ANDREW AND ERNA VITERBI FACULTY OF ELECTRICAL ENGINEERING

SIPL
Signal and Image Processing Lab.

Technion Israel Institute of Technology

דו"ח סיכום פרויקט: א'

סורק תלת מימד בעלות נמוכה להתאמת יד תותבת

Low-Cost 3D Scanner for a Prosthetic Hand Digital Fitting

מבצעים:

גיא יפה **Guy Yoffe**

אביב גולן **Aviv Golan**

מנחה: **שונית פולינסקי** **Shunit Polinsky**

סמסטר רישום: אביב תש"פ

תאריך הגשה: אוקטובר, 2020

בשיתוף עם: Haifa3D

P 1234-2-19

תודות

אנו מודים מקרב לב לעמותת Haifa3D שהציעה את הרעיון לפרויקט ולאשת הקשר בחברה ומנחת הפרויקט שונית פולינסקי שתמכה בנו במסירות בכל מהלכו. אנו מודים גם לגל מצר שעזר והציע רעיונות אשר עזרו לנו מאוד בפרויקט.

אנו מודים גם ליאיר משה שכתב את תבנית זו לדו"ח מסכם ובכך עזר לנו לכתוב דו"ח מסכם בהצלחה ☺

Table of Contents

1 Background 1

1.1 The System 1

1.2 Point Cloud 3

1.2.1 Point Cloud Normals 3

1.3 Intel RealSense SR300 depth camera 4

1.4 Previos Project 5

2 The Suggested Solution 6

2.1 Overview 6

2.2 Capturing 7

2.3 Calibration 8

2.4 Alignment 11

2.5 Segmentation 13

2.6 Registration 15

2.7 Denoising 18

2.8 Reconstruction 19

2.9 Algorithm GUI 21

2.10 Plane Cut GUI 22

3 The algorithms in used 23

3.1 Clustering 23

3.2 RANSAC & MSAC 23

3.3 Neighbors Filter 24

3.4 Guided Filter 25

3.5 Boundary Filter 26

3.6 Downsampling 28
3.7 Iterative Closest Point(ICP) 28
3.8 Poisson Surface Reconstruction 30
4 Summary 31
References 32

List of Figures

Figure 1: Scheme of the scanner setup 2
Figure 2: Picture of the real scanner setup 2
Figure 3: point cloud of a pot 3
Figure 4: Intel RealSense SR300 camera 4
Figure 5: The previous project setup 5
Figure 6: Block diagram of the project 6
Figure 7: Point cloud of a sphere 7
Figure 8: Point cloud of a hand 7
Figure 9: The centroids before(upper) and after(lower) the transformation 10
Figure 10: Point cloud of the hand before the segmentation 13
Figure 11: The point cloud of the hand after the segmantation 14
Figure 12: point cloud of a hand as seen in one camera from 4 differernt angles and the hand as seen after registration of the 4 angles point cloud 17
Figure 13: Example of 3D meshes 19
Figure 14: Examples of poisson surface reconstruction on our data 20
Figure 15: Algorithm GUI screen 21
Figure 16: The plane cut GUI screen. 22
Figure 17: RANSAC of linear fit on noisy data 24
Figure 18: Neighbors Filter on the data - the filtered out points are in red 25
Figure 19: Boundary filter with $\epsilon B \approx 0.3 \cdot rG$ 27
Figure 20: Boundary filter with $\epsilon B \approx 0.5 \cdot rG$ 27
Figure 21: Example for poisson reconstruction action 30

Abstract

Creating a prosthetic hand is a long, expensive, and complicated process – starting from the prosthetic components, which can cost thousands of dollars, and ending with the personal adjustment process which can take a lot of time. When it comes to kids, there are numerous problems added to the problems above – Not getting a funding from the government, and the fact that young kids keep on growing, so spending so much time and money on a prosthetic hand that will become unfit for the child in the near future doesn't worth the money.

In this project we created personal customized low-cost 3D-scanner for amputated hands in order to increase the accessibility of prosthetic hands using low price depth cameras on a system that reconstruct a three-dimensional model of the amputated hand.

In order to do so we planned and set up the system given the fact that the object we need to model is a living being – three cameras with minimal overlapping on a rotating ring. As a result of this setup we needed to implement a calibration algorithm, made a few capturing rounds and registered them after a segmentation and noise reduction processes took place while creating a GUI for future use of our system.

During our work we understand existing algorithms for these problems, examined the differences between them and how to use those we found fit for our project, along with developing our own solutions and algorithms to problems that appeared as this project progressed.

1 Background

1.1 The System

In this project we created a system that reconstruct the residual limb CAD model from a relatively low-cost scanner consist of 3 low-cost depth cameras (which cost 100-150 each) which can be purchased worldwide.

Our main motive for using this scanner idea is its relatively low-cost price compared to available 3D scanners today. This way, together with the open-source code, our project could potentially help increasing the accessibility of prosthetic hands.

The scanner, as can be seen in figure 1, has 3 Intel RealSense SR300 depth cameras set on a ring. They are placed on a wide enough ring with 120° from each other to prevent overlapping when simultaneously capturing the depth pictures from all the cameras. Overlapping between them causes interfering between the projected patterns of the cameras, which damages the pictures quality and creates noise in the captured point clouds.

The ring is rotatable to allow capturing the hand from various angles. This way the large distance between the cameras does not affect the coverage of the hand by the depth pictures.

We have seen in point clouds captured from the depth cameras (described in section 1.2 and 1.3) that black objects (from a middle and above distance from the cameras) are almost invisible to the depth cameras. Thus, we have added a "shell" made from black nonwoven fabric around the setup. This way the point clouds captured from the cameras contains almost the wanted objects alone. This results both faster computation and easier task for the segmentation part, as described in section 2.5.

A picture of the setup can be seen in figure 2.

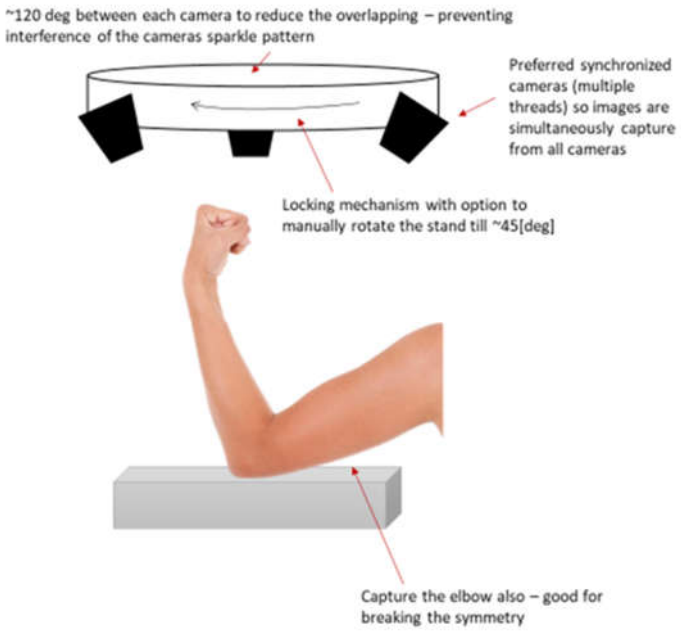

Figure 1: Scheme of the scanner setup

Figure 2: Picture of the real scanner setup

1.2 Point Cloud

The input from depth cameras, which we are using is called "Point Cloud" – a set of 3D data points in space and in our case – the position of every point captured by the camera in its coordinate system. An example of a point cloud can be seen in figure 3.

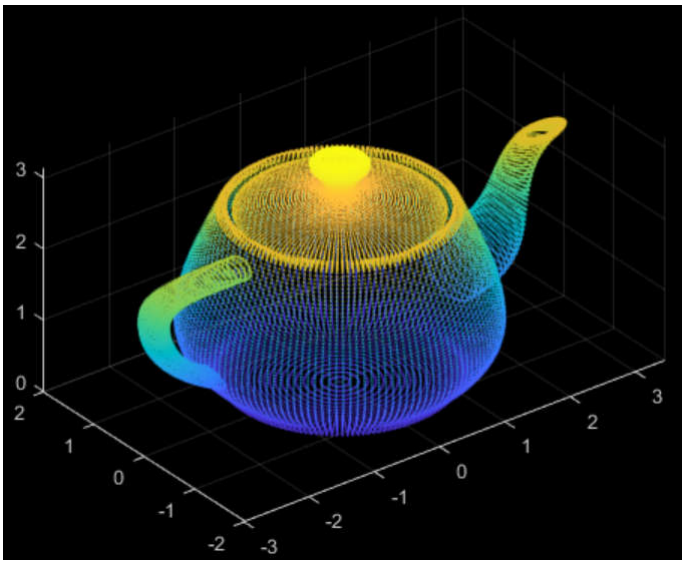

Figure 3: point cloud of a pot

1.2.1 Point Cloud Normals

Normals of a point cloud are vectors which are perpendicular to the surface of the point cloud.
Each point in the point cloud has a corresponding normal, which is the normal of the surface of the point cloud in the corresponding point.

1.3 Intel RealSense SR300 depth camera

This camera has three main components [1], as shown in figure 4 – IR camera lens, IR laser projector and color camera lens.

The acquiring process of the point cloud is done with a coded light method – a process of projecting a known pattern on to a scene, and by the way that these deform when striking surfaces allows the camera to calculate the location in the three-dimensional space of every point in the camera's coordinate system. The cameras output is a point cloud with 3D points and their corresponding point clouds (as described in section 2.2).

The access to the cameras has been done using a python module called pyrealsense2 [2].

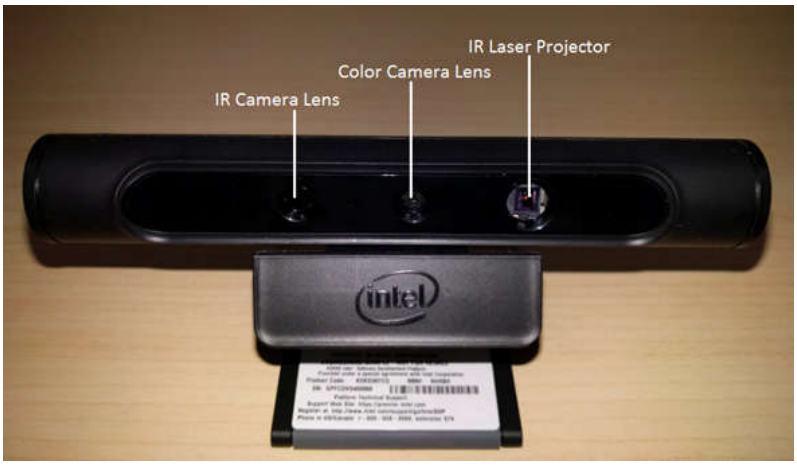

Figure 4: Intel RealSense SR300 camera

1.4 Previos Project

In 2018 the students Lital Filber and Ori Lador, under the supervision of Yair Moshe, executed a project named "Interactive Demo System for 3D Depth Data Processing" which dealt with similar problem of reconstructing three-dimensional point cloud of required shape (plasticine figure in their case).

Their solution consisted four fixed depth cameras with an overlap between neighbor cameras, as can be seen in figure 5. Each camera captured the object while the other cameras were off. The capturing process was done as such to avoid interfering.

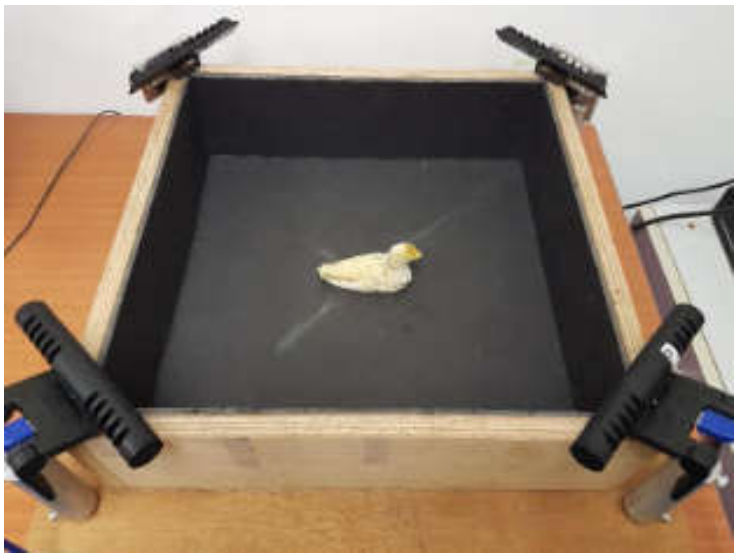

Figure 5: The previous project setup

The main differences between our project and theirs are that we need to capture a living object and that out target object is significantly bigger.

Due to the fact of possible (and almost inevitable) movements of the object, we need to capture the object from all the cameras simultaneously, unlike the previous project.

The simultaneity force us to minimize the overlap of the cameras' field of view, as described in section 1.1.

This limitation, as well as the object's size, force us to use a different setup –

Because of the low overlapping surface, we must have a rotatable setup.

The lack of overlap between the captured point clouds prevents us from using registration (described in section 2.6) alone to stitch the point clouds from different cameras.

2 The Suggested Solution

2.1 Overview

The pipeline of the project consists several components that are done chronologically:

1. Extrinsic calibration of the cameras.
1. For N times –
    2.1. Capture the hand simultaneously from all the cameras.
    2.2. Align the captured point clouds using the extrinsic transformation calculated in the calibration to the same coordinate system and merge them all to one point-cloud.
    2.3. Segmentation of the merged point cloud to remove background and far noises.
    2.4. Rotate the setup at a small angle.
2. Register all the merged point clouds.
3. Denoise the registered point cloud.
4. Reconstruct the point cloud to mesh.

The process can be described with block diagram, as can be seen in figure 6.

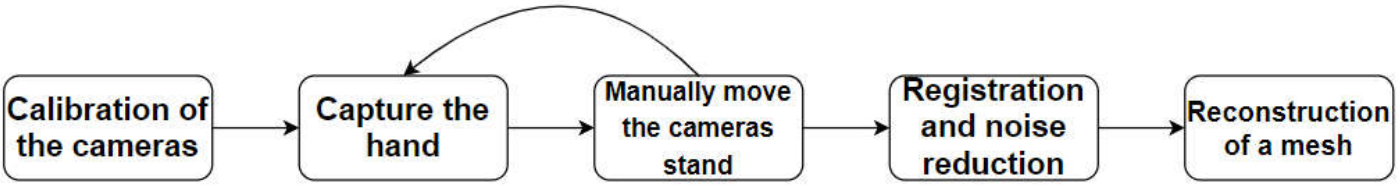

Figure 6: Block diagram of the project

2.2 Capturing

In this stage we capture both point clouds for the calibration of the cameras and point clouds of the object.

We will refer to the process of simultaneously capturing from all the cameras as round.

This stage was written in python using the module called "pyrealsense2", in which we have used for capturing, changing the preset of the cameras, etc.

Ad described in section 1.2, the point clouds contains the normal of the point cloud as well.

An example of a point cloud of a sphere captured from a camera can be seen in figure 7, and an example of a point cloud of a hand captured from a camera can be seen in figure 8.

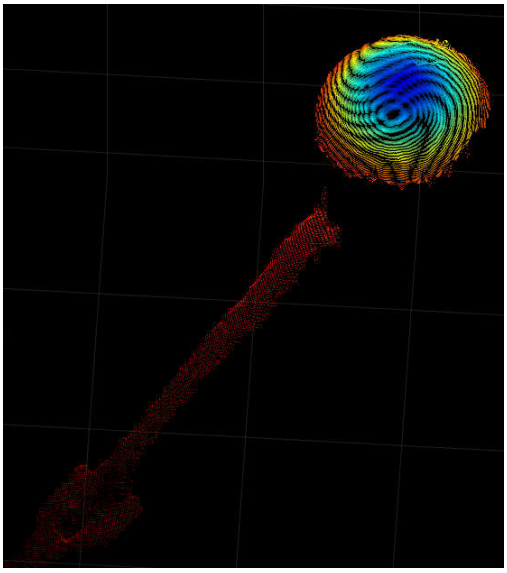

Figure 7: Point cloud of a sphere

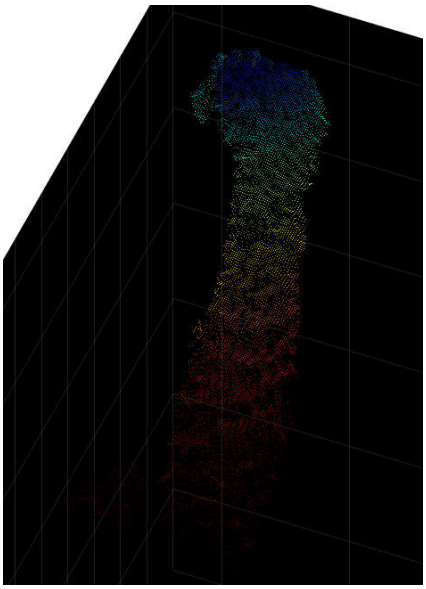

Figure 8: Point cloud of a hand

2.3 Calibration

In this stage we want to find the transformations (rotation and translation) between neighbor cameras.

At first, we have tried the classic RGB stereo calibration using the color camera in the devices. In this method, each neighbor cameras (stereo cameras) have an overlap in their field of view. They are simultaneously taking pictures of a chessboard (which his number of squares and their sizes are known) in various angles and positions, and by finding the corners on the chessboard in each picture, we can find the extrinsic transformation between the cameras.

We gave up on using this method for several reasons –

Only two cameras can be calibrated simultaneously, another transformation between the color camera and the depth camera needs to be taken in account (which increases the probability for errors), and most importantly – it gave bad results.

All the disadvantages of the previous method listed above forced us to use another method for the calibration. Fortunately, the calibration method proposed in [3] fixes all the issues above and gives us good results.

The method is described below –

We take a sphere with a known (and big enough) radius (called r_{real}) and capture it for many rounds (as described in section 2.2).

In each point cloud, we find a sphere using MSAC[4] , as described in section 3.2.

After we find a sphere in the point cloud, we check if it is a "valid" sphere –

A valid sphere is a sphere which has a radius that satisfy the condition –

$$r_{sphere} \in [r_{real} - r_{\epsilon}, r_{real} + r_{\epsilon}]$$

To reduce the effect of noise on the results, this process is done numerous times. After finding all the spheres and removing the non-valid ones, the centroid of the sphere in the point cloud is taken to be the average of the centroids of the valid spheres.

Then, we choose two neighbor cameras (1-2, 2-3, etc.). We remove rounds in which in one or more point clouds no valid sphere was found, and remain with "valid" rounds only.

We will denote each centroid as – $C_i^{(j)}$, Where i the number of the round and j is the number of the camera.

Our goal is to find the rotation matrix $R_j \in \mathbb{R}^{3\times3}$ and translation vector $t_j \in \mathbb{R}^3$ between camera j and camera $(j+1)$ which minimizes the following cost function –

$$J_j(R_j, t_j) = \sum_{i=1}^{n_r} \left\| C_i^{(j+1)} - P_j^{-1} C_i^{(j)} \right\|^2$$

Where n_r is the number of valid rounds, and P_j is the extrinsic matrix between camera j to camera $(j+1)$ –

$$P_j^{-1} = \begin{pmatrix} R_j & t_j \\ 0 & 1 \end{pmatrix}$$

This minimizing problem is the least-squared problem, and it is analytically solvable –

We first compute the covariance matrix –

$$A = \sum_{i=1}^{n_r} \left[\left(\bar{C}^{(j+1)} - C_i^{(j+1)} \right) \cdot \left(\bar{C}^{(j)} - C_i^{(j)} \right)^T \right]$$

$$\bar{C}^{(j)} = \frac{1}{n_r} \sum_{i=1}^{n_r} C_i^{(j)}$$

While using the SVD of A –

$$A = USV^T$$

We get –

$$\begin{cases} R_j = VU^T \\ t_j = \bar{C}^{(j+1)} - \bar{C}^{(j)} \end{cases}$$

The output of the calibration is –

$$\begin{cases} \{R_j\}_{j=1}^{n_c-1} \\ \{t_j\}_{j=1}^{n_c-1} \end{cases}$$

Where n_c is the number of cameras. In our case – $n_c = 3$.

An example of the detected spheres in two neighbor cameras before and after the transformation can be seen in figure 9.

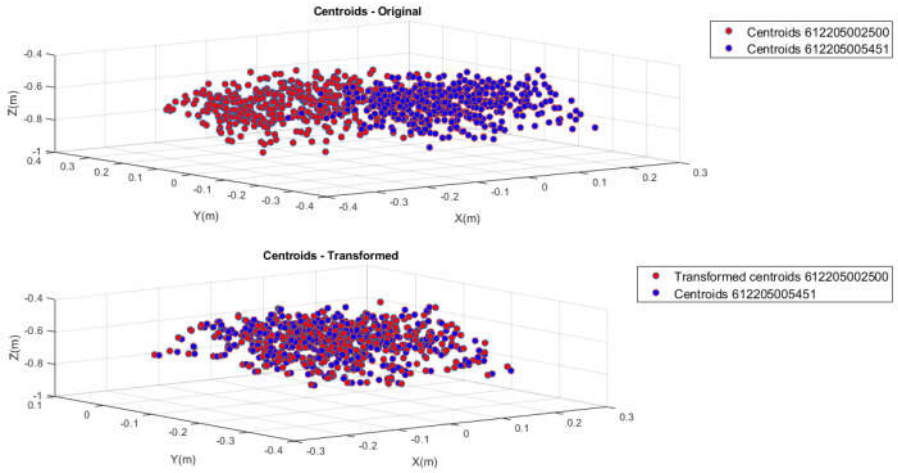

Figure 9: The centroids before(upper) and after(lower) the transformation

The results of the calibration for this experiment (and other experiments where the sphere is big enough and the cameras are not too close to each other nor too far) –

It has a low mean MSE both in the MSAC detection (about 2 mm) and after the transformations(about 0.5 mm), it has a low percentage of error in the detected sphere radius (about 1 percent), and about 99% of the captured spheres were valid.

The parameters of the calibration –

- Radius Error Percentage - an allowed error (in percentages) in the radius of the sphere that is found in the point cloud. Must be a value between 0 and 1. The smaller it is, the more likely that the found sphere is a valid one, but less spheres will be detected as valid spheres.
  As can be understood from the above - $r_\epsilon = r_{real} \cdot (radius\ error\ percentage)$.
  Default Value – 0.1.
- MaxNumTrials – the maximum number of iterations of MSAC for finding the sphere. Smaller values will result shorter runtime, but the calibration results would be less accurate.
  Default Value – 10^6.
- numOfTries - number of tries for finding the sphere (as described above). Smaller values will result shorter runtime, but the calibration results would be less accurate.
  Default Value – 20.
- MaxDistance - the maximum distance between a point to the sphere. Smaller values will result long runtime, but the calibration results would be more accurate.
  Default Value – 0.01.

2.4 Alignment

In the alignment stage we want to use the calibration results (the transformations) to transform all the point clouds to the same coordinate system, and to merge them into one point cloud.

For each round r we will denote –

$$C_r^{(j)} = \left\{C_{r,i}^{(j)}\right\}_{i=1}^{|C^{(j)}|}$$

$$N_r^{(j)} = \left\{N_{r,i}^{(j)}\right\}_{i=1}^{|C^{(j)}|}$$

Where $C_{r,i}^{(j)}$ is a point in the point cloud, i is the number of the point in the point cloud, j is the number of camera in which the point cloud was captured, r is the number of the round, $C_r^{(j)}$ is the points set of the point cloud from camera j in round r, $N_{r,i}^{(j)}$ is the corresponding normal to the point $C_{r,i}^{(j)}$, and $N_r^{(j)}$ is the set of the normals in the point cloud.

The alignment process for each round is as such –

Align_Point_Clouds($\left\{C_r^{(j)}, N_r^{(j)}\right\}_{j=1}^{n_r}$):

1. For r from 1 to n_r –
    a. For j from 1 to $(n_c - 1)$ –
        i. $\tilde{C}_r^{(j)}, \widetilde{N}_r^{(j)}$ = Transform_Point_Cloud($C_r^{(j)}, N_r^{(j)}, R_j, t_j$)
        ii. $C_r^{(j+1)}, N_r^{(j+1)}$ = Merge_Point_Clouds($\tilde{C}_r^{(j)}, \widetilde{N}_r^{(j)}, C_r^{(j+1)}, N_r^{(j+1)}$)
    b. $C_r = C_r^{(n)}$
    c. $N_r = N_r^{(n)}$
2. Return $\{C_r, N_r\}_{j=1}^{n_r}$

<u>Helper Functions</u> –

Transform_Point_Cloud($C^{(j)}, N^{(j)}, R_j, t_j$):

1. $\tilde{C}^{(j)} = \{\}$
2. $\widetilde{N}^{(j)} = \{\}$
3. For i from 1 to $\left|C^{(j)}\right|$ –
    a. $\tilde{C}_i^{(j)} = R_j \cdot C_i^{(j)} + t_j$
    b. $\widetilde{N}_i^{(j)} = R_j \cdot N_i^{(j)} + t_j$
    c. $\tilde{C}^{(j)} = \tilde{C}^{(j)} \cup \tilde{C}_i^{(j)}$

d. $\widetilde{N}^{(j)} = \widetilde{N}^{(j)} \cup \widetilde{N}_i^{(j)}$

4. Return $\tilde{C}^{(j)}, \widetilde{N}^{(j)}$

Merge_Point_Clouds($C^{(j)}, N^{(j)}, C^{(j+1)}, N^{(j+1)}$):

1. $\tilde{C}^{(j)} = C^{(j)} \cup C^{(j+1)}$
2. $\widetilde{N}^{(j)} = N^{(j)} \cup N^{(j+1)}$
3. Return $\tilde{C}^{(j)}$, $\widetilde{N}^{(j)}$

At the end of the process C_r contains all the points from all the point clouds in the coordinate system of camera number n_c, N_r contains all the normals from all the point clouds in the coordinate system of camera number n_c.

This process is done for every round. At the end of it we have n_r point clouds, all in the coordinate system of camera number n_c, containing all the point clouds from the other cameras.

Alignment parameters –

- GridStep – After the transformation, every point cloud in a 3D box with a volume of $GridStep^3$ is merged into a single point. Default value - $5 \times 10^{-4}\ m$.

2.5 Segmentation

In this stage we want to remove from the point clouds noises that are far from the hand. These noises are most likely to be caused by the background, light, etc. As described in section 1.1, we have removed most of these noises from the point clouds by using black nonwoven fabric, but we still have some noises as such, as can be seen in figure 10.

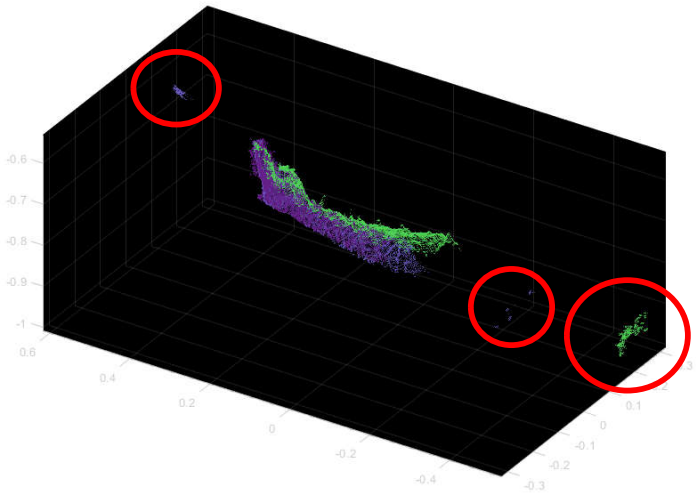

Figure 10: Point cloud of the hand before the segmentation

We first assume that the hand is the largest object (defined as dense set points) in the point cloud. The assumption is based both on the setup (as described in section 1.1) and both on experiments, as can be seen in figure 10.

Thus, we want to find the biggest object in the point cloud.

We do so by using clustering, as described in section 3.1.

We run the clustering algorithm on the point cloud and remove every cluster other than the biggest one. As can be seen in figure 11, we remain with a point cloud of the hand alone with noises close to the hand (mostly).

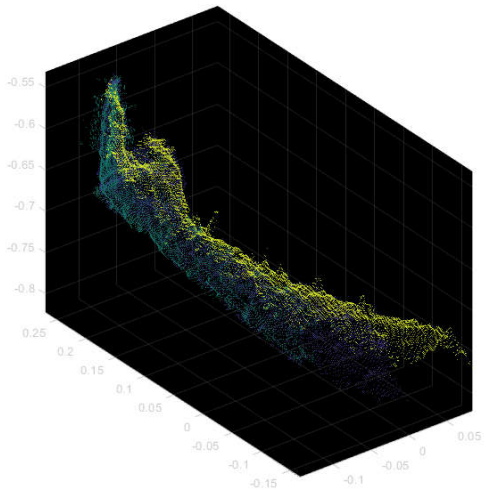

Figure 11: The point cloud of the hand after the segmantation

2.6 Registration

In this stage, we want to combine all the merged point clouds from the previous stages into a single point cloud that covers the entire hand, and get the most precise and detailed model of the residual limb that we want to reconstruct.

The registration process is as such –

Register_Point_Clouds($\{C_r, N_r\}_{r=1}^{n_r}$):

1. $C = C_1$
2. $\epsilon = \epsilon_0$
3. $Successful\_merges$ = 0
4. For r from 2 to n_r –
    a. $\tilde{R}_r, \tilde{t}_r, RMSE_r = ICP(C, C_r)$
    b. If ($RMSE_r \leq \epsilon$) –
        i. $\tilde{C}, \tilde{N}$ = Transform_Point_Cloud($C, N, \tilde{R}_r, \tilde{t}_r$)
        ii. C, N = Merge_Point_Clouds($\tilde{C}, \tilde{N}, C_r, N_r$)
        iii. $Successful\_merges = Successful\_merges + 1$
    c. Else –
        If ($Successful\_merges == 0$) –
            $C = C_r$
5. If ($Successful\_merges < T_M$) –
    a. If ($\epsilon + \Delta\epsilon < \epsilon_t$) –
        i. $\epsilon = \epsilon + \Delta\epsilon$
        ii. Return to line 1
    b. $C = C_{n_c}$
    c. $N = N_{n_c}$
6. Return C, N

At the end of the algorithm we get a single point cloud with points set C and normals N, which contains all the captured point clouds from all the rounds and cameras.

This process is done using the ICP (iterative closest point) algorithm[5], which is described in section 3.7.

As described in section 3.7, the registration's accuracy can be higher by using an initial transformation.

To find this initial transformation, we run the ICP algorithm with the downsampled point clouds, as described in section 3.6.

The downsampled point clouds are less dense and contains less points, and thus the ICP computation is faster.

We then use the same algorithm as mentioned above but add the initial transformation to the ICP algorithm. Thus, it gives better results, but at the expense of longer runtime.

As mentioned in section 3.7, outliers and noise reduce the accuracy of the ICP registration.

Thus, denoising the point clouds before the registration can increase the accuracy.

Therefore, we run the algorithm twice –

Once with the original point clouds after the alignment and segmentation (the output of this registration is called "*registered*"), and once with the denoised point clouds after the alignment and segmentation (the output of this registration is called "*registered_denoised*").

The denoising process is described in section 2.7.

We do the registration both with the original and the denoised point clouds because we've seen that for some cases the registration with the original point clouds was more accurate and for the other cases the denoised point clouds gave better results.

Thus, we ran the registration on both, at the expense of runtime, and we later give the user the option to use the best output by his opinion.

Registration parameters –

- GridStep – described in section 2.4.
- Min Error Threshold - ϵ_0.

  The minimum value of the threshold of the registration RMSE. Bigger values result in faster computation at the expense of accuracy. Default value - $0.005\ m^{0.5}$.
- Max Error Threshold - ϵ_t.

  The maximum value of the threshold of the registration RMSE. Smaller values result in faster computation. Values that are too big might consider a bad registration as a good

one, and values that are too small might ignore good registrations. Default value - $0.006\ m^{0.5}$.

- Jump Error Threshold – $\Delta\epsilon$.

  The jump value in the threshold of the registration RMSE. Bigger values result in faster computation at the expense of accuracy. Default value - $0.0002\ m^{0.5}$.

- Min Successful Registrations - T_M.

  the minimum number of good registrations. Smaller values result in faster computation at the expense of accuracy. Default value – 2.

- Inliers Ratio - described in section 3.7. Bigger values result in faster computation. Default value - 0.95.

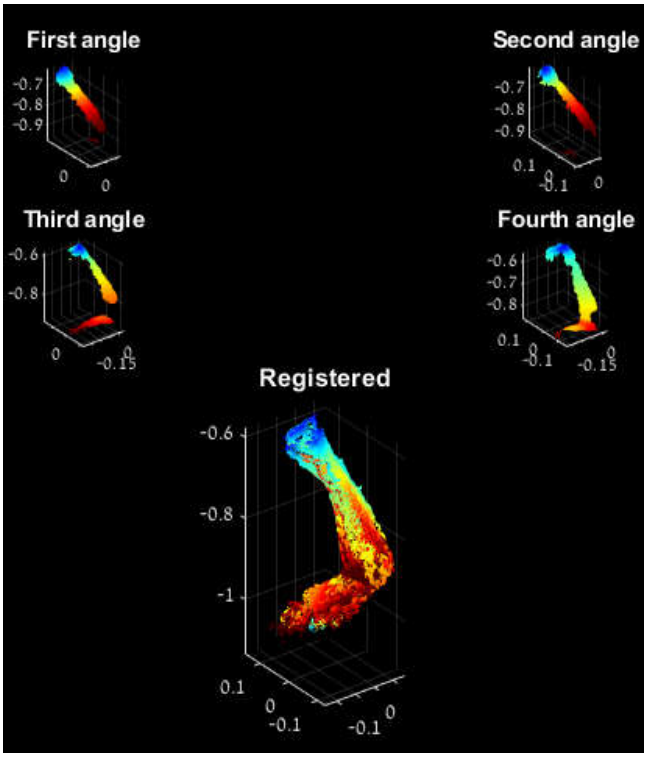

Figure 12: point cloud of a hand as seen in one camera from 4 differernt angles and the hand as seen after registration of the 4 angles point cloud

2.7 Denoising

In this stage we want to denoise the point cloud – remove noise points.

We have written and use several denoising algorithms, described in sections 3.3 to sections 3.5.

We've used the denoising process both before the registration and after it, but in each part we used a different set of algorithms.

At the end of this stage, we have four point-clouds –

"*registered*" and "*registered_denoised*", the output of the registration stage, "*final*" – the denoised point cloud of "*registered*", and "*final_denoised*" – the denoised point cloud of "*registered_denoised*".

As we said about the previous part, we save (and later reconstruct) all the 4 point-clouds because we could not find a way to be sure which one of these will result a better output.

In different experiments we have seen that it is hard to decide which one of them was the best one by vision, and thus we could not find a criterion for choosing one of them. Thus, we leave the operation of choosing the best output to the user, which can use his vision (and preferably his knowledge about prosthetic sockets) to evaluate the quality of each one of them.

The denoising process before the registration –

Denoise_Before_Registration($\{C_r, N_r\}_{r=1}^{n_r}$):

1. For r from 1 to n_r –
2. C_r, N_r = Neighbors_Filter(C_r, N_r, r_N, p_N)
3. C_r, N_r = Guided_Filter(C_r, N_r, r_G, λ_G)

Denoise_After_Registration(C, N):

1. C, N = Neighbors_Filter(C, N, r_N, p_N)
2. C, N = Guided_Filter(C, N, r_G, λ_G)
3. C, N = Neighbors_Filter(C, N, r_N, p_N)
4. C, N = Boundry_Filter(C, N, r_B, th_B)

As mentioned above, the algorithms – "Neighbor Filter", "Guided Filter" and "Boundary Filter" are described in sections 3.3 to sections 3.5, and so as their parameters.

2.8 Reconstruction

In this stage we want to reconstruct the point cloud and create a mesh – a 3D model consisting of polygons. Examples of 3D meshes can be seen in figure 13.

Previous work has been done by Alon Spinner and Yam Ben-Natan, which used the method of reconstruction using Bezier surfaces and assumptions about the model of the human hand.

Unfortunately, the results of the algorithm above on our data was not good enough in our opinion, and we have decided to use Poisson Surface Reconstruction, as described in section 3.8.

We have decided to use this algorithm instead because it smoothened the point cloud, ignored far-from-object noises, and kept the original shape of the hand.

Examples of the reconstruction on our data can be seen in figures 14.

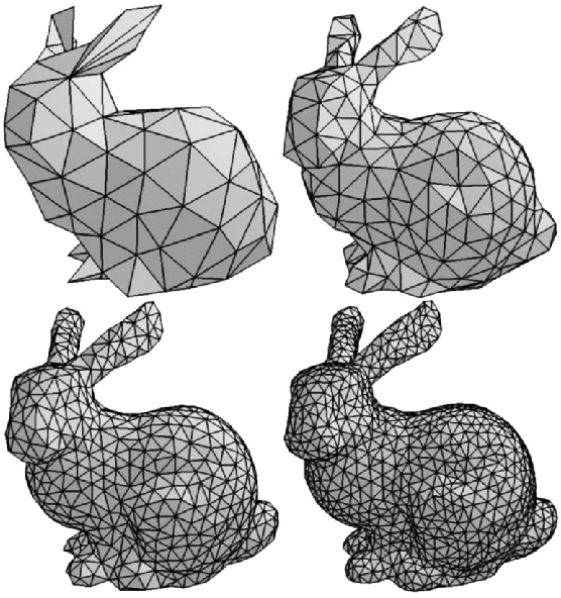

Figure 13: Example of 3D meshes

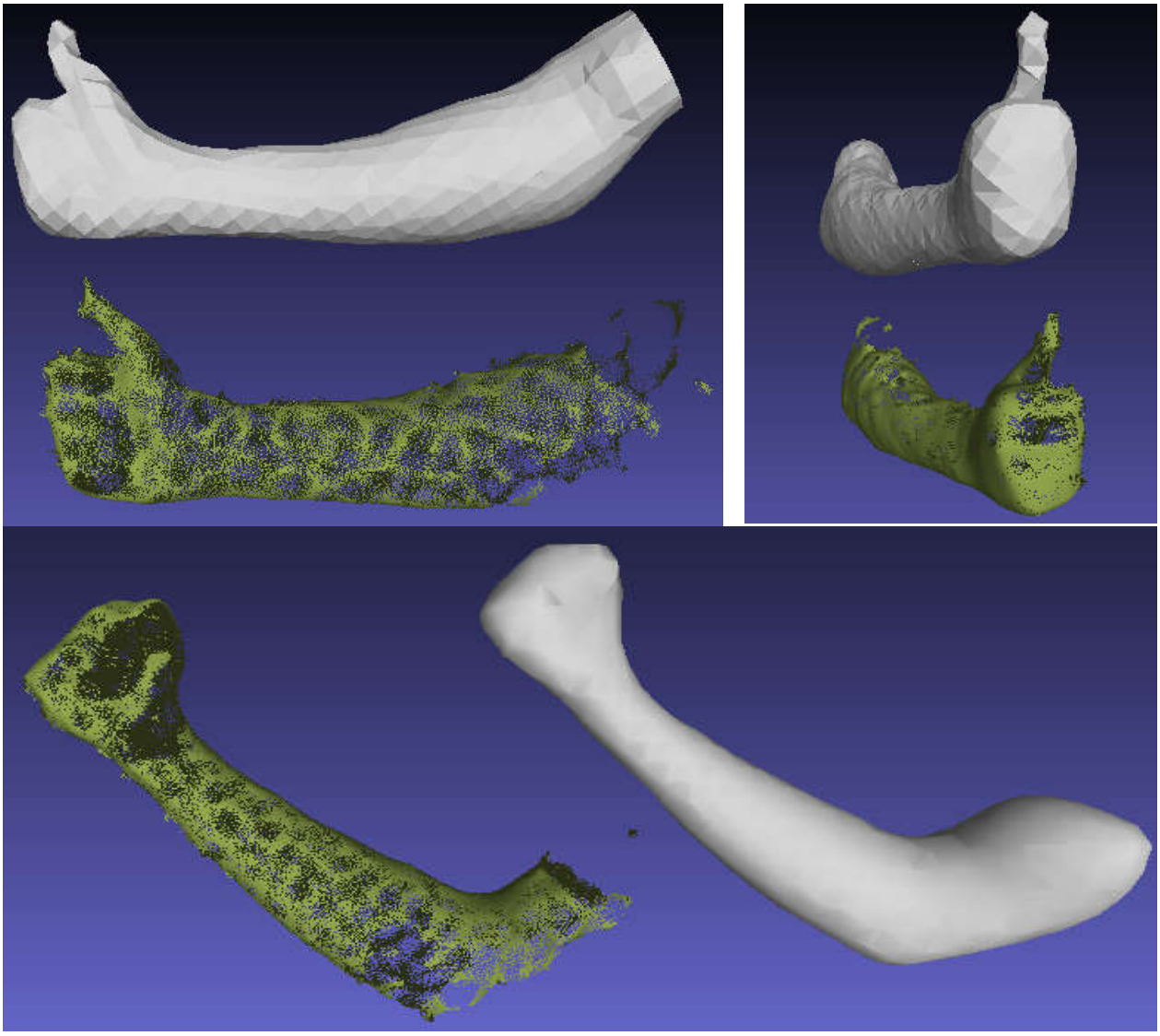

Figure 14: Examples of poisson surface reconstruction on our data

2.9 Algorithm GUI

The algorithm run is being made using a GUI. The GUI can be seen in figure 15.

In the GUI, the user has to select the directory which contains the data (calibration and hand captures), enter the circumference of the sphere used for the calibration, and choose the directories on which the user wants to run the algorithm on.

He can change the parameters of each stage and can re-run it and run over the results of previous runs.

In the text box there is the output of the algorithm, such as calibration results, registration results, etc.

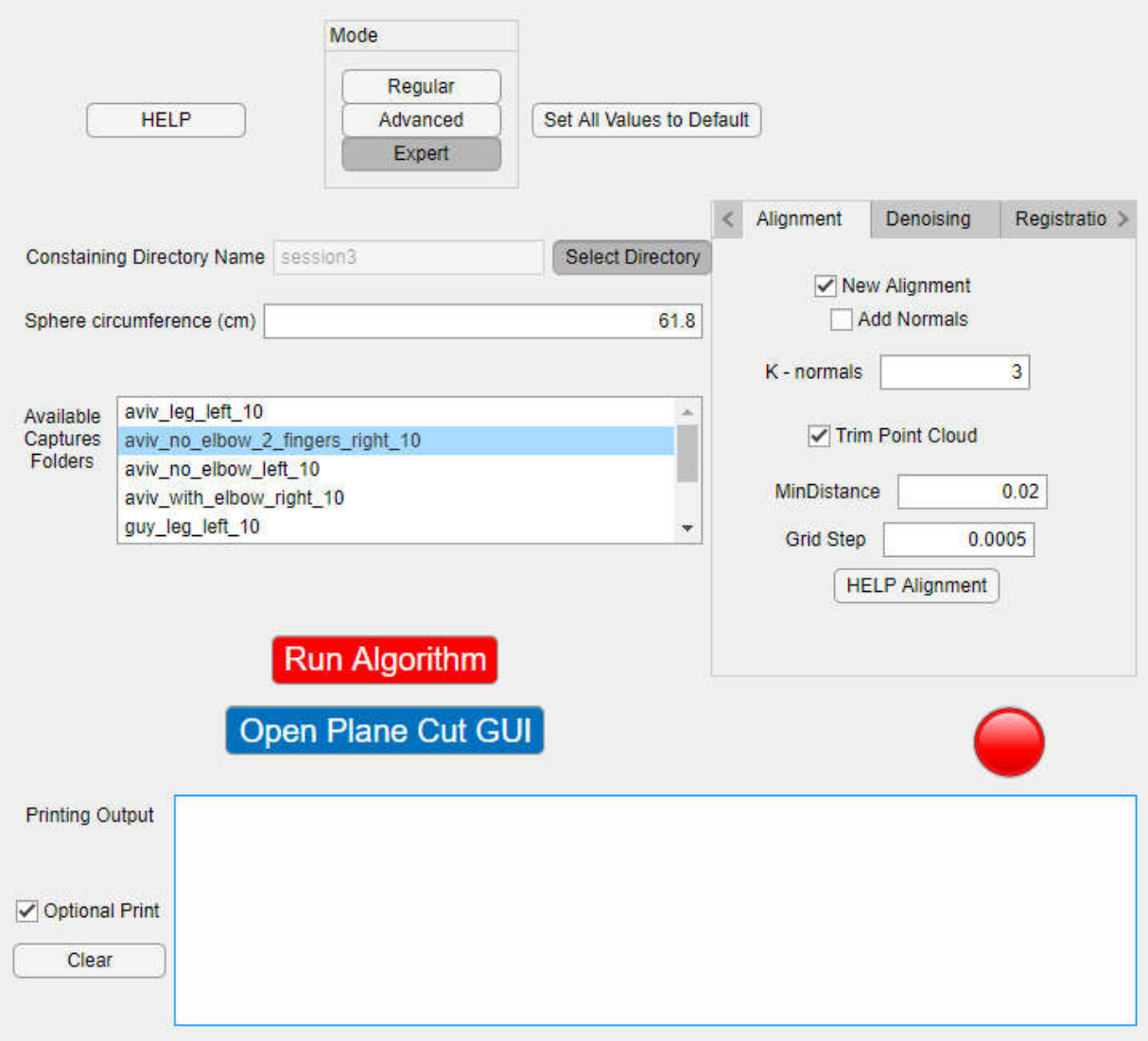

Figure 15: Algorithm GUI screen

2.10 Plane Cut GUI

In this GUI, the user can cut unnecessary parts from the point cloud, mainly the elbow and above.

The user selects a point cloud, a point and a vector (which are initialized to be the centroid and the eigenvector with the largest eigenstate respectively) which define a 3D plane.

Every point above the plane is colored with green and every point below the plane is colored with red.

After the user click the save button a new point cloud with the green points only and reconstruct a mesh from the new point cloud using the Poisson Surface Reconstruction.

This usually results a more details preserving mesh.

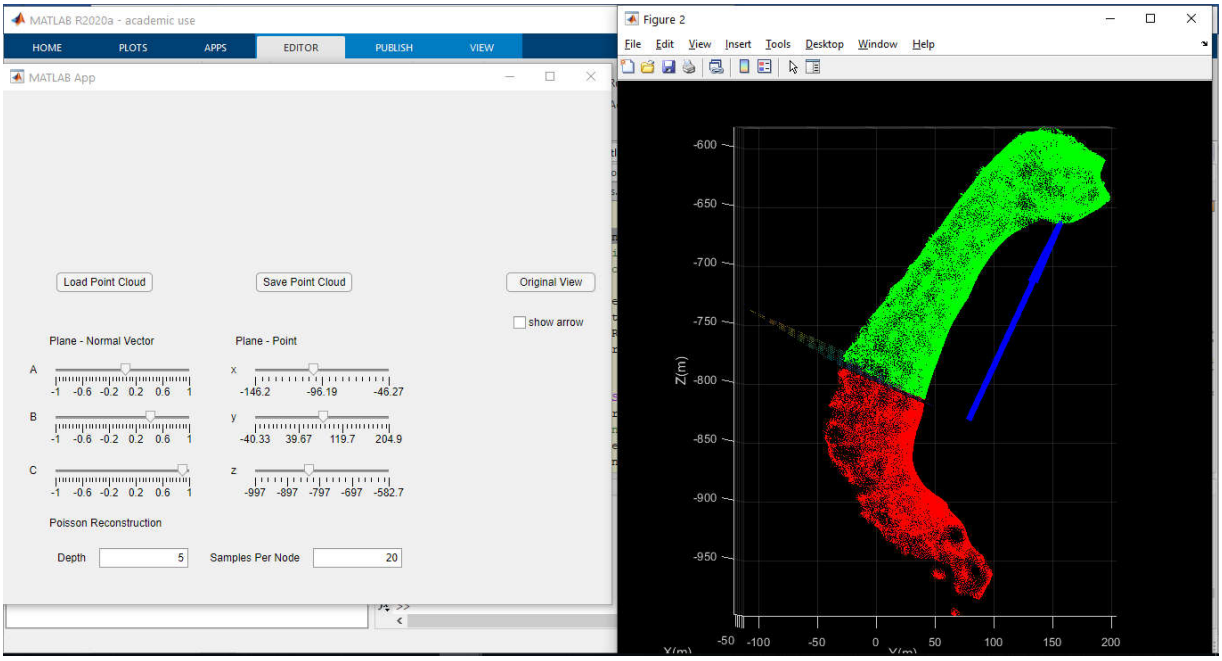

Figure 16: The plane cut GUI screen.

3 The algorithms in used

3.1 Clustering

Our clustering algorithm segments the point cloud into clusters and returns the label of every point in the point cloud and the size of every cluster. The clustering is based on minDistance parameter – if points are closer than the minDistance in the Euclidian distance they will classify as the same cluster. Different clusters will be in distance of at least minDistance from each other.

Clustering parameters –

- minDistance – minimum distance between points from different clusters. Values that are too small might delete wanted points from the point cloud, and values that are too big might keep unwanted points. Default value – $0.02\ m$.

3.2 RANSAC & MSAC

RANSAC (Random Sample Consensus) is an iterative algorithm for a robust matching of points (inliers) to a known geometric model with the presence of outliers.

The algorithm gets a set of points and a model for fitting and gives the models parameters as an output.

Other parameters of RANSAC are inliers ration and maximum distance between the model and inliers.

RANSAC ALGORITHM –

1. Random Selection of inliers.
2. Fitting the model's parameters for the inliers.
3. Checking how many points from the points are fitted to the model, and consider them as inliers.
4. If the inliers ratio is smaller than the threshold – go to stage 1.
5. Return the model's parameters.

There is an option to limit the number of runs of the algorithm, as we have done in our algorithm.

In figure (17) we can see the RANSAC when fit data to linear fit.

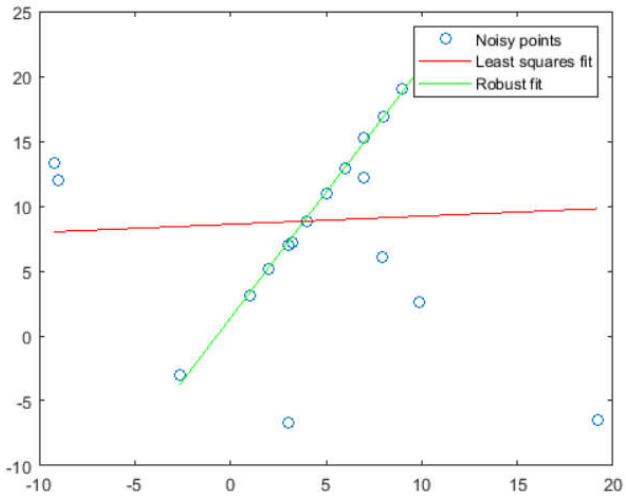

Figure 17: RANSAC of linear fit on noisy data

In our project we used a MATLAB function called *pcfitsphere*, which uses a MSAC[4] – a variant of RANSAC, to find a sphere in a point cloud.

3.3 Neighbors Filter

Neighbors Filter is a similar filter to the Radius Filter –

The Radius Filter gets a radius r_N and a threshold on the number of neighbors, and consider every point that has less neighbors (points that their distance from the center point is less than the radius) than the threshold as noise, and remove them.

The Radius Filter is implemented in many modules and libraries of point cloud processing such as PCL, Open3d, etc.

We wanted to make it more robust, so we have replaced the parameter of the threshold with noise percentage p_N –

For N points in the point cloud, c_i – point in the point cloud, and $N_i = \left\{c_j \mid \left\|c_i - c_j\right\|_2 \leq r_N\right\}$ the point's neighbors, and remove p_N percent of the points with the smallest $|N_i|$.

By replacing the threshold with the noise percentage, we make the filter more robust.

An example of the Neighbors Filter can be seen in figure 18.

Neighbors Filter parameters –

- Radius – r_N. Points within this radius are considered as neighbors. Smaller values result in faster computation, and less points will be removed. Default value – $0.01\ m$.
- Noise Percentage - p_N. The percentage of points that are considered as noise. Smaller values result in less points that will be removed. Default value – 0.15%.

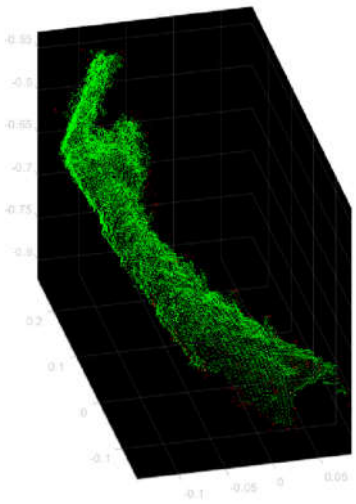

Figure 18: Neighbors Filter on the data - the filtered out points are in red

3.4 Guided Filter

The Guided Filter is a python implemented filter [6], which we implemented in MATLAB.

We assume a local linear model around each point p_i –

$$q_i = A_i p_i + b_i$$

Where q_i is the filtered point, $A_i \in \mathbb{R}^{3\times3}$, $b_i \in \mathbb{R}^3$.

Our goal is to find the A_i, b_i that minimizes the cost function –

$$J(A_i, b_i) = \sum_{j \in N(i)} \left(\left\| A_i p_i + b_i - p_j \right\|^2 + \epsilon_G \|A_i\|^2 \right)$$

Where –

$$N(i) = \left\{ j \mid \left\| p_i - p_j \right\|_2 \leq r_G \right\}$$

The analytic solution to this problem is –

$$\begin{cases} A_i = \Sigma_i^2 \left(\Sigma_i^2 + \epsilon_G I \right)^{-1} \\ b_i = \mu_i - A_i \mu_i \end{cases}$$

Where –

$$\mu_i = \frac{1}{|N(i)|} \sum_{j \in N(i)} p_j$$

$$\Sigma_i = \frac{1}{|N(i)|} \sum_{j \in N(i)} (p_j - \mu_i)(p_j - \mu_i)^T$$

$$I = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

The biggest advantage of the Guided Filter is the edge and sharp-shape preserving. Other filters such as bilateral filter can do the same, but this filter does not require normals.

The normal to the filtered point q_i is taken to be the eigenstate of Σ_i with the biggest eigenvalue, with the same direction as the normal of p_i in order to keep all the normals to point outside of the object.

Guided Filter parameters –

- Radius – r_G. Points within this radius are considered as neighbors. Smaller values result in faster computation, but a less smooth point cloud. Default value – $0.01\ m$.
- Regularization – ϵ_G. Smaller values result in a less smooth point cloud. Default value – 0.01.

3.5 Boundary Filter

In the article [6], there is a proposed method for detecting boundary points –

A point p_i is considered as boundary point if –

$$\|p_i - \mu_i\|_2 > \epsilon_B$$

Where –

$$N(i) = \left\{ j \mid \left\| p_i - p_j \right\|_2 \leq r_G \right\}$$

$$\mu_i = \frac{1}{|N(i)|} \sum_{j \in N(i)} p_j$$

Which means that the distance from the point p_i to the centroid of its neighbors is bigger than some threshold ϵ_B.

We have noticed, that for using $\epsilon_B \approx 0.3 \cdot r_B$, the algorithm does find the boundary points with good results, as can be seen in figure 19.

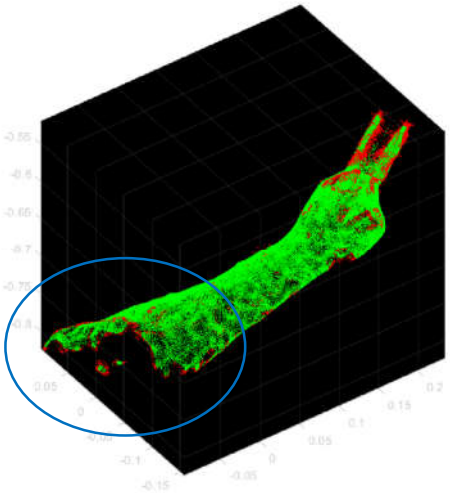

Figure 19: Boundary filter with $\epsilon_B \approx 0.3 \cdot r_B$

By trying to increase ϵ_B, we've noticed that for $\epsilon_B \approx 0.5 \cdot r_B$, the algorithm finds points that we consider as noise and consider them as boundary points. This does makes sense because noisy points are usually far from their nearest points and their neighborhood has low density.

The output of the algorithm can be found in figure 20.

As can be seen, the algorithm finds noisy points that the other algorithms couldn't detect, and it also remove numerous boundary points, but we are okay with it since nearly doesn't affect the reconstruction algorithm described in section 3.8.

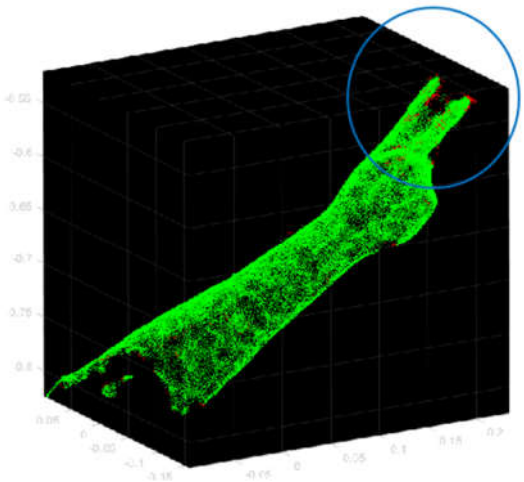

Figure 20: Boundary filter with $\epsilon_B \approx 0.5 \cdot r_B$

3.6 Downsampling

In the down sampling algorithm we used a filter, in the size of $GridStep^3$ which divide the axis-aligned bounding box of the point cloud into boxes in that size and merge all the points in every box to one single point. The normals of the points are averaged as well. This form of down sampling is good in preserving the shape of the point cloud, which very important to the reconstruction process.

Clustering parameters –

- Gridstep – every point cloud in a 3D box with a volume of $GridStep^3$ is merged into a single point. Default value - 5 mm.

3.7 Iterative Closest Point(ICP)

ICP[4] is an algorithm that aims to finding the rigid transformation between a one point cloud (which called "moving" or "source") to another (which called "fixed" or "destination"), by minimizing the RMSE between the two.

In this algorithm we assume that there is a relatively big overlap between the source and the destination. Furthermore, we assume rigid transformation (without scale changing for example).

There are few variants of this algorithm that gives better results in some cases but in our case, with a large overlap between the point clouds the standard ICP (point-to-point) is efficient and used in our project.

The steps of the standard ICP algorithm that gets the source point cloud A and the destination point cloud B:

1. Initialization of the transformation matrix T as the identity matrix or another matrix that can be defined by the user. A good initial transformation can result better registration.
2. For every point b_i in B calculate the closest point to her in A after the transformation $T \cdot a_i$.
3. Calculate T^* such that: $T^* = \arg\min_{\mathrm{T}}\{\sum_i \|b_i - Ta_i\|^2\} = \arg\min_{\mathrm{R,t}}\{\sum_i \|b_i - Ra_i - t\|^2\}$ where R is the rotation matrix and t is the translation vector.

4. Check if the average difference between estimated rigid transformations in the three most recent consecutive iterations falls below the specified tolerance value. If so – stop the algorithm and return T^*. If not – define T as T^*, add iteration and return to stage 2.

In our implementation we also have outliers filter that discard matches the percentage that defined as parameter in every iteration in stage 2.

ICP Registration parameters –

- Tolerance – Tolerance between consecutive ICP iterations, as explained above. The translation difference tolerance is set to $10^{-6}\ m$, and the angular difference tolerance is set to $(10^{-6})°$. Both values are unchangeable.
- MaxIterations – Maximum number of iterations of the ICP. It's value for the initial run is set to 10^4 and in the second run it is set to 100. Both values are unchangeable.
- Inliers Ratio – percentage of inliers in the registration. A pair of matched points is considered an inlier if its Euclidean distance falls within the percentage set of matching distances. Bigger values result in faster computation. Default value – 0.95.

3.8 Poisson Surface Reconstruction

Given a set of 3D points with oriented normals sampled on the boundary of a 3D solid, the Poisson Surface Reconstruction[8] method solves for an approximate indicator function of the inferred solid, whose gradient best matches the input normals, as can be seen in figure 21.
The method is implemented in many point cloud processing tools such as MeshLab and CGAL.
In our algorithm we used an implementation of the method as an executable file from [9].
The method has many parameters, but we chose 2 that has a great influence on the generated mesh.
Poisson Surface Reconstruction parameters –

- Poisson Depth – The higher the depth the more detailed the mesh. High depth reconstruction for noisy data keeps more vertices in the generated mesh that are outliers but the algorithm doesn't detect them as such. So, a low value provides a smoothing effect, but details will be lost. The higher the depth-value the higher is the resulting amount of vertices of the generated mesh.
- Samples Per Node – The samples per node parameter defines how many points the marching cubes algorithm puts into one node of the resulting octree. For noisy data a high sample per node value provides a smoothing with loss of detail while a low value keeps the detail level high. A high value reduces the resulting count of vertices while a low value remains them high.

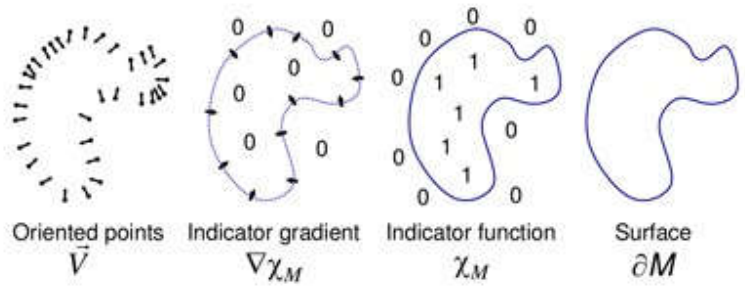

Figure 21: Example for poisson reconstruction action

4 Summary

In this paper we presented a personal customized low-cost 3D-scanner for amputated hands.
The system is built from 3 Intel RealSense SR300 depth cameras set on a rotatable ring. They are placed on a wide enough ring with 120° from each other to prevent overlapping when simultaneously capturing the depth pictures from all the cameras. There is also a "shell" of black nonwoven fabric around the setup on order to achieve better and faster results.
In the beginning of the project we examined which setup of the system will be used given our acquired data features(organ of a human – or even a child) and learned how to use the depth cameras, how to acquire data with them.
Afterwards, we implemented a calibration method including many capturing rounds of a sphere shape object and using our prior knowledge on this object and RANSAC based algorithm. As a result, we could conclude the extrinsic transformation between our cameras and put the point clouds that captured from different cameras and axis system to singular "world" system.
In every capture round of the hand we also see other butch of points and outliers that we want to filter out. Thus, we implemented segmentation algorithm which cluster the data and return the cluster of the hand. Furthermore, we implemented few filters that filters out outliers and increase the precision of our system.
We also used registration algorithm in order to combine few capturing rounds of the hand from different angles(which done with the rotatable ring) and get the fullest and the most precise point cloud of the hand.
To print a 3D prosthesis point cloud model doesn't enough, we need to convert the point cloud to a mesh it has been done using a existing code of Poisson reconstruction algorithm.
finally, we wanted that our project will be usable, thus we GUI system for wide use.
There is also a future work that can be done. In our algorithm we could not determine if we reach better result if our outlier filters activated also before the registration phase or only after this phase due to the covid-19 situation and our lack of data and measurements from different people that followed. This dilemma can be examined when the situation will allow wide data acquisition and measurement while comparing these two possibilities.

References

[1] "Intel RealSense™ Camera SR300 Product Datasheet," 2016. [Online]. Available: https://software.intel.com/sites/default/files/managed/0c/ec/realsense-sr300-product-datasheet-rev-1-0.pdf.

[2] "Welcome to pyrealsense2's documentation!" 2017. [online] https://intelrealsense.github.io/librealsense/python_docs/_generated/pyrealsense2.html

[3] Su P.C., Shen J., Xu W., Cheung S.S., Luo Y. A Fast and Robust Extrinsic Calibration for RGB-D Camera Networks. Sensors. 2018;18:235. doi: 10.3390/s18010235.

[4] Torr, P. H. S. and A. Zisserman. "MLESAC: A New Robust Estimator with Application to Estimating Image Geometry." *Computer Vision and Image Understanding*. 2000.

[5] B. Bellekens, et al.(2014) " A Survey of Rigid 3D Pointcloud Registration Algorithms". In AMBIENT 2014: The Fourth International Conference on Ambient Computing, Applications, Services and Technologies.

[6] "Github - guided-filter-point-cloud-denoise", 2019. [online]. Available: https://github.com/aipiano/guided-filter-point-cloud-denoise

[7] Chalmovianský P., Jüttler B. (2003) Filling Holes in Point Clouds. In: Wilson M.J., Martin R.R. (eds) Mathematics of Surfaces. Lecture Notes in Computer Science, vol 2768. Springer, Berlin, Heidelberg. https://doi-org.ezlibrary.technion.ac.il/10.1007/978-3-540-39422-8_14.

[8] Kazhdan, Bolitho and Hoppe. "Poisson Surface Reconstruction ". Eurographics Symposium on Geometry Processing (2006).

[9] Adaptive Multigrid Solvers (Version 13.61), [online]. Available: http://www.cs.jhu.edu/~misha/Code/PoissonRecon/Version13.61/#EXECUTABLE