b}{w} - (1 + e^2)\frac{2a}{w} & de\frac{2c}{w} - (1 + e^2)\frac{b}{w} \\ de\frac{2a}{w} - (1 + d^2)\frac{b}{w} & de\frac{b}{w} - (1 + d^2)\frac{2c}{w} \end{bmatrix}.$$

Here, the scalar $w = \sqrt{1 + d^2 + e^2}$.

In fact, we find the rest of the quantities we seek from the coefficients a through e of $T(x, y)$. The principal radii of curvature r_1, r_2 are computed as

$$\frac{1}{H + \sqrt{H^2 + K}} \quad \text{and} \quad \frac{1}{H - \sqrt{H^2 + K}},$$

respectively, using the Gaussian curvature[7] K and mean curvature H [33]. The curvatures H and K can in turn be computed from the coefficients a through e as

$$K = \frac{4ac - b^2}{(1 + d^2 + e^2)^2},$$
$$H = \frac{a(1 + e^2) - bde + c(1 + d^2)}{(1 + d^2 + e^2)^2}.$$

While the reason for determining the principal frame at $\mathbf{s}_j$ was to come up with the coordinates $[x_p, y_p, z_p]$, the principal radii of curvature r_1, r_2 are

[5] For this approximation to be good, we must assume that the curvature around $\mathbf{s}_j$ is small and bounded [34].

[6] The matrix is the so-called *Weingarten map* of $T(x, y)$.

[7] Although we have decided to use uppercase letters to denote sets, we follow convention in this case and denote the Gaussian and mean curvature by K and H, respectively.

needed to compute the weights:

$$w_j = \begin{cases} \frac{l}{l-r_j} & \text{if } l < 0,\ j = 1, 2 \\ 0 & \text{otherwise} \end{cases}$$

The scalar l is the signed distance from $\mathbf{p}_j$ to $\mathbf{s}_j$. The idea behind using these weights, is to maintain a non-negative approximant. For instance, if the signed distance is negative we see that the fraction $\frac{l}{l-r_j}$ will be positive.

Mitra et al. transform the local approximant $w_1 x_p^2 + w_2 y_p^2 + z_p^2$ to the global coordinate system such that it reads

$$w_1(\mathbf{r}_{1,j} \cdot (\mathbf{p}_j - \mathbf{s}_j))^2 + w_2(\mathbf{r}_{2,j} \cdot (\mathbf{p}_j - \mathbf{s}_j))^2 + (\mathbf{n}_j \cdot (\mathbf{p}_j - \mathbf{s}_j))^2.$$

This expression is another way of writing $A(\mathbf{p}_j)$. Hence the coefficients, i.e. the entries of $\mathbf{Q}_j$, are given by expanding the expression.

7.4 Computing A_j's using the d^2-tree

The d^2-tree method for finding the ten coefficients defining the A_j's is based on an octree data structure. The octree data structure is as the name suggests a tree data structure which partitions a three dimensional space recursively by dividing it into eights cubes. Thus, interior to each cube there will be a certain number of smaller cubes.

We will store in each cube C_i the squared distance d_i^2 from its lower left corner point (x_i, y_i, z_i) to the triangulation. Moreover, each cube will contain a collection of such lower left corner points with an associated square distance value. For each cube we will fit a quadratic trivariate function in a least square sense using the datapoints (x_i, y_i, z_i, d_i^2). The quadratic function is one of our approximants, say A_i.

Points $\mathbf{p}_j$ from P are located somewhere inside a cube C_j in the d^2-tree and we can use the approximant A_j stored in C_j to get the approximated squared distance value, i.e. $A_j(\mathbf{p}_j)$. Consider next the details of this procedure.

First we settle some notation by reviewing the initial stages of the algorithm: We start off by enclosing the triangulated surface Φ_S in a cube so that we include the space where the point cloud P lies. This cube is referred to as *the root cube* of the d^2-tree. The root cube is then partitioned into eight equally sized cubes; we will refer to this process as *subdivision*. Further, we will refer to the cube which has been subdivided as a *parent*. Consequently, we will refer to the eight sub-cubes (inside the parent) as children. These two "generations" also represent the first two levels of our octree; level zero (l_0) and level one (l_1), respectively. At this point, we say that l_1 is the finest level of the octree.

As stated, we first make a parent out of the root cube. Then we make a parent out of each of the children of the root cube. This gives us 73 cubes; the root cube, its eight children and its 64 grandchildren. The subdividing pattern then continues in the following way: We run through a list of the triangles of the triangulated surface, and for each of them we go through the 64 grandchildren of the root cube. Each time we detect that a triangle intersects one of these cubes, we subdivide the cube (if it is not a parent already) and also we subdivide each of its children. We must then traverse the cube's grandchildren and check for intersection between them and the current triangle. As before, if the triangle hits anyone of them, then they are also made grandparents if they are not already. We continue this way until we reach the finest level.

The number of levels $m + 1$ should be chosen such that the size of the cubes at this level "meets the precision requirements of the application". It is not stated what exactly those requirements are, but smaller cubes are said to result in higher accuracy of the final fit. The parameter m is given as input to the algorithm and, as we may have noticed[8], it must be an even number (greater than three). The final cluster of cubes makes up our octree.

The next step is to go through the cubes at level $l_2, l_4, \ldots, l_{m-2}$, and subdivide all of those which are neighbours of grandparents. As before, we subdivide a cube into its eight children and 64 grandchildren. Several neighbourhood definitions can be used, the most intuitive may be the one ring of direct neighbours. Generally, every cube has 26 direct neighbours; six face neighbours, eight vertex neighbours and twelve edge neighbours.

The reason for the extensive use of subdivision is to produce a large number of cubes at the same level. At the same time we obtain a cluster of small cubes close to the triangulated surface Φ_S. These are both desirable effects in the next step were we apply the sweeping method of Zhao [47]: For the cubes at level l_m that are closest to Φ_S, we will compute the exact (squared) distance from their lower left corner point to Φ_S. This information will then be extended outwards to the remaining cubes of the octree through Zhao's sweeping method.

The A_j's can be found through least square fits of the distance values. Ultimately we will compute and store in all cubes, approximations to the squared distance function based on the (squared) distance values found in their children. If there are not enough points (or no points at all) within a cube to compute a robust approximation, we store in this cube the approximant of its parent.

Zhao's sweeping method solves the so-called Eikonal equation

$$|\nabla u(\mathbf{x})| = f(\mathbf{x}), \mathbf{x} \in \mathbb{R}^n \tag{7.3}$$

[8] We always subdivide twice starting at l_2. According to [31], it is an advantage to have a large number of cubes since we will be applying a sweeping method later on.

with corresponding boundary conditions, numerically, on a rectangular grid. In particular, the distance function $d(\cdot)$ satisfies the Eikonal equation

$$|\nabla d(\mathbf{x})| = 1,$$

with the boundary condition $d(\mathbf{x}) = 0$ (when $\mathbf{x} \in \Phi_S$).

As is the usual practice, the partial differential equation above is discretized at interior grid points (according to the Godunov upwind difference scheme). That is, we compute a numerical solution u at each grid point (lower left corner point at a given level) and use this as an approximation to the exact solution. In our case, the solution u represents the distance to the triangulated surface for a given grid point. This value is updated only if the value that we try to assign is less than the already stored value.

Each grid point has a maximum of six neighbours, and at each neighbour we have stored a corresponding distance value. Initially, if the grid point is at the boundary, then the distance value at this grid point is computed exactly (For efficiency, these computations should be done during the subdivision stage.). If it is not, then we store a large positive value — this value will be updated later on.

In figure 7.1 on the next page, there are six grid neighbours next to the grid point at the question mark. Moreover, at each of the six neighbours we have stored a distance value, $u_{x,1}, u_{x,2}, u_{y,1}u_{y,2}, u_{z,1}$ and $u_{z,2}$, respectively. If we set

$$\begin{aligned} u_{x,min} &= \min(u_{x,1}, u_{x,2}), \\ u_{y,min} &= \min(u_{y,1}, u_{y,2}), \\ u_{z,min} &= \min(u_{z,1}, u_{z,2}) \end{aligned}$$

then the difference scheme at the grid point next to the question mark reads

$$(u - u_{x,min})_+^2 + (u - u_{y,min})_+^2 + (u - u_{z,min})_+^2 = h^2, \tag{7.4}$$

where h is the grid size and

$$s_+ = \begin{cases} 0 & \text{if } s \leq 0 \\ s & \text{otherwise} \end{cases}, s \in \mathbb{R}.$$

Thus, the value u which we are after, i.e. the distance to the triangulated surface at the given grid point, can be found by solving equation (7.4) above.

Assume that $u_{x,min} \leq u_{y,min} \leq u_{z,min}$, then Zhao provides an easy way of finding u: First, solve $(u - u_{x,min})^2 = h^2$. From this we get that $u = h + u_{x,min}$. If $u \leq u_{y,min}$, then we are done. Otherwise, if $u > u_{y,min}$ then we solve

$$(u - u_{x,min})^2 + (u - u_{y,min})^2 = h^2$$

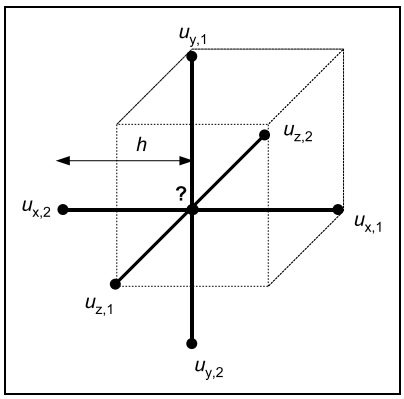


Figure 7.1: We see a selection of grid points, the one next to the question mark has six grid neighbours. At each of the grid neighbours we have stored a value u representing the distance from the grid point to the triangulated surface. We use these six values in the computation of the solution at the grid point next to the question mark. The scalar h is the grid size (which equals the length of the sides of the cubes whose lower left corner points make up the grid grid).

with respect to u and keep the solution if $u \leq u_{z,min}$. Otherwise, if $u > u_{z,min}$ then we solve

$$(u - u_{x,min})^2 + (u - u_{y,min})^2 + (u - u_{z,min})^2 = h^2.$$

However, the value for u found above is most likely not the distance we are after. We must sweep the domain eight times and thus we need to solve the equation above, eight times for each interior grid point. According to Zhao, 2^n sweeping directions in n-space is sufficient for a solution as accurate as the one we would obtain if we let the iteration converge. That is, in 3-space we have eight sweep directions and therefore we sweep eight times.

The eight directions correspond to eight ways of traversing a set of three-tuples (the coordinates of the grid points). The directions, or how we sort the cubes, are best explained graphically, see figure 7.2. Let the lower left corner points of the cubes in the figure be our current grid. We would then sort and traverse the points according to the following eight lists containing the numbers of the cubes to which the points belong,

- $1, 2, 3, 4, 5, 6, 7, 8$
- $2, 1, 4, 3, 6, 5, 8, 7$
- $3, 4, 1, 2, 7, 8, 5, 6$
- $4, 3, 2, 1, 8, 7, 6, 5$

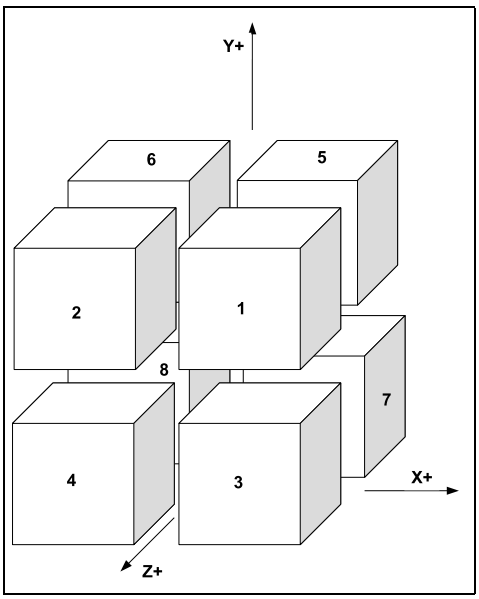


Figure 7.2: We see a small cluster of cubes.

- $5, 6, 7, 8, 1, 2, 3, 4$
- $6, 5, 8, 7, 2, 1, 4, 3$
- $7, 8, 5, 6, 3, 4, 1, 2$
- $8, 7, 6, 5, 4, 3, 2, 1$

Let us see how this method applies to our grids: When we have computed exact distances for the lower left corner points of the cubes at the finest level which intersect the triangulation, it makes sense to apply the sweeping method of Zhao to this grid. Now we will be able to update the large positive values stored at the remaining grid points.

Since the cubes are stored in an octree, sweeping within the different levels can become rather slow. Hence, at each level we sort the grid points in the eight sweeping directions before we sweep. We note that we only need four lists to be able to traverse the grid points in the eight directions: A list representing an ordering of the points, say $8, 7, 6, 5, 4, 3, 2, 1$, can be traversed backwards to yield the direction $1, 2, 3, 4, 5, 6, 7, 8$.

When the sweeping method has been applied to the cubes at the finest level l_m, it is applied to the cubes at level l_{m-2}, l_{m-4} and so on up to level l_2. At level l_{m-2}, some of the cubes may have grandchildren and hence their lower left corner point is also present at level l_m. Thus the value of the distance from their lower left corner point to the triangulated surface is already computed. This means that we have already initialized a subset of

the grid points at level l_{m-2} and we can apply the sweeping method to the corresponding grid. We continue this way up to level l_2.

When the lower left corner point of all cubes at even levels have been equipped with a distance value, we will be able to compute the quadratic functions A_j. (However, we first need to square all the stored values.) We have given the form of A_j above, but it will prove useful for later analysis to rewrite it as

$$\mathbf{x}_j^T \mathbf{A}_j \mathbf{x}_j + \mathbf{b}_j^T \mathbf{x}_j + k. \tag{7.5}$$

We start with the root cube, and try to compute a least square fit of all the squared distance values stored at the lower left corner points within the cube. Since there may be many points within this cube, we do not expect an adequate function to be found right away. If the approximant we compute does not satisfy the precision requirement of the application, we repeat the procedure for the root cube's eight children. That is, we compute eight new approximants A_j based on the points within each of the children of the root cube. We continue to traverse down the tree in this way, till we end up with an adequate approximant or till the least square fit fails.

There are some notes that should be taken regarding the last step of the algorithm: We can speed up the computation of the approximant which we try to store in the root cube by only including the points on level two, in the least square fit. Also, as only non-negative approximants are useful in our setting, we need to replace any negative eigenvalue of the matrix $\mathbf{A}$ in (7.5) by zero: If $\mathbf{D}$ is the eigenvalue matrix of $\mathbf{A}$ and $\mathbf{V}$ its eigenvector matrix, then we have

$$\mathbf{AV} = \mathbf{VD}.$$

If, say

$$\mathbf{D} = \begin{bmatrix} \lambda_1 & 0 & 0 \\ 0 & \lambda_2 & 0 \\ 0 & 0 & \lambda_3 \end{bmatrix}$$

and $\lambda_2 < 0$, we simply replace λ_2 in $\mathbf{D}$ by 0 and define

$$\mathbf{A} = \mathbf{VDV}^{-1}.$$

During registration, we find the approximants we seek by locating the cube which surrounds a given point $\mathbf{p}_j$ and extract the coefficients of the approximant stored in the cube.

7.5 Summary

We have seen in this chapter how we can align two sets of geometric data in an iterative scheme; without regard to which points are corresponding, we try to reduce the distance between the shapes taking small steps in the

direction of the negative gradient of some error function. In the following chapter we will study the third and final solution which instead tries to find and align corresponding points. Thereby it is not as sensitive to the initial positions of the input shapes as the method we have just studied. In fact, the method can be applied regardless of what orientation the input shapes may have.

Chapter 8

A Feature-based Approach

Last in our study of registration methods, we will examine a method which like voting methods, makes no assumptions about the relative initial positions of the input scans. However like most local methods, it tries to solve the problem of finding corresponding points in the input shapes prior to the registration. The method was introduced in [19].

The problem formulation is almost identical to the initial formulation of the registration problem from chapter 4. There it was stated that registration of two sets of point cloud data

$$P = \left\{\mathbf{p}_j^1\right\}_{j=1}^{n_P} \quad \text{and} \quad S = \left\{\Gamma(\mathbf{p}_j^1)\right\}$$

is equivalent to minimizing the sum

$$\sum_{j=1}^{n_P} w_j \left|\left|\Gamma(\mathbf{p}_j^1) - \mathbf{M}\mathbf{p}_j^1\right|\right|^2$$

by varying $\mathbf{M}$.

We will now see how this can be done efficiently by considering a subset of points $P' \subseteq P$ such that $n_{P'} \ll n_P$. Moreover, the problem now is to align two sets of geometric data using the sum

$$\frac{1}{n_{P'}} \sum_{i=1}^{n_{P'}} \sum_{j=1}^{n_{P'}} \left(\left|\left|\mathbf{p}_i - \mathbf{p}_j\right|\right| - \left|\left|\Gamma(\mathbf{p}_i) - \Gamma(\mathbf{p}_j)\right|\right| \right) \tag{8.1}$$

as a measure of how well this is achieved. This measure of deviation is referred to as the *distance root mean squared error*. However, the formulation does not define the function Γ nor does it say anything about how the points in P' are found. We will hopefully tie up the loose ends shortly.

8.1 A robust method

As "usual", the input to this procedure is two shapes of geometric data, P and S, specified either as meshes or as point clouds. No assumptions are made about their initial positions. The idea is to speed-up and make more robust, the schemes introduced in Chapters 6 and 7 by assigning to each point a geometric descriptor. A descriptor is a value which is computed for each point in both sets, and which says something about the behaviour of the surface in a neighbourhood around each point. High dimensional descriptors are likely to be unique for a given point but can be costly to compute. Low dimensional descriptors are often easy to compute but not unique in general. The method we are about to study, use low level descriptors due to their simplicity.

The input shapes are usually large and therefore it is common to restrict the registration to a small set of *feature* points picked from the two shapes based on the descriptor values. More precisely, the feature points are those points that have rare descriptor values. This can, however, lead to a situation where we pick points from P and S that do not correspond well. The registration method which we shall study, tries to avoid such problems.

For each of the feature points we search the entire model S for points with similar (rare) descriptor values. If we relate this work to the function Γ which was introduced in chapter 4, the function should (possibly compute and) compare descriptor values.

The purpose of the approach is to avoid bad correspondences and in addition, we will be able to align the shapes without making any assumptions about their relative initial positions. The final set of corresponding points will be aligned and their positions can be refined by, say, the ICP algorithm.

8.1.1 The Integral Volume Descriptor

Borrowing the notation from [19], the descriptor is equivalent to the volume $V_r(\mathbf{p}_j)$ of the intersection between a ball with radius r centered at $\mathbf{p}_j$, $B_r(\mathbf{p}_j)$, and the interior ($\overline{S}$) of the surface or mesh:

$$V_r(\mathbf{p}_j) = \int_{B_r(\mathbf{p}_j) \cap \overline{S}} dx.$$

It turns out that the volume integral descriptor is also related to the mean curvature H at $\mathbf{p}_j$,

$$V_r(\mathbf{p}_j) = \frac{2\pi}{3} r^3 - \frac{\pi H}{4} r^4 + O(r^5).$$

The expression is obviously the volume of the half ball centered at $\mathbf{p}_j$ corrected to a term involving the mean curvature H at $\mathbf{p}_j$. This makes sense

since if the surface is flat, i.e. the mean curvature is zero, then the descriptor is exactly the volume of the half ball.

In the actual implementation, the descriptor is computed differently. For this, the polygonal mesh is *voxelized*, i.e. converted into a volumetric representation and thus discretized. This process is know as scan conversion [36]. The "building blocks" in this representation are voxels, or volumetric pixels. This way, the interior of the surface $\overline{S}$ and the balls are represented and approximated by grids.

The integral volume descriptor can now be computed for each voxel using a particular kind of an integral transform[1] of the ball grid G_B and the grid $G_{\overline{S}}$ representing $\overline{S}$. Moreover, the value $V_r(\mathbf{p}_j)$ of the volume descriptor at $\mathbf{p}_j$ is approximated by the value of the descriptor at the voxel c containing $\mathbf{p}_j$,

$$V_r(\mathbf{p}_j) \approx \int G_B(\tau) G_{\overline{S}}(c - \tau) d\tau.$$

Gelfand et al. [19] suggest to use the Fast Fourier transform to compute this integral. The value of $G_{\overline{S}}(c)$ is zero if the voxel c does not lie on or inside the boundary, and one otherwise — $G_{\overline{S}}(c)$ is a characteristic function.

Gelfand et al. [19] show that the geometric descriptor is robust to noise and emphasize that it is invariant under rigid transformations. This is a desirable property since this means that we will be able to reuse it after applying interim transformations.

8.1.2 Picking feature points from P

As stated, the feature points of P are those that have uncommon descriptor values among all the n_P descriptor values computed. If we allow many feature points, then we will gain accuracy but the running time of the algorithm will be poor. And vice versa, if we allow only a few feature points, then we will get a fast but less accurate algorithm. Gelfand et al. compute a histogram of descriptor values and classify a point as a potential feature point if it belongs to a bin in the histogram with less than $\frac{n_P}{100}$ points in it.

Also, the method suggests defining a neighbourhood around a feature point from where no other feature points are allowed to be picked. This ensures that a large part of each shape is represented and also that points from the same bins or close to each other are not both chosen as feature points.

The robustness of the feature point selection is further improved by computing several volume descriptors for each point $\mathbf{p}_j$ according to a scaling of the radius of the ball centered at $\mathbf{p}_j$. The set R of different ball radii

[1] In general, an integral transform takes on the form $\int_{x_1}^{x_2} K(x, y) f(x) dx$ where K is a bivariate function called the kernel and the function f is given as input. The output of the transform is a new function depending on y. The integral transform used here is called a convolution.

makes up k different values,

$$R = \{r_1, r_2, ..., r_k\}, \quad r_1 < r_2 < \cdots < r_k.$$

The smallest radius r_1 is set to be ten times the voxel grid resolution, and r_k is set to a fraction of the diameter of the input shape. A feature point is chosen as a point which fulfills the requirements of a feature point as described above for at least two consecutive radii. The feature point's coordinates are stored together with the different volume descriptors V_{r_i}, $i = 1, ..., k$. We also store the radii for which the point was chosen as a feature point.

8.1.3 Potential correspondences

Let us assume that we have found n feature points $\left\{\breve{\mathbf{p}}_j\right\}_{j=1}^{n}$ in P and we want to find their corresponding points in S. First of all, for each point in S we compute the volume descriptor but we do not pick feature points. We only compute the set of different volume descriptors V_{r_i}, $i = 1, ..., k$, for each point in S. Then, for each feature point $\breve{\mathbf{p}}_j$ in P, we choose points $\mathbf{s}_i$ in S which fulfill the requirement

$$|V_{r_{max}}(\breve{\mathbf{p}}_j) - V_{r_{max}}(\mathbf{s}_i)| < \varepsilon. \tag{8.2}$$

Here r_{max} is the largest radius for which $\breve{\mathbf{p}}_j$ was chosen as a feature point and ε is the variation of the descriptor values[2]. These points in S have about the same descriptor values for $r = r_{max}$ as $\mathbf{p}_j$.

We want to avoid picking points that lie close to each other. Thus we group the $\mathbf{s}_i$s into smaller sets in such a way that the points within each set is a maximum distance of $2r_c$ apart. We then select from each of these smaller sets the point $\mathbf{s}_j$ which minimizes

$$|V_{r_{max}}(\breve{\mathbf{p}}_j) - V_{r_{max}}(\mathbf{s}_j)|.$$

That is, these points in S make up a yet smaller set of points $C(\breve{\mathbf{p}}_j)$ of correspondences for $\breve{\mathbf{p}}_j$.

8.1.4 Initializing correspondences

In order to decrease the set $C(\breve{\mathbf{p}}_j)$ further, Gelfand et al. [19] exploit the fact that they only allow for rigid transformations. Thus, a good alignment implies that the distance between two feature points in P should be approximately the same as the distance between their corresponding points. This is why Gelfand et al. use as a measure of how well the shapes are aligned,

[2] It is unclear what variation the authors refer to. In their implementation Gelfand et al. have set $\varepsilon \approx 0.75\frac{\rho}{r_{max}}$ where ρ is the voxel grid resolution.

the distance root mean squared error (8.1). This move also eliminates a potentially expensive correspondence search.

Gelfand et al. emphasize that for a feature point $\breve{\mathbf{p}}_j$, its correct corresponding point is found within a radius of r_c — this follows from the clustering above. Then from this reasoning, two feature points $\breve{\mathbf{p}}_j$ and $\breve{\mathbf{p}}_k$ and their correspondences $\mathbf{s}_j \in C(\breve{\mathbf{p}}_j)$ and $\mathbf{s}_k \in C(\breve{\mathbf{p}}_k)$ must satisfy the inequality

$$\Big|\,||\breve{\mathbf{p}}_j - \breve{\mathbf{p}}_k|| - ||\mathbf{s}_j - \mathbf{s}_k||\,\Big| < 2r_c. \tag{8.3}$$

We form correspondences for each distinct pair of feature points in P',

$$(\breve{\mathbf{p}}_j, \breve{\mathbf{p}}_k), \quad j = 1, \ldots, n-1 \quad k = j+1, \ldots, n,$$

by choosing a pair of corresponding points in S, $\mathbf{s}_j \in C(\breve{\mathbf{p}}_j)$ and $\mathbf{s}_k \in C(\breve{\mathbf{p}}_k)$ so that the left hand side of (8.3) is minimized. This gives us a $O(n^2)$ set of two-point correspondences.

We continue by combining two-point correspondences into four-point correspondences: Given a pair of feature points in the set of two-point correspondences, we pick a different pair of feature points so that the four distinct feature points and their correspondences minimize the error term (8.1). We remove from the first set of two-point correspondences, all correspondences with the same endpoints as the new four-point correspondences. We continue this way until the set of two-point correspondences is empty. Usually this procedure stops at either eight- or sixteen-point correspondences.

A rigid-body transform Ξ is then computed using the resulting eight- or sixteen-point correspondence set such that the coordinate root mean squared error is minimized:

$$\sqrt{\frac{1}{n_{P'}} \sum_{i=1}^{n_{P'}} ||\mathbf{s}_i - \Xi(\breve{\mathbf{p}}_i)||^2}.$$

It is shown by Gelfand et al. that the distance root mean squared error (8.1) is bounded above and below by the error metric[3] above.

The transformation Xi is applied to all the feature points P'. For each of the feature points $\{\breve{\mathbf{p}}_j\}$ which do not yet have a correspondence, a point $\mathbf{s} \in C(\breve{\mathbf{p}}_j)$ is assigned so that $\mathbf{s}$ is closest to $\Xi(\breve{\mathbf{p}}_j)$. We also compute and store for later comparison the error (8.1) using the feature points P' and their current correspondences.

[3] It is useful to settle this relationship since it proves that we can use the error metric in equation (8.1) instead of the coordinate mean squared error. Thereby, we can say something about the quality of the alignment without computing the rigid body transformation.

8.1.5 Determining the best correspondences

At this point we have assigned a corresponding point to each of the feature points of P. For some of the feature points we saw that their corresponding points were just the closest point in S after the rigid transformation had been applied to all of the feature points. We now want to find the *best* corresponding point for all of the feature points. We assume that we have found the best correspondence for $k-1$ of the feature points, and we want to find the best match for the kth feature point, $\breve{\mathbf{p}}_k$.

For each of the m points $\mathbf{s}_k^i \in C(\breve{\mathbf{p}}_k)$, $i = 1, \ldots, m$, that potentially corresponds with $\breve{\mathbf{p}}_k$, we need to check that

$$\left| ||\breve{\mathbf{p}}_j - \breve{\mathbf{p}}_k|| - ||\mathbf{s}_j - \mathbf{s}_k^i|| \right| < 2r_c$$

holds for $j = 1, \ldots, k-1$. For each $\mathbf{s}_k^i$ that passes this test, we compute the error (8.1). We only keep $\mathbf{s}_k^i$ if the error we get is lower than what we had at the end of the section 8.1.4.

We assign in turn each $\mathbf{s}_k^i$ that are kept at this point as a corresponding point to $\breve{\mathbf{p}}_k$ and start over again for the $k+1$th feature point. Each time all feature points are assigned correspondences, we compute the error (8.1) again and compare it to the previous value. If the value is lower than what we had we also compute the coordinate root mean square error. If this is also smaller, we have a new set of correspondences. When every potential correspondence is check this way, we are done.

8.2 Summary

We have seen in this chapter an example of a "global" registration method — it makes no assumptions about the orientation of the two sets of geometric data that we want to align. Also, the shapes may only overlap partially. We saw that in these situations, registration could nonetheless be performed by introducing feature points and determining their correspondences.

This completes our review of different solutions to the registration problem. We have covered four quite different approaches: One closed form solution and three numerical algorithms. In the next chapter we will review how the method from the previous chapter can be implemented. We will be able to see by an example how a particular registration method behaves in practice.

Chapter 9

Implementation

The C++ programming language has been used to implement the registration algorithm outlined in chapter 7. The algorithm was explained in detail in that chapter, so in this chapter we will concentrate on how the different tasks have been implemented.

Before we start, we quickly repeat how the problem was formulated: We are given a point cloud P and a surface Φ_S with known representation. The points in P represent a subset of the surface Φ; typically, the surface Φ is a complete, three dimensional representation of some object and the point cloud P is a partial scan of the same object. The task is to minimize

$$\sum_{j}^{|P|} d(\mathbf{TR}\mathbf{p}_j, \Phi_S)^2 \tag{9.1}$$

by varying the rigid body transformations **T** and **R**. In addition we use approximations to the squared distance function d^2. Recall that for a given point $\mathbf{p}_j$ in P we locate its position in an octree — the d^2-tree — and use the approximant corresponding to this position to find the approximate squared distance to Φ_S. In other words, we make use of the approximation $d(\mathbf{TR}\mathbf{p}_j, \Phi_S)^2 \approx A_j(\mathbf{TR}\mathbf{p}_j^1) = \left(\mathbf{TR}\mathbf{p}_j^1\right)^T \mathbf{Q}_j \left(\mathbf{TR}\mathbf{p}_j^1\right)$, so the function we want to minimize is

$$\sum_{j=1}^{|P|} A_j.$$

The scheme is a two step process; first we need to construct the d^2-tree and store in it the A_j's. Then we need to do the registration, that is find the optimal rigid transformation. We will go through the implementation in the same order, focusing on the d^2 tree construction first, then on the registration.

9.1 The d^2-tree construction

9.1.1 Input

A triangulation making up a complete 3D model of some object's surface should be given as input to the implementation of the d^2-tree algorithm. In addition, we must give as input the number of levels in the tree and the error threshold. The threshold value indicates the amount of error which is tolerated during the computations of the squared distance function approximations. High error thresholds make the method diverge [33].

9.1.2 Data management

The C++ programming language is an object oriented programming language and this fact is exploited. From the outline of the algorithm in chapter 7 we may conclude that we need three classes; an `Octree` class representing cubes in the octree, a `Point3D` class for their lower left corner points and a `Triangle` class. Thus these classes were created. We briefly go through some of the details in the class constructions.

The input variables to the constructor of the `OctreeNode` class are the length of the sides of the cube, the level into which the cube is "born", the coordinates of the cube's lower left corner and the cube's locational code. We will explain the computation and use of locational codes later.

We store every cube (but the root cube) in its parent (as an `OctreeNode` object in a standard template library vector of length eight — the cube is stored together with its seven siblings). There is also a global, two dimensional vector of pointers,

```
std::vector< std::vector<OctreeNode*> > octree
```

whose elements point to the `OctreeNode` objects. Let m be the number of levels in the tree. Then the size of vector `octree`, i.e. `octree.size()`, equals $m + 1$ and the size of each vector `octree[i]`, $i = 0, \ldots, m$, equals the number of `OctreeNode`-objects on level i.

In each cube we store a `Point3D` object which represents the cube's lower left corner point. Input to the constructor of the `Point3D` class is the point's coordinates. In the `Point3D` objects which represent lower left corner points of cubes at even levels, we also store the squared distance between the point and the triangulation.

We also have another global vector of `Point3D` objects,

```
std::vector<Point3D> nodes
```

which represents the points of the triangulated model. Input to the constructor of the `Triangle` class is thus the integer indices `p1,p2` and `p3` into the `nodes` vector.

From the above it is already revealed that it was made use of two of the C++ standard template library's built-in containers, list and vector. In addition, the Template Numerical Toolkit [38] was used. The toolkit provides interfaces of objects, such as arrays, useful in scientific computing. The reason for introducing a second set of containers is that the Template Numerical Toolkit is utilized by the JAMA/C++ library which is used later on for solving linear sets of equations.

9.1.3 Building the d^2-tree

As we recall, we use an octree data structure named the d^2-tree to compute and store approximants to the squared distance function d^2 to Φ_S.

For constructing the tree data structure, we first move the origin of the coordinate system in which the triangulated model is given, to the centroid of the triangulation. Although this was not suggested by [31], it turned out to be crucial for our test cases; for one case in particular, this simple move reduced the condition number of the system matrices involved in the least square fits of the approximants from order of magnitude 10^{18} to 10^{10}.

Further, the triangulation is encapsulated in a bounding box, i.e. the root cube. For this purpose, the coordinates of the two points

$$(x_{max}, y_{max}, z_{max}) \quad \text{and} \quad (x_{min}, y_{min}, z_{min})$$

are found such that for $i = 1, \ldots, n_S$ where n_S is the number of points in the triangulated model,

$$\begin{aligned} x_{max} &= \max(x_i) \\ x_{min} &= \min(x_i) \\ y_{max} &= \max(y_i) \\ &\vdots \\ z_{min} &= \min(z_i). \end{aligned}$$

Then the center of the box is computed as

$$\left(\frac{x_{max} - x_{min}}{2}, \frac{y_{max} - y_{min}}{2}, \frac{z_{max} - z_{min}}{2} \right).$$

The length of the sides of the root cube is set to be large enough to include the space were we expect the point cloud P to lie, i.e near Φ_S. In the test cases we have moved the origin in both coordinate systems to the centroids of the point sets and applied an initial rigid motion to the point cloud P. This way, we have had full control over its whereabouts.

9.1.4 Intersection testing

After having refined the root cube twice, the program refines twice all cubes at even levels that intersect a triangle of the triangulated surface. This process starts at level two. To check for intersection, the method of [2] was employed. The intersection method is based on the Separating Axis Theorem [3] which basically says that two arbitrary, convex polyhedra[1] do not intersect if one can put a line (or plane in 3D) between them. In our setting we test for intersection between triangles and axis-aligned[2] cubes. This implies doing thirteen tests.

The method first translates the origin of the coordinate system to the center of the cube. According to [3], this makes the testing a lot easier. Further, the method searches for a separating axis and terminates as soon as one is found. If all the thirteen tests fail, then the method returns `true`, i.e. the polyhedra intersect.

We have not enough space to include all thirteen tests but we include an easy example; one of the tests examines if the triangle and cube intersect in the x-direction. For this, the most negative and most positive x coordinate of the triangle's vertices are found. Also we need to compute half the length of the cube's side, $\frac{1}{2}$`side_`. Then we test whether the most negative x coordinate of the triangle is greater than $\frac{1}{2}$`side_` or if the most positive x coordinate of the triangle is less than $-\frac{1}{2}$`side_`. If either one of these two tests succeed, then the objects do not intersect in the x-direction. (Remember that the cube's center is the origin.)

We do the intersection testing at each even level, and stop when we have finished testing at level $m - 2$ were m denotes the finest level of the tree. The stop criterion follows naturally from the fact that we always refine twice; if some of the cubes at level $m-2$ intersect a triangle and thereby are refined twice, then we obtain cubes at the *finest* level m.

For each cube that intersects a triangle we also set the private variable `hit_` to `true`; we need to identify these cubes later on.

9.1.5 Squared distance from a point to a triangle, in $\mathbb{R}^3$

At this point in the algorithm, Leopoldseder et el. [31] suggest refining (twice) all cubes that are neighbours of cubes which intersect any given triangle in the triangulation. Those neighbours do not intersect the triangulation themselves and neither will their children. Thus when we refine neighbours we will construct many more cubes at the finest level but none of them will intersect the triangulation.

[1] Plural of polyhedron; geometric objects "with flat faces and straight edges" [45].

[2] An axis-aligned cube have face normals which coincides with the coordinate axes [3, 45].

Later on, we shall compute exact squared distances to the triangulation for each of the lower left corner points of the cubes at the finest level which intersect a triangle. Why not then compute squared distances to the triangulation for those cubes while we know which triangle(s) they may intersect?

It follows from the above that we compute squared distances while we refine cubes that intersect the triangulation. This way, we only need to check for intersection between cubes at the finest level and a small amount of triangles (those that intersect the cubes' grandparent). If we detect that a cube at the finest level intersects a triangle, we compute the squared distance from the lower left corner point of that cube, to the triangle.

For distance calculations, the method of Eberly [15] was employed. This method computes the squared distance from a given point to a given triangle in $\mathbb{R}^3$. The points we are given are the lower left corners of the cubes at the finest level in the octree. Again, we will not state the whole implementation, but review the ideas[3].

Consider a parametrization of the triangle in question: If the three corners of the triangle are $\mathbf{c}_1$, $\mathbf{c}_2$ and $\mathbf{c}_3$, then the triangle can be written as

$$\mathbf{c}_1 + s(\mathbf{c}_2 - \mathbf{c}_1) + t(\mathbf{c}_3 - \mathbf{c}_1) = \mathbf{c}_1 + s\mathbf{v}_1 + t\mathbf{v}_2$$

for real values s and t such that $(s,t) \in [0,1] \times [0,1]$ and $s + t \leq 1$. The squared distance between a given point $\mathbf{p}$ and the triangle can be found as the minimum of the function

$$\Psi(s,t) = ||\mathbf{c}_1 + s\mathbf{v}_1 + t\mathbf{v}_2 - \mathbf{p}||^2, \quad (s,t) \in [0,1] \times [0,1], s + t \leq 1. \tag{9.2}$$

We note that $\Psi \geq 0$ for all $s, t \in \mathbb{R}$.

By standard procedures, the global minimum of Ψ, $s, t \in \mathbb{R}$, can be found be setting the partial derivatives $\frac{\partial \Psi}{\partial s}$ and $\frac{\partial \Psi}{\partial t}$ of Ψ, equal to zero. The parameters s and t minimizing Ψ are then found as the solutions of the set of equations

$$\begin{bmatrix} ||\mathbf{v}_1||^2 & \mathbf{v}_1\mathbf{v}_2 \\ \mathbf{v}_1\mathbf{v}_2 & ||\mathbf{v}_2||^2 \end{bmatrix} \begin{bmatrix} s \\ t \end{bmatrix} = - \begin{bmatrix} \mathbf{c}_1 \cdot \mathbf{v}_1 - \mathbf{v}_1 \cdot \mathbf{p} \\ \mathbf{c}_1 \cdot \mathbf{v}_2 - \mathbf{v}_2 \cdot \mathbf{p} \end{bmatrix},$$

that is

$$s = \frac{\mathbf{v}_1 \cdot \mathbf{v}_2(\mathbf{c}_1 \cdot \mathbf{v}_2 - \mathbf{v}_2 \cdot \mathbf{p}) - ||\mathbf{v}_2||^2(\mathbf{c}_1 \cdot \mathbf{v}_1 - \mathbf{v}_1 \cdot \mathbf{p})}{||\mathbf{v}_1||^2||\mathbf{v}_2||^2 - (\mathbf{v}_1 \cdot \mathbf{v}_2)^2}$$

and

$$t = \frac{\mathbf{v}_1 \cdot \mathbf{v}_2(\mathbf{c}_1 \cdot \mathbf{v}_1 - \mathbf{v}_1 \cdot \mathbf{p}) - ||\mathbf{v}_1||^2(\mathbf{c}_1 \cdot \mathbf{v}_2 - \mathbf{v}_2 \cdot \mathbf{p})}{||\mathbf{v}_1||^2||\mathbf{v}_2||^2 - (\mathbf{v}_1 \cdot \mathbf{v}_2)^2}.$$

Since Ψ is defined on a restricted domain, we also need to check the boundary. First we take some general notes on the result.

[3]The actual implementation can be found at `http://www.geometrictools.com`.

We realize that $\Psi(s,t)$, when $(s,t) \in [0,1]\times[0,1]$ and $s+t \leq 1$, equals the squared distance from **p** to the point on the triangle which is the closest one to **p** of all the points on the triangle. This point is located in the same plane as the triangle, either within, or on, the triangle's boundary. Of course, if **p** is in the same plane as the triangle and also is located inside the triangle, then the closest point to **p** on the triangle, is **p** itself.

We also note that if **p** is in the same plane as the triangle, then the global minimum of Ψ is zero — regardless of whether **p** is inside or outside of the triangle's boundary. The vector sum $\mathbf{c}_1 + s\mathbf{v}_1 + t\mathbf{v}_2$ will in this case equal **p** and consequently, $\Psi(s,t)$ will be zero. Otherwise, if **p** is not in the same plane as the triangle, then the global minimum of Ψ will be the squared distance from **p** to the projection of **p** into the same plane where the triangle is found.

When s and t are found from the equations above, we get the *global* minimum of Ψ over $\mathbb{R}^2$. It may be the case that $s,t \in [0,1] \times [0,1]$ and that $s+t \leq 1$. If so, then we are done; the value $\Psi(s,t)$ is what we seek. However, in general we need to check the boundary; it may very well be the case that $s+t > 1$ or $s+t < 0$, and $s,t \notin [0,1]\times[0,1]$. We will then need to do some investigation to find out where, on the triangle's *boundary*, the closest point to **p** on the triangle is located. In other words, we need to find the values of s and t, with the restrictions $s+t \leq 1$ and $s,t \in [0,1] \times [0,1]$.

The boundary of the triangular region given by $s+t \leq 1$ and $s,t \in [0,1] \times [0,1]$ consists of three straight edges. These are $s = 0, t = 0$ and $s+t = 1$. This gives us three univariate functions to consider; $F_1(0,t)$, $F_2(s,0)$ and $F_3(s,1-s)$, respectively. Depending on the sum of s and t and their signs, we find the minimum of one of these three functions. We know we are seeking a minimum from the fact that

$$\Psi''(0,t) = 2||\mathbf{v}_2||^2 > 0,$$

$$\Psi''(s,0) = 2||\mathbf{v}_1||^2 > 0$$

and

$$\Psi''(s,1-s) = ||\mathbf{v}_1 - \mathbf{v}_2||^2 > 0.$$

There are a total of seven cases we need to handle. One of them is the "simple" case where $s+t \leq 1$ and $s,t \in [0,1]\times[0,1]$; in this case the method simply returns $\Psi(s,t)$, as stated above. The idea in the remaining six cases is to find the level curve of Ψ which touches one of the boundary edges. Since Ψ is a paraboloid, the level curves are ellipses. Finding the level curve of interest implies first to figure out which of the three edges is the closest one to the global minimum. Thereafter, we solve either $F_1'(0,t) = 0$, $F_2'(s,0) = 0$ or $F_3'(s,1-s) = 0$.

We look at this by an example. If say, $s+t < 1$, $0 < s < 1$ and $t < 0$, then the point we seek is found on the triangle boundary corresponding to the

edge $t = 0$. This follows, since in this case the triangle edge corresponding to the edge $t = 0$, i.e. $\mathbf{c}_1 + s\mathbf{v}_1$, is the closest one, out of the three triangle edges, to the point $\mathbf{c}_1 + s\mathbf{v}_1 + t\mathbf{v}_2$ giving the global minimum. Or, in other words, the level curve of $\Psi(s, t)$ which we now seek, will touch the triangle at the edge $\mathbf{c}_1 + s\mathbf{v}_1$. Thus, in this case we solve $F_2'(s, 0) = 0$.

When solving this equation, we may find that s is less than zero, between zero and one, or greater than one. If say, $s < 0$, then since the function in question reaches a minimum for this value and since its graph is a parabola, it is decreasing for a lower value of s. Consequently, for s in $[0, 1]$, the function is increasing. So, in that case, the minimum we seek is found when s is zero. Similarly, if we find that s is greater than one, then the parabola is decreasing for $s \in [0, 1]$ and the minimum we seek is found for s equal to one. The value for t is found as $1 - s$.

9.1.6 Producing the last cubes in the tree

The next step is to refine twice all cubes that are neighbours of cubes which intersect the triangulation. Cubes which intersect the triangulation are easily identified since the value of their private boolean variable `hit_` is set to `true`. It is pointed out by [31] that several neighbourhood definitions are possible — the definition of a neighbour as a cube which is in the 1-ring of direct neighbours is used in this setting. Specifically, the neighbours are found using the algorithms of Bhattacharya [9].

The algorithms of Bhattacharya are based on tables and on the fact that each of the 26 direct neighbours can be classified as either an edge neighbour (i.e. the cubes share an edge), a face neighbour (i.e. the cubes share a side or "face") or a vertex neighbour (i.e. the cubes share a vertex). Each of the three types of neighbours requires its own neighbour-finding algorithm and there are one or two tables associated with each algorithm. Fundamental for all three algorithms are also the existence of locational codes.

Each cube has its own locational code which can be thought of as an array of integers where each integer is a value between zero and seven. The length of the code, i.e. the length of the array, equals the level at which the cube is located, plus one. The locational code of the root cube is set to $[0]$. The root cube is located at level zero so the length of the array should be $0 + 1 = 1$, which is certainly the case.

To see how the rest of the codes are found, consider first figure 9.1. For each cube in the octree we number its eight children as is done in the figure. If the parent of the eight cubes in figure 9.1 is say, the root cube, then the eight cubes are given the codes $[0, 0]$, $[0, 1]$, $[0, 2]$, $[0, 3]$, $[0, 4]$, $[0, 5]$, $[0, 6]$ and $[0, 7]$. Further, if the cube with locational code $[0, 7]$ is refined, then its eight children are given the codes $[0, 7, 0]$, $[0, 7, 1]$, $[0, 7, 2]$, $[0, 7, 3]$, $[0, 7, 4]$, $[0, 7, 5]$, $[0, 7, 6]$ and $[0, 7, 7]$.

Given a cube which does not intersect the triangulation, we find its

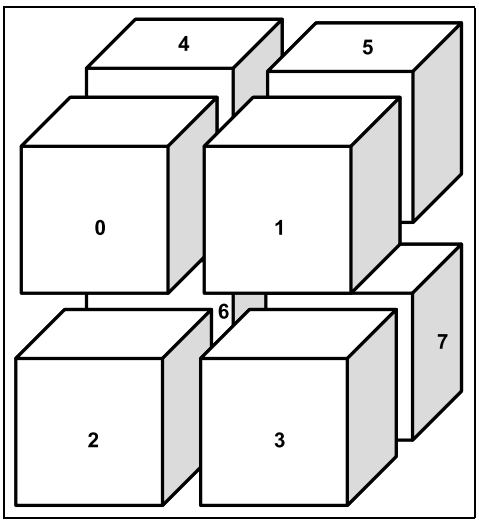


Figure 9.1: We see the numbering of the eight children of a given cube, the children are separated to enhance the visibility.

neighbours by computing their locational codes, *independent* of whether or not the code is valid. Computing codes that may not be valid is only for convenience. The algorithm is given in pseudo code and had to be "translated" into C++. Computing possibly invalid codes and then afterwards validating it, were found to be the easiest way out.

A valid code has a zero in its first entry. A code that does not have a zero in its first entry refers to a cube inside a neighbour of the root cube. Only valid codes are useful to us, and we must therefore always check if the computed code is valid. Also we must check if the neighbour actually exists — although the first entry is zero, the cube may be non existent since we have not refined every cube in the octree. If a valid code is found, if the neighbour exist and if that neighbour intersects the triangulation, only then do we refine the cube in question.

We will not pose all three algorithms here, nor all the tables[4], but we will review how a face neighbour is found. (This choice was made arbitrarily.) Input to the face-neighbour finding algorithm is the locational code of the cube in question, the direction in which we seek and two tables. The six directions are given names in [9] but are simply referred to as $0, 1, 2, 3, 4$ and 5 in the program. Which of the numbers should be up, down, left, right and so on is not important as long as one keeps track of which way is which. The directions serve as part of indices into two-dimensional tables, so giving them numbers as "names" were found convenient.

Continuing with this simple example, we refer to figure 9.1 again. Say that the cube C marked 0, has locational code $[0, 5, 0]$ and that we seek the

[4]Two of the tables contained an erroneous entry!

locational code of the face neighbour located to the left (not present in the figure). This corresponds to the direction 4 in the program. The input to the algorithm is then an integer array `code` $= [0, 5, 0]$, the number 4, a table of boolean values and a "neighbour" table. Output is the locational code, or array `ncode`, of the face neighbour.

The boolean table tells whether or not C and its neighbours have the same parent. The neighbour table simply contains the rightmost integer of the locational code of the face neighbour in each direction. Hence, there are six entries in the neighbour table of every cube, one for each direction.

We first look up in the boolean table, since if C and its neighbour have the same parent, only *one* look up in C's neighbour table is required; the rest of the code of the neighbour is equal to C's own code. As an example, if we look at the boolean table entry of the cube marked 0, corresponding to the direction 4, we find that the neighbour does not have the same parent. Thus, in this step we can only retrieve the *last* entry of the neighbour's code (by looking up in the neighbour table). From this we get the integer 1.

Face neighbour algorithm of [9] written in C++. *Input is integer array* `code`, *search direction* `d`, a *boolean table* `N` *and a neighbour table* `A`. *Output is integer array* `ncode`, *i.e. locational code of the face neighbour in the direction* `d`.

```
1  i = code.size();
2  while(i > 0){
3      if( !N[code[i]][d] ){
4          //the neighbour is outside C's parent
5          ncode[i] = A[code[i]][d];
6          --i;
7      }else{
8          ncode[i] = A[code[i]][d];
9          --i;
10         for(; i>=0; --i)
11             ncode[i] = code[i];
12     }
13 }
```

When given the complete tables and by following the algorithm above, one finds that the output of the algorithm with $[0, 5, 0]$ and 4 as input, is $[0, 4, 1]$. In other words, the locational code of the face neighbour of $[0, 5, 0]$ in the direction 4 is $[0, 4, 1]$, which is a valid code (the first entry is zero). It remains only to see if this cube exist, i.e. to check whether the cubes $[0]$ and $[0, 4]$ have been refined. In the actual program, this is done by checking whether or not their private arrays of children are empty.

9.1.7 Sorting and sweeping

When the refinement procedure is over, we must sort the cubes at each even level so that we can apply the sweeping method of Zhao [47] at each

of those levels. Recall from chapter 7 that the sweeping method solves, in particular, the following Eikonal equation (numerically):

$$|\nabla d(\mathbf{x})| = 1, \quad \mathbf{x} \in \mathbb{R}^3,$$

where $d(\cdot)$ denotes the distance function and with the boundary condition $d(\mathbf{x}) = 0$ (when $\mathbf{x} \in \Phi_S$). Recall also that in three dimensions, sweeping in eight directions guarantee a numerical solution as accurate as the solution we obtain if we let the iteration converge [47]. Each sweep implies traversing the grid points through a number of x, y and z parallel layers in turn. Moreover, we cover two directions by sweeping both ways for each ordering of the cubes. Thus at each even level we must sort the cubes four times.

The d^2-tree algorithm of [31] implies sorting the cubes at even levels, one z parallel layer at a time from top to bottom of the tree, and storing all four orderings each time. The sort routine must of course be given appropriate predicates. Then the sweeping method is applied start at the bottom of the tree, at the finest level.

However, sorting all cubes at each level in *one* operation simplifies this particular stage greatly, and the job is still done within acceptable time limits — at least for the maximum number of cubes (about eleven millions) that we were able to store in memory. We can thereby sort *and* sweep from bottom of the tree to the top, and work with one single array of pointers at each level. Consequently, we increase the running time but save a lot of memory.

It was also found that, for each cube at a given level, if all their valid face neighbours are stored the first time we sweep through this level, then the other seven sweeps through the same grid of cubes run quite fast. (The grid neighbours of a cube's lower left corner point are the lower left corner points of the cube's face neighbours.)

Recall from chapter 7 that during sweeping, we only update the distance information if the value `d` we try to store is less than the value `dist_` which is already stored. The function `setSqDist2Tri` in the `Point3D` class handles this condition in the obvious way;

```
1 dist_ = d < dist_ ? d : dist_;
```

9.1.8 Fitting the approximants

Finally, for the computations of the coefficients defining the approximants of the squared distance functions, all even level points (lower left corners) which are located within the given cubes, are considered. The octree consists of $m+1$ levels, so least square fits are computed for every cube at level

$0, 1, \ldots, m-2$ that contains more than eight points — eight points are not enough to determine the ten unknowns. Each least square fit imply solving a linear set of equations with ten unknows, i.e. the coefficients.

For solving the linear sets of equations, the JAMA/C++ linear algebra package [38] is used. The JAMA/C++ linear algebra package is a translation into C++ of the Java Matrix Library providing the eigenvalues and eigenvectors of a matrix, its singular value decomposition and its Cholesky factorization, if it exists. As stated, the library utilizes the Template Numerical Toolkit.

When solving the (many) linear sets of equations mentioned above we call for the singular value decomposition $\mathbf{USV}^T$ of the system matrices. If the vector $\mathbf{b}$ is a known vector, more specifically the right hand side of a given set of equations, then the solution vector is computed as

$$\mathbf{VS}^{-1}\mathbf{U}^T\mathbf{b}. \tag{9.3}$$

To get the singular value decomposition of a system matrix $\mathbf{A}$, we first need to create an instance, call it `asvd`, of the template class `JAMA::SVD` with $\mathbf{A}$ as input to the `SVD` class constructor. See the first line of the code snippet below. We then declare the matrices $\mathbf{U}$, $\mathbf{V}$ as two empty matrices (line two and three in the same excerpt) before we store there entries by calling for the `getU` and `getV` routines:

```
1 JAMA::SVD<double> asvd = JAMA::SVD<double>(A);
2 TNT::Array2D<double> U;
3 TNT::Array2D<double> V;
4 asvd.getU(U);
5 asvd.getV(V);
```

The matrix $\mathbf{S}$ is diagonal and instead of storing a 10×10 matrix with ninety zero entries, we only store its ten diagonal elements (the singular values of $\mathbf{A}$):

```
1 TNT::Array1D<double> s;
2 asvd.getSingularValues(s);
```

To compute the solution vector we see from the expression (9.3) that we need to compute $\mathbf{VS}^{-1}$ and thus we also need the inverse of $\mathbf{S}$. But since $\mathbf{S}$ is square and diagonal, its inverse is just the inverse of its diagonal elements. Since the diagonal elements (singular values of $\mathbf{A}$) are positive by definition, their inverses are easily computed.

Further, computing $\mathbf{VS}^{-1}$ boils down to a scaling of the columns of $\mathbf{V}$ by the inverse of the diagonal elements of $\mathbf{S}$. We store the product $\mathbf{VS}^{-1}$ in $\mathbf{V}$:

```
for(int col=0; col<V.dim2(); ++col)
    for(int row=0; row<V.dim1(); ++row)
        V[row][col] *= s[col];
```

Next, we need to compute the product of the matrices $\mathbf{VS}^{-1}$ and $\mathbf{U}^T$. The JAMA/C++ library contains a routine for computing a matrix product so all we need is the transpose of **U**. The transpose of a matrix, on the other hand, was not a routine provided by the library, so it was added. Doing so was straightforward; the routine takes as input the matrix **U** of which we want to compute the transpose, and returns a copy of a new matrix whose (i, j)-th entry is the (j, i)-th entry of **U**.

Finally, we need to do matrix-vector multiplication, i.e. we need to multiply the matrix $\mathbf{VS}^{-1}\mathbf{U}^T$ with the right hand side vector **b**. A routine for this also had to be added, see figure 9.2.

```
//Perform matrix-vector multiplication, Av, and store
//the result in a new vector C. Return a copy of C.
template <class T>
Array1D<T> vecmult(const Array2D<T> &A, const Array1D<T> &v){

    if(A.dim2() != v.dim1()){
        //take appropriate action, e.g.
        return Array1D<T>();
    }

    //A is M x N
    int M = A.dim1(); int N = A.dim2();

    Array1D<T> C(M);

    for(int i=0; i<M; ++i){
        T sum = 0;
        for(int j=0; j<N; ++j)
            sum += v[j]*A[i][j];
        C[i] = sum;
    }
    return C;
}
```

Figure 9.2: A routine for multiplying a matrix with a column vector.

9.2 Registration

9.2.1 Matrix functions

When the d^2-tree construction is finished, the registration procedure is finally called for. Registration implies solving a linear set of equations as

given in [33]. However, there are two concerns that require special attention: Firstly, the rotation matrices which are used are linearized and hence only valid at small movements. Secondly, as we saw in chapter 7, for each point $\mathbf{p}_j$ in P we must do a point location in the d^2-tree and pick out the "adequate" approximant. The adequate approximant is found within a cube that surrounds $\mathbf{p}_j$. Also, and importantly, the approximant is the best one out of all the available approximants in the cubes surrounding $\mathbf{p}_j$.

The so-called Armijo condition is used to check whether or not the computed transformation represents a small motion. When we compute a transformation taking the point cloud from one point to another, then the Armijo condition requires that we also compute the following quantities: The sum of squares at both locations and the gradient vector of the error function at the location we move *from*. This is straightforward. The difficult part turns out to be how to compute a fractional step which is what we must do if the rotational approximations are no longer valid.

If the transformation is too large, it is suggested by [33] to use the method of Alexa to compute a fractional rotation $\mathbf{R'}$ and then compute the fractional translation $\mathbf{T'}$ as

$$(\mathbf{R} - \mathbf{I})^{-1}(\mathbf{R'} - \mathbf{I})\mathbf{T}, \quad \mathbf{I} \text{ being the identity matrix.}$$

However, the matrix $(\mathbf{R}-\mathbf{I})$ did not have an inverse in our test. Fortunately, the method of Alexa applies to rigid transformation matrices as well so it was decided to apply the method to the whole transformation $\mathbf{TR}$ instead.

As we may remember from chapter 7, if the motion $\mathbf{TR}$ is found to be too large, the method of Alexa computes a $\frac{1}{n}$ fraction of $\mathbf{TR}$ as

$$e^{\frac{1}{n}\log \mathbf{TR}}.$$

Alexa has given methods for computing both the exponential and logarithm of a matrix but suggests using Rodrigues' formula when we deal with rigid transformations. The methods of Alexa were nevertheless tested and were found to be quite unstable. Especially the matrix square root function involved in the computation of the matrix logarithm, was not suited for our test data.

Rodrigues' formula gives an expression for computing the exponential map of a *rotation* matrix. There are also formulas for computing the exponential map of a rigid transformation matrix and the Rodrigues' formula are embedded in these formulas. But first we are supposed to compute the logarithm of the transformation matrix $\mathbf{TR}$. Fortunately, there are also formulas for this. These formulas imply computing the logarithm of the "rotational" part of $\mathbf{TR}$ [35], separately. Unfortunately, this requires that we have a "true" rotation at hand, i.e. an orthogonal matrix representing the rotation. Thus, we cannot use the formulas "as they are" since the rotation matrices we use, are approximations.

For computing the logarithm of **R** we have turned to the definition of the logarithmic function for a general matrix **A**,

$$\log \mathbf{A} = \sum_{n=1}^{\infty} (-1)^{n+1} \frac{(\mathbf{A} - \mathbf{I})^n}{n}$$

which is known to converge if $||\mathbf{A} - \mathbf{I}|| < 1$ [35]. This is the case for our rotation matrices as we remember that in general, they are on the form

$$\begin{bmatrix} 1 & -\gamma & \beta & 0 \\ \gamma & 1 & -\alpha & 0 \\ -\beta & \alpha & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

with $0 < \alpha, \beta, \gamma < 1$. By trial and error, including the first fifty terms of the sum have shown to give a sufficiently close approximation to the true matrix. We notice that the sum does not converge fast. However, the routine is not called for very often, at most once for each time we solve for the transformation parameters (i.e. at most eight times[5]). Also, for our 4×4 matrices the time it takes to compute the sum is negligible compared to the overall running time of the algorithm, so the slow convergence was considered not to be a big issue.

Finally, to compute the exponential of $\mathbf{S} = \frac{1}{n} \log \mathbf{TR}$, we use the exponential map formula given in [35],

$$e^{\mathbf{S}} = \begin{bmatrix} \mathbf{B} & \mathbf{Dv} \\ 0 & 1 \end{bmatrix}.$$

The vector **v** represents the translational part of **TR**. Further, letting a vector $\mathbf{r} = [\alpha, \beta, \gamma]^T$ and a matrix **R** be the leading principal 3×3 submatrix of **TR** (i.e. its rotational part), the 3×3 matrices **B** and **D** are given as

$$\mathbf{I} + \frac{\mathbf{R}}{||\mathbf{r}||} \sin ||\mathbf{r}|| + \frac{\mathbf{R}^2}{||\mathbf{r}||^2} (1 - \cos ||\mathbf{r}||)$$

and

$$\mathbf{I} + \frac{\mathbf{R}}{||\mathbf{r}||^2} (1 - \cos ||\mathbf{r}||) + \frac{\mathbf{R}^2}{||\mathbf{r}||^3} (||\mathbf{r}|| - \sin ||\mathbf{r}||),$$

respectively.

9.2.2 Point location

Each time the gradient vector is to be computed or when the linear set of equations is solved, we need to do a point location. This follows from the

[5]This number was also found by trial and error, guided by the results of [33].

fact that the approximations we use to the squared distance function depend upon the point in question, see section 7.2.1. A point location implies finding out where in the d^2-tree a given point is located. Or rather, to find out which of the cubes surrounding the point, stores the best approximant to the squared distance function. We assume that the point is inside the root cube.

First, we check whether an adequate approximant is stored in the root cube. If an adequate function is found, then we use this function. Otherwise, the center of the root cube is computed. Then a vector extending from the root cube center to the point is computed and its signs are examined. From this we can deduce which of the eight children the point is located inside. When the correct child is found, we check whether an appropriate approximant is stored there. We continue this way until we reach the bottom of the octree. See figure 9.3 on the following page for an outline of the procedure in two dimensions.

Under the d^2-tree construction above, we always store in a certain cube, the approximant which is the best one out of that computed for the cube itself and the one stored in the cube's parent. This way, during point location, we know that if we reach the bottom level without having found an appropriate approximant, we can at least rest assured that the best approximant *available* is stored there.

It should be noted that since the part of the algorithm that deals with the registration follows an iterative scheme, the procedures involved in registration (like point location and the routines for computing the matrix functions) are executed multiple times.

9.3 A test case

We show how the program performs when given as input, a complete triangulated model of a checkerboard's knight ($28,716$ points) and a scan (point cloud) from an arbitrary side of the knight. The point cloud makes up a subset ($8,375$ points) of the complete model. As will be evident from the figures, the scan is organized as a mesh (triangulation) by The Polygon Editing Tool but we treat it as a point cloud. The model of the knight was also made in The Polygon Editing Tool, from nine partial FINE mode scans taken by the VI-910.

We constructed a thirteen level octree with error threshold 0.001. That is, during the computation of the approximants we did not tolerate an error greater than 0.001. The bounding box needed to made three times as large as the height of the chess piece. As stated, these settings resulted in the construction of about eleven million cubes. Further, we let the registration iterate eight times.

The relative initial positions of P and Φ_S can be seen in figure 9.4. Both

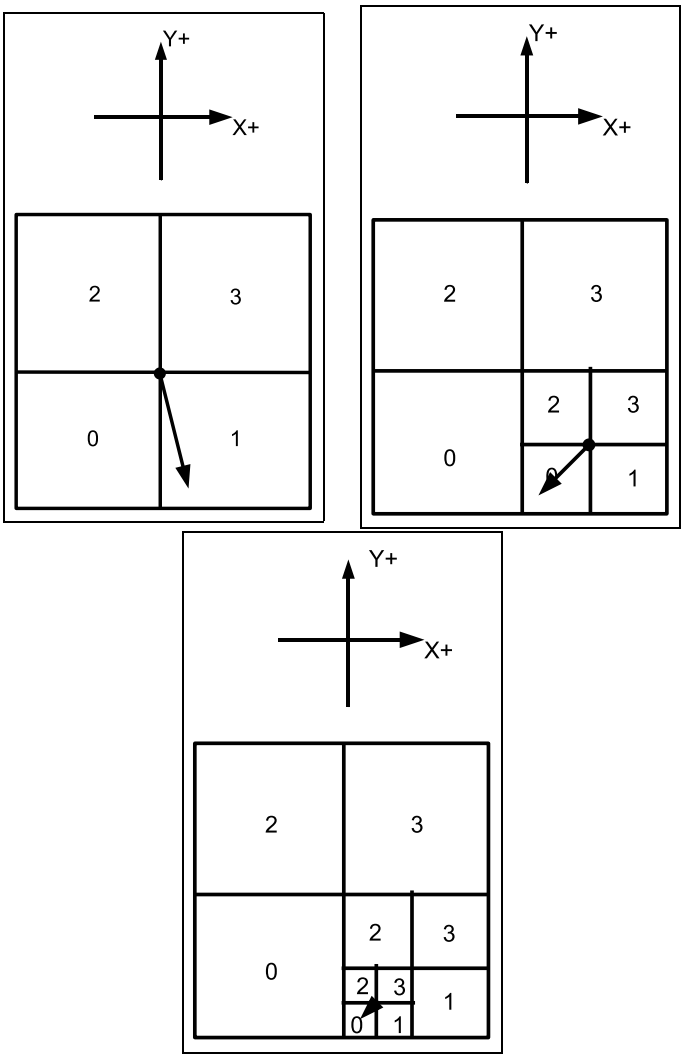


Figure 9.3: We examine the signs of the vector from the middle of the cube up to the point (at the tip of the vector), to figure out which of the four children (squares) of the enclosing cube the point is in. In the top left figure we see that the vector's x component is positive while the y component is negative. From this we deduce that the point is in the square marked '1'. Following the same procedure again, in the top right figure with both x and y components negative, we see that the point is in the square marked with a '0' — that is, the zero'th child of the cube marked '1' in the top left figure. In the last stage of the point location algorithm (bottom figure) we have either reached a childless square or a square which stores an adequate squared distance function approximant.

shapes contain a certain amount of noise.

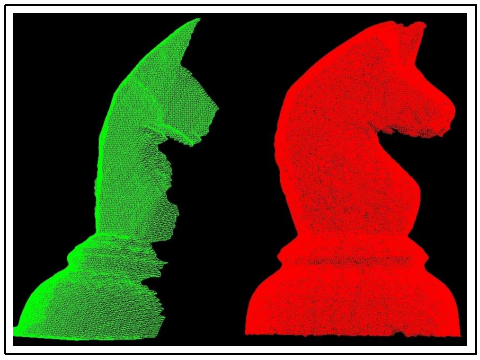


Figure 9.4: The initial positions of the point cloud P, in green, and the completed model surface Φ_S, in red.

Each time we solve for the transformation parameters, we move the point cloud P correspondingly. If we store the different positions to which the point cloud P moves, we obtain a series of positions which, when displayed, can help us to get a better "feel" for the flow of the algorithm — the different "leaps" made by the point cloud P towards the triangulated surface Φ_S can be seen in figure 9.5 and figure 9.6. The final position of the point cloud and the surface Φ_S, i.e. the final registration, can be seen in figure 9.7. For visualizing the meshes, OpenGL has been utilized.

The total running time of the program on an Intel(R) Xeon^{TM} 3.6 GHz CPU, with the model of the checkerboard's knight and the partial scan given as input, is about three minutes. In this particular test case, *all* (lower left corner) points inside a given cube were used to produce its associated approximant to the squared distance function. Thus, no attempts were made to try to improve the running time by only including points of, say, the cube's grandchildren. We discuss this matter further in the next chapter.

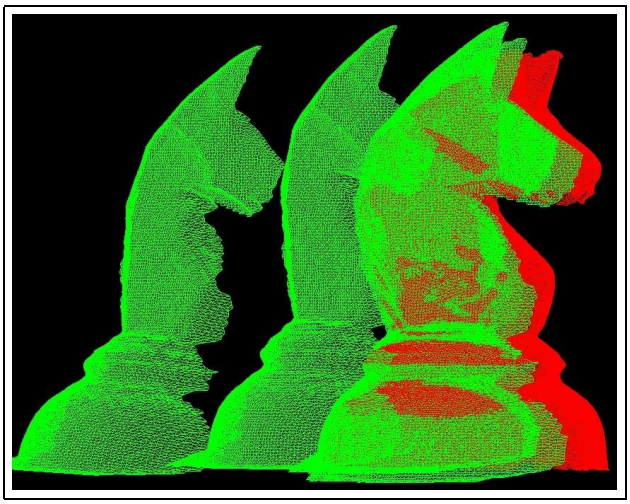


Figure 9.5: We see the (triangulated) point cloud P, in green, approaching the triangulated model. The first four positions of the point cloud is included.

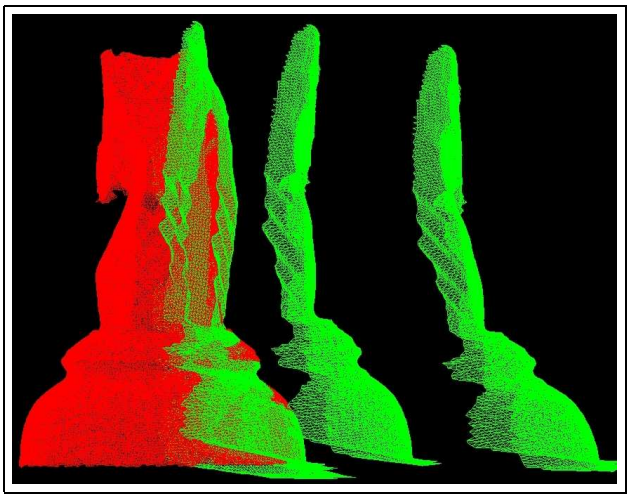


Figure 9.6: We see the (triangulated) point cloud P, in green, approaching the triangulated model from the opposite side of the previous view.

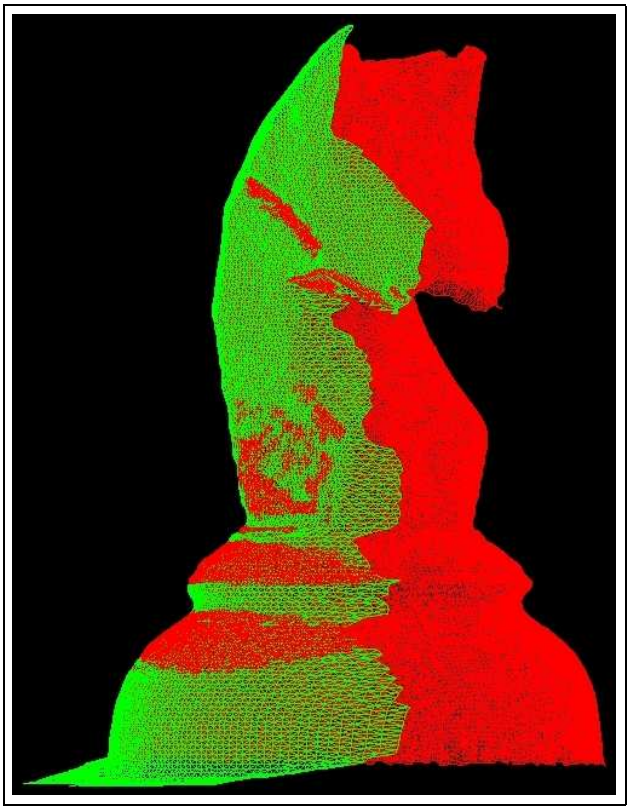


Figure 9.7: The positions of the point cloud P and the complete model after eight iterations of the algorithm. Both sets of data can be seen to contain some amount of noise.

Part IV

Summary

Chapter 10

Discussion & Conclusions

In this last chapter we discuss properties of, and share thoughts on, the registration method of Mitra et al. [33] which we have spent time studying and implementing. Also, we indicate ways to change the method so that it may provide better alignment of the input shapes.

10.1 Considerations on the running time

We start by picking up the thread from the end of the previous chapter. We referred to the running time of the algorithm in the light of a particular set of input data. In that particular case, much time is spent on sorting. The size of the enclosing cube, the number of levels (twelve) and the error threshold resulted in about eleven million cubes produced at the finest level. All these cubes were sorted simultaneously, in a straightforward manner.

The sorting could have been done more efficiently. Mitra et al. [33] try to explain one such way, but at the cost of internal memory. The large number of cubes required to obtain the wanted level of accuracy already occupied a large part of memory. Thus, in lieu of their way of sorting top to bottom, storing four copies of the cubes at each level along the way, a brute-force approach was preferred. In the end, running time was not an important issue in either of the tests.

We can, however, reduce the running time considerably by allowing for slightly bigger cubes at the finest level and still get a reasonably good result. This can be done by either reducing the number of levels while keeping the size of the enclosing cube fixed, or by keeping the number of levels fixed while increasing the size of the enclosing cube. Note that reducing the number of points representing the model and point cloud does not affect the running time at all. It is the area of the model surface that determines how many cubes are produced and this number in turn, is what affects the execution time of the current program.

10.2 Convergence

Another aspect to consider is in what situations the algorithm converges to a correct situation. Some observations on this matter are already made in [33]. To sum up, Mitra et al. state that the method converges for a larger set of initial positions than the ICP algorithm and also that the initial positions for which the method converges, are clustered. This last property has also been observed in all test cases. It also seems like a translational displacement between the model and the partial scan has little effect on the convergence, whereas a rotational displacement plays a central role.

Let us try to figure out why the initial rotational displacement is of such importance to the final alignment. For this we have given a partial scan of a knight (from a chess game) as input to the d^2-tree construction algorithm, and constructed a ten level octree with error threshold 0.001. According to our previous notation, this scan represents the surface Φ of our model. Further, we have given a copy of the same scan as input to the registration algorithm and we will think of this copy as our point cloud P.

The idea is to apply successive rotations to the point cloud P about the z axis, and to sample the error function at each orientation *without* doing any registration. That is, we apply rotations $\mathbf{R}_{z,i}$,

$$\mathbf{R}_{z,i} = \begin{bmatrix} \cos(\theta_i) & -\sin(\theta_i) & 0 \\ \sin(\theta_i) & \cos(\theta_i) & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad \theta_i = \frac{i\pi}{180},$$

for $i = 180, 179, \ldots, -178, -179$ to all the points in P and after each rotation we use the points in their current positions to compute the sum of squared residuals. The motion applied to the point cloud P is illustrated in figure 10.1.

The sum of squares is plotted in figure 10.2 as a function of the rotational angle θ_i. Recall that the method follows a gradient descent approach in the search for the optimal transformation parameters. Assume now that we start the registration procedure with the above data after having first applied an initial rotation of say, 120 degrees about the z axis. We see from the plot that at the moment we start registration, we are to the right of a (global) maximum (the point $(90.0, 125)$) and cannot possibly reach optimal alignment (global minimum of the plotted function). It follows from this that if the rotational displacement is large when the registration procedure starts, then we should expect to get stuck at a local minimum, say the point $(166, 57.3)$ in the same figure.

The registration procedure solves for six transformation parameters, and not only one rotational angle. Thus, it is not possible to see from the plot in figure 10.2 where we end up if we start registration from a position corresponding to a point between the global maximum and the local minimum $(166, 57.3)$.

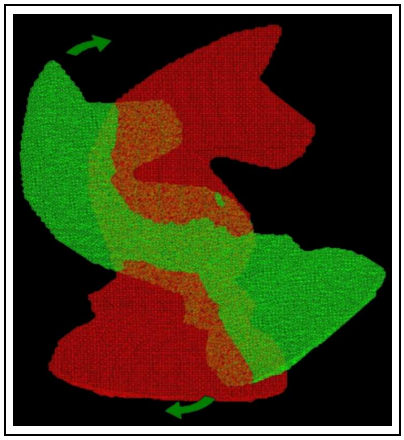


Figure 10.1: The figure indicates how the point cloud, in green, is rotated in the xy plane around the z axis. We apply 180 anti-clockwise rotations ($\theta = 180, 179, \cdots, 1$ degrees) followed by 180 clockwise rotations ($\theta = 0, -1, \cdots, -179$ degrees). The scan of the model (in red) stays fixed.

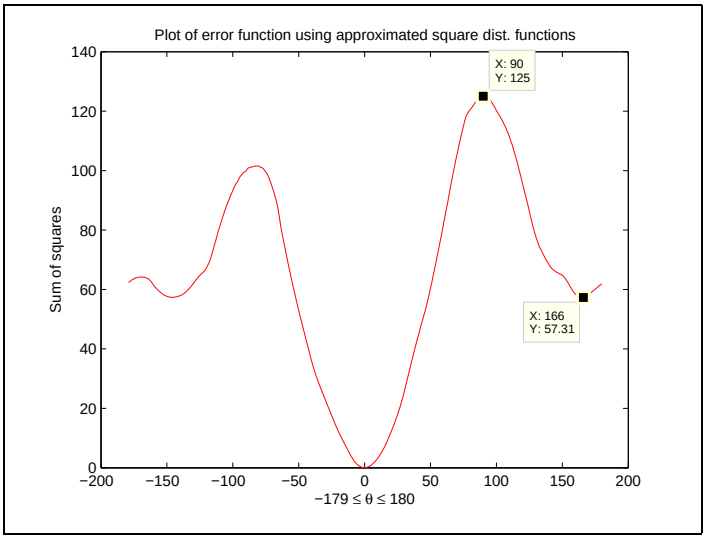


Figure 10.2: A plot of the sum of squares in the knight scan setting as a function of successive rotations about the z-axis.

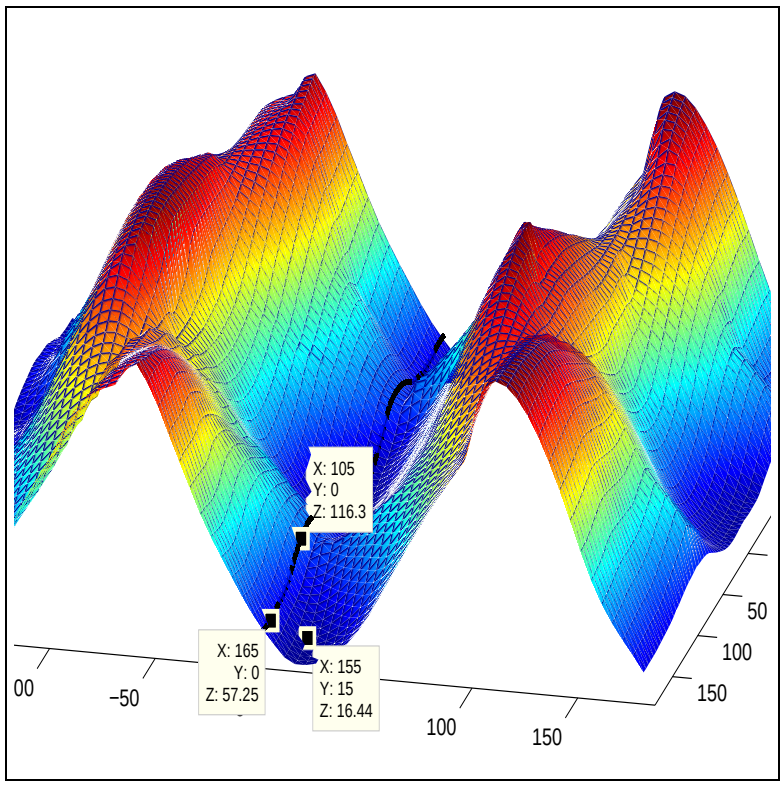


Figure 10.3: A plot of the error function sampled at 72×72 positions corresponding to 72×72 different rotations applied to the point cloud P.

From experiments with the above data, we know for a fact that the successive transformations found by the registration procedure contain a negligible amount of translation and also almost no rotation about the y axis. Thus, in this case the error function can be seen as a function of only two variables, the rotational angles about the x and z axes, and we can easily plot it. To make such a plot, we have sampled the error function in a similar manner to above. This time, for each $\mathbf{R}_{z,i}$, $i = 0, 5, 10, \ldots, 355$, we applied successive rotations $\mathbf{R}_{x,j}$ $j = 0, 5, 10, \ldots, 355$ about the x axis as well. For each position of the point cloud corresponding to a rotation $\mathbf{R}_{z,i}\mathbf{R}_{x,j}$, we computed the sum of squared residuals. The plot is shown in figure 10.3 and for comparison, we have added the plot from figure 10.2.

In figure 10.3 we have included three points; a local minimum **a** $(X = 155, Y = 15.0, Z = 16.4)$, the point **b** $(X = 105, Y = 0, Z = 116.3)$ and a local minimum on the graph from figure 10.2, **c** $(X = 166, Y = 0, Z = 57.5)$. The points' x and y values correspond to the rotational angles about the z and x axes, respectively. The z values are the sum of squares.

The z value of the point **b** is the sum of squares when the point cloud P have been rotated 105 degrees about the z axis (and zero degrees about

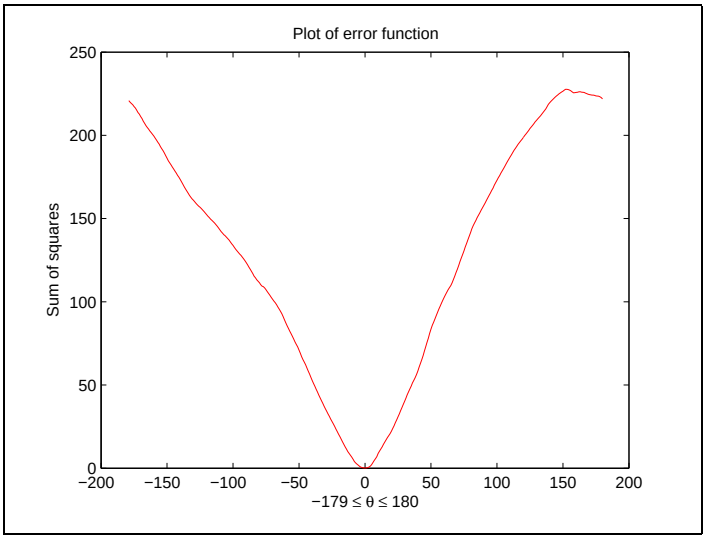


Figure 10.4: A plot of the sum of squares in the frog scan setting as a function of successive rotations about the z-axis.

the x axis). If we look at the plot in figure 10.2 we expect to get stuck at **c** if we start the registration procedure from the position corresponding to **b**. However, we see from the plot in figure 10.3, which is a more correct image of the error function, that we actually should get stuck at **a**. Experimental results is in accordance with this reasoning.

If we rotate about an axis different from the z axis we do not in general get the same sinusoidal graph as in plot 10.2. In other words, the form of the plot depends on the shape of the object and about which axis we rotate. See figure 10.4 for a display of a plot made in exactly the same way as when we made the plot in figure 10.2, this time with two partial scans of a rubber frog. Again, by just looking at the plot we believe that we will reach the global minimum even for large rotational displacements. From the reasoning above, we now know however, that life is not that simple.

10.3 Improvements

In this last section we indicate ways to improve the algorithm of Mitra et al. [33]. By improvements we first of all think of ways to alter the method so that it will not be as sensitive to the initial displacement between the input shapes as it is in its current form. The first thought that may come to mind in that respect is whether or not the local approximants to the exact squared distance function are as good as Mitra et al. argue that they are. This line of thought is followed up in the subsection below.

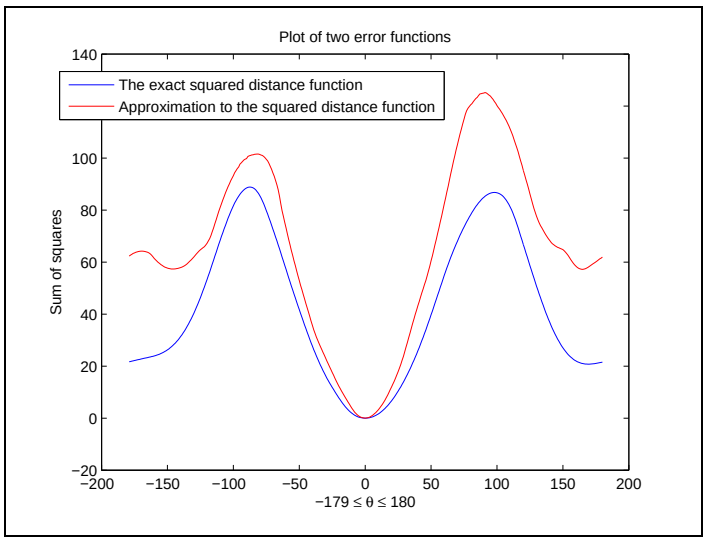


Figure 10.5: A plot of the sum of squares using the exact squared distance function (in blue) and its local approximations (in red).

10.3.1 Change of basis

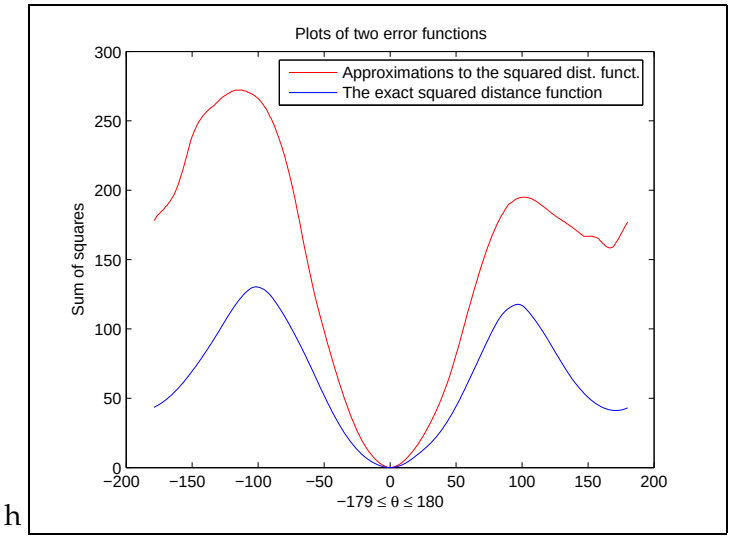
Expressing the approximants in a different basis other than the monomials could possibly provide better alignment of the input shapes. A nice replacement for the current approximants could be tensor product spline surfaces.

A way to figure out how good the approximations are, is to compare the sum of the squared residuals when using the exact squared distance function to the sum we get when using approximants. This has been done for the data which was used to produce the plots in figure 10.2 and figure 10.4. That is, we have rotated the point cloud 360 degrees around the z axis, one degree at a time, and sampled the error function at each position using both the local squared distance function approximations and the exact squared distance function.

The result of that experiment for the knight data is shown in figure 10.5. We see from the figure that the approximants produce values close to those of the exact function when the initial (counter-clockwise or clockwise) rotation θ is no larger than, say fifty degrees. If we increase the size of the bounding box, and thus construct larger cubes, the approximations get slightly worse, see figure 10.6.

The result of sampling the error function using both the exact squared distance function and its local approximants for the frog data is shown in figure 10.7.

We have seen that the local approximations *are* quite good, and more importantly, they are convex within the same regions as the exact squared



h

Figure 10.6: A plot of the sum of squares using the exact squared distance function (in blue) and its local approximations (in red). The cubes at the finest level are larger in this example than in the previous plot and consequently, the approximations are not as good.

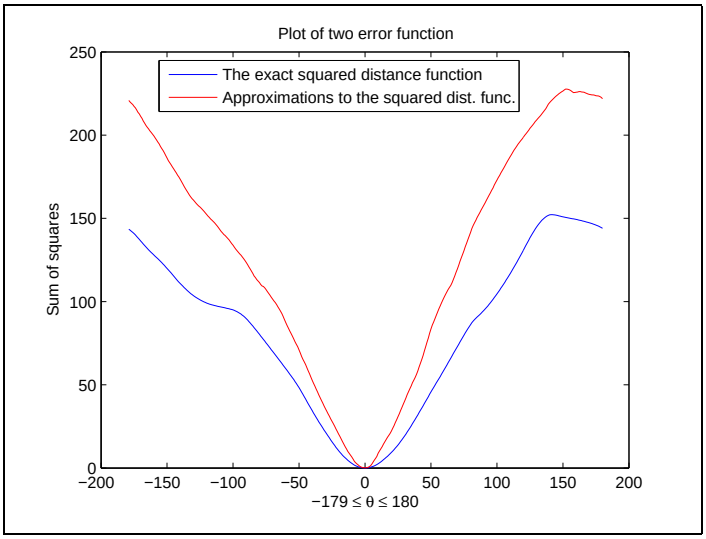


Figure 10.7: A plot of the sum of squares using the exact squared distance function (in blue) and its local approximations (in red).

distance function. We could perhaps have approximated the function better for large rotation angles but since the registration follows a gradient descent approach, this would still not have "carried" us over the local maximum(s) during registration.

Consider next another possible way to change the method into something that may provide better alignment.

10.3.2 Feature points

How can we ensure optimal alignment when we do not know the relative initial positions of the point cloud and the model? We think that the answer may be; by making use of feature points. Recall from chapter 4 that there are no precise definition of feature points, they can be thought of as "characteristic" points. Examples are points on a surface corresponding to maximum surface curvature. By settling a set of feature points and their correspondences we could be able to compute, rather exact, the initial rotational displacement, even if it is large. Thereby we would expect to get into the convex setting where we know that the method ensures good alignment. Note that, this way, we could also be able to align shapes that may overlap only partially.

We realize that the algorithm outlined in chapter 8 is appropriate for this purpose. The method gives both a way to determine which points should be chosen as features from the point cloud and the model and also a way of matching them. However, the procedure for computing the integral volume descriptor is rather complicated, and there has not been enough time to implement it.

In an attempt to follow this line of thought, however, we have tried to implement the point matching procedure of [22]. Given two sets of points P and S, and a match matrix with entries $m_{i,j}$, the authors seek to minimize the sum

$$\sum_{i=1}^{n_S}\sum_{j=1}^{n_P} m_{i,j}\left|\left|\mathbf{s}_i - \mathbf{TR}\mathbf{p}_j\right|\right| - \alpha\sum_{i=1}^{n_S}\sum_{j=1}^{n_P} m_{i,j}$$

by varying the $m_{i,j}$'s and the transformation parameters $\mathbf{T}$ and $\mathbf{R}$. The $m_{i,j}$'s take on values between 0 and 1 where 1 denotes a match. Clever as this formulation might be, the resulting algorithm is too slow to be given large point sets as input. Also, there are several parameters that need to be set but the authors suggest doing so by trial and error. At the time of writing the correct set of parameters and appropriate subsets of points have not been settled.

10.3.3 Nonlinear least squares

Recall that the algorithm uses a linearized rotation matrix and thus it only works well for small rotational angles. The work that has to be done if the computed motion is too large implies computing matrix functions which adds to the complexity of the implementation. This could be avoided if we instead solve the nonlinear system of equations which arise if we use an exact representation of the rotation. Preferably we would want to employ a globally convergent method solving a nonlinear system of equations.

Typically, when solving nonlinear systems of equations we must provide an initial guess for the solution. Some methods may not converge at all if the initial guess is not good. This is certainly the case for Newton's method. This is bad news if we want to align shapes that are separated by, say, a large rotation. A naive guess, like the identity transformation, would probably not be sufficient for the Newton's method to converge. However, there exists methods that converge even when the initial guess is not very good.

List of Figures

Bibliography

[1] 3rdtech. http://www.3rdtech.com.

[2] T. Akenine-Möller. Fast 3d triangle-box overlap testing. *Journal of graphics tools*, 6, 2001.

[3] T. Akenine-Möller and E. Haines. *Real-Time Rendering*. A K Peters, 2002.

[4] M. Alexa. Linear combination of transformations. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 380–387. ACM Press, 2002.

[5] E. Angel. *Interactive Computer Graphics - A Top Down Approach Using Open GL*. Addison Wesley, 2003.

[6] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu. An optimal algorithm for approximate nearest neighbor searching. *Journal of the ACM*, 45:891–923, 1998.

[7] P. J. Besl. Active, optical range imaging sensors. *Mach. Vision Appl.*, 1(2):127–152, 1988.

[8] P. J. Besl and N. D. McKay. A method for registration of 3d-shapes. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 14, 1992.

[9] P. Bhattacharya. Efficient neighbor finding algorithms in quadtree and octree. Technical report, Indian Institute of Technology, Department of Computer Science and Engineering, Kanpur, 2001.

[10] F. Cazals and M. Pouget. Estimating differential quantities using polynomial fitting of osculating jets. *Comput. Aided Geom. Des.*, 22(2):121–146, 2005.

[11] Immersion Corporation. http://www.immersion.com.

[12] S. J. Cunnington and A. J. Stoddart. N-view point set registration: A comparison. In *British Machine Vision Conference*, volume 1, pages 234–244, 1999.

[13] B. Curless. New methods for surface reconstruction from range images. Technical report, Stanford University, 1997. http://graphics.stanford.edu/papers/curless_thesis/.

[14] B. Curless and S. Seitz. 3d photography. ACM Siggraph '00 Course Notes, Course No. 19, 2000.

[15] D. Eberly. Distance between point and triangle in 3d. Geometric Tools for Computer Graphics (The Morgan Kaufmann Series in Computer Graphics), 1999. http://www.geometrictools.com/Documentation/DistancePoint3Triangle3.pdf%/.

[16] Faro Europe GmbH et. co. http://www.iqvolution.com/en/.

[17] A. Fitzgibbon. Robust registration of 2d and 3d point sets. In *British Machine Vision Conference*, pages 411–420, 1996.

[18] Max-Planck-Institut für Informatik, Polytechnical University of Catalonia, Istituto di Scienza e Tecnologie dell'Informazione, Konica Minolta Europe, gedas Iberia S.A., and Soprintendenza per i beni ambientali. http://www.vihap3d.org.

[19] N. Gelfand, N. J. Mitra, L. J. Guibas, and H. Pottmann. Robust global registration. In *Proceedings of Symposium on Geometry Processing 2005*, pages 197–206, 2005.

[20] Leica Geosystems. http://www.leica-geosystems.com.

[21] Breuckmann GmbH. http://www.breuckmann.com.

[22] Steven Gold, Chien Ping Lu, Anand Rangarajan, Suguna Pappu, and Eric Mjolsness. New algorithms for 2d and 3d point matching: Pose estimation and correspondence. In *Advances in Neural Information Processing Systems*, volume 7, pages 957–964. The MIT Press, 1995.

[23] A. P. Gueziec, X. Pennec, and N. Ayache. Medical image registration using geometric hashing. *IEEE Computational Science and Engineering*, 4(4):29–41, 1997.

[24] A. Guéziec and N. Ayache. Smoothing and matching of 3-d space curves. *International Journal of Computer Vision*, 12(1):79–104, 1994.

[25] Konica Minolta Holdings. http://www.konicaminolta-3d.com/.

[26] B. K. P. Horn. Closed-form solution of absolute orientation using unit quaternions. *Journal of the Optical Society of America*, 4(4):629–642, 1987.

[27] inSpeck. http://www.inspeck.com.

[28] Capture Geometry Internally. http://www.cgiinspection.com.

[29] J. B. Kuipers. *Quaternions and Rotation Sequences - A primer with applications to orbits, aerospace and virtual reality*. Princeton University Press, 2002.

[30] D. C. Lay. *Linear Algebra and its applications*. Addison Wesley, Greg Tobin, 2003.

[31] S. Leopoldseder, H. Pottmann, and H. Zhao. The d^2-tree: A hierarchical representation of the squared distance function. Technical report, Institute of Geometry, Vienna University of Technology, Vienna, Austria, 2003. http://www.geometrie.tuwien.ac.at/ig/papers/t_rep101.pdf.

[32] A. Lorusso, D. W. Eggert, and R. B. Fisher. A comparison of four algorithms for estimating 3-d rigid transformation. In *British Machine Vision Conference*, pages pages 237 – 246, 1995.

[33] N. J. Mitra, N. Gelfand, L. J. Guibas, and H. Pottmann. Registration of point cloud data from a geometric optimization perspective. In *SGP '04: Proceedings of the 2004 Eurographics/ACM SIGGRAPH symposium on Geometry processing*, pages 22–31. ACM Press, 2004.

[34] N. J. Mitra and A. Nguyen. Estimating surface normals in noisy point cloud data. In *SCG '03: Proceedings of the 19th annual symposium on Computational geometry*, pages 322–328. ACM Press, 2003.

[35] R. M. Murray, Z. Li, and S. S. Sastry. *A Mathematical Introduction to Robotic Manipulation*. CRC Press, 1994.

[36] F. Nooruddin and G. Turk. Simplification and repair of polygonal models using volumetric techniques. Technical report, Georgia Institute of Technology, 1999. http://citeseer.ist.psu.edu/article/nooruddin99simplification.html.

[37] nub3d. http://www.nub3d.com/english/.

[38] U.S.A.'s National Institute of Standards and Technology. http://math.nist.gov/tnt.

[39] Steinbichler Optotechnik. http://www.steinbichler.de.

[40] X. Pennec. Registration of uncertain geometric features: Estimating the pose and its accuracy. In *Proceedings of the First Image Registration Workshop*, 1997. http://citeseer.ist.psu.edu/pennec97registration.html.

[41] H. Pottmann and M. Hofer. Geometry of the squared distance function to curves and surfaces. *Visualization and Mathematics III*, pages 221–242, 2003.

[42] S. Rusinkiewicz and M. Levoy. Efficient variants of the icp algorithm. In *Proceedings of the Third International Conference on 3D Digital Imaging and Modeling (3DIM)*, pages 145–152, 2001. http://citeseer.ist.psu.edu/rusinkiewicz01efficient.html.

[43] Scantech. http://www.scantech.dk.

[44] INUS Technology. http://www.rapidform.com/.

[45] Wikipedia. A free-of-charge, multilingual, web-based encyclopedia written by volunteers. http://en.wikipedia.org/.

[46] H. J. Wolfson and I. Rigoutsos. Geometric hashing: An overview. *IEEE Computational Science and Engineering*, 4(4):10–21, 1997.

[47] H. Zhao. A fast sweeping method for eikonal equations. *Mathematics of Computation*, 74(250):603–627, 2004.