Institute for Computer Science VII
Robotics and Telematics

Master's thesis

Registration and iterative scale estimation of differently scaled point clouds

David Redondo

May 2020

First supervisor: Prof. Dr. Andreas Nüchter
Second supervisor: Prof. Dr. Klaus Schilling

Zusammenfassung

Die vorliegende Arbeit beschreibt einen Algorithmus zur Registrierung und gleichzeitigen Skalierungsschätzung von Punktwolken verschiedener Quellen. Insbesondere die Registrierung von dichten, absolut skalierten Laserscan Punktwolken und Punktwolken, die mit Hilfe des Structure from Motion Verfahrens rekonstruiert wurden, lagen dabei im Fokus. Letzteren Punktwolken fehlt aufgrund des verwendeten Verfahrens ein absoluter Bezugsrahmen.

Der vorgeschlagene Algorithmus basiert auf dem 4-points-congruent-sets-Algorithmus. Dieser verwendet Eigenschaften von Mengen von vier Punkten, die unter affinen Transformationen invariant sind, um diese in beiden Punktwolken zu finden. Jedoch ist er auf rigide Transformationen beschränkt. Er wurde angepasst, so dass er auch in dem oben beschriebenen Szenario von verschieden skalierten Punktwolken verwendet werden kann. Dies wurde durch Ergänzen einer iterativen Komponente zur Schätzung der Skalierung erreicht. Die beste Schätzung wird vorgehalten und je nach der geschätzen Güte dieser werden gesuchte Distanzen auf einen Bereich um diesen Skalierungsfaktor eingeschränkt. Die initiale Schätzung wird mittels einer Hauptkomponentenanalyse ermittelt.

Auch verbessert der neu entwickelte Hauptteil des Algorithmus dessen Laufzeit im Vergleich zum Original. Die Suche von Kandidaten für Punktpaare war in Tests 2-3-mal schneller. Die Konstruktion der namensgebenden Mengen von vier Punkten war mit der originalen Methode nach Wegfall der Beschränkungen für den rigiden Fall gar nicht mehr durchführbar.

Abstract

The present works describes an algorithm for registering and simultaneous scale estimation of point clouds gathered by different means. The focus was in particular on laser scans that are dense and absolutely scaled and point clouds that were reconstructed using Structure from Motion. The latter point clouds lack an absolute reference because of the used method.

The proposed algorithm is based on the 4-points-congruent-sets-algorithm. This algorithm uses properties of sets of four points that are invariant under affine transformations to find them in both point clouds. However, it is restricted to finding rigid transformation. The algorithm has been adapted for the above described scenario of differently scaled point clouds. This has been achieved by adding an iterative component to the algorithm in order to estimate the scale. The best scale factor is saved and depending on its quality, searched distances are restricted to a range around this scale factor. The initial scale estimation is determined by employing principal component analysis.

Additionally, the newly developed main part of the algorithm improves its performance compared to the original. The search for point pair candidates was in tests 2 to 3 times faster. The construction of the eponymous four point sets was without the restrictions for the rigid case not able to be carried out.

Contents

Acronyms

3DTK	The 3D Toolkit
4PCS	4-points congruent sets
ANN-tree	approximate nearest neighbor tree
ICP	Iterative Closest Points
PCA	Principal Component Analysis
RANSAC	Random Sample Consensus
SfM	structure from motion
SVD	singular value decomposition

Chapter 1

Introduction

1.1 Motivation

Today, 3D data sets are used in various applications: They complement geographic information systems, they are used for autonomous navigation, can be used in archaeology to identify heritage sites or for recording a site before further destructive excavations. Numerous others fields have been enriched by the employment of 3D data. They are used for example for monitoring large agricultural and silvicultural areas or to support urban planning and surveying.

There are multiple ways to gather 3D Data that are in widespread use. Laser scanning (also called Lidar) measures the time of flight of directed light to create high fidelity point clouds. Most laser scanners will also measure the reflectivity of the objects that are surveyed. However, because most laser scanners emit monochromatic light in the non-visible spectrum, data gathered this way does not include color information about the 3D space. Reflective surfaces like mirrors also pose a challenge for laser scanning. Light reflected from a mirror and returning to the scanner can create a duplicated geometry in the scene with the mirror appearing as a window into this ghost geometry. Applications of Lidar are wide ranging, they include airborne laser scanning aboard an aircraft to create terrain models, scanning objects in order to generate 3D models of them and usage in robotics and autonomous driving for collision avoidance and mapping. Using cameras to capture 3D data, it is possible to also include color information with the 3d data. The field of Photogrammetry uses multiple cameras with a known pose difference and multiple images of scenes that contain a known reference object to make measurements and reconstruct 3D data from images. When no such reference is available, structure from motion (SfM) can be used to extract 3D information from a set of overlapping images. This is done by extracting features from the images and matching them. The drawback of this technique is that the resulting 3D model is not as dense and can be less accurate than one captured by for example laser scanning and that no information about absolute dimensions can be gained because of the lack of a known reference. There can also be unexpected holes when no suitable features could be found e.g. on large areas like smooth surfaces. Another camera based technique is using structured light. A light pattern, usually laser lines or a pattern of dots or stripes, is projected onto a scene. A camera filming this scene can detect this pattern and deformations in it allowing to draw conclusions about the 3D structure of the scene. However

Figure 1.1: A sensor system consisting a video camera and thermal camera mounted on of a laser scanner. (Source: [5])

this does not allow to construct a true 3D representation of the scene, only depth information of the points that are visible to the camera is gathered. The advantages of combining data that was captured using multiple different measurement methods are obvious. First, some methods can fail to produce data under specific circumstances or of objects. The resulting holes in such data sets can be filled by combining it with data from a measurement method that does not exhibit this weakness. It can also be imagined that one has two sensors, one is more accurate when measuring near objects, the other is more precise when doing more distant measurements. By combining the data from both a good quality result in near and far distance can be achieved. Second, by combining data coming from multiple measurement methods it is possible to add an additional dimension to a dataset. For example, point clouds acquired via laser scanning can be combined with the images taken with a camera in order to color its points and generating textured meshes of objects. Or information gathered via thermal imaging can be used to identify sources of heat loss in the 3D structure of buildings. If the different measurements are taken at the same time and place or the relative positions of the measurements are otherwise known, the data fusion is straightforward. Figure 1.1 shows a sensor system that is able capture range data with a laser scanner, thermal data using a thermal camera and images using a video camera at the same time. The cameras are mounted on top of the laser scanner at fixed position. The constant offset between the measurement devices makes the relation between the different forms of data measured by the sensors known at all times. This makes it relatively easy to combine the information gathered by the different sensors.

When this information is not available, merging of multiple dataset is harder because the fusion has to be done on the basis of the data itself. It can also be in the nature of the used

sensors that it is not possible to make measurements at the same time. For example thermal images should be taken at night but photo cameras need the sun as a light source to capture images of good quality. Merging of datasets captured using the same method and sensors is an already very good explored problem. Fusion of 3D point clouds coming from different sources on the other hand faces a set of difficulties. Most of them stem from the different characteristics of the sensors and methods that were used to gather the data affecting the properties of it. Point clouds can for example vary in density, scale, extent, occluded regions and accuracy. Most problematic of these is difference in scale between 3D data captured by different sources. Algorithms developed for registering multiple point clouds by the same sensor cannot be applied to this problem because they assume a rigid relationship between them. Nevertheless, there is a high interest in aligning 3D data gathered via laser scanning and reconstructed one via SfM which exhibits this isssue of an unknown scale factor. This can be seen by huge variety of schemes that have been proposed to tackle the problem of registering cross-source 3D data with different scale.

1.2 Related Work

The standard method for registering point clouds of the same scale is the Iterative Closest Points (ICP) algorithm by Besl and McKay [4] and its variants. While originally only capable of aligning point clouds rigidly it can be extended to also estimate a scale factor. However, as Zinßer et al. showed it only works reliably for a small region of scale factors [42]. Du et al. counters this by bounding the range of admissible scale factors [11]. Other authors use manually crafted inputs to "kick-start" the algorithm. Borrmann selects pairs of known correspondences in order to calculate an initial transformation [5]. Peng et al. manually select dense point regions in [35] to facilitate matching [20].

Another approach is representing point clouds as Gaussian mixture models and matching those. Bing and Vemuri [21] proposed a possible representation of points clouds and an algorithm to match them by aligning the Gaussian mixture models that they are represented by. This method is able to align point sets rigidly and non-rigidly. Evangelidis et al. [13] extended this strategy and developed a scheme for matching multiple point clouds based on this idea. In 2016, Campbell and Peterson [7] proposed a method for aligning two mixture models that is globally optimal. This was later adapted into a framework for determining camera poses inside 3D scans. [8].

Bülow and Birk [6] use a series of integral transform to calculate the alignment between two point clouds. First the 3D Data is resampled from a spectrum produced by applying a fast Fourier transform. Then in a first step the rotation is calculated using the $SO(3)$ Fourier transform. The rotation is applied to both point clouds and the new spectrum of the data is calculated. The scale is determined by employing a Mellin transform on the rotated spectra. After applying the scale to the rotated point clouds their spectra are recalculated again and are used to determine the missing translation.

Feature recognition and matching is a very common technique in Computer Vision for analyzing images. There is also work that uses feature-based techniques on 3D data. Li and Guskov [25] transferred the famous SIFT descriptor of Lowe [26] to the 3D space. A similar approach is

taken by Tombari et al. [39] employing the histogram of normals as a descriptor called SHOT. However, Theiler [38] found that " descriptors like SIFT are of limited use for laser scans: many studies confirm that they can handle viewpoint differences only up to $\approx$ 25 degrees". Huang et al. [20] identify clusters inside the data and assign ESF descriptors to them. Then they arrange these clusters as nodes of a graph and proceed with a graph matching based approach.

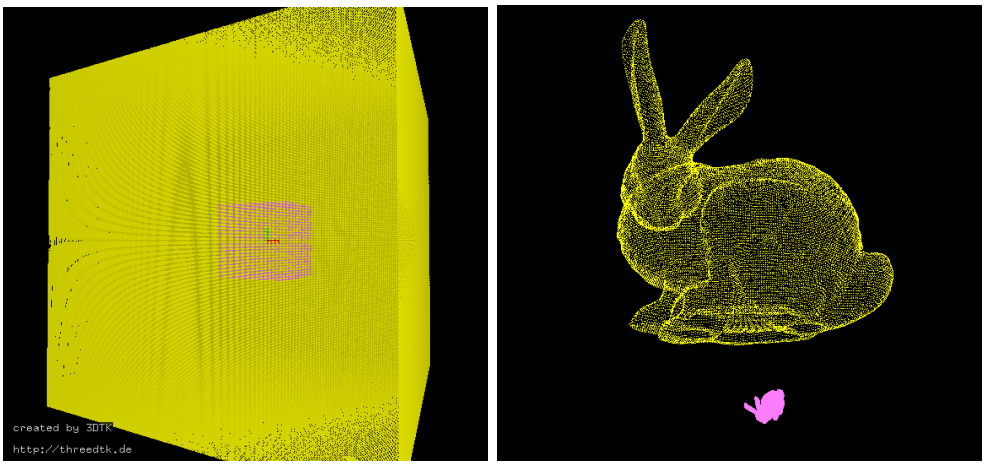
Some authors try to reduce the problem space by reducing the degrees of freedom of the sought after transformation. A popular technique is identification of the ground plane. Novak and Schindler [29] generate heightmaps after identifying the ground and try to align these. A method that is only applicable for urban scenes was shown by Moussa and Elsheimy in [28]. Here matching is done by recognizing intersections of buildings with the ground and aligning the so created floor plans of the scene. A similar idea is used by Yoshimura et al. [41]. Here intersection points of vertical lines with the ground plane (i.e. corners of buildings) are used as feature points. Triangles of feature points are randomly created in one point cloud and matching is done by trying to find similar triangles in the other one using properties of these triangles that are invariant under transformation such as the corner angles.

The 4-points congruent sets (4PCS) algorithm has a similar idea at its basis. Point clouds are matched using properties of sets of four points that stay constant under affine transformation. The algorithm was proposed by Aiger et al. in 2008 [1], however they restricted it to rigid transformations for an efficient implementation. Mellado et al. [27] published an version of the Algorithm titled "Super 4PCS" that further improved performance. It also only applies to rigid problems. Theiler et al. [38] adapted the 4PCS-algorithm for use with feature points similar to those used by SIFT and align point clouds by aligning these feature points. Another method for aligning SfM point clouds and those captured using a laser scanner was published by Corsini et al. [10]. However, they deconstruct the data as a set of planar regions and restrict their matching to include only one point per region.

1.3 Contribution and Outline

This thesis builds on the 4PCS-algorithm which was chosen because of the relative simple but elegant idea behind it. It does not make any assumption about the structure of the point clouds - in contrast to methods that for example try to detect the ground plane - and works directly on the points constituting the data. Within this thesis, the 4PCS-algorithm has been adapted in order to align point clouds of different scales - which fits the usage of affine invariant properties. Besides, a more efficient method to find congruent bases in two sets of points was developed, which constitutes the majority of the algorithm. Thereby, the adapted algorithm is able to align differently scaled point clouds much more efficiently than a version of the original algorithm in which the assumptions of rigid transformations have been removed.

The thesis is structured as follows: After this introduction, Chapter 2 briefly explains how point clouds can be stored and the algorithms that are used when analyzing and working with point clouds that are used in this thesis. The original 4PCS-algorithm is described in Chapter 3. It is followed by the modifications and adaptions to it that are the main work of this thesis in Chapter 4. Chapter 5 presents tests comparing the runtime of the developed algorithm to the original one and experiments using it to align different point clouds.



(a) The smaller pink cube inside the bigger yellow cube.

(b) The Stanford bunny.

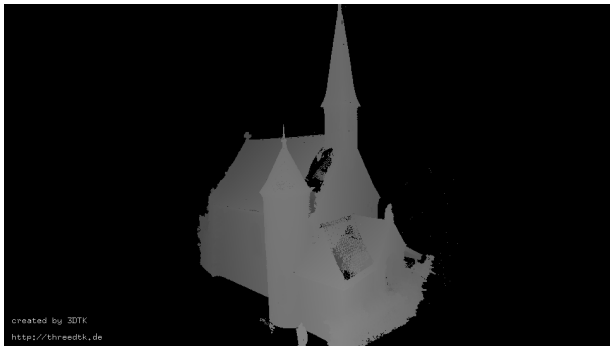
Figure 1.2: Two of the datasets.

1.4 Data sets

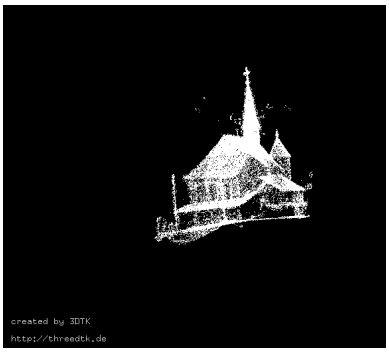
Multiple data sets were used for development and testing of this work. The first data set is an artificial one consisting of two cubes of different sizes. It was mainly used because of its relatively small size for quick tests during development. Figure 1.2a shows both cubes. The larger yellow cube has edge lengths of $100 \times 100 \times 100$ units and the average distance between points is 0.5. The smaller cube has a four times shorter edge length of $25 \times 25 \times 25$ units and is less dense than the larger cube with the average distance between points being 1. The larger cube consists of 960 002 points and the small cube is comprised of 15 002 points.

The second data set is based on the famous Stanford bunny [24]. The model consists of 35 947 points as does the original. A second version of the bunny has been scaled down and depending on the test rotated and its point density varied. In Figure 1.2b the smaller bunny has been rotated by 45° around each axis.

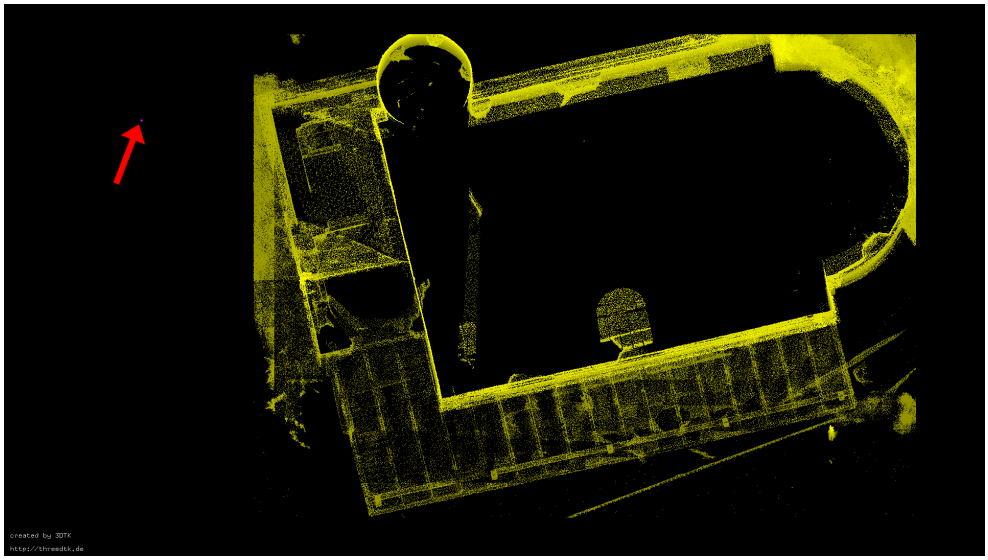
Finally the last data set consists of two models captured by two different measurements of the same scene. It shows a chapel in village of Randersacker (located at N 49°45′49.32″ E 9°58′52.536″). The first measurement has been taken with Riegl VZ-400 laser scanner and is available at [31]. The scene has been edited so that only the chapel is shown and its point density reduced to make the original size of 5117 MiB and 86 585 411 points more manageable, resulting in a size of 5 714 307 points and 337 MiB including normals. The second one is reconstructed using SfM from multiple camera images and shows the same chapel. It consists of 789 248 points which amounts to 43 MiB including their surface normals. Both models are shown in Figure 1.3, on the left the model captured with the laser scanner can be seen, on the right is the chapel as reconstructed from the photos. Showing an image of both is difficult because of the large size difference between the two. The laser scanner model is approximately 345 times larger than the SfM model (Section 4.2). An attempt can be seen in bottom picture and in Figure 2.4 but the pink colored SfM model is almost invisible on those pictures because it is so small.



(a) The measurement taken with a laser scanner



(b) The model reconstructed from camera images



(c) Both models in one image as seen from the top. The magenta SfM point cloud is almost invisible.

Figure 1.3: The chapel data set.

Chapter 2

Fundamentals

This chapter summarizes the principle and methods required to understand the original algorithm and the adaptions. It starts with the introduction of the kD-tree which is the data structure containing the 3D data. The section afterwards introduces affine transformation which are required to relate two sets of 3D points. Also it contains methods for calculations of such translations. The proposed method relies on several standard algorithm which are also explained in this chapter: ICP, Principal Component Analysis (PCA) and Random Sample Consensus (RANSAC).

2.1 *k*D-tree

kD-trees are generalized binary search trees adapted to store k-dimensional data developed by Bentley [3]. Their main advantages are storing abitrary multi-dimensional data without much overhead and allowing for efficient searches inside the resulting structure. For this reasons, they are a popular method for storing 3D point clouds. The same is true for this work. The developed algorithm performs extensive searches for points inside the data. A kD-tree is a perfect fit for this type of application and is used to store the point clouds used in this thesis.

Each node of the tree stores a k-dimensional data point. On each level the data is split along one of the k dimensions called the discriminator. The data is divided into two parts along the $k-1$-dimensional hyperplane perpendicular to the discriminator axis: one part which discriminator value is smaller than the point stored in the current node and one part with larger discriminator values. Traditionally, the discriminator only depends on the current level. The root (level 0) uses dimension 0 as it's discriminator, the children of the root use dimension 1 as discriminator (the dimensions are labeled from 0 to $k-1$ here) and so on. In general, the discriminator for a level l can be calculated by l modulo k. To construct from N data points an optimal kD-tree - the level of all leave nodes differs at most by one, in other words the tree has $\lfloor \log_2 N \rfloor$ levels - it is sufficient to choose at each level the median of the remaining points to be stored in the node and use its discriminator value to partition the other points [3].

An example of an optimal kD-tree in two dimensions is shown in Figure 2.1. It is constructed from 2D points each consisting of a x and y coordinate. The first discriminator is the x-coordinate of the points. The point A has the median x-coordinate of all points and is stored

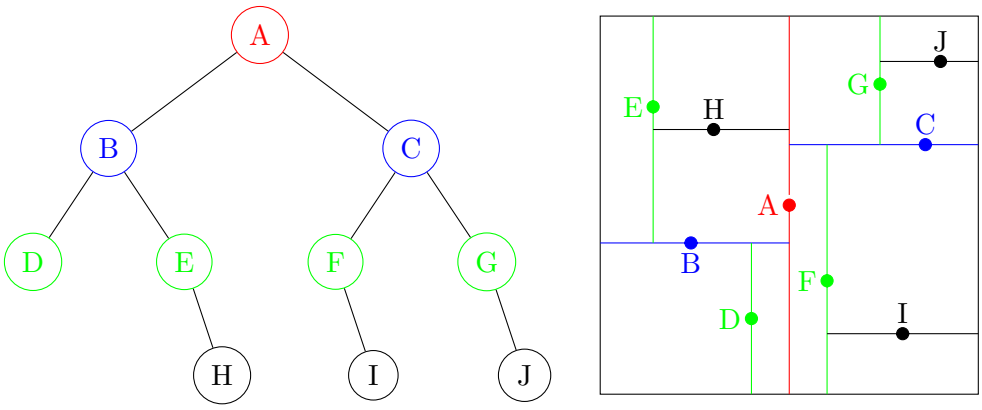


Figure 2.1: An optimal two dimensional kD-tree. (Reproduced based on [5])

into the root node. The values of the x-coordinates of Points B, D, E and H have smaller values than the one of A and form the left subtree of A, the ones of Points C, F, G, I and J are larger and form the right subtree. In the next step, the subtrees are divided based on the y-coordinate of their points. The left subtree has an even number of points, either B or H could have been selected as the median, in the example B was chosen. Only D has a smaller y-coordinate than B and forms the left subtree of B on its own. The right subtree of B consists of E and H. Now the discriminator is again the x-coordinate. However, as only two points are left either one could be chosen to be stored as the root of the subtree, here again the first one is chosen so the right subtree of B is formed by E with its sole right child H. The same holds for the right subtree of A. Here C has the median y-coordinate with F and I forming the left subtree and G and J forming the right one. These two subtrees only have two points each and are partitioned the same way as E and H before.

Creating an optimal kD-tree is expensive as all points forming a particular subtree need to be queried in order to find their mean. Also, storing only one point per node can be inefficient as each node needs to store pointers to its two subtrees in addition to its point. Therefore, different implementations exist that differ in their tree creation scheme and how the data points are stored. A comparison of some implementations is given in [12]. In this work the kD-tree implementation included in The 3D Toolkit (3DTK) [32] is used for working with 3D points as it has been shown to perform well. It differs from the original described kD-tree in that it stores the points only in leaf nodes which can contain multiple points and in intermediate nodes which dimension was used to split the points and the discriminator value used for partitioning. All points also store the bounding volume that their subtrees encompass. During tree construction the point set is not split along its median for performance reason but in each step the bounding volume is split in half by choosing the mean of the longest dimension of the bounding box as discriminator.

2.1.1 Searching in a tree

Searching for a nearest neighbor in a tree for a given point $\mathbf{p}$ is a classical problem and is implemented in the chosen implementation as shown in Algorithm 1. It has an additional

Algorithm 1: Searching for a nearest neighbor in a kD-tree

```
   input  : kD-tree t, point p, maximum distance d
   output: Point p_c that is closest to p in t
 1 FindClosest(t, p, d)
 2 if t is a leaf node then
 3 |  forall Points p_i in the leaf do
 4 |  |  if ||p_i − p|| < d then
 5 |  |  |  d ← ||p_i − p||
 6 |  |  |  p_c ← p_i
 7 |  |  end
 8 |  end
 9 else
10 |  if distance of bounding box of t to p > d then
11 |  |  return
12 |  end
13 |  i ← discriminator value of t - value of the corresponding dimension of p
14 |  if i ≥ 0 then
15 |  |  FindClosest(left child of t, p, d)
16 |  |  if |i| < d then
17 |  |  |  FindClosest(right child of t, p, d)
18 |  |  end
19 |  else
20 |  |  FindClosest(right child of t, p, d)
21 |  |  if |i| < d then
22 |  |  |  FindClosest(left child of t, p, d)
23 |  |  end
24 |  end
25 end
```

parameter in the maximum search distance d. Starting with the root the tree is traversed recursively. If the node, that is currently examined, is a leaf node then all points in this leaf are checked if they are closer to $\mathbf{p}$ than the maximum search distance d. If this is the case, the point is the new candidate for the closest point $\mathbf{p_c}$. The maximum search distance is lowered to the distance of $\mathbf{p}$ to $\mathbf{p_c}$ in order to only consider the points that are closer than $\mathbf{p_c}$ from now on. If the current node is not a leaf, first a quick check is performed if the bounding box of the node is within the maximum distance of $\mathbf{p}$. If it is farther away, it does not need to be further examined. If this is not the case the distance of $\mathbf{p}$ to the hyperplane that was used to split the current node is calculated. If its value is greater than zero, that means that $\mathbf{p}$ is in the left subtree of the current node and the algorithm descends down into into it. Otherwise $\mathbf{p}$ is in right subtree and the method is recursively executed for it. Finally, if the absolute value of i is smaller than d, then the bounding box of the other is closer than the current closest point and needs to be searched, too. When the algorithm terminates, $\mathbf{p_c}$ will be the closest point to $\mathbf{p}$ in the tree. If

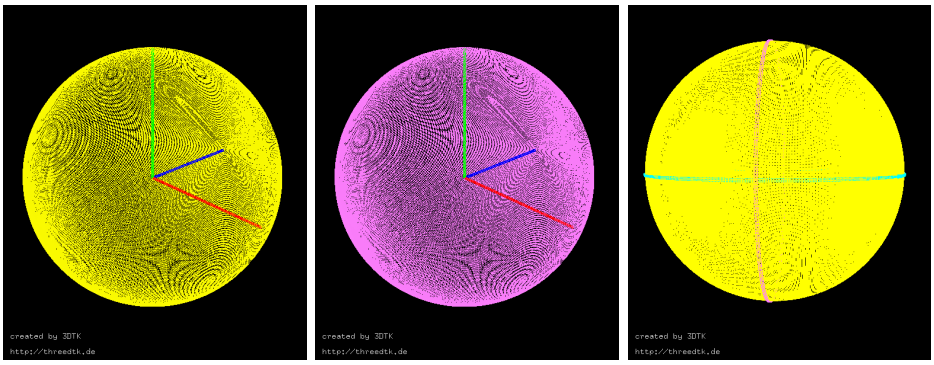


Figure 2.2: Result of searching for points on a sphere/circles. From left to right: The original sphere, the result of searching for all points on that sphere, the result of searching for points on two different circles.

one is interested in all points within a specified radius around a point the algorithm can easily be adapted. Line 5 is dropped and Line 6 is changed such that the candidate is appended to a set of points that is returned as the result.

This work is also interested in the points lying exactly on the shell of a sphere or on the rim of a circle around a selected point. Algorithms for these two problems can be easily derived from Algorithm 1 and have been implemented as presented in Algorithm 2. The parts of the algorithm specific for finding points on a sphere are colored in red, the parts specific to the circle case are in blue. The inputs in addition to the center point $\mathbf{p}$ are the radius of the sphere or circle r and an allowed error distance d that points can have to the sphere or circle. For the circle, a normal $\mathbf{n}$ to the circle plane is required to fully define it. The first check is again if the current node is a leaf and if the contained points lie on the circle or the sphere. For a point to lie on the sphere, its distance to $\mathbf{p}$ has to be in the range $[r-d, r+d]$. To check that a point lies on the border on a circle it additionally has to fulfill the requirement to be in the same plane as the circle, so $|\mathbf{n} \cdot (\mathbf{p_i} - \mathbf{p})| \leq d$. If the current node is not a leaf node a quick check is performed if the bounding box of it fully lies inside or outside of the sphere of radius r as it does not need to be evaluated further if this is the case. In the recursive case, if the indicator value i is smaller or larger than the minimum or the maximum allowed radius respectively, the sphere that needs top be searched is fully contained in the subtree that $\mathbf{p}$ is in in the discriminator dimension and only that subtree needs to be examined. Otherwise the sphere intersects the plane that partitions t and both subtrees have to be explored. Figure 2.2 shows the results of searching for points on a sphere or on different circles. The leftmost image shows the original sphere of points that the kD-treeconsists of. The middle one is the result of searching for points on that exact sphere which looks identical as expected. The right image shows the points that were found to be on two different specified circles. The found points are colored in blue and magenta and again the expected result is displayed.

Algorithm 2: Searching points on a sphere or on the border of a circle

```
   input  : kD-tree t, point p radius r, tolerance d, normal n
   output: Points P in t that are on the sphere circle
 1 FindPointsOnSphere/FindPointsOnCircle(t, p, r, d, n)
 2 if t is a leaf node then
 3 |  forall Points p_i in the leaf do
 4 |  |  if p_i is on the sphere  is on the circle with radius r around p and lies in the
   |  |   plane with normal n with a tolerance up to d then
 5 |  |  |  P ← P ∪ p_i
 6 |  |  end
 7 |  end
 8 else
 9 |  if bounding box of t does not overlap with the sphere with radius r around p then
10 |  |  return
11 |  end
12 |  i ← discriminator value of t - value of the corresponding dimension of p
13 |  if i ≥ r + d then
14 |  |  FindPointsOnSphere/FindPointsOnCircle(left child of t, p, r, d, n)
15 |  else if i ≤ −(r + d) then
16 |  |  FindPointsOnSphere/FindPointsOnCircle(right child of t, p, r, d, n)
17 |  else
18 |  |  FindPointsOnSphere/FindPointsOnCircle(left child of t, p, r, d, n)
19 |  |  FindPointsOnSphere/FindPointsOnCircle(right child of t, p, r, d, n)
20 |  end
21 end
```

2.2 Transformation between two sets of points

3D points are defined by three coordinates. The reference of these coordinates is different for each point set. For example in a laser scan the origin is typically the laser scanner. In order to merge two different sets of 3D data the transformation between the coordinate systems in which the coordinates of the points are expressed needs to be found. If this transformation consists only of a translation and rotation, it is called rigid. If an additional scaling between the point clouds is required, the transformation will be in the class of affine transformations.

2.2.1 Affine transformations

Affine transformations are a class of transformations and can be understood as generalized linear transformations. Affine transformations map between two vector spaces and are characterized by the properties that they preserve affine combinations. That is if $T : V \to W$ is an affine transformation, $\mathbf{v}_{1\ldots n}$ vectors in A and $a_{1\ldots n}$ some scalars such that $\sum_{i=1}^{n} a_i = 1$, the following holds [40]:

$$T(a_1\mathbf{v}_1 + \cdots + a_n\mathbf{v}_n) = a_1T(\mathbf{v_1}) + \cdots + a_nT(\mathbf{v}_n). \tag{2.1}$$

Important properties of affine transformations are preservation of collinearity and relative proportions [40]. Given two points $\mathbf{a}$ and $\mathbf{b}$ that form a line and a third point

$$\mathbf{c} = (1 - s)\mathbf{a} + s\mathbf{b} \tag{2.2}$$

that lies at the s distance of the full distance between $\mathbf{a}$ and $\mathbf{b}$, the relative distances between these three points will not have changed after applying an affine transformation T:

$$T\left((1 - s)\mathbf{a} + s\mathbf{b}\right) = (1 - s)T(\mathbf{a}) + sT(\mathbf{b}). \tag{2.3}$$

Affine transformations can be written as a combination of a linear transformation and a translation:

$$T(\mathbf{v}) = \mathbf{R}\mathbf{v} + \mathbf{t}. \tag{2.4}$$

If T maps from A to B, $\mathbf{v} \in A$ is an m-dimensional vector and $\mathbf{t}$ is a suitable n-dimensional vector in $\mathbf{B}$, then $\mathbf{R}$ will be a $n \times m$-matrix. However it is much more convenient to use so called homogeneous coordinates that add one extra dimension when dealing with affine transformations. Equation (2.4) can be written in homogeneous coordinates as

$$\begin{pmatrix} \mathbf{R} & \mathbf{t} \\ 0 & 1 \end{pmatrix} \begin{pmatrix} \mathbf{v} \\ 1 \end{pmatrix} = \begin{pmatrix} \mathbf{R}\mathbf{v} + \mathbf{t} \\ 1 \end{pmatrix}. \tag{2.5}$$

This work is mostly interested in affine transformations mapping from $\mathbb{R}^3$ to $\mathbb{R}^3$. In particular this includes the rigid transformations translation and rotation and the deforming scaling operation.

Translation shifts the points in the target space B by $\mathbf{t} \in B$ compared to the source space. Because all points in the domain $\mathbf{A}$ are shifted by the same amount, translating preserves lengths between points and angles between vectors. The transformation matrix for a translation $\mathbf{t}$ is given by

$$T_{\text{trans}}(\mathbf{t}) = \begin{pmatrix} \mathbf{I} & \mathbf{t} \\ 0 & 1 \end{pmatrix}. \tag{2.6}$$

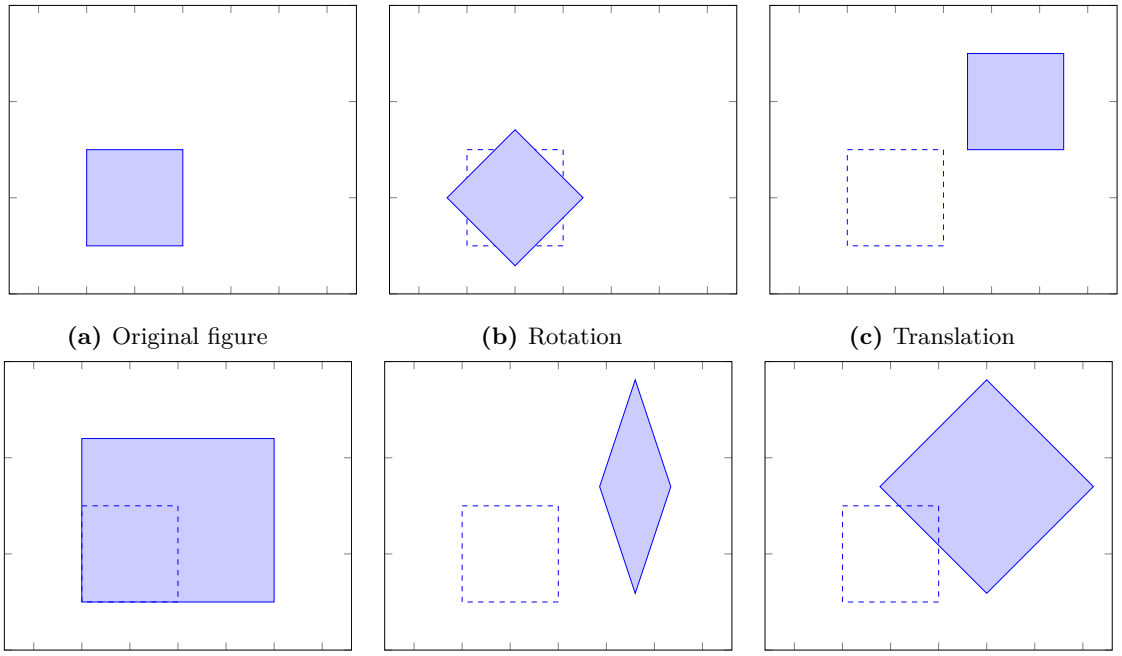
Rotation is performed by moving points around a static rotation axis by some angle without changing distances. If $\mathbf{R}$ is a rotation matrix the equivalent homogeneous transformation matrix is

$$T_{\text{rot}}(\mathbf{R}) = \begin{pmatrix} \mathbf{R} & 0 \\ 0 & 1 \end{pmatrix}. \tag{2.7}$$

Rotation is a rigid transformation as well and preserves distances and angles. In contrast to the two former transformations, scaling is not a rigid transformation and is called a deformation. Given the scale factors $s_1, \ldots, s_n$ for each dimension, scaling can be written in matrix form as

$$T_{\text{scale}}(s_1, \ldots, s_n) = \begin{pmatrix} \text{diag}(s_1, \ldots, s_n) & 0 \\ 0 & 1 \end{pmatrix} \tag{2.8}$$

In general, scaling does not preserve angles and distances but if the same scale factor is used in each dimension, angles between vectors are unchanged. Performing each transformation one



(a) Original figure **(b)** Rotation **(c)** Translation

(d) Scaling **(e)** Combined transformations **(f)** Combined transformation using equal scale factors

Figure 2.3: Affine transformations, dashed the original figure, in blue the figure after the transformation.

after another, first rotation, followed by scaling and finally translation creates a new combined affine transformation for which an explicit form can be found:

$$\begin{aligned} T(\mathbf{R}, s_1, \ldots, s_n, \mathbf{t}) &= T_{\text{trans}(\mathbf{t})} T_{\text{scale}}(s_1, \ldots, s_n) T_{\text{rot}}(\mathbf{R}) \\ &= \begin{pmatrix} \mathbf{I} & \mathbf{t} \\ 0 & 1 \end{pmatrix} \begin{pmatrix} \text{diag}(s_1, \ldots, s_n) & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} \mathbf{R} & 0 \\ 0 & 1 \end{pmatrix} \\ &= \begin{pmatrix} \text{diag}(s_1, \ldots, s_n)\mathbf{R} & \mathbf{t} \\ 0 & 1 \end{pmatrix} \end{aligned} \tag{2.9}$$

Figure 2.3 shows an example for each of the above mentioned special affine transformations. Figures 2.3e and 2.3f show the result of applying different affine transformations consisting of a rotation, scaling and transformation component each. In Figure 2.3e the deforming nature of the scaling operation can be observed, distances and angles are changed compared to the original figure. However, Figure 2.3f shows that when using the same scale factor in each dimension the angles are preserved.

2.2.2 Calculating the transformation between two sets of points

Two measurements from different sets of points are called a point correspondence if they refer to the same point. If the point correspondences for two sets of point measurements $P = \{\mathbf{q}\}$

and $Q = \{\mathbf{q}\}$ (for example two 3D scans taken from two different locations) are known it is possible to obtain a closed form solution for the transformation between the two set of points. A method for this was first devised by Horn in 1987 [18] using unit quaternions to obtain the optimal rotation which is described briefly below for obtaining the optimal translation and scale. There are also multiple other methods to obtain this transformation, for example one using orthonormal matrices instead of quaternions by the same author [17] or using the singular value decomposition (SVD) of a cross-correlation matrix of the measurements by Arun et al. [2] which is described briefly below. Nüchter et al. evaluate some of these methods for obtaining the transformation between two or more sets of measurements for the rigid case in [30]. An application including scaling can be found in [15]. Let $\mathbf{p}_i \in P$ and $\mathbf{q}_i \in Q$ be measurements that refer to the same point. The goal is to find a transformation T such that for every pair as above the following holds:

$$\mathbf{p}_i = T(\mathbf{q}_i) \, \forall i : 1 \leq i \leq n. \tag{2.10}$$

In other words T maps coordinates from L to R. T can also be decomposed into tree components: A translation $\mathbf{t}$, a rotation $\mathbf{R}$ and a scaling s which can be calculated independently from one another. With the above the equation can be written as

$$\mathbf{p}_i = s\mathbf{R}\mathbf{q}_i + \mathbf{t} \, \forall i : 1 \leq i \leq n \tag{2.11}$$

These elements represent seven degrees of freedom in 3d space, three each of translation and rotation and one of scaling. That means that three points are needed to determine the wanted transformation, two points would only provide six constraints.

Translation

Suppose that P and Q contain n measurements. If both are perfect then the T will map each point from Q perfectly to the corresponding point in P. However, in a real world scenario this is often not the case and small differences will remain between each $\mathbf{p}_i$ and transformed $\mathbf{q}_i$:

$$\mathbf{e}_i = \mathbf{p}_i - s\mathbf{R}\mathbf{q}_i - \mathbf{t}. \tag{2.12}$$

To achieve a least squares solutions the following error function has to be minimized

$$E = \sum_{i=1}^{n} \|\mathbf{e}_i\|^2 = \sum_{i=1}^{n} \|\mathbf{p}_i - s\mathbf{R}\mathbf{q}_i - \mathbf{t}\|^2. \tag{2.13}$$

By shifting each measurement such that their centroids are the origin of their coordinate systems:

$$\mathbf{p}'_i = \mathbf{p}_i - \bar{\mathbf{p}} \qquad \mathbf{q}'_i = \mathbf{q}_i - \bar{\mathbf{q}}, \tag{2.14}$$

where $\bar{\mathbf{p}}$ and $\bar{\mathbf{q}}$ are the centroids of the measurements in P and Q

$$\bar{\mathbf{p}} = \frac{1}{n} \sum_{i=1}^{n} \mathbf{p}_i \qquad \bar{\mathbf{q}} = \frac{1}{n} \sum_{j=1}^{n} \mathbf{q}_i, \tag{2.15}$$

the error function can be rewritten as

$$\begin{aligned} E' &= \sum_{i=1}^{n} \|\mathbf{p}'_i - s\mathbf{R}\mathbf{q}'_i - \mathbf{t} + \bar{\mathbf{p}} - s\mathbf{R}\bar{\mathbf{q}}\|^2. \\ &= \sum_{i=1}^{n} \|\mathbf{p}'_i - s\mathbf{R}\mathbf{q}'_i\|^2 - 2\left(\mathbf{t} - \bar{\mathbf{p}} + s\mathbf{R}\bar{\mathbf{q}}\right) \sum_{i=1}^{n} \mathbf{p}'_i - s\mathbf{R}\mathbf{q}'_i + \sum_{i=1}^{n} \|\mathbf{t} - \bar{\mathbf{p}} + s\mathbf{R}\bar{\mathbf{q}}\|^2. \end{aligned} \tag{2.16}$$

Equation (2.16) is a sum whose three summands can be minimized independently. The middle sum always evaluates to zero because the new centroids of the $\mathbf{p}'_i$ and $\mathbf{q}'_i$ being the origin by design. The last term also can be set to zero by choosing $\mathbf{t}$:

$$\mathbf{t} = \bar{\mathbf{p}} - s\mathbf{R}\bar{\mathbf{q}}. \tag{2.17}$$

The searched translation is the difference of the centroids after rotating and scaling $\bar{\mathbf{q}}$ to the coordinate system of R.

Scale

Now to determine s and R the remaining first term of the error function needs to minimized. Its expanded form is

$$\sum_{i=1}^{n} \|\mathbf{p}'_i - s\mathbf{R}\mathbf{q}'_i\|^2 = \sum_{i=1}^{n} \|\mathbf{p}'_i\|^2 - 2s \sum_{i=1}^{n} \mathbf{p}_i'^T \mathbf{R}\mathbf{q}'_i + s^2 \sum_{i=1}^{n} \|\mathbf{R}\mathbf{q}'_i\|^2. \tag{2.18}$$

Because rotation matrices are length preserving (Section 2.2.1) this is equivalent to

$$\sum_{i=1}^{n} \|\mathbf{p}'_i\|^2 - 2s \sum_{i=1}^{n} \mathbf{p}_i'^T \mathbf{R}\mathbf{q}'_i + s^2 \sum_{i=1}^{n} \|\mathbf{q}'_i\|^2. \tag{2.19}$$

Regarding the above as a quadratic in the variable s the minimum can be found at

$$s = \frac{\sum_{i=1}^{n} \mathbf{p}_i'^T \mathbf{R}\mathbf{q}'_i}{\sum_{i=1}^{n} \mathbf{q}'_i} \tag{2.20}$$

It has to be noted that when calculating the inverse transformation from P to Q, i.e. $\mathbf{q}_i = \tilde{s}\widetilde{\mathbf{R}}\mathbf{p}_i + \tilde{\mathbf{t}}$, one arrives at

$$\tilde{s} = \frac{\sum_{i=1}^{n} \mathbf{q}_i'^T \widetilde{\mathbf{R}}\mathbf{p}'_i}{\sum_{i=1}^{n} \mathbf{p}'_i}. \tag{2.21}$$

It might be desirable that the two solutions are the inverses of one another. However, as Horn notes in [18] in the general case this is not the case and so $\tilde{s} \neq \frac{1}{s}$. A symmetric solution can be obtained by considering a symmetric error term and the resulting expanded remaining error (cf. (2.19))

$$\begin{aligned} \mathbf{e_i} &= \frac{1}{\sqrt{s}}\mathbf{p}_i - \sqrt{s}\mathbf{R}\mathbf{q}_i - \mathbf{t}\, \forall i : 1 \leq i \leq n \\ &\frac{1}{s} \sum_{i=1}^{n} \|\mathbf{p}'_i\|^2 - 2 \sum_{i=1}^{n} \mathbf{p}_i'^T \mathbf{R}\mathbf{q}'_i + s \sum_{i=1}^{n} \|\mathbf{q}'_i\|^2. \end{aligned} \tag{2.22}$$

Differentiating with regards to s results in

$$-\frac{1}{s^2}\sum_{i=1}^{n}\|\mathbf{p}'_i\|^2+\sum_{i=1}^{n}\|\mathbf{q}'_i\|^2. \tag{2.23}$$

Setting equation (2.23) to zero and solving for s yields

$$s=\sqrt{\frac{\sum_{i=1}^{n}\|\mathbf{p}'_i\|^2}{\sum_{i=1}^{n}\|\mathbf{q}'_i\|^2}}. \tag{2.24}$$

This expression for the scale not only solves the problem of inverse transformations as described above but also has the advantage that it can be calculated without the rotation $\mathbf{R}$ in contrast to equation (2.20).

Rotation

Independent of the formula selected to calculate the scale (Equation (2.24) or Equation (2.20)), the error in Equations (2.16) or (2.22) is minimized in regards to R by maximizing

$$\sum_{i=1}^{n}\mathbf{p}_i'^{T}\mathbf{R}\mathbf{q}'_i. \tag{2.25}$$

All three methods that were mentioned above, do this by first constructing a cross-correlation matrix $\mathbf{H}$

$$\mathbf{H}=\sum_{i=1}^{n}\mathbf{p}'_i\mathbf{q}_i'^{T}=\begin{pmatrix}S_{xx} & S_{xy} & S_{xz}\\ S_{yx} & S_{yy} & S_{yz}\\ S_{zx} & S_{zy} & S_{zz}\end{pmatrix} \tag{2.26}$$

where the matrix entries can be calculated as

$$S_{xx}=\sum_{i=1}^{n}r'_{i,x}\cdot l'_{i,x}\quad S_{xy}=\sum_{i=1}^{n}r'_{i,x}\cdot l'_{i,y}\quad\dots\quad. \tag{2.27}$$

The singular value decomposition $VH=\mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$ of S can be used to obtain the rotation matrix

$$\mathbf{R}=\mathbf{V}\mathbf{U}^T \tag{2.28}$$

as shown by Arun et al. in [2]. Algorithm 3 on Page 17 summarizes the above equations into an efficient method for calculating the transformation between two sets of points.

2.3 Iterative Closest Points

The prerequisite to applying Algorithm 3 to two sets of points is that the point correspondences between them have to be known in order to calculate the rotation (see Equation (2.26)). In most cases this relation is unknown, may be hard to obtain and the unknowingness is one of the reason why a transformation between the two sets is sought after. One of the most well-known and

Algorithm 3: Calculating the least squares solution to the transformation between two sets of points

input : Two sets of points P and Q
output: A rotation matrix $\mathbf{R}$, a translation $\mathbf{t}$ and a scale s
1 Construct the cross-correlation Matrix $\mathbf{H}$ as described in equation (2.26)
2 Calculate the SVD of $\mathbf{H}$ as $\mathbf{H} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$
3 Calculate the rotation matrix $\mathbf{R} = \mathbf{V}\mathbf{U}^T$
4 Calculate s either according to equation (2.20) or equation (2.24)
5 Find the centroids $\bar{\mathbf{p}}$ and $\bar{\mathbf{q}}$ of P and Q
6 Calculate the translation $\mathbf{t} = \bar{\mathbf{p}} - s\mathbf{R}\bar{\mathbf{q}}$

widely used algorithms that solve this problem is the Iterative Closest Points (ICP) algorithm. First proposed in 1992 [4] it is now state of the art for aligning multiple point clouds. It requires no prior knowledge about the point clouds that should be registered but can be enhanced by it.

The main idea of the algorithm shown in Algorithm 4 is that instead of selecting the correct point correspondences in line 3, corresponding point pairs are formed by taking a point $\mathbf{p} \in P$ and the point $\mathbf{q}$ that is the closest point in Q to $\mathbf{p}$. Using these approximate correspondences a transformation is calculated by minimizing the error function as outlined as in Section 2.2.2. The assumption is that by using the closest points pairs to calculate the transformation one finds a transformation that is close enough to the real one and moves the two point clouds closer together. By iterating this process the computed transformation converges towards the correct one. The procedure is repeated until the mean square error does not change anymore or stays in a τ-region. Besl and McKay proved that the algorithm reaches a local minimum [4]. Therefore, it is important to have an initial estimate of the transformation or point clouds that are close together.

Algorithm 4: The ICP algorithm

input : two sets of points P and Q
desired precision τ
output: Rotation $\mathbf{R}$ and translation $\mathbf{t}$
1 Initialize $\mathbf{R}$ and $\mathbf{t}$
2 **repeat**
3 Find point correspondences $(\mathbf{p}_i, \mathbf{q}_i) \in P \times Q$
4 Minimize $E_j = \sum \|\mathbf{p}_i - \mathbf{R}_j\mathbf{q}_i - \mathbf{t}_j\|$
5 Apply $\mathbf{R}_j$ and $\mathbf{t}_j$ to Q
6 $\mathbf{R} \leftarrow \mathbf{R}_j\mathbf{R}$
7 $\mathbf{t} \leftarrow \mathbf{t} + \mathbf{t}_j$
8 **until** $E_{j-1} - E_j < \tau$;

The algorithm spends the most time searching for the point pairs. Using a kd-tree for point storage and lookup can offer tremendous speedup. Over time, many different variants of the original algorithm and methods to increase speed and quality of the result have been developed.

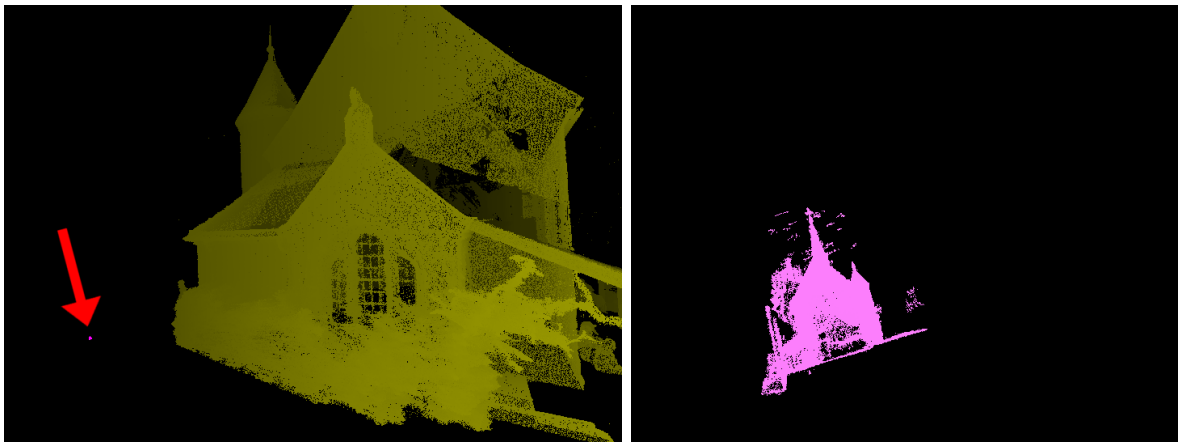


Figure 2.4: A large scale difference in point clouds that should be registered. On the right the second one zommed in.

These include only using a sampled subset of points, caching of points or rejecting pairs with a distance greater than some threshold. Criteria for forming the point pairs include searching along the normal of a point, searching in the view direction or enhancing the search with additional criteria for example normals. A weighting of the pairs in the error function can also be performed based on the distance between the two points or on some other information like color or normals if available. Finally the error function and therefore the way the transformation is calculated can be changed. As before if additional information is available, it can be included into the error term. Instead of using the point to point distance using a point to the local plane of the destination point is also common. Rusinkiewicz and Levoy give an overview and comparison of these methods and their properties [36].

It is also possible to include a scale term in the error function (cf. Equation (2.13)). However, when exposed to large scale differences they are prone to failure. An integrated method is described in [42] and it shows only a small range of scale factors where it can be used successfully. When exposed to the situation in Figure 2.4, namely huge scale differences (notice the little purple dot) combined with relatively large displacement, a tested implementation was not able to calculate a meaningful translation. Some authors bound the range of the scale because of this [11] or manually select point pairs to calculate an initial transformation as an input to the algorithm [5].

2.4 Principal Component Analysis

PCA is a statistical method used to reduce the dimensions of a multi dimensional data set while still retaining as much information as possible. It is used in this work for estimating the scale factor between two point clouds. PCA works by transforming the data set to the eigenspace spanned by the eigenvectors of the covariance matrix of the data points. The eigenvalues correspond to the variance observing that the larger the eigenvalue the larger the variance in that direction. If only k dimensions are to be retained, the first k eigenvectors with the greatest

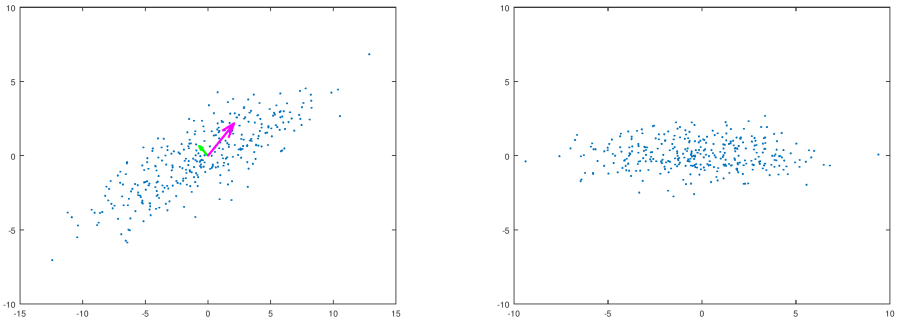


Figure 2.5: On the left some points, in green and magenta the eigenvectors of the points. On the right the points after PCA.

eigenvalues are used [22].

An algorithm for carrying out PCA can be found in [16]. Given the d-dimensional data points $\mathbf{x}_{1\ldots n}$ the covariance matrix $\mathbf{C}$ needs to be calculated first

$$\mathbf{C} = \sum_{i=1}^{n} (\mathbf{x}_i - \bar{\mathbf{x}}) (\mathbf{x}_i - \bar{\mathbf{x}})^T \tag{2.29}$$

with $\bar{\mathbf{x}}$ being the mean of all data points

$$\bar{\mathbf{x}} = \frac{1}{n} \sum_{i=1}^{n} \mathbf{x}_i. \tag{2.30}$$

The next step is calculating the eigenvalues $\lambda_{1..d}$ and eigenvectors $\mathbf{v}_{1..d}$ of of the covariance matrix $\mathbf{C}$. There are many methods to do this and the exact details of such a computation are left out here. Now the K eigenvector corresponding to the k largest eigenvalues are selected and arranged in a $d \times k$ matrix $\widetilde{\mathbf{V}}$. The data points can be transformed into the new principal components space by applying this matrix

$$\boldsymbol{\xi}_i = \widetilde{\mathbf{V}} \mathbf{x_i}. \tag{2.31}$$

Reconstruction of the original points if it is desired can be achieved in the following way:

$$\mathbf{x}_i = \widetilde{\mathbf{V}}^T \boldsymbol{\xi}_i + \bar{\mathbf{x}} \tag{2.32}$$

An example for a PCA of set of two dimensional points can be seen in in Figure 2.5. While not for reducing dimensionality, PCA can also be used when dealing with 3D point clouds. One application is for example fitting of planes to a set of points [34]. As the input is 3D points, the covariance matrix is a 3×3-matrix. If the points lie in a plane then there is relatively few variance of the position of the points in the direction of the normal vector of the plane. Applying PCA this fact can be exploited to get an estimation for the normal of the plane as the eigenvector corresponding to the smallest eigenvalue. The other two eigenvectors span the plane at the mean of the set of points.

2.5 RANSAC

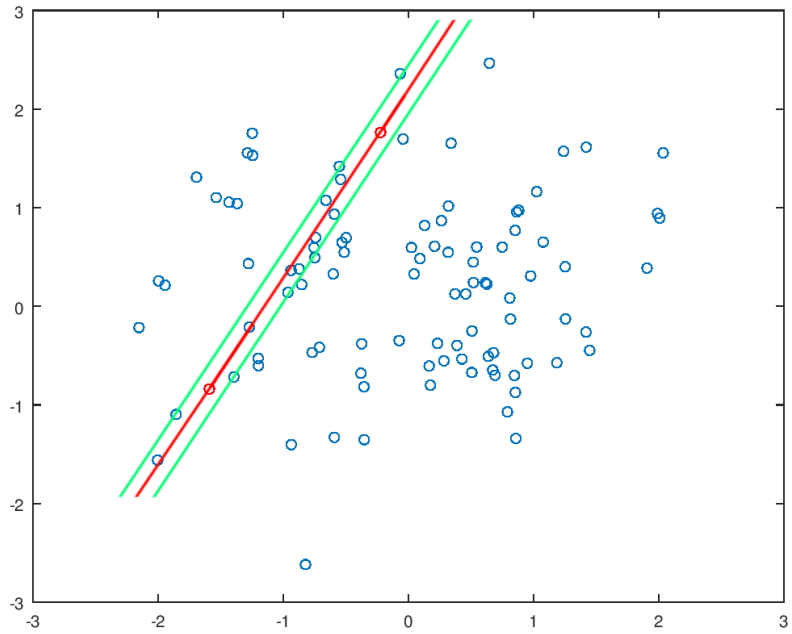


Figure 2.6: Fitting a line into set of Points. The two red points form the set S_i for this iteration with the red model line fitted to them. The points inside the green tolerance lines from the consensus set S_i^*. The figure does not show the end result but rather an intermediate step with a bad model.

RANSAC is the name of a method introduced by Fischer and Bolles in 1981 [14] for fitting a model to data. The 4PCS algorithm is a RANSAC style algorithm. The algorithm developed in this work is a hybrid approach of the RANSAC idea and iterative features. The RANSAC method can be described in general as follows:

1. Given the set of data points P, select a subset S_i of n points randomly where n is the number of points needed to compute a model.
2. Create a model M_i using the points in S_i.
3. Determine the set of points S_i^* that agree with the model. S_i^* is called the consensus set of M_i.
4. If S_i^* is larger than some threshold accept it or repeat steps 1 to 3.
5. Create a new final model M^* using the accepted consensus set. If no set was accepted after the maximum number of iterations l, use the largest consensus set or fail.

Applied to the example of fitting a line into a set of points, each iteration 2 points S_i are randomly drawn from the entirety of all points P as 2 points are sufficient to determine a line.

Then the line M_i is fitted to the two points and all points are checked if they are within a predefined small tolerance distance to the line. These points form the consensus set S_i^*. The situation is visualized in Figure 2.6. If the consensus set is large enough the best fitting line is calculated M^* according to it and returned as the final result. Otherwise the process is iterated by randomly selecting two new points.

The RANSAC technique can be applied to a wide variety of problems as it is very general. Examples include the fitting of lines or planes as described above, shape detection in point clouds [37] or even feature-based registration of point clouds [19]. Depending on the application is also the choice of the parameters not given by the description of the algorithm. For the required number of iterations l an estimation is given dependent on the wanted probability that an error-free subset of n points is chosen once p_s and the probability that a data point is within the specified tolerance to a model p_e:

$$l = \frac{\log(1 - p_s)}{\log(1 - p_e^n)} \tag{2.33}$$

For the other two parameters it is harder to reach general criteria. Most importantly, no general recommendation can be given for the error tolerance for which a data point is thought to agree with a model as it is highly dependent on the data itself and wanted accuracy. The second parameter is the number of data points required to accept a model. It is also hard to determine for the arbitrary case and should be generally chosen as a higher value or if so desired this early stop criterium can be dropped completely.

Chapter 3

Four point congruent sets

The original 4-points congruent sets (4PCS)-algorithm is based on the RANSAC method. It exploits the fact that affine transformations preserve the ratios of distances between points lying on a straight line (Section 2.2.1) in order to calculate the transformation between two point clouds. It was first described by Aiger et al. in 2008 [1]. Here, only the rigid application is described as was done originally. Section 3.1 describes the set of points that the algorithm operates on and some properties of these that can be calculated easily. An overview of the algorithm is given in Section 3.2. Sections 3.3 and 3.4 describe specific steps of the algorithm in more detail.

3.1 Coplanar bases

The algorithm selects coplanar bases in one point cloud and searches for corresponding candidates in a second point cloud. A coplanar base is a set of four points **a**, **b**, **c** and **d**, which are contained in one geometric plane. An example of such a base can be seen in Figure 3.1, showing the intersection point **e** and the angle α between the line segments connecting **a** and **b** and **c** and **d**. The intersection point **e** always exists because the diagonals of a quadrilateral always intersect. If the four points **a**, **b**, **c** and **d** do not form a valid quadrilateral, they can be reordered such that they form a quadrilateral. The ratios r_1 and r_2 of the lengths along the diagonals to the intersection point to their total length (Equation 3.1) are the main features used to identify a coplanar base in conjunction with the angle $\alpha = \angle \mathbf{a} - \mathbf{b}, \mathbf{c} - \mathbf{d}$ between $\mathbf{a} - \mathbf{b}$ and $\mathbf{c} - \mathbf{d}$.

$$r_1 = \frac{\|\mathbf{a} - \mathbf{e}\|}{\|\mathbf{a} - \mathbf{b}\|} \qquad r_2 = \frac{\|\mathbf{c} - \mathbf{e}\|}{\|\mathbf{c} - \mathbf{d}\|} \tag{3.1}$$

The intersection point can be calculated from the four points of a base and the ratios as follows:

$$\mathbf{e} = \mathbf{a} + r_1(\mathbf{b} - \mathbf{a}) \qquad \mathbf{e} = \mathbf{c} + r_2(\mathbf{d} - \mathbf{c}) \tag{3.2}$$

The ratios r_1, r_2 can be calculated directly from the four base points by equating both right-hand terms of Equation (3.1):

$$\mathbf{a} + r_1(\mathbf{b} - \mathbf{a}) = \mathbf{c} + r_2(\mathbf{d} - \mathbf{c}) \tag{3.3}$$

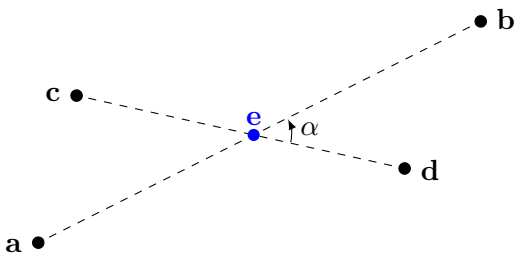


Figure 3.1: A coplanar base formed by the points **a**, **b**, **c** and **d**

Equation (3.3) represents a system of three linear equations of which any two equations are sufficient to determine r_1 and r_2 by solving for r_1 and r_2 by eliminating one of them (here r_2). For example, using the first two vector components and with **ab** being the vector from **b** to **a** (likewise for **cd**):

$$\begin{aligned} r_1 \cdot ab_i - r_2 \cdot cd_i &= (c_i - a_i) \; \forall 1 \leq i \leq 3 \\ r_1 \left(ab_1 - \frac{cd_1}{cd_2} ab_2 \right) + r_2 \left(-cd_1 + \frac{cd_1}{cd_2} cd_2 \right) &= (c_1 - a_1) - \frac{cd_1}{cd_2}(c_2 - a_2) \\ r_1 &= \frac{c_1 - a_1 - \frac{cd_1}{cd_2}(c_2 - a_2)}{ab_1 - \frac{cd_1}{cd_2} ab_2} \\ r_2 &= \frac{c_2 - a_2 - r_1 \cdot ab_1}{-cd_2} \end{aligned} \tag{3.4}$$

The unused equation of the third components can be used to verify the results and to assess the planarity of the four points. If they all are contained in a plane, the line segments will intersect. However if they are only coplanar with regards to some tolerance value, the line segments will not intersect but come into a small distance of eachother. The third equation will not hold anymore and there will be a small of difference in the values of both sides of it.

3.2 The algorithm

A high level overview of of the 4PCS-algorithm is shown in Algorithm 5. Given four points from point cloud P that form a coplanar base B, the main idea behind the algorithm is to find four points in Q that are congruent to B. The identifying features of such coplanar bases are the ratios r_1 and r_2 which are unaffected by affine transformations. Therefore, they can be used to find candidates for the corresponding points forming B in Q. Selecting the best matching set of points is done by calculating the transform that would align the candidate to B optimally in least-squares sense (see Section 2.2.2). Then each of these transforms is judged by how well it aligns the whole point clouds. This is done by simply comparing the number of points in Q for which at least one point in a small region around it can be found in P (in a kD-tree this can be done by using Algorithm 1). This procedure is done in a RANSAC-style loop where in each iteration multiple models are created (the transformations obtained from the candidates). In difference to the original RANSAC formulation, the models are not created randomly but

Algorithm 5: The 4PCS algorithm

input : Two point clouds P and Q
output: Transformation t_{best} to register Q with P

```
t_best ← null
for 1 ≤ i ≤ l do
    Select a coplanar base B in P
    C ← 4 point sets in Q that are congruent to B
    T ← {Transformation that aligns c to B|c ∈ C}
    for t ∈ T do
        Apply T to Q and calculate point correspondences of P and Q if t aligns Q and
         P better than t_best then
            t_best ← t
        end
    end
end
return t_best
```

selected according to the chosen coplanar base in P using the algorithm described in Section 3.4. The consensus sets consist of the point pairs $\in P \times Q$ that agree with a transformation, the number of pairs is used to judge the quality of each transformation. These pairs can be found by applying the transformation and then searching for each point in Q for an equivalent point in P in a small δ region. The number of required iterations L can be calculated according to Equation (2.33) as

$$L > \frac{\log 1 - p_s}{\log 1 - p_g^N}, \tag{3.5}$$

where p_s is the desired success probability and p_g is the probability that a randomly chosen point from P can also be found in Q. This probability either has to be available by knowledge of the data set or can be estimated. If this is not possible, the number of iterations can also be set manually.

3.3 Selecting a coplanar base

Extracting a coplanar base from a point cloud is straightforward, any three randomly selected points will form a plane. A forth point can be found by iterating through the points or randomly selecting one until the selected point is coplanar with the previously selected set of three points. However, the selection of the target base has a major influence on the quality of the resulting calculated transformation. At least one of the diagonals has to be sufficiently long to reduce the influence of noise and other errors as can be seen in Figure 3.2. This also has advantages when matching to a lower density point cloud, where the impact of the lower sampling rate of 3D geometry decreases with higher lengths. On the other hand it has to be ensured that the matching points can be found in the other point cloud that is going to be matched. Selecting the maximum length between two points can fail because it is susceptible to outliers. Arriving

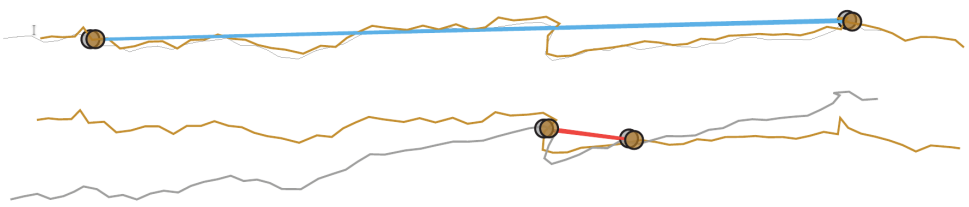


Figure 3.2: Long distances are more robust against noise. (Source: [1])

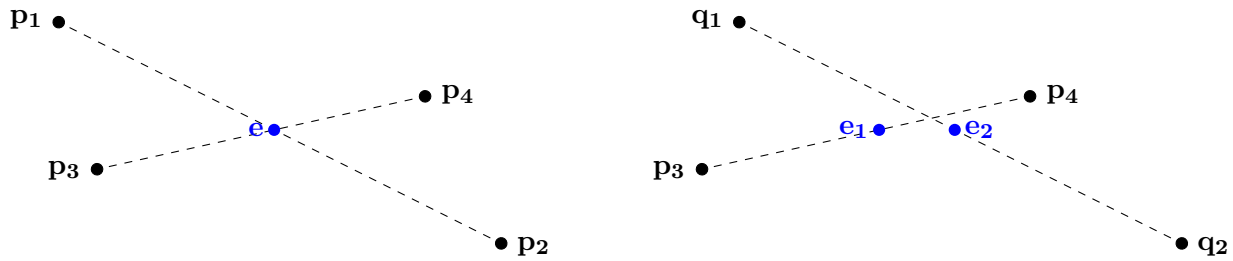


Figure 3.3: The four points on the right are not congruent to the ones in Figure 3.1, the four points on the left are

at a suitable length can be either done through prior knowledge of the two point clouds being registered or through a iterative process. First, a relatively long goal length l is chosen. If the selected length results in no matching case being found, it could be reduced in the following steps to for example $\frac{1}{2}l$, $\frac{1}{4}l$, $\frac{1}{8}l$ and so on.

3.4 Finding corresponding bases

For finding a 4-point set in a scan Q congruent to a coplanar base B in scan P Aiger et. al describe a straightforward but naive method (see Algorithm 6). They start by finding candidates for the point pairs $(\mathbf{a}, \mathbf{b})$ and $(\mathbf{c}, \mathbf{d})$ by looping over all possible point pairs in Q. Because they restrict the searched transformations to rigid ones the pairs can be filtered by comparing their distance to the distance $d_1 = \|\mathbf{a} - \mathbf{b}\|$ or $d_2 = \|\mathbf{c} - \mathbf{d}\|$. For each of these pairs a potential intersection point is calculated according to the ratios r_1 and r_2 of B as in Equation (3.2) - $\mathbf{e_1}$ when the pairs corresponds to $(\mathbf{a}, \mathbf{b})$, $\mathbf{e_2}$ when it corresponds to $(\mathbf{c}, \mathbf{d})$. To facilitate fast lookup the points $\mathbf{e_1}$ are inserted into a range tree structure E_{1t} like a kd-tree.

To form candidates for matching from these virtual intersection points, the tree E_{1t} is searched for nearby points E_{1m} around the points E_2. In order to be congruent to B, two points pairs that both have the correct distance, also have to share their virtual intersection points (cf. Figure 3.3). The set of candidates is formed by combining the two points that were used to calculate $\mathbf{e_{1}}_m$ and the two points that were used to calculate $\mathbf{e_{2}}_i$. Finally, additional criteria such as the angle α between the two connecting vectors and the planarity of the four points are enforced.

The described algorithm has some obvious inefficiencies. For example the last described step

Algorithm 6: `FindCongurent` as described by Aiger et al. in [1]

input : Coplanar base $B \equiv \{\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}\}$ and its ratios r_1 and r_2, Set of 3D points Q, uncertainty δ
output: Set of congruent coplanar bases C congruent to B

$d_1 \leftarrow \|\mathbf{a} - \mathbf{b}\|$
$d_2 \leftarrow \|\mathbf{c} - \mathbf{d}\|$
$R_1 \leftarrow \{(\mathbf{p_i}, \mathbf{p_j} \in Q \times Q) \,|\, |\|\mathbf{p_i} - \mathbf{p_j}\| - d_1| \leq \delta\}$
$R_2 \leftarrow \{(\mathbf{p_i}, \mathbf{p_j} \in Q \times Q) \,|\, |\|\mathbf{p_i} - \mathbf{p_j}\| - d_2| \leq \delta\}$
$E_{1t} \leftarrow \emptyset$; `/* A range tree, for example a kd-tree */`
foreach $(p_i, p_j) \in R_1$ **do**
  $\mathbf{e_1} = \mathbf{p_i} + B.r_1\,(\mathbf{p_j} - \mathbf{p_i})$
  $E_{1t} \leftarrow E_{1t} \cup \{\mathbf{e_1}\}$
end
$E_2 \leftarrow \emptyset$
foreach $(p_i, p_j) \in R_2$ **do**
  $\mathbf{e_2} \leftarrow \mathbf{p_i} + B.r_2\,(\mathbf{p_j} - \mathbf{p_i})$
  $E_2 \leftarrow E_2 \cup \{\mathbf{e_2}\}$
end
$C \leftarrow \emptyset$
foreach $\mathbf{e_{2}}_i \in E_2$ **do**
  $E_{1m} \leftarrow$ all points in a δ-region around $\mathbf{e_2}$ in E_{1t}
  foreach $\mathbf{e_{1}}_m \in E_{1m}$ **do**
    $C \leftarrow C \cup \{$4-point set with the points that $\mathbf{e_{2}}_i$ and $\mathbf{e_{1}}_m$ were calculated from$\}$
  end
end
$C \leftarrow \{c \in C | c$ is congruent to $B\}$
return C

enforcing of constraints is done after the sets were constructed which means they were saved into memory only to be discarded. The algorithm developed in this work improves upon these inefficiencies it and its improvements are described in the next chapter.

Chapter 4

Point cloud alignment with iterative scale estimation

4.1 Introducing scale estimation

The motivation behind this thesis is to find the transformation between point clouds acquired from different sources and captured at different times so no prior knowledge about the relation between them is known. The algorithm in Chapter 3 while relying on the fact that ratios of distances are preserved across affine transformation, achieves its main performance improvement over the naive approach of checking all quadruples from the fact that when studying rigid transformations it is possible to filter point pairs by their distance. The described approach is not applicable in a cross-source scenario as the scale of point clouds acquired with different methods is not necessarily consistent. For example, point clouds produced by SfM lack any absolute reference and can only provide information about relations inside the point cloud. The end result can be arbitrarily scaled and oriented with respect to the real world. In contrast, Laser scanning produces absolute distance measurements because the constant speed of light serves as a reference to tie the measurements to the space. Figures 4.1 and 2.4 show this difference between both measurement methods, in yellow a point cloud obtained via laser scanning is displayed with absolute dimensions and in magenta a point cloud of the same object is shown that was reconstructed via SfM. The difference in dimension is apparent, the magenta SfM point cloud is tiny compared to the yellow one - almost invisible at the chosen zoom level. Therefore, an algorithm for aligning point clouds of different scales cannot enjoy the performance benefit achieved by comparing point distances across point clouds. With this thesis, an algorithm is proposed, which is able to estimate the unknown scale between two point clouds while alignment is in progress. However, in order to shrink the search space an initial guess is used to start the process, which is improved iteratively through application of a transformation. A hybrid approach has been implemented randomly creating transformations and seeing which fits best in a RANSAC fashion and using the previous iterations results to improve the current guess and speed it up by narrowing the search space. In addition, individual point properties can be used to improve the result. The first that comes to mind are surface normals which can be calculated from the underlying 3D data in a straightforward fashion when not readily available by applying

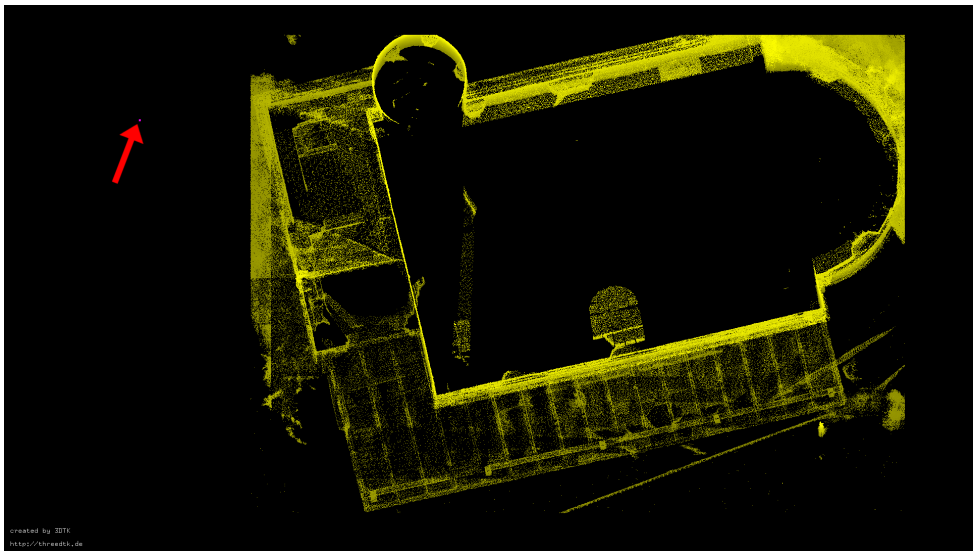


Figure 4.1: The same scene as in Figure 2.4 from a bird's eye view. The magenta point cloud is almost invisible when compared to the yellow one.

PCA (cf. Section 2.4). Hereby, the normals of two or more points are used to identify structures consisting of multiple points. The absolute direction of normals is not usable in comparisons due to the unknown orientation between both datasets. Other data that could be compared directly if it is available could be color information or the reflectance value of points for example.

Algorithm 7 shows an overview of the proposed algorithm. At first glance, it follows the same basic structure as the 4PCS algorithm discussed in the previous chapter. The main difference is created by the unknown scale between the two input point clouds. At the beginning an initial scale factor s is estimated as a starting point for the algorithm. The first iteration is restricted to finding matching sets of points using only this scale factor. After each iteration, s is assigned the scale component of the current best transformation. Depending on how many points are aligned correctly by this transformation, the quality of the scale factor included in the transformation is estimated, which is used to determine an acceptable range for distance comparisons in the next iteration. Creating the hybrid approach mentioned above, each iteration improves the following one. If a transformation aligns more than 90 % of points it is deemed good enough and the iteration is terminated. This early termination condition corresponds to step 4 as described in the original RANSAC algorithm (Section 2.5). In a last step, the ICP algorithm is applied to both point clouds as a refinement by the identified transformation. It is restricted to rigid transformations and is used for improving the rotation and translation. The scale should have been identified by the algorithm beforehand.

Section 4.2 describes the estimation of an initial scale value comparing four different methods. Several steps of the original algorithm had to be adapted to support an unknown scale factor including the search for points having congruent bases. Simultaneously, performance improvements have been introduced such as replacing tree-based pair matching with direct construction of candidate sets. Finding congruent bases is the main task within the algorithm and is performed entirely different than in Algorithm 6. The proposed changes are described in

Algorithm 7: Algorithm to align two point clouds of different scales

input : Two point clouds P and Q
output: Transformation t_{best} to register Q with P

$t_{best} \leftarrow$ null
$s \leftarrow$ initial scale estimation
for $1 \leq i \leq l$ **do**
  Select a coplanar base B in P
  $C \leftarrow$ 4 point sets in Q that are congruent to B using current best scale s
  T $\leftarrow$ {Transformation that aligns and scales c to $B|c \in C$}
  for $t \in T$ **do**
    if t aligns Q and P better than t_{best} **then**
      $t_{best} \leftarrow t$
      $s \leftarrow$ scale component of t
    end
  end
  if t_{best} aligns 90% of points **then**
    break
  end
end
return result of ICP-algorithm of P and Q with t_{best} as starting transformation

Section 4.4. In Section 4.3 properties of two point clouds of varying scale and density such as the maximum deviation of the distance between a pair of two points and their corresponding pair in a second dataset are derived which are required later on.

4.2 Initial scale estimation

The initial scale estimation impacts the quality of the first estimated transformation which is able to align many points, reducing the range that needs to be considered in subsequent transformation and enabling a faster convergence. A good initial estimation therefore reduces the required iterations and has a positive influence on the algorithm's execution time. In this section, four different methods have been evaluated. The resulting scale factors are either compared against the true scaling value if available or - in the case of true cross source data sets - against a manually estimated value. These values were obtained by careful extraction of coordinates of prominent points like the tip of a tower and calculation of the resulting scale according to equation (2.24). Table 4.1 lists the results of applying the different heuristics to four different pairs of data sets. The "Bunny" and "Bunny rotated" pairs are based on the same point clouds. However, in the second pair the data sets are rotated by 45° about each axis against each other. The simplest heuristics is taking the axis-aligned bounding box of a point cloud and comparing the dimensions of it to the bounding box of another point cloud. This approach is dependent on point cloud orientations and therefore not applicable to arbitrary oriented data sets. A variant of the heuristics relies on the third root of bounding box volume

Data Set	bounding box dimension	bounding box volume	bounding sphere	PCA	true value/- manually selected points
Cube	4	4	4	3.99903	4
Bunny	10.0072	10.0072	11.4274	10.0177	10
Bunny rotated	9.33278	9.25935	11.4274	10.0177	10
Chapel	272.677	269.995	243.042	348.29	345.481

Table 4.1: Scale values gathered via some heuristics.

for scale estimation. Due to the axis-alignment of the bounding box, this variant only brings an advantage when the rotation of the point clouds against each other is close to multiples of 90° around each axis. The scale between the two bunny data sets estimated using these methods varies by 7.47 %. An alternative to bounding boxes which are fragile when orientations differ is the usage of bounding spheres which are axis-independent. But the results obtained from either comparing bounding sphere radii or volumes (both methods delivering the same result because the volume of a sphere depends only on the radius) vary largely in quality. Both, significant over- and underestimation, can be observed in Table 4.1.

The last heuristic is based on PCA. Noting that the covariance matrix $\mathbf{C}$ is symmetric (Equation (2.29)) it is possible to diagonalize it in a way that

$$\mathbf{\Lambda} = \mathbf{V}^{-1}\mathbf{C}\mathbf{V} \tag{4.1}$$

with $\mathbf{\Lambda}$ being a diagonal matrix with its entries consisting of the eigenvalues of $\mathbf{C}$ and $\mathbf{V}$ being the matrix whose columns are the eigenvectors of $\mathbf{C}$. Equation (4.1) can be interpreted as a change of basis using basis matrix $\mathbf{V}$. The m-th eigenvalue of the covariance matrix corresponds to the variance of the data in the direction of the m-th eigenvector [22]. The size of the eigenvalues is related to the extent of the point cloud along the eigenvectors. Because the orientation of the eigenvectors inside the data is only dependent on the data itself and is rotated with the data and the variance of the points along them is also unaffected by rigid transformations, the magnitude of the eigenvalues can be used for a rough comparison of the size of the point clouds and to derive an estimated scale factor. This is done by taking the square root of each eigenvalue and using the average of the three eigenvalue ratios. Let $\lambda_{p,1\ldots3}$ be the eigenvalues of point cloud P sorted either ascending or descending and $\lambda_{q,1\ldots3}$ the eigenvalues of point cloud Q sorted in the same fashion, then approximated scale factor s_{PCA} is calculated as

$$s_{\text{PCA}} = \frac{1}{3}\left(\sqrt{\frac{\lambda_{p,1}}{\lambda_{q,1}}} + \sqrt{\frac{\lambda_{p,2}}{\lambda_{q,2}}} + \sqrt{\frac{\lambda_{p,3}}{\lambda_{q,3}}}\right) \tag{4.2}$$

In addition to Table 4.1, Figure 4.2 gives a visual overview of the quality of the different heuristics utilizing the “Chapel” data set. In green the point cloud is displayed with the scale factor derived from a bounding sphere is displayed, for the red and blue ones are the bounding box dimensions and volumes used. The point cloud scaled according to PCA is shown in cyan.

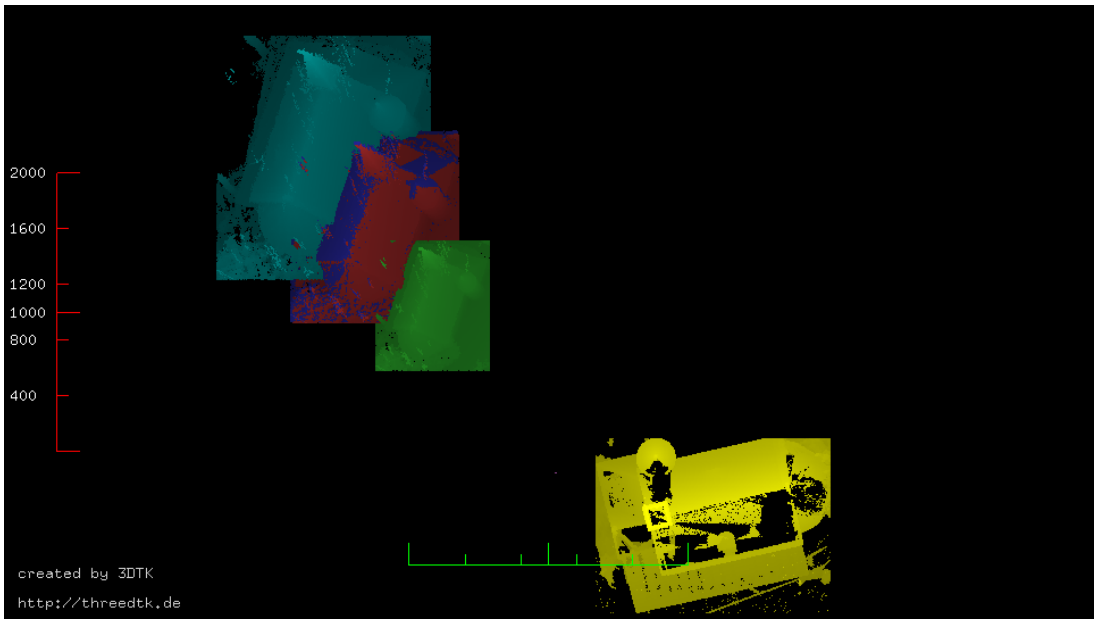


Figure 4.2: Comparison of different heuristics for an initial scale factor estimation. The initial situation is displayed in Figure 4.1.

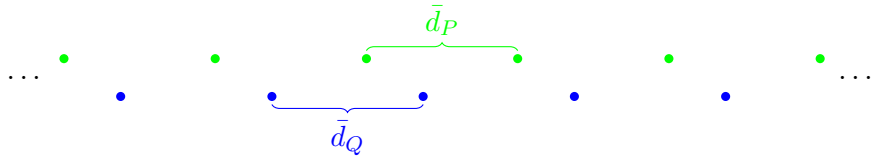
As can be seen in the figure and observed from the values in the table, the scale factor calculated based on PCA is producing the point cloud which size is closest to the reference point cloud.

4.3 Derivation of useful point cloud properties

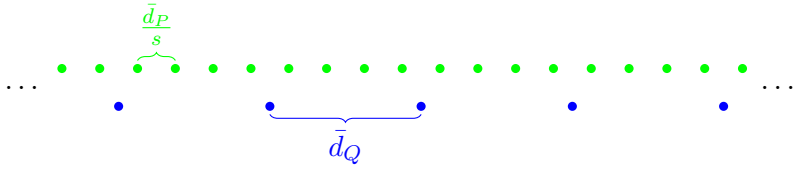
A pair of point clouds with different point densities and scales has certain properties which are useful in the further implementation process. It is assumed that two given point clouds P and Q are aligned correctly but have a different point density and hence differing average point distances $\bar{d}_P$ and $\bar{d}_Q$. Further, the scale difference s between the two point clouds that correctly scales Q to P is known. Let, without loss of generality, P be the point cloud with higher average point density. Then the effective differences in point density and average point distance are not only dependent on the density and average distance values of the scans itself but also on the scale factor s. In Figure 4.3 both point clouds initially have the same average point distance. But when they are resized by applying the known scale factor s it is clear that in reality P is much more dense than Q.

For registering point clouds, points, point pairs and distances between points in different point clouds have to be compared. This is especially true when the 4PCS method is applied and corresponding points to various points in P are searched in Q. Assuming the same average point distances as above and that P and Q are scaled to the same scale, the distance $\|\boldsymbol{\delta}_{\mathbf{p}'}\|$ between a random point $\mathbf{p}$ in Q and the corresponding point $\mathbf{p}'$ can be up to

$$\max\|\boldsymbol{\delta}_{\mathbf{p}'}\| = \frac{\bar{d}_Q}{2}. \tag{4.3}$$



(a) Point clouds P and Q have the same average point distance.



(b) P scaled to the same size as Q.

Figure 4.3: Resized to the same scale as point cloud Q (in blue), has point cloud P (in green) higher point density.

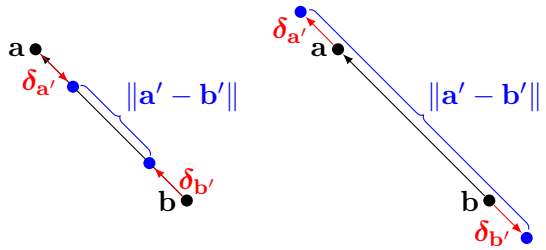


Figure 4.4: $\boldsymbol{\delta}_{\mathbf{a}'}$ and $\boldsymbol{\delta}_{\mathbf{a}'}$ pointing in the same/opposite direction of $\mathbf{a} - \mathbf{b}$ maximizes the difference $|\|\mathbf{a} - \mathbf{b}\| - \|\mathbf{a}' - \mathbf{b}'\||$

Note that a point Q can be considered to multiple points in P. Using this result the maximum deviation of the distance d between a pair of two points $\mathbf{a}$ and $\mathbf{b}$ and the corresponding pair $\mathbf{a}'$ and $\mathbf{b}'$ in Q can be found. Each of both points could be shifted a distance of up to $\frac{\bar{d}_Q}{2}$ in some direction described by the unit vectors $\hat{\boldsymbol{\delta}}_{\mathbf{a}'} = \frac{\boldsymbol{\delta}_{\mathbf{a}'}}{\|\boldsymbol{\delta}_{\mathbf{a}'}\|}$ for $\mathbf{a}$ and $\mathbf{b}'$ and $\hat{\boldsymbol{\delta}}_{\mathbf{b}'} = \frac{\boldsymbol{\delta}_{\mathbf{b}'}}{\|\boldsymbol{\delta}_{\mathbf{b}'}\|}$ for $\mathbf{b}$ and $\mathbf{b}'$.

$$\begin{aligned} \max\left(\left|\|\mathbf{a} - \mathbf{b}\| - \|\mathbf{a}' - \mathbf{b}'\|\right|\right) &= \max\left(\left|d - \|\mathbf{a} + \boldsymbol{\delta}_{\mathbf{a}'} - (\mathbf{b} + \boldsymbol{\delta}_{\mathbf{b}'})\|\right|\right) \\ &= \max\left(\left|d - \|\mathbf{a} - \mathbf{b} + \boldsymbol{\delta}_{\mathbf{a}'} - \boldsymbol{\delta}_{\mathbf{b}'}\|\right|\right) \end{aligned} \tag{4.4}$$

In order to maximize the absolute value of an expression both the maximum and minimum value of the argument have to be examined. Here, only the maximum case is performed, the argumentation for the minimum value is symmetric and can be done analogously. The term $\mathbf{a} - \mathbf{b}$ is constant, the expression is minimized when the combination of the other vectors points in the opposite direction:

$$\begin{aligned} \max\left(d - \|\mathbf{a} - \mathbf{b} + \boldsymbol{\delta}_{\mathbf{a}'} - \boldsymbol{\delta}_{\mathbf{b}'}\|\right) &= d - \min\left(\left\|(\mathbf{a} - \mathbf{b}) + \|\boldsymbol{\delta}_{\mathbf{a}'} - \boldsymbol{\delta}_{\mathbf{b}'}\|\frac{\mathbf{b} - \mathbf{a}}{d}\right\|\right) \\ &= d - \min\left(d - \|\boldsymbol{\delta}_{\mathbf{a}'} - \boldsymbol{\delta}_{\mathbf{b}'}\|\right) \end{aligned} \tag{4.5}$$

The magnitude of the difference of the error terms is maximized if both point in the opposite direction of each other and have their maximum value of $\frac{\bar{d}_Q}{2}$ (Equation (4.3).)

$$\begin{aligned} d - \min\left(d - \|\boldsymbol{\delta}_{\mathbf{a}'} - \boldsymbol{\delta}_{\mathbf{b}'}\|\right) &= d - \left(d - \left(\frac{\bar{d}_Q}{2} + \frac{\bar{d}_Q}{2}\right)\right) \\ &= \bar{d}_Q \end{aligned} \tag{4.6}$$

This fits with the case that is intuitively imagined that $\boldsymbol{\delta}_{\mathbf{a}'}$ and $\boldsymbol{\delta}_{\mathbf{a}'}$ shorten $\mathbf{a} - \mathbf{b}$. For the opposite case of minimizing the difference, $\mathbf{a} - \mathbf{b}$ is lengthened by the maximum amount. The situation is also displayed in Figure 4.4.

4.4 Finding corresponding bases

Section 3.4 and Algorithm 6 describe a method for finding sets of points in Q that are congruent to a base $(\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d})$ found previously in P. However, this naive approach has some drawbacks that make the application to larger scale problems unpractical. The main obstacle is it's multi-pass data flow and quadratic nature. First, all possible pairs of points in Q are considered candidates for either $\mathbf{a}$ and $\mathbf{b}$ or $\mathbf{c}$ and $\mathbf{d}$. For each of these pairs, a possible intersection point is calculated and stored. Then these intersection points are matched via lookup in a tree and combined to 4 point sets. Finally, the sets are filtered such that only sets congruent to base in P are kept. The area where this algorithm can be primarily improved is the memory consumption. For each of the pairs that have a similar distance as $\mathbf{a}$ and $\mathbf{b}$ or $\mathbf{c}$ and $\mathbf{d}$ two additional points (the points $(\mathbf{p_i}, \mathbf{p_j})$ could either correspond to $(\mathbf{a}, \mathbf{b})$ or to $(\mathbf{b}, \mathbf{a})$) have to be stored. Also, a lookup from these additional points to the points that they were calculated from is necessary to form a candidate set if two potential intersection points fall together (see line 19 of Algorithm 6). This implies the requirement of either an additional data structure like a map or some other way to store this extra metadata. Furthermore, a complete set of potential candidates is built first and filtered only afterwards. In summary, much data is processed potentially multiple times only to be discarded afterwards. Avoiding this and directly constructing only candidate sets that are congruent to the target base not only reduces memory requirements but also improves runtime because less data has to be processed.

The goal is to find sets of four point in Q that are congruent to a given base $(\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d})$. Congruent sets will have the same ratios r_1, r_2 and angle α (up to some uncertainty). Using these properties of congruent sets it is possible to calculate possible positions for the other two points from $\mathbf{a}$ and $\mathbf{b}$. The set of possible point pairs of candidates for $\mathbf{c}$ and $\mathbf{d}$ can therefore be constructed from points which have been detected at the calculated positions. Therefore, invalid candidates are omitted from the set of point pairs from the beginning. The line segments $\mathbf{a} - \mathbf{b}$ and $\mathbf{c} - \mathbf{d}$ intersect at $\mathbf{e}$ with angle α. If $\mathbf{a}$, $\mathbf{b}$ and $\mathbf{e}$ or r_1 are known, the location of $\mathbf{c}$ can be narrowed down to a circle. Figure 4.5 shows that the circle is the base of a right cone with tip $\mathbf{e}$ and whose axis runs along $\mathbf{a} - \mathbf{b}$. Its slant height d is the distance from $\mathbf{e}$ to $\mathbf{c}$. The coordinate of the center, a radius length and a surface normal are required to properly specify a circle in 3D space. The normal of the circle is the unit vector along $\mathbf{a} - \mathbf{b}$, the position of the center can be found by moving from $\mathbf{e}$ into the direction to $\mathbf{b}$ by the length h which as the radius can be derived from the slant height:

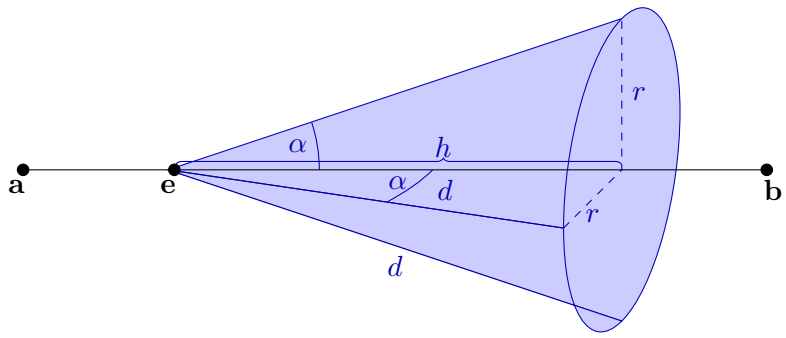


Figure 4.5: Possible positions of point **c** from given points **a** and **b** and properties of a coplanar base as indicated by the base of the cone.

$$\begin{aligned} d &= \|\mathbf{e} - \mathbf{c}\| \\ r &= d \cdot \sin\alpha \\ h &= d \cdot \cos\alpha \end{aligned} \tag{4.7}$$

The quantities required to construct the circle and constrain the possible locations of **c** are either directly available or can be derived from the properties of the input congruent set from P when searching for corresponding congruent sets in Q. Therefore, with the above method it is possible to describe a new algorithm for finding congruent bases which is more performant and requires less memory than Algorithm 6. The algorithm is described as Algorithm 8.

The advantages compared to algorithm 6 are obvious: The check for a set of candidates for **c** and **d** is obsolete as only candidates for **a** and **b** are initially required. The range of allowed point distances is determined by the current best transformation as described in section 4.3. Additionally, the angle between the normals of each point of a point pair is calculated and compared to the angle between normals of the reference points. The notation $n(\mathbf{p})$ describes the surface normal of point **p**. Using this approach, at first the algorithm performs a search for candidate points of **a** and **b**. If a given point **p** is considered for **a** then the candidates for **b** have to be on the surface of a sphere with radius $\|\mathbf{a} - \mathbf{b}\|$ scaled to the scale of Q around **p**. The search for points on a sphere can be performed efficiently for point clouds stored as kD-trees using the method described in Algorithm 2. The allowed tolerance is the average point distance as calculated in Section 4.3 plus the length scaled by the confidence of the current best transformation. If the confidence is at its maximum of 1, this additional term $((1-c)\frac{1}{s}\|\mathbf{a} - \mathbf{b}\|)$ is zero, and the lower the confidence is, points further away are considered. To avoid the need for a map or a similar data structure the calculated intersection points $\mathbf{e}'$ are stored in together with indices to the points that they were calculated from.

For each of these virtual intersection points the point cloud is searched for potential candidates of point $\mathbf{c}'$ as described above. Here the scale s' that relates the distance $\|\mathbf{a} - \mathbf{b}\|$ to the one between the source points of $\mathbf{e}'$ is calculated and used afterwards. Allowing a range of distances here as well, would make no sense as the four point sets correspond to one possible transformation each and should be internally consistent as a single global scale factor is searched and not different scale factors in different directions. With a possible candidate $\mathbf{c}'$ for **c** it is

Algorithm 8: Finding congruent bases in a different scale point cloud

input : Coplanar base $B \equiv \{\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}\}$, Set of 3D points Q, current best scale s, confidence value of current best transformation c

output: Set of congruent coplanar bases C congruent to B

foreach $\mathbf{q_i} \in Q$ **do**
  $J \leftarrow$ `FindPointsSphere`$(Q, \mathbf{q_i}, \frac{1}{s}\|\mathbf{a}-\mathbf{b}\|, \bar{d}_Q + (1-c)\frac{1}{s}\|\mathbf{a}-\mathbf{b}\|)$; `/* Algorithm 2 */`
  foreach $\mathbf{q_j} \in J$ **do**
    if $\angle(n(\mathbf{q_i}), n(\mathbf{q_j}) \approx \angle(n(\mathbf{a}), n(\mathbf{b}))$ **then**
      $E \leftarrow E \cup \{(i, j, \mathbf{q_i} + B.r_1(\mathbf{q_j} - \mathbf{q_i})\}$
    end
  end
end
foreach $\mathbf{e}' \in E$ **do**
  $s' \leftarrow \frac{\|\mathbf{q_i}-\mathbf{q_j}\|}{\|\mathbf{a}-\mathbf{b}\|}$
  $d \leftarrow s'\|\mathbf{e}-\mathbf{c}\|$
  $\mathbf{n} \leftarrow \frac{\mathbf{q_j}-\mathbf{q_i}}{\|\mathbf{q_j}-\mathbf{q_i}\|}$
  $\mathbf{m} \leftarrow \mathbf{e}' + \cos(\alpha) \cdot d \cdot \mathbf{n}$
  $CC \leftarrow$ `FindPointsOnCircle`$(Q, \mathbf{m}, d \cdot \sin(\alpha), \frac{\bar{d}_Q}{2}, \mathbf{n})$; `/* Algorithm 2 */`
  foreach $\mathbf{c}' \in CC$ **do**
    $\mathbf{r} \leftarrow \frac{\mathbf{e}-\mathbf{c}}{\|\mathbf{e}-\mathbf{c}\|}$
    $\mathbf{d}' \leftarrow$ `FindClosest`$(Q, c + s'\|\mathbf{c}-\mathbf{d}\|\mathbf{r}, \bar{d}_Q)$; `/* Algorithm 1 */`
    if $\angle(n(\mathbf{c}'), n(\mathbf{d}')) \approx \angle(n(\mathbf{c}), n(\mathbf{d}))$ **and**
    $\angle(n(\mathbf{c}'), n(\mathbf{d}')) - \angle(n(\mathbf{q_i}), n(\mathbf{q_j})) \approx \angle(n(\mathbf{c}), n(\mathbf{d})) - \angle(n(\mathbf{a}), n(\mathbf{b}))$ **then**
      $C \leftarrow C \cup \{(\mathbf{q_i}, \mathbf{q_j}, \mathbf{c}', \mathbf{d}')\}$
    end
  end
end

straightforward to also verify if the point cloud contains a suitable candidate for $\mathbf{d}$. The direction $\mathbf{r}$ in which it should lie is fixed by the direction to $\mathbf{c}'$ from the point $\mathbf{e}'$ where the search initially started. Again, the distance where the point should be, can be calculated from the corresponding points $\mathbf{c}$ and $\mathbf{d}$. In addition to checking that the directions of the normals of the point pair $\mathbf{c}', \mathbf{d}'$ are in similiar relation to one another as the ones of $\mathbf{c}$ and $\mathbf{d}$, the angles of the normals of the whole set of four points including $\mathbf{q_i}$ and $\mathbf{q_i}$ are compared to the original ones from B to make sure that even less unmatching sets are considered candidates. Only if this last test passes, the four point set is assembled as $(\mathbf{q_i}, \mathbf{q_j}, \mathbf{c}', \mathbf{d}')$ and appended to the list of possible candidates.

Listing 4.1: Paralleizing a for-loop with OpenMP by adding two compiler directives to it.

```
#pragma omp declare reduction (merge: std::vector<estruct> :\
omp_out.insert(omp_out.end(), omp_in.begin(), omp_in.end()))
#pragma omp parallel for reduction(merge: E)
std::vector<estruct> E;
for(size_t i = 0; i < Q.size(); ++i) {
  const auto J = tree.find_points_on_sphere([...]);
  for (size_t k = 0; k < J.size(); ++j) {
    const size_t j = J[k];
    const double normal_angle = angle(n(Q[i]), n(Q[J]));
    if (fabs(normal_angle - angle(n(B.a), n(B.b)) <= rad(1)) {
      E.emplace_back(i, j, Q[i] + B.r1 * (Q[j] - Q[i]));
    }
  }
}
```

4.5 Implementation Details

The algorithm was implemented within the 3DTK framework [32]. The original 4PCS-algorithm has been implemented for comparison. During the implementation of the proposed algorithm a main focus was on good performance and low memory consumption because of its demanding nature. Below, a few approaches are described which have been implemented to achieve these objectives.

The algorithm relies on surface normals. These can either be calculated during run-time or be pre-calculated before algorithm start. As the 3DTK framework only offers methods to recalculate the complete set of surface normals without allocating an unnecessary amount of memory per point, the pre-calculation option has been chosen. The normals are stored with the point data in one file and are read together. This data is then stored in a central location in the implemented program, copies are avoided where possible.

This is supported by one of the available kD-tree implementations in 3DTK. The point data is not stored directly but rather the tree is constructed from indices which can be used to reference data stored in an array or another similar data structure. The indexed kD-tree class has been extended with a wrapper so it can be used as a tree type in the central "Scan" class[1] among other options like a normal kD-tree or an approximate nearest neighbor tree (ANN-tree). That enables classes and code created during this work to always reference and store data by a single index or a pointer to the actual data. Only data like the calculated intersection points is stored directly and only if really necessary.

The two main loops of the algorithm have been parallelized using the OpenMP 4.5 framework [33]. OpenMP allows the easy parallelization of sequential code by adding directives to the code instructing the compiler to generate machine code that splits the work across multiple threads.

[1] `https://sourceforge.net/p/slam6d/code/HEAD/tree/trunk/include/slam6d/scan.h`

Listing 4.1 shows how a simplified implementation of the loop in line 1 of Algorithm 8 can be easily transformed. The annotation `#pragma omp parallel for` instructs the compiler that the following loop should be parallelized across multiple threads. Each thread will receive a separate copy of the vector `E` in which it writes the results it found. Often parallel loops produce results that need be unified or merged across threads. For example when searching for the maximum value inside an array of numbers, each threads finds the maximum value of the portion of the array that it had inspected. The maximum value of the array is the maximum of the maxima found by each thread. OpenMP includes the reduction concept for such tasks. The second part of the directive above the loop (`reduction(merge: E)`) tells the compiler that at the end of the execution of each the thread the previously declared reduction called "`merge`" is to be executed for the private copies of `E`. A reduction is introduced using `#pragma omp declare reduction`. In parentheses follows the name of the reduction and for which types of variables it is applicable, in this case that is a vector of a type able store two indices and one additional point. The last part of the reduction describes the actual reduction operation. This can be for example a function call or any other valid expression of the C++ programming language. OpenMP provides the special variables `omp_in` and `omp_out` that can be used in such a statement. When the reduction is executed `omp_in` will refer to the private copy of the variable of the thread and `omp_out` will be the combined variable after the reductions have been executed. So after the execution of the loop in Listing 4.1, `E` will be the union of the private `E` variables of each thread as the reduction clause simply appends the thread local vectors to the resulting one. Aside from these two compiler directives that have to be added, the rest of the code can be left unmodified and can be written normally like for the sequential case.

Chapter 5

Experimental Results

Multiple experiments have been carried out. In Section 5.1 the runtime of algorithm as described in 4 is compared against the original one described in 3. Also the speedup gained by parallelizing parts of it are investigated. In Sections 5.2 and 5.3 the developed algorithm is used to align two data sets. First it is applied to variations of the Stanford bunny dataset that have been scaled, rotated and sampled down and the qualities of the found transformations are compared. Finally a real world data set consisting of a laser scan and a SfM point cloud of the same chapel is aligned.

5.1 Runtime comparison between different methods for finding corresponding bases and parallel versions

The algorithms for finding corresponding bases (Algorithms 6 and 8) can be be divided into two main parts. First the candidate pairs for **a** and **b** and for the first algorithm also **c** and **d** are formed and then these pairs are used to construct the four point sets that are used to calculate possible transformations between both point clouds. For both of these parts the runtime of the implementations in both algorithms have been compared.

5.1.1 Constructing the set of point pair canidates

First the construction of the set of candidates for the point pairs **a** and **b** and if required **c** and **d** was examined. Figure 5.1 shows the runtimes of different methods for constructing the candidate pairs as first step of the algorithms using the cube dataset. In addition to the so far described methods of simply iterating over all pairs and searching for points that are on the surface of a sphere, an additional method has been implemented and tested. Here instead of searching directly for points that are at a given distance from the query point (in other words lying on the surface of the sphere with the query point in its center), first all points inside a maximum distance from the query point are searched in the kD-tree which can be imagined as searching for all points inside a solid sphere around the query point and then filtered depending on if they fall into the desired distance range. This was done as the the query for all points inside a maximum range inside a kD-tree is a much more common operation as the for this

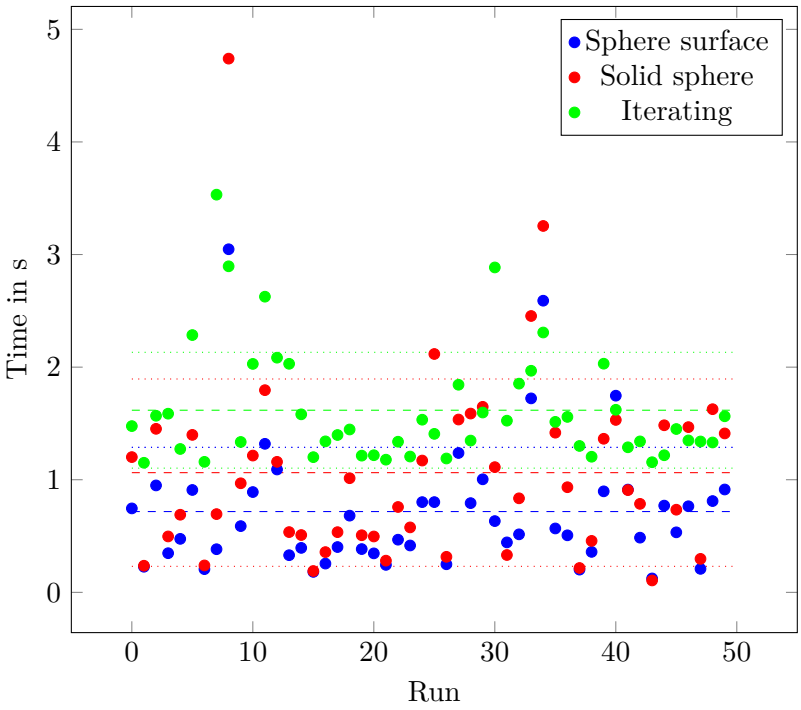


Figure 5.1: Comparing the runtime of different methods for constructing point pair candidates using the cube dataset. The dashed lines represents the mean and the dotted lines show the standard deviations.

work implemented search for points on the surface of a sphere. It was also used as a verification tool by comparing the amount of points returned by both procedures. This additional query procedure can be created easily by adapting Algorithm 1 to not return the closest point found but rather all points inside the given distance d. The test was carried out on a Laptop with a Intel Core i3-7100U CPU with a clock rate of 2.40 GHz. It can be seen that across the 50 runs that were carried out the sphere method was almost always faster than just iterating over all point pairs, its mean runtime is 0.717 s compared to 1.617 s. The difference between querying the kD-tree for points on a sphere and for points in a sphere and filtering afterwards with a mean runtime 1.063 s is generally smaller but the surface method seems the best for the average case. Its runtime is also more predictable. While the former method's runtime has a variance of $0.692\,\mathrm{s}^2$, the variance of the surface method is with $0.325\,\mathrm{s}^2$ almost as low as the one of the simple iteration ($0.265\,\mathrm{s}^2$).

Testing the three methods on the the Stanford bunny data set shows different results. As can be seen in Figure 5.2, searching for points on the surface sphere is still the fastest method with an average runtime of 4.792 s. Iterating over point pairs shows a relatively stable runtime of 15.517 s with variance of $0.680\,\mathrm{s}^2$ because the workload mostly stays the same between multiple runs. The kD-tree based methods show a more volatile behavior. Here, depending on the requested distances, a different amount of tree nodes and leaves has to be visited. The surface method seems to be moderately impacted by this but is almost all the time three times faster than simply iterating over all point pairs. Searching all points inside a sphere defined by a given maximum

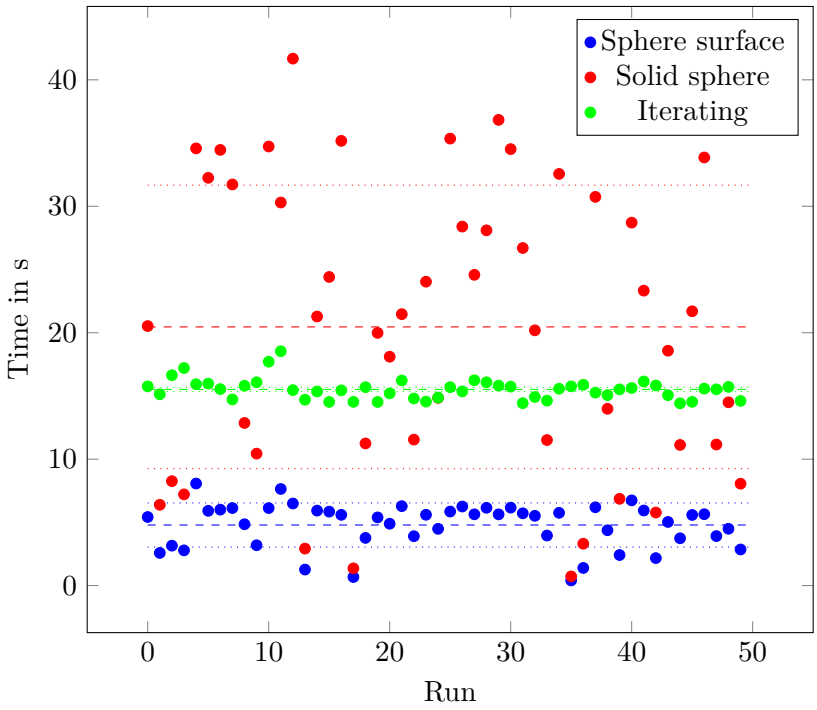


Figure 5.2: Comparing the runtime of different methods for constructing point pair candidates using the bunny dataset. The dashed lines represents the mean and the dotted lines show the standard deviations.

distance and then discarding all points not inside the actually wanted distance range performs much worse than in the previous test. It seems also highly impacted by the aforementioned effect, it has an enormous variance of $125.706\,\mathrm{s}^2$. In singular instances it is almost as fast as the surface method or sometimes the run time falls into the middle of the two approaches. But most of the time it is the worst method out of the three in regards to performance and takes on average longer than the naive iteration with $20.459\,\mathrm{s}$.

In summary, it can be concluded that the sphere surface based method seems to be the best one for the general case. It consistently performs better by a huge margin than simply iterating over all point pairs. The other kD-tree based method of querying all points inside a solid sphere and afterwards discarding all points which are not on its surface is unreliable. Although it performs good in very simple datasets as the two cubes, even a slightly more complicated example like the Stanford bunny shows that it is ultimately flawed and should be avoided as it can perform much worse than the other methods.

The above tests were carried out on already parallelized versions of the algorithms using the OpenMP framework[33]. This was done after determining that sharing the work between multiple threads proved to be beneficial and offered significant speedup. Using the sphere surface method, the runtime for constructing the set of point pair candidates was measured using one, two and four parallel thread with the processor of the used laptop being able to execute 4 threads in parallel. Figure 5.3 shows the results. On the left the raw runtime is shown, on the right it is divided by the time needed for the sequential case. Aside from a few outliers the speedup

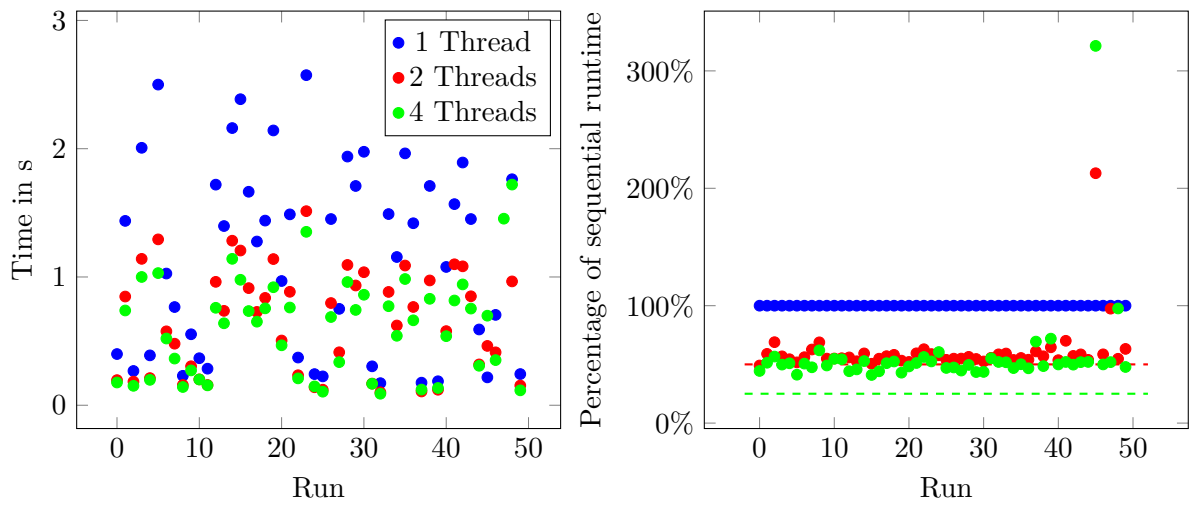
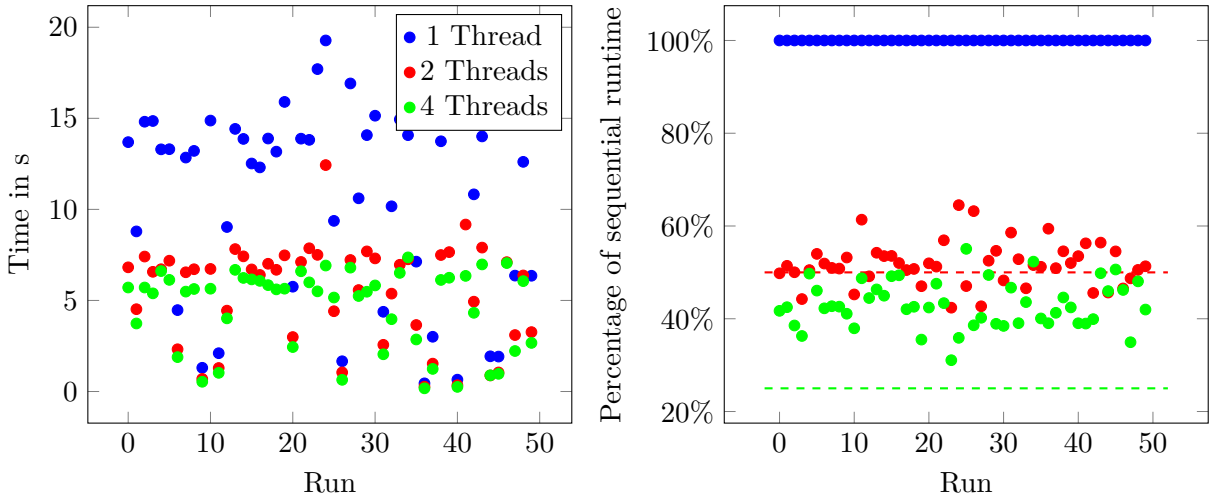


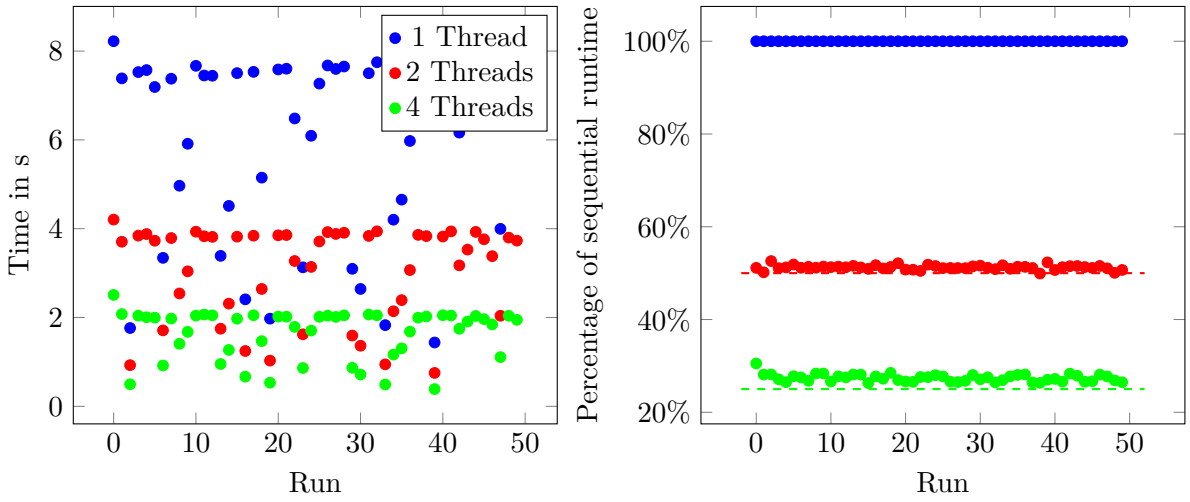
Figure 5.3: Comparing the runtime of constructing point pair candidates using the cube dataset and the sphere surface method with 1, 2 or 4 parallel threads. Dashed lines indicate the optimal speedup gained from using multiple threads.

from using two threads is near the optimal case indicated by the dashed red line. However using four threads doesn't improve performance any further in a significant manner. The runtime improvement is only very slight to using two threads. Here, the overhead of creating multiple threads and merging their results does seem to outweigh the benefit of using more threads. To confirm this the same test was performed on the bunny data set where the pair extraction takes longer. The results can be seen in Figure 5.4a. Again using two threads performs very good and achieves a speedup near the optimum indicated by the dashed line. While using four threads increases the time difference compared to two threads, the performance gain is still not as high as the step from sequential execution to using two threads. A reason for the observed not significant speedup when increasing the thread-count could be that the CPU of the laptop that the tests were performed on, although it is capable of executing four parallel threads at the same time, only consists of two physical cores [9].

In order to confirm this theory the test was repeated on a desktop computer with an Intel Core i5-7600K processor. The CPU has a clock rate of 3.80 GHz and consist of four physical processor cores [9]. Figure 5.4b shows the results. The runtimes are as expected lower compared to the ones from the laptop because of the more powerful hardware. Also, the graph of execution time percentages shows that the potential runtime advantage gained from using multiple could be almost fully exploited. Nearly all data points are near the dashed lines indicating the optimum speedup. It can be concluded that previously not seen performance increase can be traced back to the laptop hardware which is able to execute four parallel threads but cannot exploit the full performance benefit gained from doing so.



(a) Test performed on the laptop.



(b) Test performed on the desktop computer.

Figure 5.4: Comparing the runtime of constructing point pair candidates using the bunny dataset and the sphere surface method with 1, 2 or 4 parallel threads. Dashed lines indicate the optimal speedup gained from using multiple threads.

5.1.2 Assembling the congruent four point sets

A detailed performance comparison of the methods of the algorithms for assembling the candidate four point sets was not possible. Due to the inefficiencies described in Section 4.4 the execution of the implementation of Algorithm 6 took a long time and the 16 GB of available memory didn't suffice causing the program to be aborted early. Table 5.1 shows the results gathered across 10 runs on the desktop computer using the cube data set. The time for Algorithm 6 includes additionally the time needed to filter the constructed four point sets from the

Table 5.1: Comparison of the runtimes of the methods for assembling the congruent four point sets using the cube dataset, values are in seconds. Entries marked with * mean that the program was terminated early due to insufficient available memory.

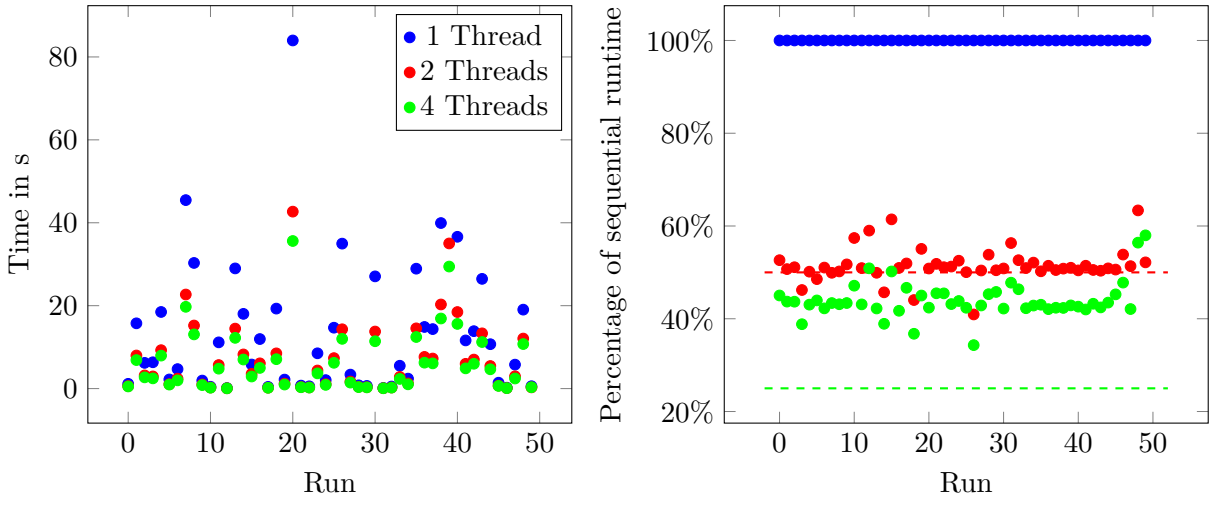
Algorithm 8	15.125	83.577	37.656	8.793	8.330
Algorithm 6	362.814*	667.456*	480.076*	896.429	200.038*
Algorithm 8	4.677	12.070	41.210	6.987	44.187
Algorithm 6	15.139	1442.960*	320.433*	1225.490*	831.573*

ones that are not congruent to the base. This was done because for Algorithm 8 such a step is not necessary and the approach developed in this work of constructing the point sets directly is a major improvement over the one based on tree-matching. The original algorithm could not be successfully executed more often than it managed to finish due to the vast required amount of memory. It was also significantly slower by a huge amount regardless if it succeeded or was terminated during execution. There is one instance where it worked reasonably fast (but still more than 3 times slower) but that seems like an outlier given the other data.

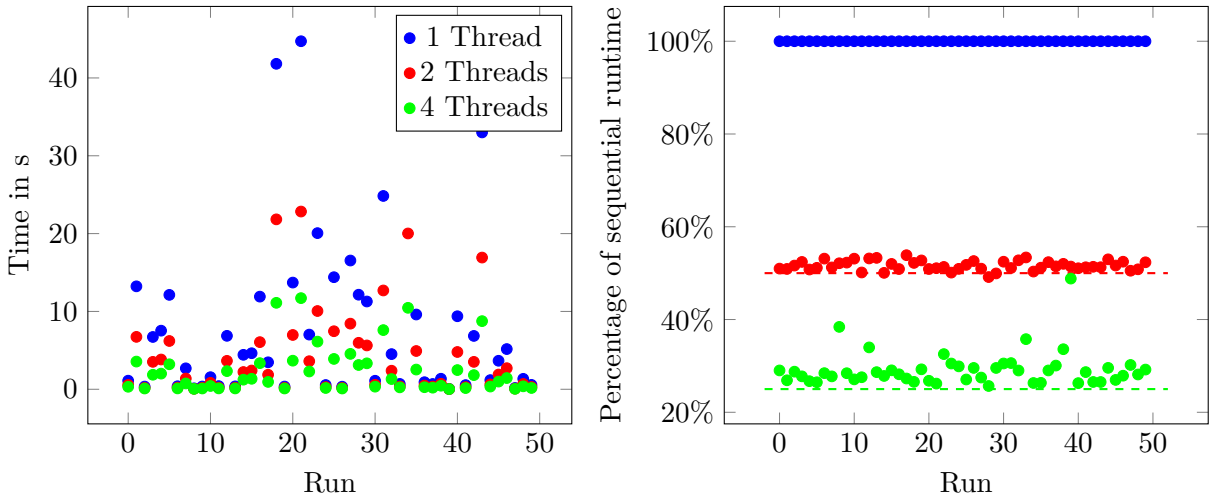
This second part of the algorithm is a heavy work loop that also processes data in each iteration independently from those before, which makes it a prime candidate for parallelizing in order to improve the execution time. Like the first loop and part of the algorithm the OpenMP framework was used for this and tested using the Stanford bunny dataset. As before multiple thread counts were tested on the laptop and on the more powerful desktop computer. The results in Figure 5.5 are similar to the ones in Figure 5.4. The potential runtime improvement gained from using two threads can be exploited by both systems. On the laptop the speedup seems more consistent as was the case in the previous section. Most of the measured runtimes are close to the dashed line indicating the optimal runtime. However, as before the hardware is not able to efficiently utilize additional threads. While the difference is also more consistent than in the previous case the speedup is quite small and in terms of the actual runtime inconsequential. As expected, the desktop computer is able to fully make use of two and four parallel threads. Although closer inspection and comparing Figure 5.5b to Figure 5.4b conveys the impression that there is more parallel overhead in this instance or the work was not shared equally between the threads. OpenMP will divide the iterations of the outer loop equally across the threads but depending on the point pairs that were found in the previous step of the algorithm and the stucture of the point cloud that is being searched the amount of work between these iterations can vary depending on how many points are found in each step of the loop.

5.2 Aligning the Stanford bunny

Applying the algorithm to the data set consisting of two models of the Stanford bunny shows how it operates. First the algorithm was tested on a the original model and a scaled down version of it that is 10 times smaller. Afterwards the smaller model was rotated by 45° around each axis. Then the amount of points of the second model was reduced to approximately three fourths of the original by randomly sampling of points and both tests repeated. In each tests 10



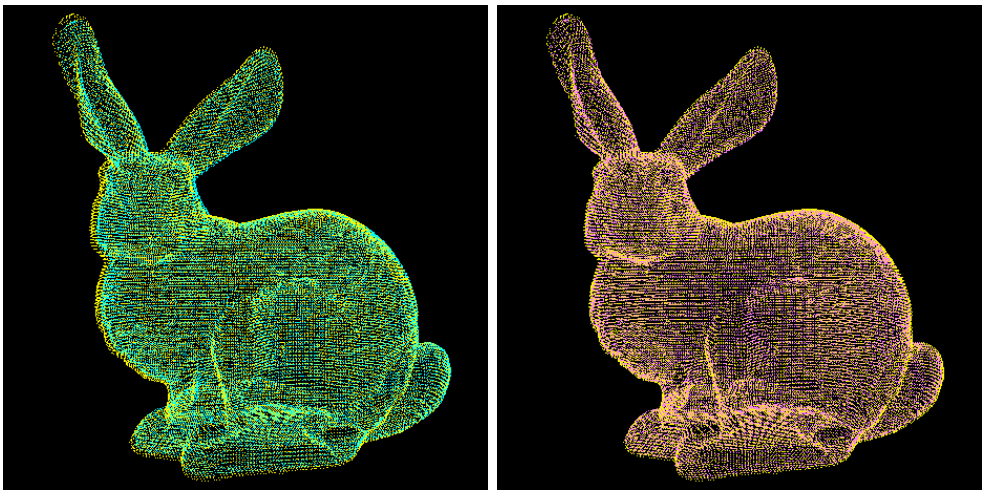
(a) Test performed on the laptop.



(b) Test performed on the desktop computer.

Figure 5.5: Comparing the runtime of assembling the congruent for point sets as per Algorithm 8 using the bunny dataset with 1, 2 or 4 parallel threads. Dashed lines indicate the optimal speedup gained from using multiple threads.

iteration of the algorithm were performed but if the termination condition of 90 % aligned points was reached the early, fewer iterations have been performed. Figure Table 5.2 shows the results of these tests in the order they were described. The first column displays the number iterations that were taken until the result. The following columns show the transformation matrix that was found to align the two point clouds the best after the iterations of the algorithm, the result of the refinement step after applying the ICP algorithm and the last column shows the expected

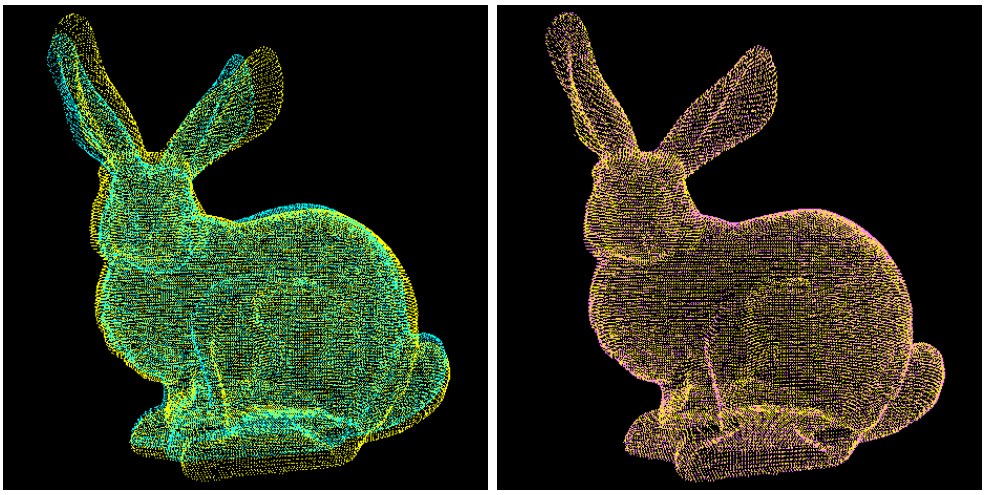


(a) Before applying ICP **(b)** After applying ICP

Figure 5.6: Result of aligning the bunny data set with an initial scale difference of 10.

outcome which is the true transformation between the two models. In table 5.3 the errors between the calculated transformations and the known true ones are listed. The scale error is given as a relative error and the translation as the added distance along all three dimensions. For the rotation the distance as described by Kuffner [23] between the rotation angles described by the rotational parts of the matrices is given. The Figures 5.6 to 5.9 display the results visually. The left image of each figure shows the first scan of the data set in yellow and the second one in cyan with the found transformation applied before employing the ICP algorithm. On the right of each figure the same comparison can be seen except that the transformation is applied to the second point clouds shown in magenta after undergoing the steps of ICP.

The transformation found for the pair with only a scale difference is already quite good without ICP. There is a slight rotation error but it is corrected after the execution of the ICP step. Visually both point clouds lie on top of each other (Figure 5.6) but inspecting the transformation matrix reveals that the scale factor is a bit low. Instead of 10 it is only a bit larger than 9.9 which amounts to a scaling error of under 1 %. Starting with a rotational difference between point clouds also produces a larger rotational offset of about 7° in the result that is clearly visible in Figure 5.7. As before applying ICP improves the alignment such that there is no noticeable difference in orientation. The scale factor is overestimated this time but is with 10.038 the closest one to the real value in this series of experiments. The tests using the downsampled data show the same pattern. If there is an initial orientation difference, the result will also show a larger discrepancy in orientation. However the additional ICP refinement is able to correct it. In general the iterative process produces transformations for these data sets that are less exact than for the two before. The scaling factor is also overestimated in both cases. Here the largest error occurs on the non-rotated dataset, clearly visible in Figure 5.8. The model is also shifted upwards but this translational offset is like the rotational one, that is about the same as in the previous test, corrected by ICP. Figure 5.9 shows the largest disparity when applying the transformation produced by iterative process in this series of tests. Both



(a) Before applying ICP **(b)** After applying ICP

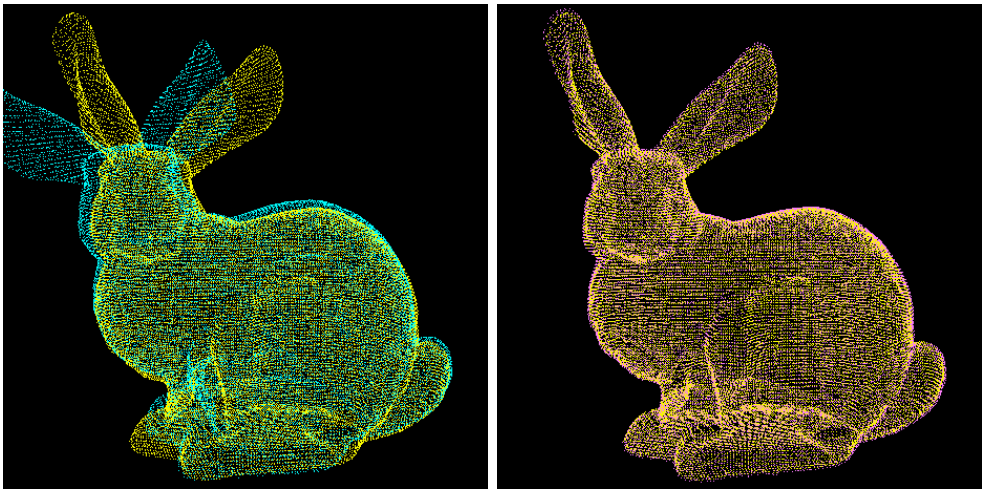
Figure 5.7: Result of aligning the bunny data set with an additional rotation between the two models.

the rotational and translational errors of almost 17° and 2 units are substantially larger as the next largest one in Table 5.3. Nevertheless, both point clouds are still correctly aligned after applying the ICP algorithm on them. The scale factor is estimated quite well, the error being around 1 % as in the first test.

In general in all of the above described initial situations the algorithm was able to produce a transformations that aligns both point clouds well. Minor and larger rotational and translational offsets could be fixed in its final ICP step. The scale was both over- and underestimated but was always quite close to the true value.

(a) Before applying ICP **(b)** After applying ICP

Figure 5.8: Result of aligning the bunny data set after resampling the second model at a lower density.



(a) Before applying ICP **(b)** After applying ICP

Figure 5.9: Result of aligning the bunny data set after resampling the second model at a lower density and rotating it.

Table 5.2: Results of applying the algorithm to the Stanford bunny dataset.

	Iterations	Transformation Matrix	ICP Applied	Expected Outcome
scaled	5	$\begin{pmatrix} 9.897 & 0.008 & 0.304 & 0.012 \\ -0.004 & 9.901 & -0.104 & 0.122 \\ -0.304 & 0.104 & 9.897 & -0.051 \\ 0 & 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 9.902 & 0.003 & 0.002 & -0.028 \\ -0.003 & 9.902 & 0.003 & 0.087 \\ -0.002 & -0.003 & 9.902 & 0.012 \\ 0 & 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 10 & 0 & 0 & 0 \\ 0 & 10 & 0 & 0 \\ 0 & 0 & 10 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$
rotated	10	$\begin{pmatrix} 6.248 & 7.628 & 1.880 & -0.211 \\ -5.532 & 2.567 & 7.973 & 0.420 \\ 5.578 & -5.999 & 5.802 & -0.717 \\ 0 & 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 5.019 & 8.568 & 1.470 & 0.010 \\ -5.019 & 1.470 & 8.568 & -0.036 \\ 7.098 & -5.019 & 5.019 & -0.003 \\ 0 & 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 5 & 8.536 & 1.464 & 0 \\ -5 & 1.464 & 8.536 & 0 \\ 7.071 & -5 & 5 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$
reduced	8	$\begin{pmatrix} 10.171 & 0.427 & 1.036 & -0.660 \\ -0.508 & 10.190 & 0.791 & 0.028 \\ -0.998 & -0.837 & 10.150 & 0.990 \\ 0 & 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 10.233 & 0.077 & -0.029 & -0.016 \\ -0.077 & 10.233 & 0.013 & -0.244 \\ 0.029 & -0.013 & 10.233 & -0.014 \\ 0 & 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 10 & 0 & 0 & 0 \\ 0 & 10 & 0 & 0 \\ 0 & 0 & 10 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$
reduced & rotated	10	$\begin{pmatrix} 6.634 & 7.365 & 1.971 & -0.033 \\ -2.890 & 0.011 & 9.684 & 0.117 \\ 7.056 & -6.920 & 2.113 & 1.861 \\ 0 & 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 5.052 & 8.628 & 1.476 & 0.030 \\ -5.055 & 1.484 & 8.625 & -0.093 \\ 7.146 & -5.050 & 5.057 & -0.014 \\ 0 & 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 5 & 8.536 & 1.464 & 0 \\ -5 & 1.464 & 8.536 & 0 \\ 7.071 & -5 & 5 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$

Table 5.3: Errors for the found transformation shown in Table 5.2 compared to the expected outcome.

		Transformation Matrix		ICP Applied	
	scaling error	rotational error	translation error	rotational error	translation error
scaled	−0.981 %	1.688°	0.133	0.023°	0.092
rotated	0.379 %	7.116°	0.857	0.018°	0.037
reduced	2.329 %	7.063°	1.190	0.425°	0.245
reduced & rotated	1.064 %	16.809°	1.865	0.036°	0.099

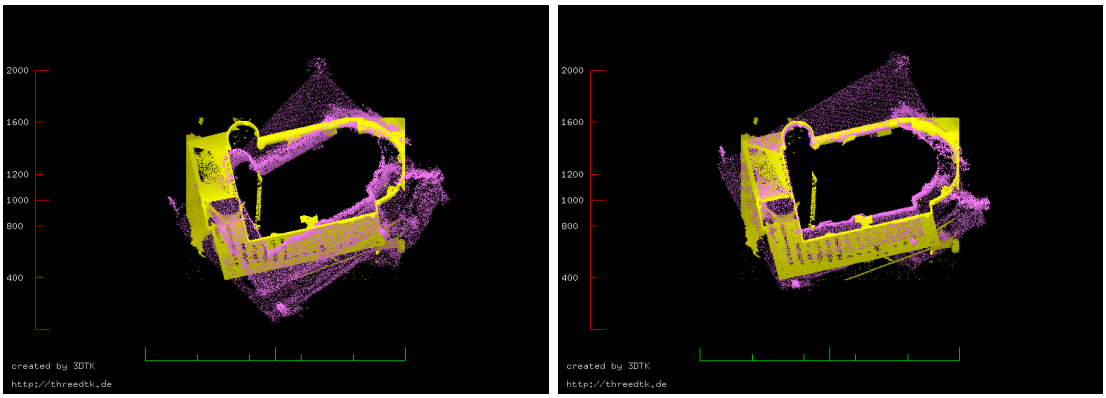
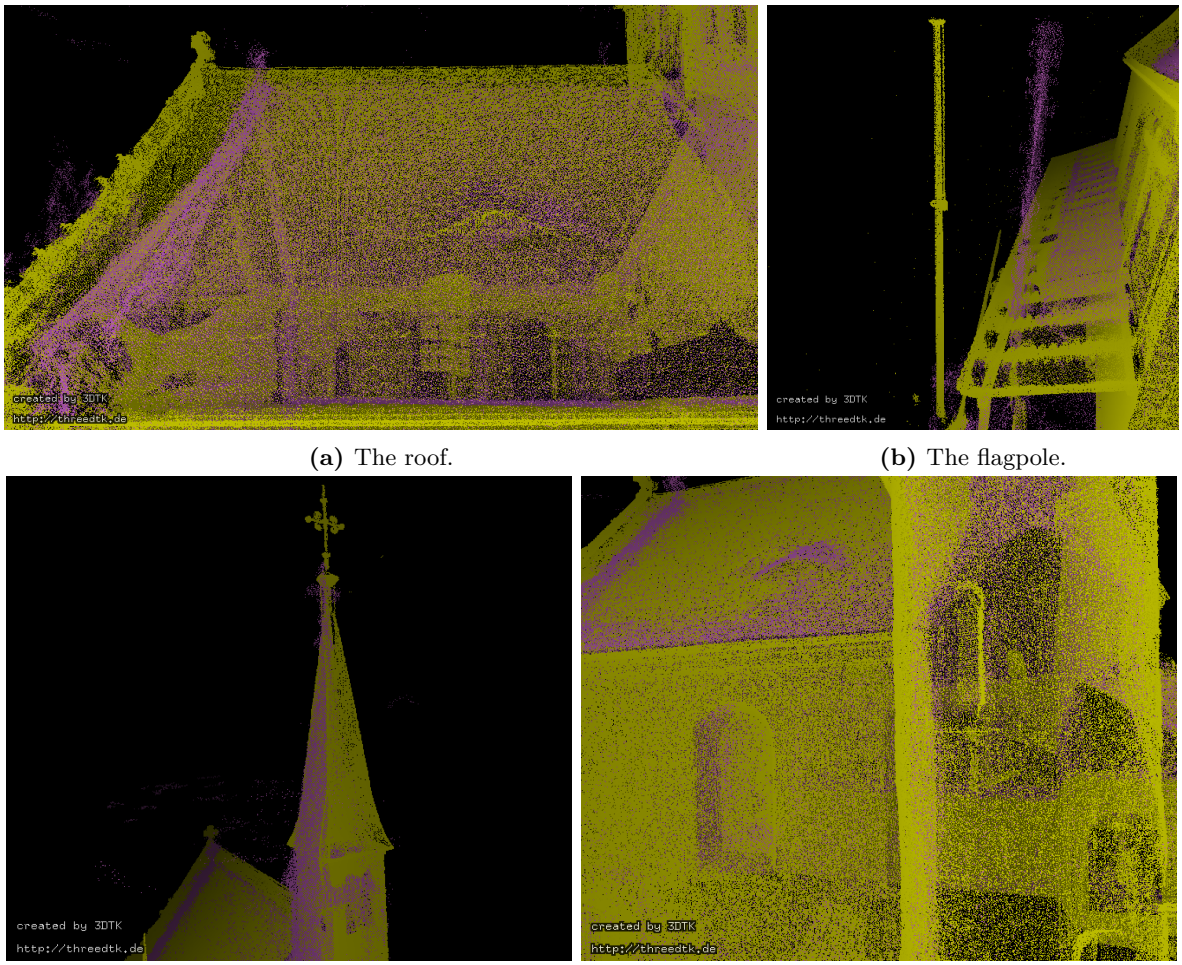


Figure 5.10: Top view of the data set. On the left, the laser scan of the chapel in yellow and the reduced SfM pointcloud with the found transformation applied. On the right the same situation but after ICP has ben applied.

5.3 Aligning the chapel data set

Finally, the algorithm was tested on the real world chapel data set. While the SfM point cloud being still being smaller than the one from the laser scanner by a huge amount it was still too large for being processed efficiently. For this reason, this point cloud was sampled in order to produce a transformation in a reasonable time. Visual inspection of the found transformation can be done with the whole point cloud again. This test was not a total success. The developed algorithm is probabilistic in its nature. The first few runs limited to 10 iterations on this dataset didn't produce a satisfactory transformation. The output of the sixth run is shown in Figure 5.10 on the left. As can be seen the rotational and transitional error is still large. After the pass through the ICP step however both point clouds are better aligned. However, it also shows that the recovered scale factor is too small. The round tower at the top right is aligned well but the chapel is clearly too small. The right and left walls don't reach the respective walls of the yellow laser scan. The extracted scale factor of 320.93 has an error of $-7.85\,\%$ relative to the value of 348.29 calculated from manually selected point pairs in Section 4.2. Figure 5.11 shows some areas of the registered data sets in more detail.

The picture on the top left shows a side top view of the chapel. The side surfaces of the roofs are aligned. This view also shows clearly that the scaling is too small. The half-circular back wall of the magenta SfM model doesn't reach the yellow one and the crest of the roof is too low. The same observations about the roof can be made in the image on the lower right. It shows additionally a part of the side wall and secondary tower. The wall is well aligned as does the detail of the window in it. The same holds for the tower. The parts of the tower that are of lower density in the laser scan are filled appropriately by the other point cloud. The main tower is in a similiar position in both point clouds as displayed in the lower left image. Here the low scale factor is again apparent. The cross on top of the tower ends midway of the cross of the laser scan. In the background the described features of the roof can be spotted again from another viewpoint. This picture also shows that the found rotation while quite good is not perfect. Due to the higher elevation and distance from the ground, smaller rotational errors are



(a) The roof. **(b)** The flagpole.

(c) The main tower. **(d)** Windows in the side wall.

Figure 5.11: Detail views of the alignment between the two data sets.

more visible. The magenta tower is skewed to left compared to the yellow one. This is reinforced by another vertical feature of the scene - the flagpole on the far side of the chapel displayed on the top right. It shows a rotation in the same direcretion exacerbated by the viewing angle. Note that the viewing direction is roughly from the opposite site compared to the picture of the main tower. Again the falling short of the scale factor can be seen at the roof and smaller roof on the side of the wall. The algorithm might produce a better result with an increased number of iterations. However, it is recommended to run tests on a more powerful machine due to the required runtime.

Chapter 6

Conclusions

In this thesis an algorithm for aligning point clouds of different scales has been developed on the basis of the 4PCS algorithm. It uses affine invariant properties of point sets to match them across point clouds and calculate a transformation. It was shown that the implementation of the developed algorithm performed faster than an equivalent implementation of the original one. While tests on simple data sets showed encouraging results - the point clouds being fully aligned after execution or at least close together so that the ICP refinement step could fully align them - the algorithm was not able to fully register a real world data set consisting of two point clouds that have been gathered by the means of laser scanning and SfM. Here additional work is needed so that the algorithm performs better in these kind of scenarios.

Another area were the developed algorithm can still be improved is performance. Despite taking huge care to not introducing performance hindrances in the implementation, it took still a long time for registering the chapel data set although the point clouds had been sampled down. Currently all normals are loaded at the start of the program. However, only a tiny percentage of the normals of the points of P will be used. An idea to further reduce memory consumption could be to only load or calculate the surface normals on demand when they are to be used. This should be at least an improvement for the point cloud P, all points of Q are at least queried once each iteration and the probability that their normals are required in a later step is high. Because the absolute orientation of normals is of no relevance to this work, only relative orientations between normals are, there is no need to transform them when testing the found transformations. However, currently the 3DTK framework will also transform the normals when transforming the point cloud. This introduces an overhead of factor two for each transformation.

Also related to normals is the way that they are used when comparing them between points to restrict the set of candidates in Q to those that have similar normals as the reference points in P. Here instead of comparing angles a signature of normal orientations in a four point set similar to the SHOT descriptor could be used, with further potential performance gains because the comparison if a set of points has similar normals is more accurate. Additional criteria that can be used for identifying points like color or reflectance values will also further restrict the set of candidates that needs to be checked and help performance.

Bibliography

[1] D. Aiger, N. J. Mitra, and D. Cohen-Or. 4-points congruent sets for robust surface registration. *ACM Transactions on Graphics*, 27(3):#85, 1–10, 2008.

[2] K. S. Arun, T. S. Huang, and S. D. Blostein. Least-squares fitting of two 3-d point sets. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-9:698–700, 1987.

[3] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, September 1975.

[4] P.J. Besl and Neil D. McKay. A method for registration of 3-d shapes. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 14(2):239–256, February 1992.

[5] Dorit Borrmann. *Multi-modal 3D mapping - Combining 3D point clouds with thermal and color information*. PhD thesis, Universität Würzburg, 2018.

[6] H. Bülow and A. Birk. Spectral registration of volume data for 6-dof spatial transformations plus scale. In *2011 IEEE International Conference on Robotics and Automation*, pages 3078–3083, 2011.

[7] Dylan Campbell and Lars Petersson. GOGMA: Globally-optimal gaussian mixture alignment. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, jun 2016.

[8] Dylan John Campbell, Lars Petersson, Laurent Kneip, and Hongdong Li. Globally-optimal inlier set maximisation for camera pose and correspondence estimation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 1–1, 2018.

[9] Intel Corporation. Intel® Core™ i3-7100U Processor Product Specifications. `https://ark.intel.com/content/www/us/en/ark/products/95442/intel-core-i3-7100u-processor-3m-cache-2-40-ghz.html`.

[10] M. Corsini, M. Dellepiane, F. Ganovelli, R. Gherardi, A. Fusiello, and R. Scopigno. Fully automatic registration of image sets on approximate geometry. *International Journal of Computer Vision*, 102(1):91–111, Mar 2013.

[11] Shaoyi Du, Nanning Zheng, Shihui Ying, Qubo You, and Yang Wu. AN extension of the ICP algorithm considering scale factor. In *2007 IEEE International Conference on Image Processing*. IEEE, 2007.

[12] J. Elseberg, S. Magnenat, R. Siegwart, and A. Nüchter. Comparison on nearest-neigbour-search strategies and implementations for efficient shape registration. *Journal of Software Engineering for Robotics (JOSER)*, 3(1):2–12, 2012.

[13] Georgios D. Evangelidis, Dionyssos Kounades-Bastian, Radu Horaud, and Emmanouil Z. Psarakis. A generative model for the joint registration of multiple point sets. In *Computer Vision – ECCV 2014*, pages 109–122. Springer International Publishing, 2014.

[14] Martin A. Fischler and Robert C. Bolles. Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography. *Commun. ACM*, 24(6):381–395, June 1981.

[15] F. Grosan, A. Tandrau, and A. Nüchter. Localizing Google SketchUp Models in Outdoor 3D Scans. In *Proceedings of the XXIII International Symposium on Information, Communication and Automation Technologies (ICAT '11)*, Sarajevo, Bosnia, October 2011. IEEE Xplore.

[16] Joachim Hertzberg, Kai Lingemann, and Andreas Nüchter. *Mobile Roboter*. Springer Berlin Heidelberg, 2012.

[17] Berthold Horn, Hugh Hilden, and Shahriar Negahdaripour. Closed-form solution of absolute orientation using orthonormal matrices. *Journal of the Optical Society of America A*, 5:1127–1135, 07 1988.

[18] Berthold K. P. Horn. Closed-form solution of absolute orientation using unit quaternions. *J. Opt. Soc. Am. A*, 4(4):629–642, April 1987.

[19] H. Houshiar, J. Elseberg, D. Borrmann, and A. Nüchter. A Study of Projections for Key Point Based Registration of Panoramic Terrestrial 3D Laser Scans. *Journal of Geo-spatial Information Science*, 18(1):11–31, January 2015.

[20] X. Huang, J. Zhang, L. Fan, Q. Wu, and C. Yuan. A systematic approach for cross-source point cloud registration by preserving macro and micro structures. *IEEE Transactions on Image Processing*, 26(7):3261–3276, July 2017.

[21] Bing Jian and B C Vemuri. Robust point set registration using gaussian mixture models. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33(8):1633–1645, aug 2011.

[22] I. T. Jolliffe. *Principal Component Analysis*. Springer Series in Statistics. Springer-Verlag, New York, 2002.

[23] James J Kuffner. Effective sampling and distance metrics for 3d rigid body path planning. In *IEEE International Conference on Robotics and Automation, 2004. Proceedings. ICRA'04. 2004*, volume 4, pages 3993–3998. IEEE, 2004.

[24] Stanford University Computer Graphics Laboratory. Stanford Bunny. `http://graphics.stanford.edu/data/3Dscanrep/#bunny`. Accessed: 2020-03-19.

[25] Xinju Li and Igor Guskov. Multi-scale features for approximate alignment of point-based surfaces. In *Proceedings of the Third Eurographics Symposium on Geometry Processing*, SGP '05, Aire-la-Ville, Switzerland, Switzerland, 2005. Eurographics Association.

[26] D. G. Lowe. Object recognition from local scale-invariant features. In *Proceedings of the Seventh IEEE International Conference on Computer Vision*, volume 2, pages 1150–1157 vol.2, 1999.

[27] Nicolas Mellado, Dror Aiger, and Niloy J. Mitra. Super 4pcs fast global pointcloud registration via smart indexing. *Computer Graphics Forum*, 33(5):205–215, 2014.

[28] A Moussa and N Elsheimy. Automatic registration of approximately leveled point clouds of urban scenes. *ISPRS Annals of Photogrammetry, Remote Sensing and Spatial Information Sciences*, pages 145–150, 2015.

[29] D. Novak and K. Schindler. Approximate registration of point clouds with large scale differences. *ISPRS Annals of Photogrammetry, Remote Sensing and Spatial Information Sciences*, II-5/W2:211–216, oct 2013.

[30] A. Nüchter, J. Elseberg, P. Schneider, and D. Paulus. Study of Parameterizations for the Rigid Body Transformations of The Scan Registration Problem. *Journal Computer Vision and Image Understanding (CVIU)*, 114(8):963–980, August 2010.

[31] Andreas Nüchter and Helge Andreas Lauterbach. Maria-Schmerz-Kapelle Randersacker. `http://kos.informatik.uni-osnabrueck.de/3Dscans/`. Data Set No. 26.

[32] A. Nüchter et al. 3DTK - The 3D Toolkit. `http//www.threedtk.de/`. Accessed: 2020-03-19.

[33] OpenMP Architecture Review Board. OpenMP application program interface version 4.5. `https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf`, November 2015.

[34] M. Pauly, M. Gross, and L.P. Kobbelt. Efficient simplification of point-sampled surfaces. In *IEEE Visualization, 2002. VIS 2002.* IEEE, 2002.

[35] Furong Peng, Qiang Wu, Lixin Fan, Jian Zhang, Yu You, Jianfeng Lu, and Jing-Yu Yang. Street view cross-sourced point cloud matching and registration. In *2014 IEEE International Conference on Image Processing (ICIP).* IEEE, October 2014.

[36] S. Rusinkiewicz and M. Levoy. Efficient variants of the ICP algorithm. In *Proceedings Third International Conference on 3-D Digital Imaging and Modeling.* IEEE Comput. Soc, 2001.

[37] R. Schnabel, R. Wahl, and R. Klein. Efficient RANSAC for Point-Cloud Shape Detection. *Computer Graphics Forum*, 2007.

[38] Pascal W Theiler, Jan D Wegner, and Konrad Schindler. Markerless point cloud registration with keypoint-based 4-points congruent sets. *ISPRS Annals of Photogrammetry, Remote Sensing and Spatial Information Sciences*, 2(5/W2):283–288, 2013.

[39] Federico Tombari, Samuele Salti, and Luigi Di Stefano. Unique signatures of histograms for local surface description. In *Computer Vision – ECCV 2010*, pages 356–369. Springer Berlin Heidelberg, 2010.

[40] James M. Van Verth and Lars M. Bishop. *Essential Mathematics for Games and Interactive Applications*. A. K. Peters, Ltd., USA, 3rd edition, 2015.

[41] Reiji Yoshimura, Hiroaki Date, Satoshi Kanai, Ryohei Honma, Kazuo Oda, and Tatsuya Ikeda. Automatic registration of mls point clouds and sfm meshes of urban area. *Geo-spatial Information Science*, 19(3):171–181, 2016.

[42] Timo Zinßer, Jochen Schmidt, and Heinrich Niemann. Point set registration with integrated scale estimation. In *International conference on pattern recognition and image processing*, pages 116–119, 2005.

Proclamation

Hereby I confirm that I wrote this thesis independently and that I have not made use of any other resources or means than those indicated.

Würzburg, May 2020