University of Burgundy

Software Engineering

January 9, 2017

3D Human Body Scanner

Team Members:
AbdelRahman ABUBAKR
Gopikrishna ERABATI
Ziyang HONG
Chunxia LI
Yu LIU
Avinash NARAYANA
Anirudh PULIGANDLA
Darja STOEVA
Shubham WAGH
Hassan ZAAL

Supervisors:
Yohan Fougerolle
Cansen Jiang

Acknowledgement

We would like to express our special thanks of gratitude to our supervisors Dr. Yohan Fougerolle and Cansen Jiang who gave us the golden opportunity to do this wonderful project on the topic (3D scanner), which also helped us in doing a lot of research and we came to know about so many cutting edge technologies and novel algorithms. We are really thankful to them. Secondly we would also like to thank all of our team members and classmates who helped us a lot in finalizing this project within the limited time frame.

Contents

List of Figures

Chapter 1

Introduction

Development and integration of scanners for 3D reconstruction, prototype printing, augmented and virtual reality had made 3D scanning an increasingly common part of the world we live in. With more and more affordable depth sensors around such as Microsoft Kinect, 3D scanning is no longer restricted to industrial or research.

Among 3D scanning applications, human body scanning has especially gained immense popularity due to its versatile use in the movie, video game, clothing and medical industry, as well as anthropometry and ergonomic applications. As new technologies and tools were continuously developed with massive cost reduction, more realistic figures can be obtained under brief period of scanning. As the number of non-technical user increase to employ these high-end technologies, a solution that allows user to perform minimum amount of operations to acquire final scan result is not only ideal but also demanded. This can be achieved using a 3D scanning rig, which combines point-and-shoot functionality with portability.

The main objective of this project is to develop a human 3D scanner software able to fully interface with a scanner rig composed of a turning table and a stationary depth sensor. The software is aimed to perform full body scan under 90 seconds. A friendly, interactive graphical user interface provides simple control and outputs watertight mesh results that can be used mainly but not limited to 3D printing.

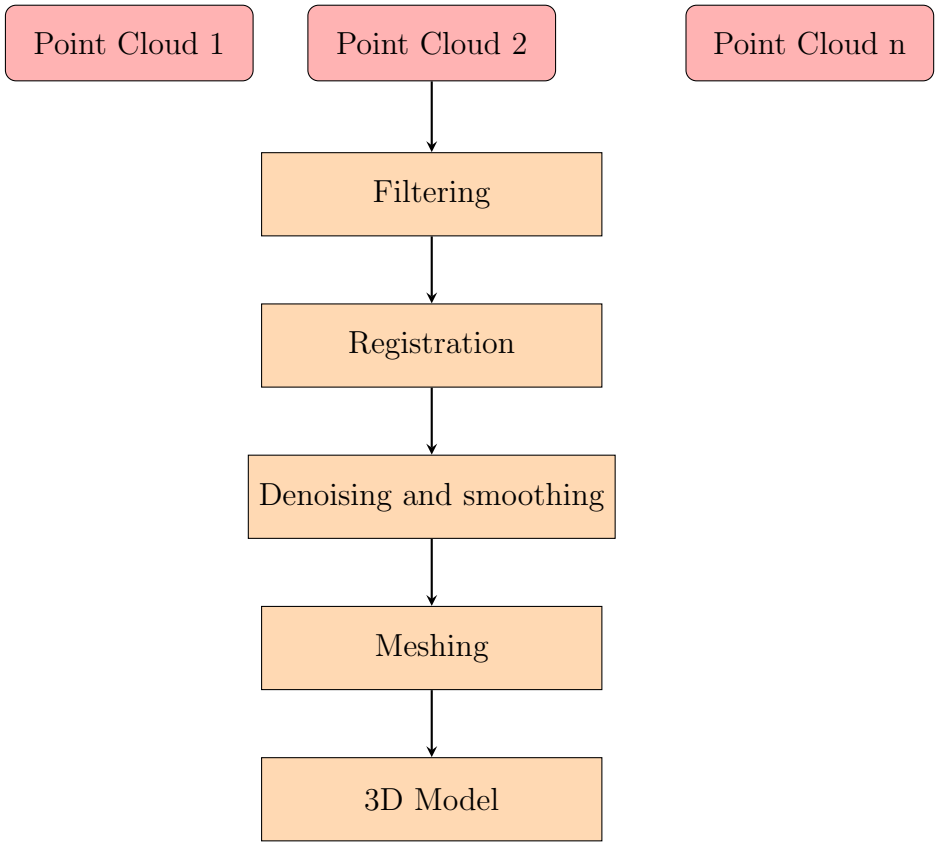
All source codes are avalaible on Github repository.

Click here - `https://github.com/AnirudhPuligandla/3D_scanner`

1.1 Proposed Software Architecture

Firstly, we will use the depth sensor to scan the object which can deliver real-time good accurate and dense 3D scenes at economical cost. However, if the depth map is noisy and contains holes due to surface property, we may require model denoising.

After that we start the process to reconstruct 3D model of the captured frames. The following flow-chart shows the project's architecture.



The architecture of the project can be summarized as follows:

1. **Acquisition**: In this framework, a sequence of depth frames of the rotating object is acquired using Kinect.

2. **Filtering**: Object segmented from the background for all frames.

3. **Registration**: Stitch all frames together; application of initial/coarse alignment (if required) refined by ICP.

4. **Denoising and Smoothing**: 3D model point denoising to remove misaligned point clouds, followed by model smoothing to create a clean 3D model

5. **Meshing**: Watertight triangulation mesh result

1.2 Software Tools

1. **Qt**: The software is implemented using c++ programming. Our team choose to use Qt Creator, the cross-platform IDE to realize the software. The Qt GUI module provides classes for a variety of GUI tools such as windowing system integration, event handling and 2D graphics, an ideal consolidation for all functions of 3D scanning.

2. **Point Cloud Library (PCL)**: For various point cloud processing, we apply state-of-the-art algorithms from the open project framework PCL. Specifically, PCL version 1.8 is used, mainly due to its compatibility to interface with our selected scanner.

3. **Git**: Github is used for version control and code management.

1.3 Library installation

In this project, Point Cloud library 1.8 is used to build our 3D scanner software. The original PCL 1.8 all-in-one installer could be found here: **Click here**. The Point Cloud Library (or PCL) is a large scale, open project for 2D/3D image and point cloud processing. The PCL framework contains numerous state-of-the art algorithms including filtering, feature estimation, surface reconstruction, registration, model fitting and segmentation.

Point Cloud Library 1.8 installation instruction prerequisites: PC Windows 10 (x64) and a USB 3.0 port for Kinect V2. I am assuming that you do not change the default paths of any of the installers. If you change installation directories, its up to you to modify the steps and files.

1. Download PCL 1.8 all-in-one installer x64 **Click here** and the .pdb files with PCL-1.8.0-AllInOne-msvc2015-win64.exe **Click here**

   Install the .exe remembering to tick the option to add path for all users, and then extract the files in the .zip files and copy them inside the bin folder at `C:\ProgramFiles\PCL1.8.0\bin`

   Make sure to update your environment variables to include: User Variables: - `PCL_ROOTC:\ProgramFiles\PCL1.8.0` (your PCL folder) System Variable Path (already exist. make sure following value are written):

- `%PCL_ROOT%\bin`
- `%PCL_ROOT%\3rdParty\FLANN\bin`
- `%PCL_ROOT%\3rdParty\VTK\bin`
- `%OPENNI2_REDIST64%`

and RESTART your computer for the path to update.

2. Download and install Kinect SDK 2.0 from: **Click here**
3. Download and install MSVC2015 compiler from: **Click here**. Remember to add the new Qt5.7-MSVC libraries to QtCreator.
4. Make a new kit for compilation that includes the MSVC2015 compiler and Qt5.7-MSVC.
5. Change the build type to Release in Qtcreator.

Chapter 2

Project Management

2.1 Objectives

The following major objectives are targeted by our group as specified in the following chapters.

- Targets home-made and quick scanning (acquisition time less than 90 seconds)
- Stationary Kinect V2 for depth data acquisition
- Available Point Cloud Library used as the main resource for depth data processing
- Implementation of fully-functional 3D scanning software outputting watertight triangulation meshes
- User-friendly interface designing to allow the user to interact with our program.
- Rotating turntable construction to target serial communication

2.2 Management Structure

A management structure was established for successful collaborative work amongst our group. We divided into four teams across the group and undertook different division of labor as below:

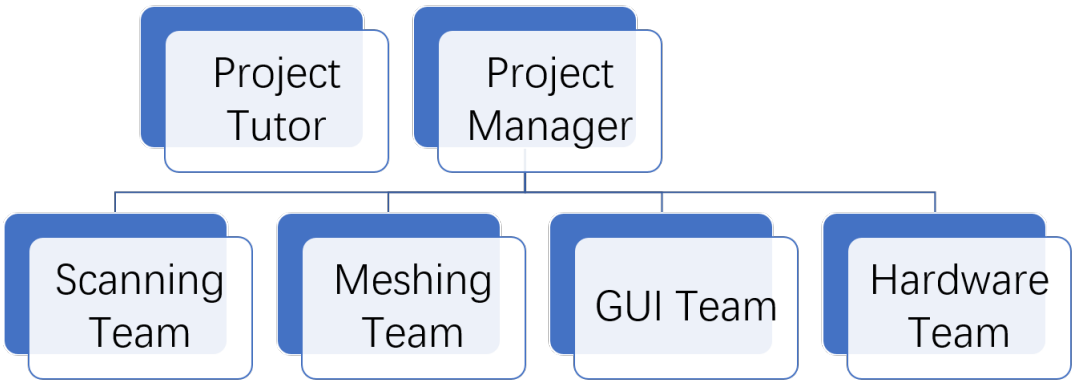


Figure 2.1: Project Structure

1. **Scanning Team** – Interfacing Kinect v2, grabbing PCD frames, registration.
2. **Meshing Team** – Normal estimation, upsampling, meshing techniques, mesh smoothing.
3. **GUI Team** – QVTK widgets, user friendly interface designing.
4. **Hardware Team** – Rotating turntable construction, serial communication.

Name	Main personal responsibilities
Project Tutors: Yohan Fougerolle, Cansen Jiang	
Project Manager: Shubham Wagh	
Scanning Team-Team Leader : Yu Liu	
Yu Liu	Multiple ICP and ICP with Normal implementation; Incremental registration, Kdradius noise removal
Abdelrahman	Frame preprocessing and ICP implementation; Filter/registration OOP organization
Hassan	ICP and Ransac, MLS smoothing, Filter/registration OOP organization
Meshing Team -Team Leader : Shubham	
Shubham	Interfacing Kinect v2 with Grabber, Greedy Triangulation, Grid Projection, Poisson Surface Reconstruction, Code Organization
Gopikrishna	Upsampling point clouds, Laplacian Smoothing of mesh, Code Organization
Ziyang	Normals Estimation, k-d tree search
Avinash	Geometrical background regarding triangulation and watertight meshes, Class Diagram
GUI Team-Team Leader : Anirudh	
Anirudh	Class Diagram, QVTKWidget display, Initial GUI, Interactive GUI test, Code Organization
Chunxia	Code organization and Demo development, Designing and Finalizing GUI, team support
Darja	Code organization and Demo development, Designing and Finalizing GUI, team support
Hardware Team- Team Leader : Ziyang	
Ziyang	Construction of scanning rig, designing and testing
Avinash	Configuring Qt GUI with Turntalble via serial communication, designing and testing

Figure 2.2: Main Personal Responsibilities

2.3 Management Tools

We used the **FreedCamp** as one of project management tools across our group to increase our group's productivity. Documents and information are uploaded and posted regularly. Tasks are assigned to each group members. Milestones and deadline are set and backup solutions are proposed also with the events scheduled on calendar. We used discussion boards to comment and give feedback to each other, tracking time spent on tasks.

What's more, we set up the Gantt chart to plan all the tasks involved in our project.

As part of the process, we have worked out who will be responsible for each task, how long each task will take, and what problems our team may encounter. This Gantt chart helped us ensure that the schedule was workable, that the right people were assigned to each task, and that we had workarounds for potential problems before we started. The Gantt chart tool also helped us work out practical aspects of this project, such as the minimum time it will take to deliver. Plus, we can use them to identify critical procedure - the sequence of tasks that must individually be completed on time if the whole project is to deliver on time. At last, we updated the chart to show schedule changes and their implications, using it to communicate the key tasks that have been completed. We used them to keep our group and our tutors informed of progress

2.4 Quality plan

A quality plan (shown as **Gantt chart** and every weekly report) was prepared to clearly define the personal and sub-team responsibilities for the execution for the project tasks, for supervision of the project tasks and control the project progress. As the quality plan (Gantt chart) was regularly updated in every weekly report, it gave a clear and actual status of the project.

2.5 Project meetings

During the duration of the project, several overall project meetings, sub-group meetings, workshops and review meetings with tutors were held as the main forum for interaction between groups, see Figure 2.3. Sub-group meetings were held regularly after the weekly overall group meetings.

Weekly meetings	
[1]	Kick-off meeting, October.3rd,2016
[2]	Project meeting-week 1, October.9th,2016, Meeting recorder: Abdel
[3]	Project meeting-week 2, October.15th,2016, Meeting recorder: Darja
[4]	Project meeting-week 3, October.22th,2016, Meeting recorder: Ziyang
[5]	Project meeting-week 4, October.29th,2016, Meeting recorder: Avinash
[6]	Project meeting-week 5, November.5th,2016, Meeting recorder: Chunxia
[7]	Project meeting-week 6, November.12th,2016, Meeting recorder: Hassan
[8]	Project meeting-week 7, November.19th,2016, Meeting recorder: Anirudh
[9]	Project meeting-week 8, November.26th,2016, Meeting recorder: Shubham
[10]	Project meeting-week 9, December.3rd,2016, Meeting recorder: Gopi
[11]	Project meeting-week 10, December.17th,2016, Meeting recorder: Alpha
[12]	Project meeting-week 11, December.24th,2016, Meeting recorder: Darja
[13]	Project meeting-week 12, December.31st,2016, Meeting recorder: Chunxia
Workshops	
[1]	Study session 1, October.30th,2016
[2]	Study session 2, December.15th,2016,
[3]	Study session 3, January.1-2,2016
Review meeting with Tutors (Yohan and Cansen)	
[1]	Review session 1, October.25th,2016, Meeting with Cansen
[2]	Review session 1, November.4th,2016, hosted by Yohan
[3]	Review session 2, December.5th,2016, hosted by Yohan
[4]	Review session 3, December.16th,2016, hosted by Yohan and Cansen
[5]	Review session 4, December.19th,2016, Meeting with Cansen

Figure 2.3: Key Project Meetings

2.6 Milestones

All milestones listed in the Figure 2.4 as below have been achieved in due time.

No.	Milestone name	Achieved(Yes/No)	Achievement week
1.	Software Tools installation	Yes	Week 2
2.	Kinect v2 Interfacing	Yes	Week 3
3	Full-body scan	Yes	Week 5
4.	Full-body auto-scan	Yes	Week 10
5.	PCL Available for depth data processing	Yes	Week 5
6.	Registration	Yes	Week 10
7	Watertight Meshing and Comparison	Yes	Week 10
8	Initial GUI	Yes	Week 9
9	Designing and Finalizing GUI	Yes	Week 11
10.	Interactive GUI	Yes	Week 12
11.	Turntable Hardware Setup	Yes	Week 10
12.	Turntable Interfacing	Yes	Week 11
13.	Final successful Testing	Yes	Week 12

Figure 2.4: Milestones

Chapter 3

Depth Sensors and Interfacing

3.1 Depth Sesnor

3.1.1 Intel Realsense R200

The Intel RealSense Development Kit R200 (referred to as the R200) is a small, USB-powered camera that includes a depth sensor and an RGB sensor. The R200 supports enhanced photography by providing a standard Full HD image, and in addition the camera will enable the manipulation of color and depth from multiple angles and perspectives. This allows the user to create artistic and creative depth-based filters and manipulate images through parallax, dolly zoom, motion affects, object segmentation, color popping, and more. The R200 is capable of scanning people or objects and digitally capturing the 3D world. Body scans can be used to build 3D printable busts or to create personalized avatars. Objects can be measured and incorporated into real-world spaces to visualize possibilities.

Figure 3.1: Realsense R200

Specification:

- Up to 3.5 meters indoors, longer range outdoors

- Depth/IR: Up to 640x480 resolution at 60 FPS
- RGB: 1080p at 30 FPS
- USB 3.0 port required
- Dimensions: 130mm x 20mm x 7mm

An example from here Click here shows that R200 can be interfaced with Point Cloud Library on Linux operation system Ubuntu.

3.1.2 Microsoft Kinect V2

Kinect V2 was released with Xbox One on November 22, 2013. The hardware included a time-of-flight sensor developed by Microsoft. Kinect V2 is a camera using time-of-flight technology. It is a physical device with depth sensing technology, built-in color camera, infrared (IR) emitter, and microphone array, can sense the location and movements of people as well as their voices. Kinect V2 has higher depth fidelity and a significantly improved noise floor, the sensor gives you improved 3D visualization, improved ability to see smaller objects and all objects more clearly, and improves the stability of body tracking

Kinect V2 Specifications:

- Depth sensing: 512 x 424 30 Hz. FOV: 70 x 60. One mode: .5–4.5 meters
- 1080p color camera: 30 Hz (15 Hz in low light)
- New active infrared (IR) capabilities: 512 x 424 30 Hz
- sensor dimensions (length x width x height): 24.9 cm x 6.6 cm x 6.7 cm. Sensor FOV: 70 x 60

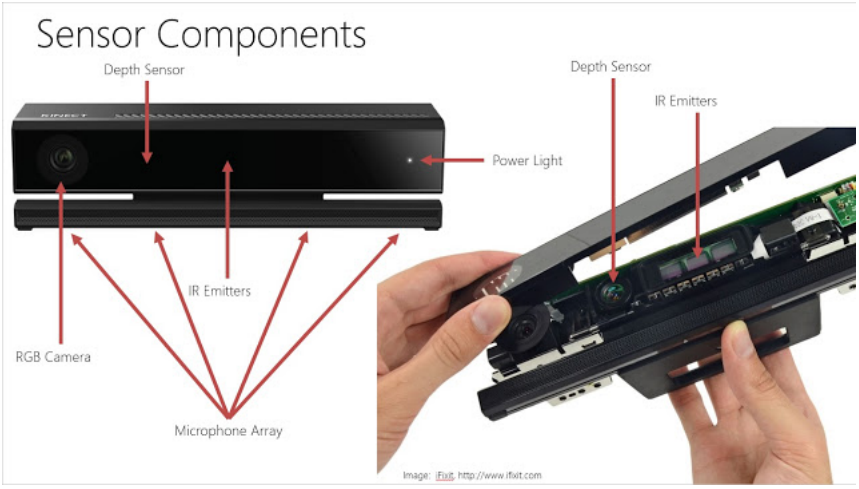


Figure 3.2: Kinect Version 2

3.2 Kinect v2 Interfacing using Grabber

Point Cloud Library (PCL) have some modules for input data that called "**Grabber**".

- OpenNI2Grabber: This module is grabber for input data from PrimeSensor based on OpenNI2.
- HDLGrabber/VLPGrabber: This module is grabber for input data from Velodyne LiDAR based on Boost.Asio and PCAP.
- RealSenseGrabber: This module is grabber for input data from Intel RealSense cameras based on RealSense SDK and librealsense.

3.2.1 PCL with Kinect v2

PCL doesn't have grabber for input data from Kinect v2. So, we take input data from Kinect v2 based on Kinect SDK v2.

Kinect2Grabber is implemented as class that inherits from **pcl::Grabber**. We can implement original Grabber that are not included in PCL by implementing class that inherits from **pcl::Grabber**.

Kinect2Grabber is grabber to input data from Kinect v2 based on Kinect SDK v2. This Grabber only depends on Kinect SDK v2. We don't need to have other libraries to use Kinect2Grabber. In addition, this Grabber has been implemented in only one header file (**kinect2_grabber.h**). We use Kinect2Grabber by just including this header file.

3.2.2 Retrieving Point Cloud

Firstly, we make an instance of pcl::Kinect2Grabber.

```
boost::shared_ptr<pcl::Grabber> grabber = boost::make_shared<pcl::
    Kinect2Grabber>();
```

Listing 3.1: Grabber instance

Secondly, we define callback function to retrieve point cloud using pcl::Kinect2Grabber. This callback function is called each time to retrieve point cloud of one frame. We will write point cloud processing in here, and returns shared pointer of point cloud to outside.

```
pcl::PointCloud<PointType>::ConstPtr cloud;
boost::mutex mutex;
boost::function<void( const pcl::PointCloud<PointType>::ConstPtr& )>
    function =
    [&cloud, &mutex]( const pcl::PointCloud<PointType>::ConstPtr& ptr ){
        boost::mutex::scoped_lock lock( mutex );

        /* Point Cloud Processing */

        cloud = ptr->makeShared();
    };
```

Listing 3.2: Point Cloud callback function

Lastly, we will register callback function to pcl::Kinect2Grabber, followed by giving a function pointer to pcl::Kinect2Grabber::registerCallback().

```
boost::signals2::connection connection = grabber->registerCallback(
    function );
```

Listing 3.3: Register callback function

By this way, we retrieve real time display of point cloud from Kinect v2

```
// Start Grabber
grabber->start();

while( !viewer->wasStopped() ){
    // Update Viewer
```

```
6       viewer->spinOnce();
7
8       boost::mutex::scoped_try_lock lock( mutex );
9       if( lock.owns_lock() && cloud ){
10          // Update Point Cloud
11          if( !viewer->updatePointCloud( cloud, "cloud" ) ){
12              viewer->addPointCloud( cloud, "cloud" );
13          }
14      }
15  }
16
17  // Stop Grabber
18  grabber->stop();
```

Listing 3.4: Sample code for Kinect v2

Figure 3.3: Real-time display of Point Cloud

Chapter 4

Registration

4.1 Filtering

After acquiring the 3D point clouds from Kinect, the next step is to filter them to retain only the points of the scanned body, removing all unwanted points from background and the outliers of the body. For this purpose we use many filter functions provided by the pcl_filters library from PCL [1].

In this chapter, we will introduce the filters we used, in addition to the class "cfilter" in our code containing the implementation of all these filters. Next snippet of code shows the header file of "cfilter" class, it has 4 methods, one for each filter used in our code. For each section in this chapter, we will introduce the idea of each filter, how to use it, then the details of the method in our "cfilter" class implementing it.

```
1  // The "cfilter" class implements all filtering functions to be used in
       other classes
2  class cfilter {
3  public:
4  cfilter();  // constructor
5  \~cfilter(); // destructor
6  PointCloud::Ptr PassThrough(PointCloud::Ptr, float, float, float, float,
       float, float);
7  PointCloud::Ptr OutlierRemoval(PointCloud::Ptr , float);
8  PointCloud::Ptr VoxelGridDownSample(PointCloud::Ptr , float);
9  PointCloud::Ptr Smoothing(PointCloud::Ptr, float );
10 };
```

Listing 4.1: Header file for **cfilter** class

4.1.1 Passthrough filtering

The passthrough filter will remove any point from the cloud whose values do not fall in a certain user-given range. For example, if you wanted to discard all points farther away than, say, 3 meters, you would run the filter on the Z coordinate with a range of [0,3]. This filter can be useful to discard unneeded objects from the cloud, but you may have to adapt a different reference frame if the default one (relative to the sensor) is not fitting. For example, filtering on the Y value to remove all points not sitting on a turning table will yield unwanted results if the camera was at an odd angle. For our project, we try to make the kinect horizontal as we can, to get good results regarding Z and Y filtering [7] [3].

In our project, we are allowing filtering in X, Y, and Z planes, by setting the cube size from the GUI interface. the values obtained by the cube is then passed to the paththrough method. Next code snippet shows the implementation of this method in the cfilter class

```
// Implementation of PassThrough() method in cfilter class
PointCloud::Ptr cfilter::PassThrough(PointCloud::Ptr cloud, float z1, float
    z2 , float y1, float y2, float x1, float x2) {
    PointCloud::Ptr cloud_filteredz (new PointCloud);
    PointCloud::Ptr cloud_filteredy (new PointCloud);
    PointCloud::Ptr cloud_filtered (new PointCloud);

    // Create the passthrough filtering object
    pcl::PassThrough<PointT> pass;
    pass.setInputCloud (cloud);
    pass.setFilterFieldName ("z");
    pass.setFilterLimits (z1,z2);
    pass.filter (*cloud_filteredz);

    pass.setInputCloud (cloud_filteredz);
    pass.setFilterFieldName ("y");
    pass.setFilterLimits (y1,y2);
    pass.filter (*cloud_filteredy);

    pass.setInputCloud (cloud_filteredy);
    pass.setFilterFieldName ("x");
    pass.setFilterLimits (x1, x2);
    pass.filter (*cloud_filtered);
```

```
23     return cloud_filtered;
24 }
```

Listing 4.2: PassThrough Filter implementation

the PassThrough method takes seven arguments, first the point cloud pointer to the input point cloud to be filtered, the next six are the limits of the filtering in Z, Y, and X respectively. The code works in the way that after filtering in Z plane, the filtered output is the input of the Y filter, then the output is filtered in X plane. the function then return a pointer the point cloud filtered in X-Y-Z planes.

4.1.2 Statistical Outlier removal

Another kind of data to be removed is the outliers points. Due to measurement errors, certain datasets present a large number of shadow points. This complicates the estimation of local point cloud 3D features. Some of these outliers can be filtered by performing a statistical analysis on each point's neighborhood, and trimming those which do not meet a certain criteria [7].

The sparse outlier removal implementation in PCL is based on the computation of the distribution of point to neighbors distances in the input dataset. First, for every point, the mean distance to its K neighbors is computed. Then, if we assume that the result is a normal (gaussian) distribution with a mean μ and a standard deviation σ, we can deem it safe to remove all points with mean distances that fall out of the global mean plus deviation. Basically, it runs a statistical analysis of the distances between neighboring points, and trims all which are not considered "normal", you define what "normal" is with the parameters of the algorithm. this goal can be achieved in PCL using StatisticalOutlierRemoval function [5].

Next code snippet introduces the method OutlierRemoval(), in our "cfilter" class:

```
1 // Implementation of OutlierRemoval() method in cfilter class
2 PointCloud::Ptr cfilter::OutlierRemoval(PointCloud::Ptr cloud, float
    Threshold) {
3     PointCloud::Ptr cloud_filtered (new PointCloud);
4
5     pcl::StatisticalOutlierRemoval<PointT> sor;
6     sor.setInputCloud (cloud);
7     sor.setMeanK (50);
```

```
8     sor.setStddevMulThresh (Threshold);
9     sor.filter (*cloud_filtered);
10
11    return cloud_filtered;
12 }
```

Listing 4.3: Outlier Removal Filter implementation

The method take 2 arguments. First, a pointer to the input point cloud to be filtered, second the deviation threshold, after which the point will be removed.

4.1.3 Downsampling

The data acquired by kinect is huge, if we were to perform a simple operation on every point of a cloud, it would be of O(n), n being the number of points. If we had to compare every point with its k nearest neighbors, it would be O(nk). We noticed from the first time we worked with PCL, that even simple operations take much time. Some solutions may consider working on GPUs, which allow more processing power, but because this is not our case, we should find better solution. A common operation is also downsampling, which will give you back a cloud that, for all intents and purposes, equivalent to the original one, despite having less points [7].

The solution we are working with is downsampling. Downsampling is done via voxelization, which introduce many voxel grids (think about a voxel grid as a set of tiny 3D boxes in space). The cloud is divided in multiple cube-shaped regions with the desired resolution. Then, all points inside every voxel are processed so only one remains. The simplest way would be to randomly select one of them, but a more accurate approach would be to compute the centroid, which is the point whose coordinates are the mean values of all the points that belong to the voxel. If done with sensible parameters, downsampling will yield results that are precise enough to work with, at the cost of less CPU power [7] [6].

We implemented the downsampling using VoxelGrid() function in PCL. The next code snippet shows our VoxelGridDownSample() method inside the "cfilter" class:

```
1 // Implementation of VoxelGridDownSample() in cfilter class
2 PointCloud::Ptr cfilter::VoxelGridDownSample(PointCloud::Ptr cloud, float
    LeafSize) {
3     PointCloud::Ptr cloud_filtered (new PointCloud);
```

```
4
5       pcl::VoxelGrid<PointT> vox;
6       vox.setInputCloud (cloud);
7       vox.setLeafSize (LeafSize, LeafSize, LeafSize);
8       vox.filter (*cloud_filtered);
9
10      return cloud_filtered;
11  }
```

Listing 4.4: Voxel Grid Filter implementation

The VoxelGridDownSample() method takes 2 arguments. first, a pointer to the input point cloud, and second the leaf size of the voxelgridbused. Although the original PCL setLeafSize() function takes three parameters for each dimension of the grid, in our code we assume the voxel grid to be cubic, so that the three leaf sizes are the same. It is important to point out that bigger the leaf size, less points you get in output (more downsampling).

4.1.4 Smoothing

As stated, depth sensors are not very accurate, and the resulting clouds have measurement errors, outliers, holes in surfaces, etc. Surfaces can be reconstructed by means of an algorithm, that iterates through all points and interpolates the data, trying to recreate the missing parts of the surface by higher order polynomial interpolations between the surrounding data points. By performing resampling, these small errors can be corrected and the “double walls” artifacts resulted from registering multiple scans together can be smoothed. [4]

PCL uses the Moving Least Squares algorithm. Performing this step is important, because the resulting cloud’s normals will be more accurate, as we will see later. Next code snippet shows the smoothing() method used to implement the smoothing in ”cfilter” class.

```
1 // Implementation of smoothing() method in cfilter class
2 PointCloud::Ptr cfilter::Smoothing(PointCloud::Ptr cloud, float
    SearchRadius) {
3     PointCloud::Ptr cloud_smoothed (new PointCloud);
4
```

```
    pcl::search::KdTree<PointT>::Ptr tree (new pcl::search::KdTree<PointT>);  // Create a KD-Tree
    // Output has the PointNormal type to store the normals calculated by MLS
    pcl::PointCloud<pcl::PointNormal> mls_points;
    pcl::MovingLeastSquares<PointT, pcl::PointNormal> mls;

    mls.setComputeNormals (true);
    mls.setInputCloud (cloud);
    mls.setPolynomialFit (true);
    mls.setSearchMethod (tree);  // KD-Tree search method
    // Set the sphere radius that is to be used for determining the k-nearest neighbors used for fitting.
    mls.setSearchRadius (SearchRadius);
    mls.process (mls_points);
    // To convert from XYZNormals to XYZ
    pcl::copyPointCloud(mls_points, *cloud_smoothed);

    return cloud_smoothed;
}
```

Listing 4.5: Smoothing implementation

The smoothing() method takes 2 arguments, first one is the pointer to the input point cloud, and the second is the Search radius for which the algorithm will choose the neighbors to use in the algorithm.

4.2 Iterative Closest Points (ICP)

The problem of registering two point clouds consists in finding the rotation and the translation that maximize the overlap between the two clouds [18]. ICP is an algorithm employed to minimize the difference between two clouds of points.

$$T = \begin{pmatrix} R & t \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{4.1}$$

Following the principle of the ICP algorithm iteratively register point clouds [12]:

4.2.1 Correspondence Estimation

registering large point clouds is considerably more computationally expensive than registering clouds of smaller cardinality. registering only subsets of the original point clouds can yield sufficient results while saving computation time. In principle, two methods of data reduction can be distinguished: automatically extracting a small set of unique and repeatable key points and sampling of the original data with respect to a desired target distribution.

Correspondence estimation is the process of pairing points p_i from the source point cloud P to their closest neighbors q_j in the target cloud Q.

4.2.2 Rejecting and Filtering Correspondences

1. Correspondence rejection based on distance: This method filters out point pairs with a distance larger than a given threshold (see Figure 4.1a).

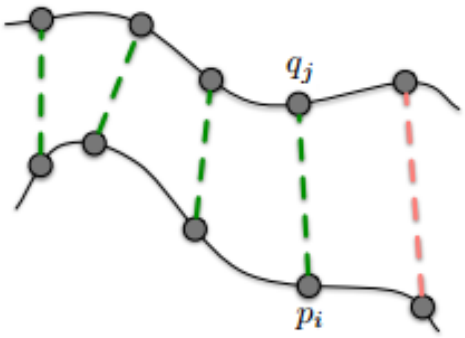
2. Rejection based on median distance: Unlike the previous rejector, this one does not use a fixed threshold, but computes it as the median of all point-to-point distances in the input set of correspondences. Hence, it considers the distribution of the distances between the points and adapts to it, becoming smaller as the two point clouds get closer during the ICP iterations.

3. Rejecting pairs with duplicate target matches: Usually, each sampled point in the source cloud is assigned to a correspondence in the target cloud. Hence, it might happen that a point in the target cloud is assigned multiple corresponding source points. This rejector only keeps a single such pair (p_i ,q_j), the one with the minimum distance out of all the pairs ($p(i_{min})$, q_j) (see Figure 4.1c).

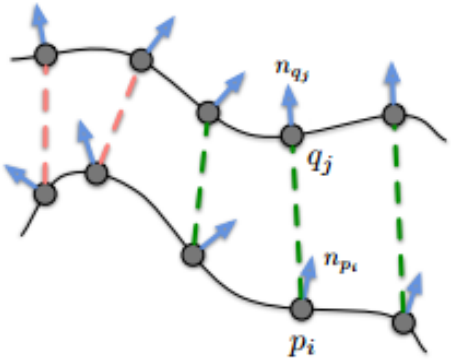
4. RANSAC-based rejection: This method applies Random Sample Consensus (RANSAC) to estimate a transformation for subsets of the given set of correspondences and eliminates the outlier correspondences based on the Euclidean distance between the points after the computed transformation is applied to the source point cloud. It is very effective in keeping the ICP algorithm from converging into local minima, as it

always produces slightly different correspondences and is good at filtering outliers. In addition, it provides good initial parameters for the transformation estimation with all inlier correspondences that follows.

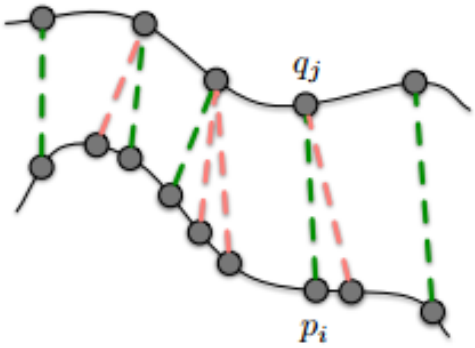
5. Rejection based on normal compatibility: This filter uses the normal information about the points, and rejects those pairs that have inconsistent normal (see Figure 4.1a).



(a) Rejection based on the distance between the points.



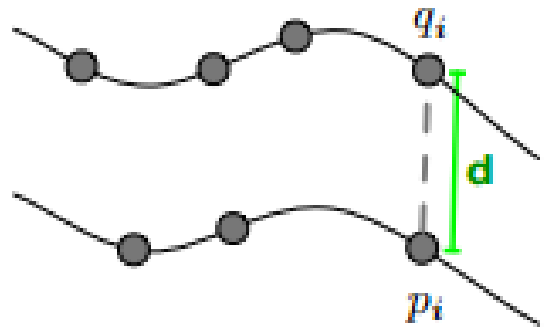
(b) Rejection based on normal compatibility.



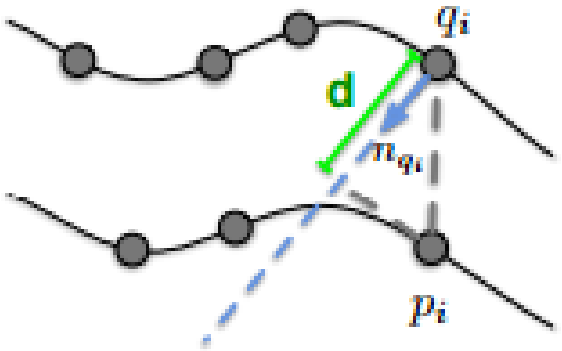
(c) Rejection of pairs with duplicate target matches.

Figure 4.1: Correspondence rejection. Good correspondence pairs (green) are kept while outliers (red) are sorted out to improve convergence.

6. Rejector pipelines: Most often, not only a single rejector is applied, but instead several correspondence rejectors are queued in order to implement a filtering pipeline.



(a) Point to point error



(b) Point to plane error

Figure 4.2: Error metrics and transformation estimators.

4.2.3 Alignment-Error Metrics and Transformation Estimation

There have been numerous mathematical approaches for solving for the rigid transformation T that minimizes the error of the point pairs. T is composed of a rotation R and a translation t. There are two main error metrics to be minimized that have been considered in literature: point-to-point (Eq.(4.2)) and point to-plane (Eq.(4.3)), where (p_k, q_k) is the $k-th$ of the N pair correspondences from the source cloud to the target cloud.

$$E_{point-to-point}(T) = \sum_{k=1}^{N} w_k \|Tp_k - q_k\|^2 \tag{4.2}$$

$$E_{point-to-plane}(T) = \sum_{k=1}^{N} w_k ((Tp_k - q_k).n_{q_k}^2 \tag{4.3}$$

The optional w_k can be used for weighting the pairs in order to give them more or less importance in the least squares formulation ($w_k = 1$ if no weighting is applied):

- Standard point-to-point error metric: The standard error metric used in the ICP algorithm is the point-to-point error metric (Eq.(4.2)).
- Point-to-plane error metric(Eq.(4.3)): It has proved it to be more stable and converge faster than the previous approaches. It uses the distance between the source point p_k and the plane described by the target point q_k and its local surface normal n_{q_k}.
- Linear least squares point-to-plane.
- Weighted point-to-plane error metric: Assigning a different weight to each correspondence can improve convergence. The weighting of the point pairs can be seen as a soft correspondence rejection, adjusting the influence of noisy corresponding points in the minimization process. The weighting can be a function of the point-to-point or point to-plane distance between the points, a function of the angle between the normals corresponding to the points, or a function of the noise model of the sensor that has been used.
- Weighted linear least squares point-to-plane.

4.2.4 Termination Criterian

1. Maximum number of iterations: Exceeding the number of iterations means that the optimizer diverged. This threshold has to be tuned depending on the complexity of the registration problem; expect that registering a pair of scans that are far away from each other will require more iterations, than two scans that have a good initial alignment.
2. Absolute transformation threshold: The iterations are stopped when the currently estimated transformation is far away from the initial transformation. This is an early termination criteria for registration procedures that diverge. The intuition behind it is that the two clouds to be aligned are expected to be within a certain range of distances from each other, and so transformations that are outside that range need to be rejected.
3. Relative transformation threshold: It specifies the minimum transformation difference from one iteration to the next that is considered small enough for the optimizer

to have converged.

4. Maximum number of similar iterations: we allow the optimizer to spend a certain number of iterations around a minimum point before considering it converged.

5. Relative mean square error.

6. Absolute mean square error: It stops the iterations when the error between the two aligned clouds is below a certain value.

All of these techniques represent ways of balancing between the quality and the runtime of the registration procedure.

4.3 Implementation

In this section, we describe how registration is realized in our software.

We divide the registration task into two main parts: preprocessing and alignment. We create a filter class **cfilter** which contains all functions for preprocessing and smoothing. Below is a code snippet of its header file and the declarations of its member functions.

```
// The "cfilter" class implements all filtering functions to be used in
    other classes
class cfilter {
public:
    cfilter();
    ~cfilter();
    ...
    PointCloud::Ptr PassThrough(PointCloud::Ptr, float, float, float, float
    , float, float);
    PointCloud::Ptr OutlierRemoval(PointCloud::Ptr , float);
    PointCloud::Ptr VoxelGridDownSample(PointCloud::Ptr , float);
    PointCloud::Ptr Smoothing(PointCloud::Ptr, float);
    ...
};
```

Listing 4.6: Header file for **cfilter** class

Similarly, we create a registration and meshing class **regmesh**. This class contains all functions addressing alignment of point clouds, as well as taking the alignment result for meshing (to be discussed in Chapter 4). Below is a code snippet of its header file and the declarations of its member functions. Details regarding each function embedded in the above two classes are discussed in the following sections.

```
// Class regmesh implements all registration and meshing functions. For the
    topic of this chapter, only the registration portion is displayed.
class regmesh{
public:
    regmesh();
    ~regmesh();
    ...
    PointCloud::Ptr ICP(PointCloud::Ptr, PointCloud::Ptr, float, float,
    float);
    PointCloud::Ptr ICPNormal(PointCloud::Ptr, PointCloud::Ptr, float,
```

```
    float , float);
    PointCloud::Ptr ICP2(PointCloud::Ptr, PointCloud::Ptr, float, float,
    float, float, float, float);

    PointCloud::Ptr Register(std::vector<PointCloud::Ptr> & );
    PointCloud::Ptr RegisterNormal(std::vector<PointCloud::Ptr> & );
    PointCloud::Ptr Register2(std::vector<PointCloud::Ptr> & );
    ...
};
```

Listing 4.7: Header file for **regmesh** class

The diagram below summaries our pipeline of the registration.

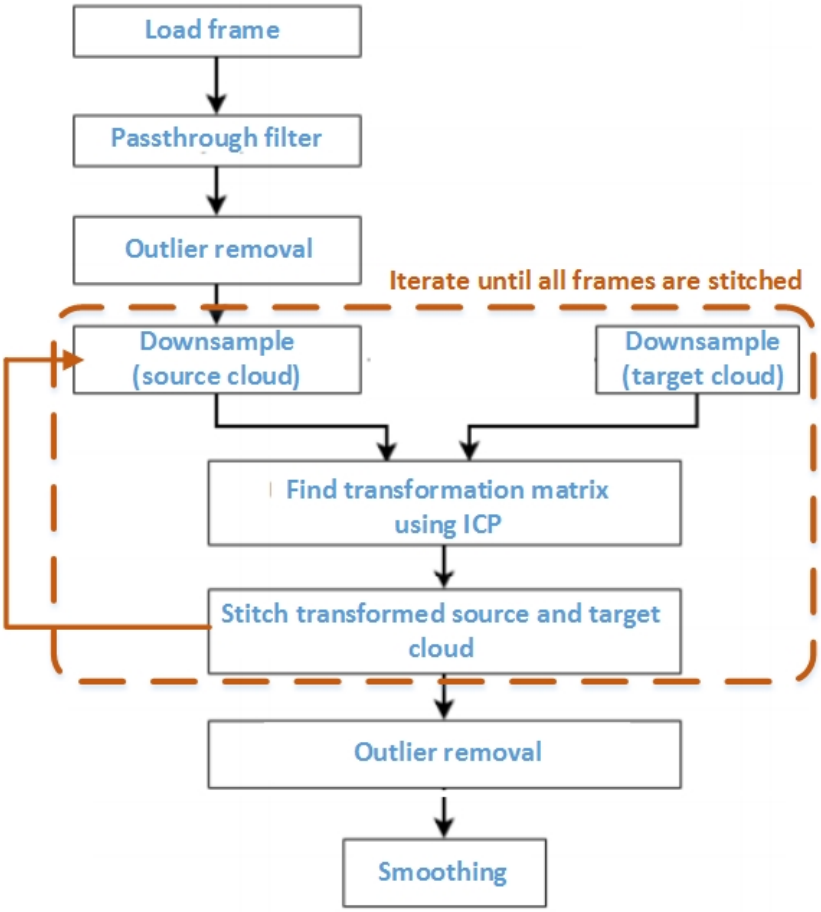


Figure 4.3: Registration Pipeline

4.3.1 Preprocessing - Passthrough Filter

The raw point clouds contain roughly 220,000 points per frame. To extract meaningful points corresponding to the subject, we first apply PCL passthrough filter to simply cut

off points that lie outside the bounded range.

The function cfilter::PassThrough takes a raw point cloud pointer as the input to the filter, and returns a filtered point cloud pointer. We initially specified fixed-bounded range (-0.4, 0.4), (-0.85, 1.2), and (0.1, 2.0) meter for x, y, and z direction respectively. Points confined within the range would be kept while others were to be filtered. Later in the design, an adaptive passthrough filter is implemented to flexibly take any filtering parameters such that prior to scanning, user is able to view the scene in real-time and adjust the cut-off range accordingly (details to be discussed in the GUI chapter).

Following is the result after passthrough filtering is applied. For a sequence of cloud frames, the same filtering parameters are applied. After passthrough filtering, approximately one-tenth of the points are left.

Figure 4.4: A frame acquired by kinect before and after passthrough filtering

4.3.2 Preprocessing - Outlier Removal Filter

We define outliers as points that are significantly far apart from the main cluster of point cloud. We are able to remove outliers by applying PCL statistical outlier removal after passthrough filtering.

The function cfilter::OutlierRemoval takes the point cloud pointer processed by the passthrough filter as the input, and returns the noise-removed point cloud pointer. We define the number of neighbors to be analyzed for each point as 50; this means that we compute the mean distance from every point to up to its 50^{th} neighbors. The standard

deviation multiplier is set to 5. With such configuration, all points who have a distance larger than 5 standard deviation of the mean distance to the nearest 50 neighbors will be marked as outliers and removed. By setting the standard deviation multiplier high, we ensure that only true outliers are to be removed. A sample of the resulted point cloud is shown as follows.

Figure 4.5: A frame before and after outlier removal

4.3.3 Voxel Grid Filter

As we address aligning a sequence of cloud frames using an incremental approach (to be shown in the next section), number of points will accumulate as we register more frames. The computation complexity greatly increases when more points are considered. Hence, we take the step of downsampling to only keep a portion of the original point cloud.

The function cfilter::VoxelGridDownSample takes a point cloud pointer as the input, and returns the pointer of the downsampled version of that point cloud. The gridsize is set to 0.002 for every edge of the voxel cube.

It should be noted that the downsampling step is performed iteratively for every registration source and target cloud prior to transformation estimation. Unlike passthrough filtering and outlier removal, downsampling is not only conducted on individual cloud frames, but also on the stitched result which is discussed in the following section.

4.3.4 Alignment - ICP

An incremental approach is employed to register a sequence of point clouds. This means the following: we first register frame 1 (source cloud) and frame 2 (target cloud). The result of that (source cloud) is then registered with frame 3 (target cloud), and so on. Below illustrates this approach. As mentioned in the previous section, points accumulate as more frames are registered. Hence downsampling is applied on both source and target frames prior to registration.

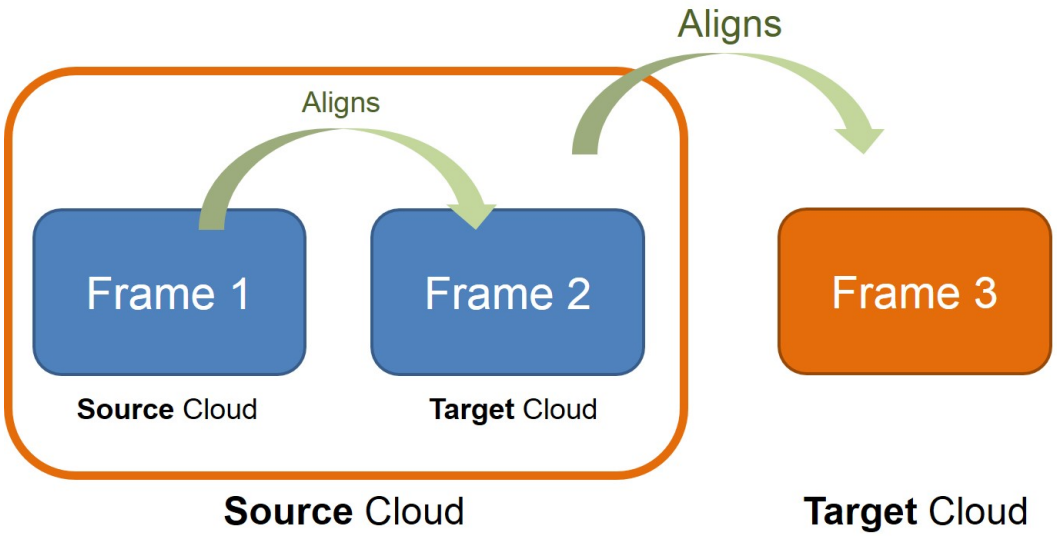


Figure 4.6: Incremental Registration Three Frames

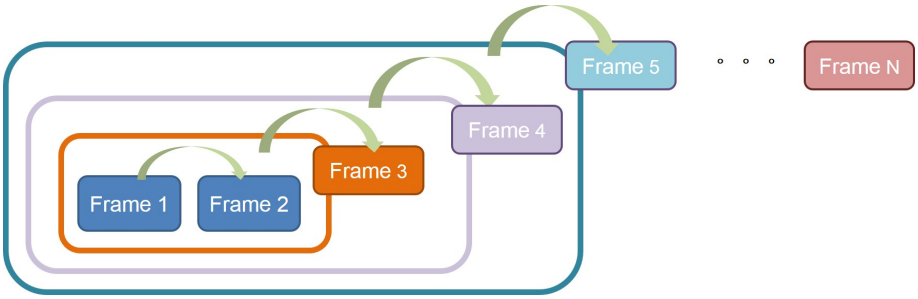


Figure 4.7: Incremental Registration N Frames

We have experimented three approaches to align the sequence of preprocesssed cloud frames. Each approach uses two functions in the **regmesh** source file.

The first approach is the standard ICP. Defined in function regmesh::ICP, this function takes two point cloud pointers as inputs (one points to the source cloud, and the other one to target point cloud). We create an instance of the PCL class IterativeClosestPoint and

configure correspondence estimation, correspondence rejector, and termination conditions via this instance. The maximum correspondence distance is set 0.015 meter (in other words, assuming no correspondences of two frames are beyond 1.5 centimeter apart). Two termination conditions are defined: transformation epsilon (transformation difference = 1e-8) and number of iterations (iteration = 100). After setting the source and target, we align the source with target in the target's frame. The "align()" function inherited from pcl::Registration estimates the transformation and returns the transformed source. The transformed source is the output of our ICP function. See below the code snippet for the single ICP function.

```
ICP(source cloud, target cloud, parameters){
    ...
    pcl::IterativeClosestPoint<PointT, PointT> icp;
    icp.setMaxCorrespondenceDistance (MaxDistance); //0.015
    icp.setRANSACOutlierRejectionThreshold (RansacVar); //0.05
    icp.setTransformationEpsilon (1e-8);
    icp.setMaximumIterations (Iterations); //100
    icp.setInputSource(src);
    icp.setInputTarget(tgt);
    icp.align(*Final);
    return Final; //transformed source cloud
}
```

Listing 4.8: Standard single ICP implementation

The above function deals with aligning any given two frames. To incrementally align a sequence of frames, the function regmesh::Register is created to handle a vector of point cloud pointers (point to frames that have undergone preprocessing) and iteratively downsamples and updates the source and target cloud to regmesh::ICP. In addition, actual stitching of the transformed source and the target is performed here (the stitched result becomes the new source cloud for the next iteration). The code snippet and some registered results are shown as follows.

```
Register(point cloud vector){
    ...
    // Iterate through cloud vector and update source and target clouds for
    ICP
    for (size_t i = 1; i < Data.size (); ++i)
    {
```

```
6           cloud_src = result2; // source
7           cloud_tgt = Data[i]; // target
8
9           // downsample source and target
10          src = FilterObj.VoxelGridDownSample(cloud_src, VoxelGridLeafSize);
11          tgt = FilterObj.VoxelGridDownSample(cloud_tgt, VoxelGridLeafSize);
12
13          // Call ICP
14          ICPResult = ICP(src, tgt, MaxDistance, RansacVar, Iterations);
15
16          // Stitch transformed source to target cloud
17          *ICPResult += *cloud_tgt;
18      }
19      ...
20  }
```

Listing 4.9: Registration using single ICP

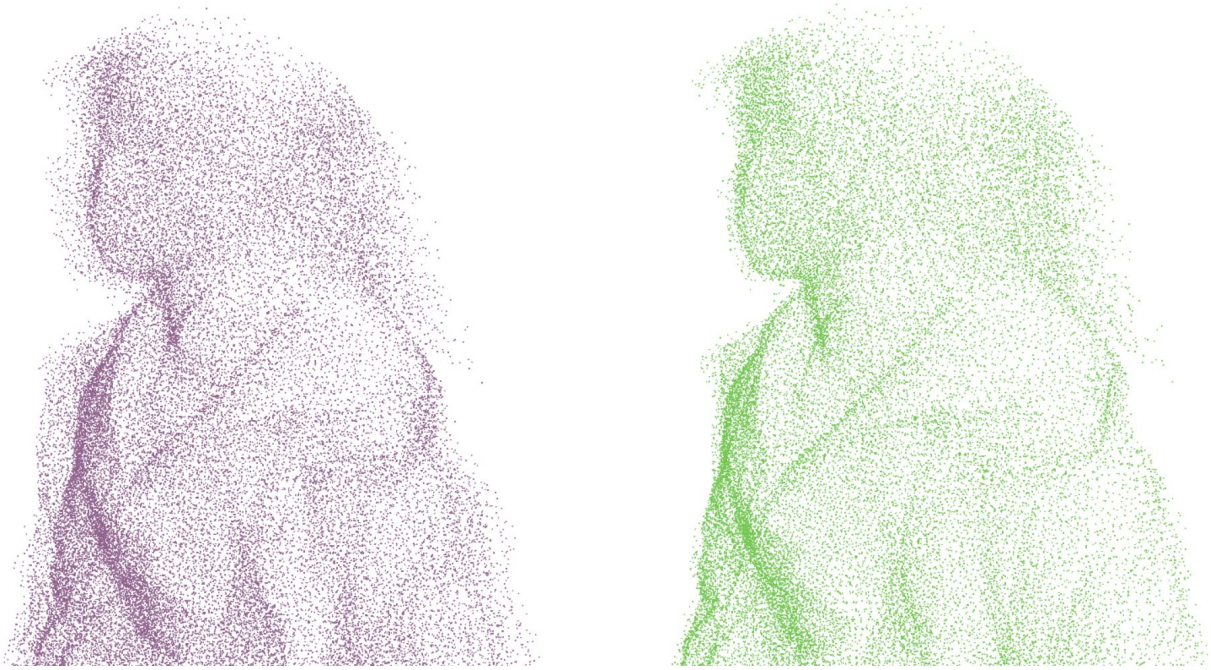


Figure 4.8: Registration using ICP: MaxCorrespondenceDistance = 0.05 (left) and 0.015 (right)

4.3.5 Alignment - Two ICPs

The second approach involves using two ICPs in consecutive order. The additional ICP is added as a step of initial alignment to enhance the robustness of the registration. Our

algorithm does not handle initial alignment using feature descriptors given that we perform scanning at high frequency to increase overlapping regions between frames. However, to test if using half of the current number of frames would be enough for registration, and also to prevent sudden change of orientations in two consecutive frames, we make the two-ICPs approach.

The working flow for applying two ICPs is similar to that of just one. The functions regmesh::ICP2 and regmesh::Register2 handle applying two ICPs. regmesh::ICP2 calls the function regmesh::ICP and uses the transformed cloud returned by that function to continue with the second ICP. In regmesh::Register2, a set of parameters for correspondence estimation and terminations are set to configure either the first or the second ICP. Same to the single ICP approach, actual stitching is achieved in regmesh::Register2 (regmesh::ICP2 only handles transformation of source cloud to the target frame). In fact, parameters are the same for both the first and second ICP except for maximum correspondence distance (looser condition for the first ICP).

```
ICP2(source cloud, target cloud, parameters){
    ...
    // Call ICP
    out1 = ICP(src, tgt, MaxDistance1, RansacVar1, Iterations1);

    //Call seond ICP using out1 as input
    out2 = ICP(out1, tgt, MaxDistance2, RansacVar2, Iterations2);
    return out2; // transformed (two ICPs) source cloud
}
```

Listing 4.10: Standard two ICP implementation

4.3.6 Alignment - ICP with Normals

The idea of using the third approach - ICP with normals, came from [12]. It was stated by Chen and Medioni [12] , that the point-to-plane metric proves to be more stable and converges faster than the standard error metric used in the previous ICP algorithm. Because normals have been computed on both source and target point clouds, they approximate the underlying surface which is preferred as a better estimation of the distance between the source and target surfaces (as opposed to point-to-point measurements).

The working flow for applying ICP with normals is similar to that of just one ICP. In

fact, pcl::IterativeClosestPointWithNormals, which inherits from pcl::IterativeClosestPoint, uses a estimated transformation based on point-to-plane distances.

In our class regmesh , the regmesh::ICPNormal and regmesh::RegisterNormal functions handle applying ICP with normals. In regmesh::ICPNormal, we create an instance of the PCL class IterativeClosestPointWithNormals and configure correspondence estimation, correspondence rejector, and termination conditions via this instance (identical to what we have done in regmesh::ICP). The maximum correspondence distance is set 0.015 meter (in other words, assuming no correspondences of two frames are beyond 1.5 centimeter apart). Two termination conditions are defined: transformation epsilon (transformation difference = 1e-8) and number of iterations (iteration = 100).

For point-to-plane, normals on both source and target point clouds are used to measure error. Hence, we create two pointers that point to two point clouds with normals (source cloud with normals and target clouds with normals respectively). Normals on source and target clouds are computed via pcl::NormalEstimation and pcl::search::KdTree. The number of neighbors for Kdsearch is set to 12 so that clouds' surfaces do not get overly approximated by points far apart. Once normals are computed, we set the source cloud, now with normals, as the source to align with the target cloud (also with normals). The "align()" function inherited from pcl::Registration estimates the transformation and returns the transformed source. It should be noted that in regmesh::ICP, the transformed source (of type PointCloud as we define) is the direct output of the function. However in regmesh::ICPNormal, the direct output of the align function is the transformed source of type PointCloudWithNormals (since we take normals into account to measure error). To output an equivalent point cloud but without normals, we first obtain the estimated transformation via the command getFinalTransformation, which corresponds to the affine transformation that map source cloud with normals to target cloud with normals. We then apply this transformation on the original source cloud (prior to addition of normals) and return the result. Below is a portion of the code snippet from the regmesh::ICPNormal.

```
1 ICPNormal(source cloud, target cloud, parameters){
2     ...
3     pcl::IterativeClosestPointWithNormal<PointNormalT, PointNormalT> icpN;
4     icpN.setInputSource (points_with_normals_src);
5     icpN.setInputTarget (points_with_normals_tgt);
6     icpN.align (*transformed_src_with_normals); //
```

```
7
8       Eigen::Matrix4f transform_normals = reg.getFinalTransformation ();
9       pcl::transformPointCloud (*src, *transformed_src, transform_normals);
10
11      return transformed_src;
12  }
```

Listing 4.11: ICP with normals implementation

Similar to regmesh::Register paired to work with a single ICP, the above function deals with aligning any given two frames. To incrementally align a sequence of frames, the regmesh::RegisterNormal function is created to handle a vector of point cloud pointers (point to frames that have undergone preprocessing) and iteratively downsamples and updates the source and target cloud to regmesh::ICPNormal. In addition, actual stitching of the transformed source and the target is performed here (the stitched result becomes the new source cloud for the next iteration). Some registered results are shown as follows.

Figure 4.9: Registration using ICP with Normals (front)

Figure 4.10: Registration using ICP with Normals (bottom)

4.3.7 Noise Removal and Smoothing

As shown from previous sections, the result of the registration is noisy on the surface ("double" or "multi" layer). To remove sparse points from the main cluster, we again apply statistical outlier removal. In every regmesh::Register (single ICP, two ICPs, and ICP with normals), cfilter::OutlierRemoval is called after all frames have been stitched.

To address smoothing the surface, we apply moving least squares algorithm for surface reconstruction. The function cfilter::smoothing takes the registered point cloud (after noise removal from the above step) as the input, and returns a smoothed version of that point cloud. In order to balance surface smoothing and detail preservation (e.g., facial details), we set kdradius small to 0.02. Below shows the result before and after smoothing is performed.

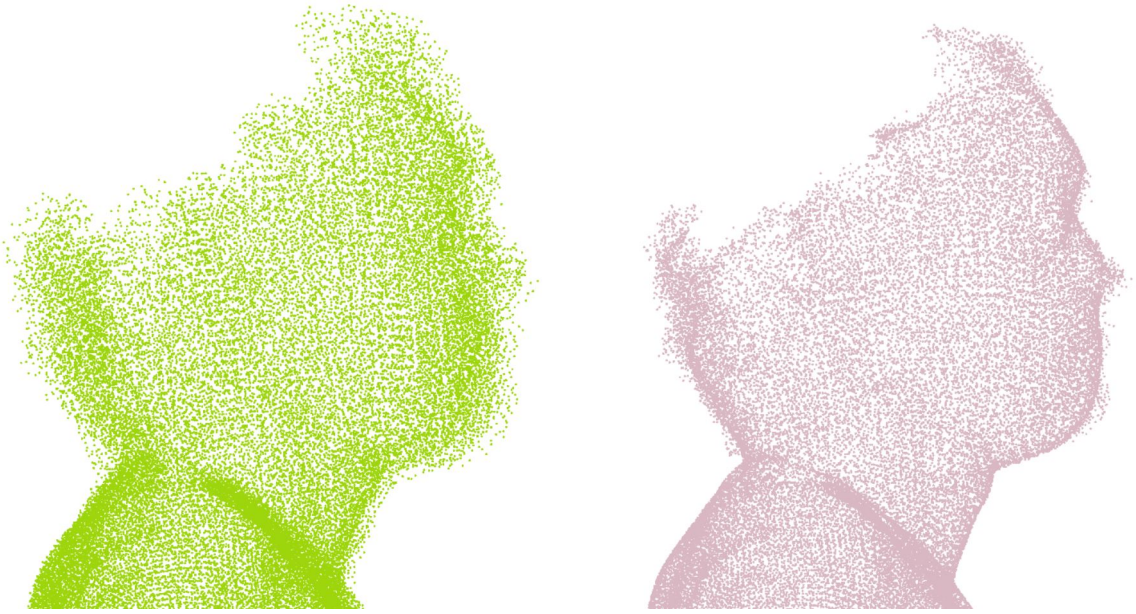


Figure 4.11: Unsmoothed (left) and Smoothed (right) result

Figure 4.12: Unsmoothed (left) and Smoothed (right) result

Figure 4.13: Unsmoothed (left) and Smoothed (right) result

Chapter 5

Meshing

This chapter will introduce the concepts regarding 3D reconstruction which will lead to the process of implementation. Firstly, an introduction about Surface Reconstruction using Point Cloud Data. Finally, a short discussion of the criteria leading to a successful reconstruction in the implementation will be described.

5.1 Point Cloud Data

A point cloud represents a collection of various points set in different dimensions which are commonly used to represent 3 dimensional data. In point clouds the points refer to the X, Y and Z geometric coordinates of a surface. Here, for the project purpose, Point clouds are acquired by using Kinect v2 device. It is a *Time of flight* laser scanner which measure the elapsed time between a light emission and sensor detection. The data acquired from Kinect v2 can be used to recreate certain objects digitally, represent shapes or detect particular surfaces and their topology. With the point data it is possible to create a mesh of the surface.

There are four main steps to be considered when using the point cloud data to reconstruct a surface:

- Once the point cloud data has been gathered it is important to eliminate irrelevant data and sample points to reduce the computational time.
- Finding the relations between the data points their connectedness, continuity and boundaries.
- Start the process of generating the surface using polygons and create the meshes.

- Edit the model created to perfect the polygonal surface resulted. [10]

Triangulation or mesh generation is the most important part in the reconstruction of the model from the data. Methods that aim to recreate detailed object from the point data use as output either a discrete surface or an implicit function. Implicit functions are based on an underlying grid while the reconstructed surface will be achieved via isocontouring. Other methods use grids such as octrees or adaptive 3D triangulations for the reconstruction; however contouring octrees can bring up difficulties in creating models with completely closed meshes. [8]

Many methods of surface reconstructions require normal associated with the point clouds, the normal can either be oriented or unoriented. Normals that do not have a direction are called unoriented normal, which means it is to be expected the normal to be pointing either on the inside or outside of the surface. This sort of information is useful to determine planar regions in a point cloud, projecting a point onto an approximated surface or achieving covariance matrix. An oriented normal on the other hand have consistent directions and it is known which point outside or inside of the surface. [8]

5.1.1 Greedy Projection Triangulation

The focus of Greedy Triangulation (GT) is to keep a list of possible points that can be connected to create a mesh. Greedy triangulation is obtained by adding short compatible edges between points of the cloud, where these edges cannot cross previously formed edges between other points.

GT can, however, over-represent a mesh with triangles when these are represented by planar points. GT contains accurate and computationally efficient triangulations when the points are planar. However, a big difference can be noticed when the triangulations are non-planar, this can be fixed by increasing the number of boundary vertices or linearly interpolating between the boundary vertices. [17]

An approach at computing the GT is to compute all distances, sort them and examine each pair of points in length and compatibility with edges created. The Greedy Triangulation algorithm is simple at its core but also not very reliable. With a point cloud, S, of n points, the algorithm looks for the closest point where a compatible edge can be created between the two. A compatible edge can be described as an edge between two points that does not intersect with any other edge.[11] [9] [15]

In the Point Cloud Library, greedy projection triangulation works locally, however it allows the specification of the required features to be focused on. Such parameters are neighbourhood size of the search, the search radius and the angle between surfaces. [13]

5.1.2 Grid Projection

A different approach to the 3D reconstruction is to use Grid Projection (GP) [16]. GP is an extremal surface reconstruction algorithm that tries to create seamless surfaces that lack boundaries or proper orientation. One of the defining points for GP is that a continuous surface is created, as opposed to GT where the surface is not continuous and can contain holes.

5.1.3 Poisson Surface Reconstruction

There are a multitude of surface reconstruction techniques, however each one of them poses a number of difficulties when applied to data points. Some schemes can be classified in global and local fitting methods. Global fitting methods are represented in implicit form and can be represented as the sum of radial basis functions (RBFs) at the centre of the points while local fitting methods use the distance from tangent planes to the nearest points. Poisson surface reconstruction combines both the local and global methods; however the RBFs functions are associated with the ambient space and not the data points. The Poisson reconstruction uses the Poisson equation, which is also known to be used in systems that perform tone mapping, fluid mechanics and mesh editing. The Poisson reconstruction approached by Michael Kazhdan, Matthew Bolitho and Hugues Hoppe has set as goal to reconstruct a watertight model by using the indicator function and extracting the isosurface. The indicator function is a constant function whose computation would result in a vector with unbounded values. The challenge these three researchers were faced with was to accurately compute the indicator function form the sampled data points. They approximated the surface integral of the normal field using a summation of the point samples and lastly reconstructed the indicator function using its gradient field as a Poisson problem.

One limitation of the Poisson reconstruction is dependent on the acquisition of the data points. The time taken to perform the Poisson reconstruction increases by adding octree depth while the number of output triangles increases by a factor of four. The

Poisson reconstruction uses the indicator function that works best with noisy data points and can therefore recover fine details. [14]

5.1.4 Success Criteria

A criteria for the success of the various reconstruction algorithms is that they are fast and reliable. For that purpose, the computation time is recorded for each algorithm with various settings and each mesh is manually looked at for checking how reliable and accurate the mesh is. For computation time, a simple C++ code snippet is added around each algorithm.

For the reliability and accuracy of the mesh, three different topics are looked at; holes, amount of details and noise.

- Holes: how many holes are there and how do they affect the finished mesh?
- Details: is the mesh too smooth or are there enough details to see the smaller changes in mesh?
- Noise: how much noise is present in the final mesh? Noise is defined as polygons that manually need to be removed in another software package.

These criteria combine both subjective measurements with objective observations, which in the end should have a proper conclusion on what algorithms to use on what types of meshes.

Based on initial theory the following is proposed:

- Greedy Projection Triangulation works best for smaller meshes with few details and closed point cloud devoid of noise.
- Grid Projection works best for larger meshes that are somewhat at and without too much noise.
- Poisson works best for enclosed meshes that do not contain any edges or larger holes.

The three above mentioned methods were chosen because they are part of the Point Cloud Library and allows the reconstruction of the point clouds gathered.

A final point to keep in mind when using a reconstruction technique is the time it takes to record images, compute the point cloud and reconstruct the mesh should not take significantly longer than creating the mesh in a 3D software.

5.2 Implementation

Before going into the implementation of greedy triangulation, grid projection and Poisson [2], it is necessary to look at the input data that is provided to these algorithms and how it is processed.

The basis for 3D reconstruction is a proper registered point cloud. With the .pcd file ready to use, it is possible to start on the filtering and smoothing of the point cloud. Before showing how this is done, a few of the important variables used will be explained. Code 4.1 shows the initial setup of variables.

```
1 pcl::PointCloud<pcl::PointXYZ>::Ptr cloud (new pcl::PointCloud<pcl::
    PointXYZ>);
2 pcl::PointCloud<pcl::PointNormal>::Ptr cloud_with_normals (new pcl::
    PointCloud<pcl::PointNormal>);
3 pcl::search::KdTree<pcl::PointXYZ>::Ptr tree (new pcl::search::KdTree<pcl::
    PointXYZ>);
4  pcl::search::KdTree<pcl::PointNormal>::Ptr tree2 (new pcl::search::KdTree<
    pcl::PointNormal>);
5  int main (int argc, char** argv)
6 {
7 pcl::PointCloud<pcl::PointXYZ>::Ptr cloud (new pcl::PointCloud<pcl::
    PointXYZ>);
8 pcl::PCLPointCloud2 cloud_blob;
9 pcl::io::loadPCDFile ("bun0.pcd", cloud_blob);
10 pcl::fromPCLPointCloud2 (cloud_blob, *cloud);
11 }
```

Listing 5.1: Setup of the most used variables

Code 4.1 shows the initial setup of the variables. Line 1 and 2 defines the two clouds that are used. **cloud** stores the x, y and z information in a pointer and **cloud_with_normals** stores the information of the normals for each point. Line 3 and 4 represent two kdimensional trees which are used to organize the cloud points in k dimensions. These four variables will be used throughout the whole program.

Next thing is to create a search three and perform a normal estimation.

```
1 pcl::NormalEstimation<pcl::PointXYZ, pcl::Normal> n;
```

```
2 pcl::PointCloud<pcl::Normal>::Ptr normals (new pcl::PointCloud<pcl::Normal
    >);
3 pcl::search::KdTree<pcl::PointXYZ>::Ptr tree (new pcl::search::KdTree<pcl::
    PointXYZ>);
4 tree->setInputCloud (cloud);
5 n.setInputCloud (cloud);
6 n.setSearchMethod (tree);
7 n.setKSearch (20);
8 n.compute (*normals);
9   pcl::concatenateFields (*cloud, *normals, *cloud_with_normals);
```

Listing 5.2: Setup search tree and normal estimation of variables

The search tree is based on the previously created **KdTree** and takes the **cloud** as input. The normal estimation is stored in a new variable called **normals**. These two variables, the sorted cloud and the normals, need to be stored together in a new variable. The **cloud** only contains information about x, y and z while the **normals** only contains information of the normals. In order to use the cloud properly for the reconstruction, a cloud with both x, y and z values and normal information is needed. Line 9 in Code 4.2 shows how it is possible to take the **cloud** data and the **normals** data and combine it and store it in **cloud_with_normals**. The cloud is now ready for use and the next section will go into details with the meshing process.

5.2.1 Reconstruction

The Reconstruction section will introduce; Greedy Triangulation, Grid Projection and Poisson. These three reconstruction methods will be used to reconstruct the point clouds acquired.

In greedy triangulation and Poisson a new variable called **triangles** is used. It is initialized as seen in Code 4.3

```
1 boost::shared_ptr<pcl::PolygonMesh> triangles (new pcl:: PolygonMesh);
```

Listing 5.3: Initialization of **triangles** variable

The **PolygonMesh** data type contains information about vertex position, edge data and face data. This data is required for **PCL** to save the new reconstructed mesh as a file that can be read by another 3D software.

5.2.1.1 Greedy Triangulation

As mentioned, the idea behind Greedy Triangulation (GT) is straightforward. To accomplish GT using the PCL a new variable **gt** is created which can be seen in Code 4.4.

```
1  pcl::GreedyProjectionTriangulation<pcl::PointNormal> gp3;
2  pcl::PolygonMesh triangles;
3
4  gp3.setSearchRadius (R);
5  gp3.setMu (mu);
6  gp3.setMaximumNearestNeighbors (D);
7  gp3.setMaximumSurfaceAngle(M_PI/4);
8  gp3.setMinimumAngle(M_PI/18);
9  gp3.setMaximumAngle(2*M_PI/3);
10 gp3.setNormalConsistency(false);
11 gp3.setInputCloud (cloud_with_normals);
12 gp3.setSearchMethod (tree2);
13 gp3.reconstruct (triangles);
```

Listing 5.4: Greedy Triangulation implementation

Triangulation is performed locally, by projecting the local neighborhood of a point along the point's normal, and connecting unconnected points. Thus, the following parameters can be set:

- *setMaximumNearestNeighbors(D)* and *setMu(mu)* control the size of the neighborhood. The former defines how many neighbors are searched for, while the latter specifies the maximum acceptable distance for a point to be considered, relative to the distance of the nearest point (in order to adjust to changing densities). Typical values are 50-100 and 2.5-3 (or 1.5 for grids).

- *setSearchRadius(R)* is practically the maximum edge length for every triangle. This has to be set by the user such that to allow for the biggest triangles that should be possible.

- *setMinimumAngle(double)* and *setMaximumAngle(double)* are the minimum and maximum angles in each triangle. While the first is not guaranteed, the second is. Typical values are 10 and 120 degrees (in radians).

(a) Front View

(b) Back View

Figure 5.1: Greedy Triangulation

- *setMaximumSurfaceAgle(double)* and *setNormalConsistency(bool)* are meant to deal with the cases where there are sharp edges or corners and where two sides of a surface run very close to each other. To achieve this, points are not connected to the current point if their normals deviate more than the specified angle (note that most surface normal estimation methods produce smooth transitions between normal angles even at sharp edges). This angle is computed as the angle between the lines defined by the normals (disregarding the normal's direction) if the normal-consistency-flag is not set, as not all normal estimation methods can guarantee consistently oriented normals. Typically, 45 degrees (in radians) and false works on most datasets.

Fig. 5.1 shows the greedy triangulation results. Here we keep the value od D, mu and R as 1200, 3 and 0.025 respectively. As we lower the value of D (the maximum number of nearest neighbors which the algorithm should search for) will make the mesh have a bit fewer holes. Lowering it more than 50 will not have any substantial effect because not enough neighbors are considered and the library automatically increases D until it finds enough neighbors. There is only one probleem in Fig. 5.1 is that the normals are flipped for half the triangles. This is supposedly a known problem with PCL.

5.2.1.2 Grid projection

With PCL it is possible to implement Grid Projection using Code 4.5.

```
1 pcl::PolygonMesh grid;
2 pcl::GridProjection<pcl::PointNormal> gp;
3
4 gp.setInputCloud(cloud_with_normals);
5 gp.setSearchMethod(tree2);
6 gp.setResolution(res);
7 gp.setPaddingSize(3);
8 gp.reconstruct(grid);
```

Listing 5.5: Implementation of Grid Projection

The code is very similar to the greedy triangulation code (see Code 4.4) where a polygon mesh and a grid projection variable are created. The g variable is then used to set the input cloud, search method, resolution and padding properties before the reconstruction begins.

(a) Front View

(b) Back View

Figure 5.2: Grid Projection

Fig. 5.2 shows the grid projection results. Here the value of *res* is set to 0.005. If the value of *res* is increased the grid projection could not find enough points and the results are small scattered lumps of mesh. The topography is so deformed that is is impossible to see what is it supposed to resemble. And if the value of *res* is decreased, the mesh becomes a bit more rough and the normals skewed in some place. If this value is further decreased, it gives a lot more clear and smooth mesh. The problem with grid projection is that they are so smooth that a lot of details are lost.

5.2.1.3 Poisson Surface

The implementation of Poisson in this project is done as described in Code 4.6.

```
1 pcl::Poisson<pcl::PointNormal> poisson;
2 poisson.setDepth(D);
3 poisson.setInputCloud(cloud_with_normals);
4 boost::shared_ptr<pcl::PolygonMesh> triangles (new pcl::PolygonMesh);
5 poisson.reconstruct(*triangles);
```

Listing 5.6: Implementation of Poisson

Fig. 5.3 shows reconstruction using Poisson. It should be noted that the figures have their normals inverted and is represented from below (comapared to Fig. 5.1 and Fig. 5.2), because the Poisson algorithm closes all borders to the models which become spherical and with no openings. With a compute time of over 11 minutes is by far the slowest and it still has a lot of mesh leftover that shouldn't be a part of the model.

The biggest problem with Poisson and the reason why it is not usable for this project, is the fact that so much extra mesh is created. The point cloud is pretty clean and without too much noise but the mesh is still not clean and too much manual cleanup is needed for Poisson to be a valid implementation method.

5.2.2 Inferences

Based on the three reconstruction method's accuracy level, time, computational complexity and memory it takes up, it can be assessed that the Greedy Triangulation presents the most positive features for both the small scaled objects and the medium scaled objects, however in the case of the large scale object Grid Projection seems to be most optimal. Therefore, based on the reconstruction method being applied using PCL it can be assessed

(a) Front View

(b) Side View

Figure 5.3: Poisson Surface Reconstruction

that Greedy Triangulation would provide the most optimal mesh when working with small and medium sized objects. Large objects reconstructed using GP would prove to be less accurate and more time consuming however with a lower computational complexity and memory usage, to get an accurate and fast reconstruction GT would be most optimal.

Chapter 6

Graphical User Interface

6.0.1 Software Design

The implemented GUI has a user friendly interface and simple operation. The software design framework is based on the class diagram shown in Figure 6.1. For this approach a diagram network structure is used to show the system built by a combination of different programs or processes. In the diagram, each box represents a class and the member functions. The dashed arrows represent unidirectional dependencies between two classes, meaning that the main class **MyVTKWidget** depends on instances from the other classes.

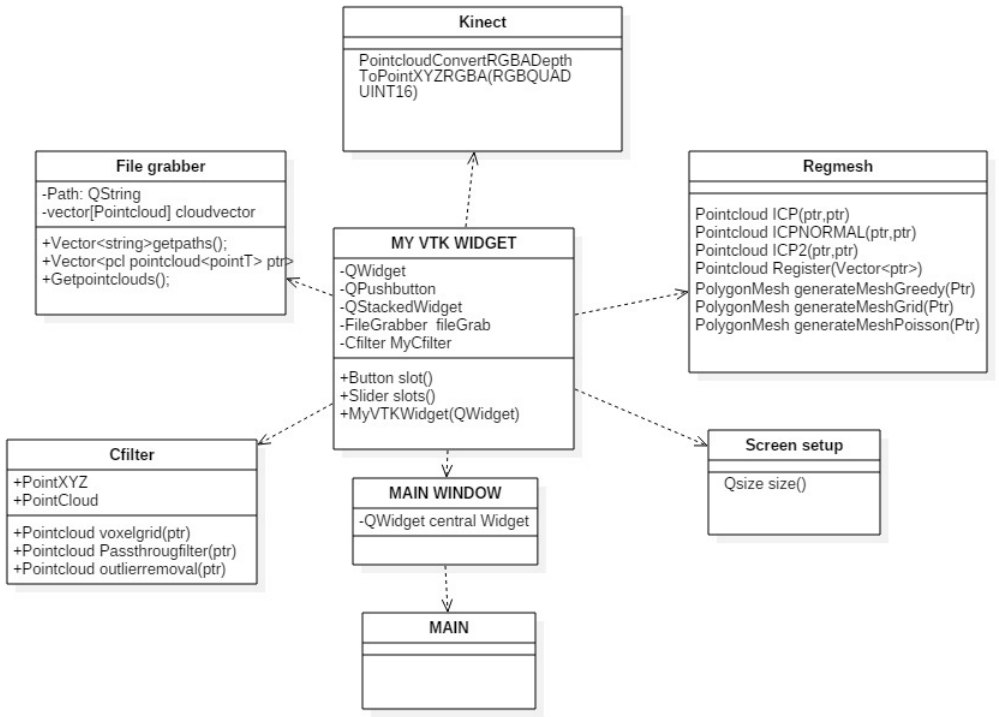


Figure 6.1: Class diagram of the software

As shown in the class diagram the **MainWindow** class contains the central Widget **MyVTKWidget** with several subclasses from where all of the program is executed sequentially. The **MyVTKWidget** class includes the following widgets:

- **QPushButtons** used for executing a command
- **QRadioButtons** used for choosing one of the offered options
- **QSliders** used for adjusting parameters

The **ScreenSetup** class adjusts the size of the display screen according to the dimensions of the user's screen.

The **FileGrabber** class captures depth map of the scene, which is converted to point cloud with the **Kinect** class. They are saved as point cloud (.pcd) files which are used for the subsequent steps for obtaining the 3D model. The **cfilter** class is used to filter each of the captured frames, which afterwards are aligned by using ICP after the filtering procedure is done. The class has three methods: **voxelgrid** for downsampling, **Passthroughfilter** for filtering and **outlierremoval** for removing the outliers from the point clouds.

The **Regmesh** class is used for performing the registration and meshing of the point clouds after the filtering procedure is done. The filtered frames are aligned using different approaches for registration by ICP. The triangulation and obtaining the complete 3D reconstruction of the scanned object are done by three different meshing techniques. The class has six methods for registering and three methods for meshing. Each Register method is used for registering sequence of point clouds. For aligning two point clouds the methods **ICP**, **ICPNORMAL** and **ICP2** are used. The meshing methods **generateMeshGreedy**, **generateMeshGrid** and **generateMeshPoisson** are used for performing greedy projection triangulation, grid projection and poisson reconstruction, respectively.

6.0.2 GUI Design

As it can be seen in Figure 6.2 the home page of the graphical user interface consists of:

- Five push buttons: Read, Load Kinect, Register, Meshing and Save

- PCL Visualizer
- Sidebar on the right showing the captured point clouds

Figure 6.2: Home Page of the GUI

In order to render a PCL Visualizer onto the application window a **QVTKWidget** was used, so the VTK was built and integrated into the Qt. To communicate between different objects when a widget is changed, the signals and slots mechanism was used. To update the Widget when a button is clicked pages were implemented using the **QStackedWidget** class.

To allow the user to choose already saved .pcd files for further processing when the read button is pressed, the **QFileDialog** class was used. The method executed when the read button is pressed is not fully implemented due to crashes that were not solved. The load kinect button, when pressed, first child page is called that updates the window. The page consists of several sliders allowing the user to adjust the parameters of the red cube, a radio button to lock the parameters, and three icon buttons for starting the capturing of frames, stopping and going back to the home page. The red cube is used to set the boundaries for the passthrough filter, everything outside the cube would be not considered. The sliders are connected with signals and slots to update the size and position of the red cube in the PCL Visualizer. Each time a frame is captured it is saved and shown in the right sidebar.

When the register button is clicked, a second child page is called which contains three radio buttons for each ICP method, and two icon buttons; done and go back to the home page. The user can choose one of the techniques and press the done button, which using signal and slots is connected to a method executing the chosen ICP technique.

A third child page is called when the user presses on the meshing button. The updated widget has a similar structure as the second page, consisting of three radio buttons for choosing a meshing technique. When the user selects one of the methods, a command with the chosen meshing technique is executed.

The save button was not completed. The aim of the save button would be for the user to choose a directory using the **QFileDialog** and save the final 3D model.

Chapter 7

Hardware Configuration

The aim is to create a functioning 3D scanner that has a work envelope of at least a cubic foot, while being generally affordable. The 3D scanner will be able to scan items within the work envelope, create a point-cloud of the scanned points, and finally provide an exportable file or data stream representing the scanned object. It will be targeted toward hobbyists and consumers, as its cost will not be extraordinary and its accuracy will be relatively reliable and practical for small- to medium-sized objects.

- Ply Wood
- Twist board (90 kgs max.)
- DC geared Motor (4 rpm)
- Timing Belts
- Arduino board
- Motor Driver
- Power Supply (12V)

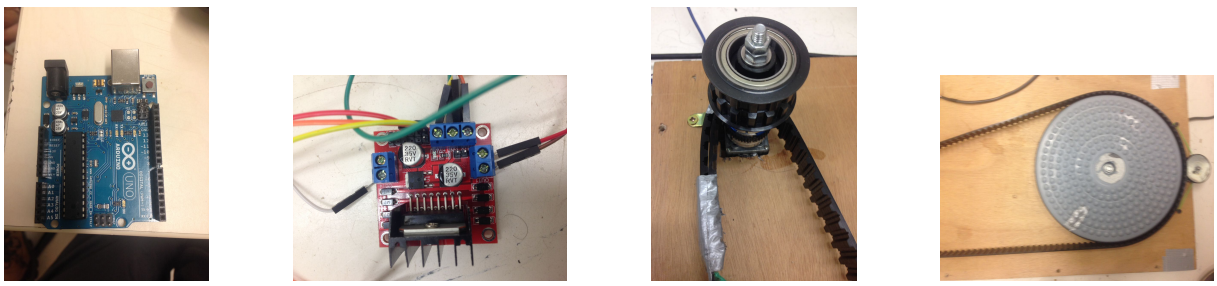


Figure 7.1: Materials for 3D Scanner

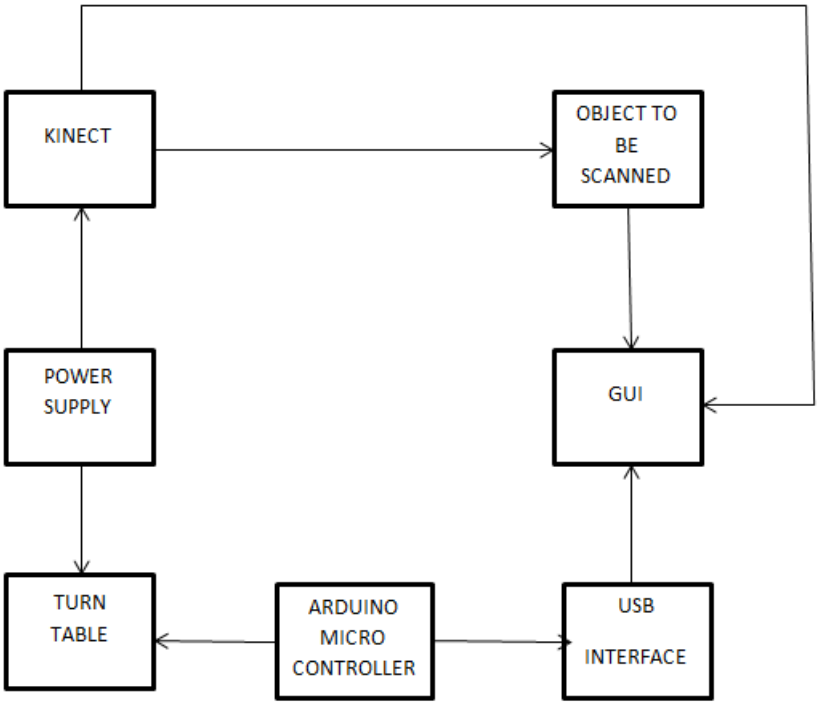


Figure 7.2: 3D Scanner Hardware block diagram

7.1 Interfacing Arduino with Qt

To interface between Qt and Arduino, we need to use specific classes for the application. We need to include the following header file and Qt library.

```
Header: #include <QSerialPort>
qmake:  QT += serialport
```

QtSerialPort is for serial but the actual USB device on the Arduino is a USB-Serial converter so it's basically serial port but through USB. The operating system is usually able to handle these USB2Serial converters out of the box meaning that they will appear as serial ports. After setting the port, you can open it in read-only (r/o), write-only (w/o), or read-write (r/w) mode using the open() method.

QSerialPort tries to determine the current configuration of the port and initializes itself. We can reconfigure the port to the desired setting using the **setBaudRate()**, **setDataBits()**, **setParity()**, **setStopBits()**, and **setFlowControl()** methods.

7.2 Driving motor with Arduino

We use a DC geared motor, the rotating speed can be controlled with the Arduino as well as from Qt GUI. QtSerialPort enables to access Arduino via serial communication..

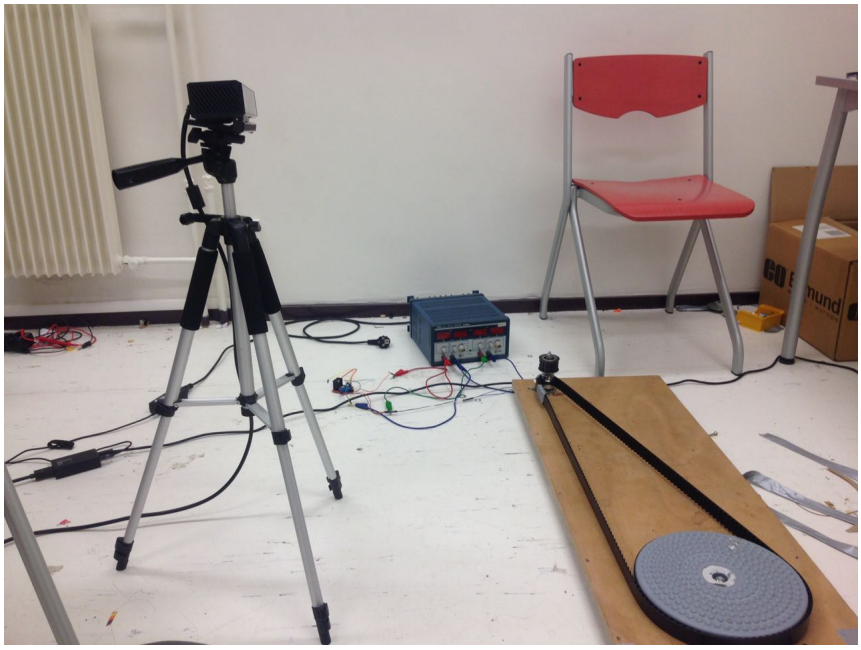


Figure 7.3: 3D Scanner Setup Kinect + Turning Table

7.3 Requirements met and Numerical Analysis

- **Cost**: Cost of the scanning turntable and development was under 100 euros.
- **Time**: The project was completed on schedule.
- **Physical Attributes**: The scanner is reasonably sized and is easy to carry.
- **Power**: The scanner uses a 120VAC to 12VDC wall wart, and draws about 1.25A from the 12VDC supply, when scanning.
- **Autonomy**: The scanner operates autonomously, and when done provides the resulting model file.
- **Ease of Use**: Since the scanner appears as a USB mass storage device, no extra software or drivers are needed for a user to be able to scan an object and get the resulting model file. The scanner is also mechanically sturdy, so the user doesn't need to fiddle with it.

Chapter 8

Conclusion

For this project, we worked for more than 3 months on the design and implementation of 3D human body scanner using C++ with Qt. Throughout the project, we learned many things about depth sensors, 3D modeling, point cloud library, well-implemented coding, time management, and team work.

In this report, we justified our approach for the problem, with all technical details, codes, and results. We presented our use of depth sensors, scanning and registration techniques, meshing algorithms, and finally the handy GUI integrating all theses classes together to produce our final software.

We thank every tough problem that sharpened our programming and problem solving skills. Our confidence grew once we realize all we need is just more hardworking and persistence.

However, our output is still not perfect yet. We can recommend many modifications to be done as future work. For example, the reconstruction of colored registered point clouds using texture mapping would be a great addition to the project. In addition, we can increase the robustness of the registration process by applying loop closure, which will increase the need for good scanning conditioned.

Chapter 9

Appendix A

9.1 Gantt Chart

See next page.

Figure 9.1: Gantt Chart

Chapter 10

Appendix B

10.1 QtCreator Project file Settings

In the *.pro file include the paths and libraries and per your PC directory:

```
1
2 QT += core gui opengl
3 greaterThan(QT_MAJOR_VERSION, 4): QT += widgets
4 TARGET = kinect2grabbersample
5 TEMPLATE = app
6
7 INCLUDEPATH += "C:\Program Files\PCL 1.8.0\include\pcl-1.8
8 INCLUDEPATH += "C:\Program Files\PCL 1.8.0\3rdParty
9 \VTK\include\vtk-7.0"
10 INCLUDEPATH += "C:\Program Files\PCL 1.8.0\3rdParty
11 \Boost\include\boost-1_61"
12 INCLUDEPATH += "C:\Program Files\PCL 1.8.0
13 \3rdParty\Qhull\include"
14 INCLUDEPATH += "C:\Program Files\PCL 1.8.0
15 \3rdParty\FLANN\include"
16 INCLUDEPATH += C:\Program Files\PCL 1.8.0
17 \3rdParty\Eigen\eigen3"
18 INCLUDEPATH += "C:\Program Files\Microsoft SDKs
19 \Kinect\v2.0_1409\inc"
20 INCLUDEPATH += "C:\Program Files\OpenNI2\Include"
21
22 LIBS += opengl32.lib advapi32.lib Ws2_32.lib
23 LIBS += user32.lib shell32.lib gdi32.lib kernel32.lib\par
24 LIBS += "-LC:\Program Files\PCL 1.8.0\lib"
25 LIBS += "-LC:\Program Files\PCL 1.8.0\3rdParty\VTK\lib"
```

```
LIBS += "-LC:\Program Files\PCL 1.8.0\3rdParty\Qhull\lib"
LIBS += "-LC:\Program Files\PCL 1.8.0\3rdParty\FLANN\lib"
LIBS += \path{"-LC:\Program Files\PCL 1.8.0
\3rdParty\Boost\lib"
LIBS += "-LC:\Program Files\Microsoft SDKs
\Kinect\v2.0_1409\Lib\x64"
LIBS += "-LC:\Program Files\OpenNI2\Lib"
LIBS += -lpcl_common_release
LIBS += -lpcl_features_release
LIBS += -lpcl_filters_release
LIBS += -lpcl_io_ply_release
LIBS += -lpcl_io_release
LIBS += -lpcl_kdtree_release
LIBS += -lpcl_keypoints_release
LIBS += -lpcl_ml_release
LIBS += -lpcl_octree_release
LIBS += -lpcl_outofcore_release
LIBS += -lpcl_people_release
LIBS += -lpcl_recognition_release
LIBS += -lpcl_registration_release
LIBS += -lpcl_sample_consensus_release
LIBS += -lpcl_search_release
LIBS += -lpcl_segmentation_release
LIBS += -lpcl_stereo_release
LIBS += -lpcl_surface_release
LIBS += -lpcl_tracking_release
LIBS += -lpcl_visualization_release
LIBS += -llibboost_atomic-vc140-mt-1_61
LIBS += -llibboost_chrono-vc140-mt-1_61
LIBS += -llibboost_container-vc140-mt-1_61
LIBS += -llibboost_context-vc140-mt-1_61
LIBS += -llibboost_coroutine-vc140-mt-1_61
LIBS += -llibboost_date_time-vc140-mt-1_61
LIBS += -llibboost_exception-vc140-mt-1_61
LIBS += -llibboost_filesystem-vc140-mt-1_61
LIBS += -llibboost_graph-vc140-mt-1_61
LIBS += -llibboost_iostreams-vc140-mt-1_61
LIBS += -llibboost_locale-vc140-mt-1_61
LIBS += -llibboost_log-vc140-mt-1_61
```

```
65 LIBS += -llibboost_log_setup-vc140-mt-1_61
66 LIBS += -llibboost_math_c99-vc140-mt-1_61
67 LIBS += -llibboost_math_c99f-vc140-mt-1_61
68 LIBS += -llibboost_math_c99l-vc140-mt-1_61
69 LIBS += -llibboost_math_tr1-vc140-mt-1_61
70 LIBS += -llibboost_math_tr1f-vc140-mt-1_61
71 LIBS += -llibboost_math_tr1l-vc140-mt-1_61
72 LIBS += -llibboost_mpi-vc140-mt-1_61
73 LIBS += -llibboost_prg_exec_monitor-vc140-mt-1_61
74 LIBS += -llibboost_program_options-vc140-mt-1_61
75 LIBS += -llibboost_random-vc140-mt-1_61
76 LIBS += -llibboost_regex-vc140-mt-1_61
77 LIBS += -llibboost_serialization-vc140-mt-1_61
78 LIBS += -llibboost_signals-vc140-mt-1_61
79 LIBS += -llibboost_system-vc140-mt-1_61
80 LIBS += -llibboost_test_exec_monitor-vc140-mt-1_61
81 LIBS += -llibboost_thread-vc140-mt-1_61
82 LIBS += -llibboost_timer-vc140-mt-1_61
83 LIBS += -llibboost_type_erasure-vc140-mt-1_61
84 LIBS += -llibboost_unit_test_framework-vc140-mt-1_61
85 LIBS += -llibboost_wave-vc140-mt-1_61
86 LIBS += -llibboost_wserialization-vc140-mt-1_61
87 LIBS += -lflann_cpp_s
88 LIBS += -lqhullstatic
89 LIBS += -lvtkalglib-7.0
90 LIBS += -lvtkChartsCore-7.0
91 LIBS += -lvtkCommonColor-7.0
92 LIBS += -lvtkCommonComputationalGeometry-7.0
93 LIBS += -lvtkCommonCore-7.0
94 LIBS += -lvtkCommonDataModel-7.0
95 LIBS += -lvtkCommonExecutionModel-7.0
96 LIBS += -lvtkCommonMath-7.0
97 LIBS += -lvtkCommonMisc-7.0
98 LIBS += -lvtkCommonSystem-7.0
99 LIBS += -lvtkCommonTransforms-7.0
100 LIBS += -lvtkDICOMParser-7.0
101 LIBS += -lvtkDomainsChemistry-7.0
102 LIBS += -lvtkexoIIc-7.0
103 LIBS += -lvtkexpat-7.0
```

```
104 LIBS += -lvtkFiltersAMR-7.0
105 LIBS += -lvtkFiltersCore-7.0
106 LIBS += -lvtkFiltersExtraction-7.0
107 LIBS += -lvtkFiltersFlowPaths-7.0
108 LIBS += -lvtkFiltersGeneral-7.0
109 LIBS += -lvtkFiltersGeneric-7.0
110 LIBS += -lvtkFiltersGeometry-7.0
111 LIBS += -lvtkFiltersHybrid-7.0
112 LIBS += -lvtkFiltersHyperTree-7.0
113 LIBS += -lvtkFiltersImaging-7.0
114 LIBS += -lvtkFiltersModeling-7.0
115 LIBS += -lvtkFiltersParallel-7.0
116 LIBS += -lvtkFiltersParallelImaging-7.0
117 LIBS += -lvtkFiltersProgrammable-7.0
118 LIBS += -lvtkFiltersSelection-7.0
119 LIBS += -lvtkFiltersSMP-7.0
120 LIBS += -lvtkFiltersSources-7.0
121 LIBS += -lvtkFiltersStatistics-7.0
122 LIBS += -lvtkFiltersTexture-7.0
123 LIBS += -lvtkFiltersVerdict-7.0
124 LIBS += -lvtkfreetype-7.0
125 LIBS += -lvtkGeovisCore-7.0
126 LIBS += -lvtkgl2ps-7.0
127 LIBS += -lvtkhdf5-7.0
128 LIBS += -lvtkhdf5_hl-7.0
129 LIBS += -lvtkImagingColor-7.0
130 LIBS += -lvtkImagingCore-7.0
131 LIBS += -lvtkImagingFourier-7.0
132 LIBS += -lvtkImagingGeneral-7.0
133 LIBS += -lvtkImagingHybrid-7.0
134 LIBS += -lvtkImagingMath-7.0
135 LIBS += -lvtkImagingMorphological-7.0
136 LIBS += -lvtkImagingSources-7.0
137 LIBS += -lvtkImagingStatistics-7.0
138 LIBS += -lvtkImagingStencil-7.0
139 LIBS += -lvtkInfovisCore-7.0
140 LIBS += -lvtkInfovisLayout-7.0
141 LIBS += -lvtkInteractionImage-7.0
142 LIBS += -lvtkInteractionStyle-7.0
```

```
143 LIBS += -lvtkInteractionWidgets-7.0
144 LIBS += -lvtkIOAMR-7.0
145 LIBS += -lvtkIOCore-7.0
146 LIBS += -lvtkIOEnSight-7.0
147 LIBS += -lvtkIOExodus-7.0
148 LIBS += -lvtkIOExport-7.0
149 LIBS += -lvtkIOGeometry-7.0
150 LIBS += -lvtkIOImage-7.0
151 LIBS += -lvtkIOImport-7.0
152 LIBS += -lvtkIOInfovis-7.0
153 LIBS += -lvtkIOLegacy-7.0
154 LIBS += -lvtkIOLSDyna-7.0
155 LIBS += -lvtkIOMINC-7.0
156 LIBS += -lvtkIOMovie-7.0
157 LIBS += -lvtkIONetCDF-7.0
158 LIBS += -lvtkIOParallel-7.0
159 LIBS += -lvtkIOParallelXML-7.0
160 LIBS += -lvtkIOPLY-7.0
161 LIBS += -lvtkIOSQL-7.0
162 LIBS += -lvtkIOVideo-7.0
163 LIBS += -lvtkIOXML-7.0
164 LIBS += -lvtkIOXMLParser-7.0
165 LIBS += -lvtkjpeg-7.0
166 LIBS += -lvtkjsoncpp-7.0
167 LIBS += -lvtklibxml2-7.0
168 LIBS += -lvtkmetaio-7.0
169 LIBS += -lvtkNetCDF-7.0
170 LIBS += -lvtkNetCDF_cxx-7.0
171 LIBS += -lvtkoggtheora-7.0
172 LIBS += -lvtkParallelCore-7.0
173 LIBS += -lvtkpng-7.0
174 LIBS += -lvtkproj4-7.0
175 LIBS += -lvtkRenderingAnnotation-7.0
176 LIBS += -lvtkRenderingContext2D-7.0
177 LIBS += -lvtkRenderingContextOpenGL-7.0
178 LIBS += -lvtkRenderingCore-7.0
179 LIBS += -lvtkRenderingFreeType-7.0
180 LIBS += -lvtkRenderingGL2PS-7.0
181 LIBS += -lvtkRenderingImage-7.0
```

```
182 LIBS += -lvtkRenderingLabel-7.0
183 LIBS += -lvtkRenderingLIC-7.0
184 LIBS += -lvtkRenderingLOD-7.0
185 LIBS += -lvtkRenderingOpenGL-7.0
186 LIBS += -lvtkRenderingVolume-7.0
187 LIBS += -lvtkRenderingVolumeOpenGL-7.0
188 LIBS += -lvtksqlite-7.0
189 LIBS += -lvtksys-7.0
190 LIBS += -lvtktiff-7.0
191 LIBS += -lvtkverdict-7.0
192 LIBS += -lvtkViewsContext2D-7.0
193 LIBS += -lvtkViewsCore-7.0
194 LIBS += -lvtkViewsInfovis-7.0
195 LIBS += -lvtkzlib-7.0
196 LIBS += -lOpenNI2
197 LIBS += -lkinect20
```

Bibliography

[1] http://docs.pointclouds.org/trunk/group__filters.html.

[2] http://pointclouds.org.

[3] http://pointclouds.org/documentation/tutorials/passthrough.php#passthrough.

[4] http://pointclouds.org/documentation/tutorials/resampling.php#movingleast-squares.

[5] http://pointclouds.org/documentation/tutorials/statistical_outlier.php#statistical-outlier-removal.

[6] http://pointclouds.org/documentation/tutorials/voxel_grid.php#voxelgrid.

[7] http://robotica.unileon.es/index.php/pcl/openni_tutorial_2:_cloud_processing_(basic).

[8] M. Berger, A. Tagliasacchi, L. Seversky, P. Alliez, J. Levine, A. Sharf, and C. Silva. State of the art in surface reconstruction from point clouds. In *EUROGRAPHICS star reports*, volume 1, pages 161–185, 2014.

[9] M. T. Dickerson, R. L. Scot Drysdale, S. A. McElfresh, and E. Welzl. Fast greedy triangulation algorithms. In *Proceedings of the tenth annual symposium on Computational geometry*, pages 211–220. ACM, 1994.

[10] R. Fabio et al. From point cloud to surface: the modeling and visualization problem. *International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences*, 34(5):W10, 2003.

[11] S. A. Goldman. A space efficient greedy triangulation algorithm. *Information Processing Letters*, 31(4):191–196, 1989.

[12] D. Holz, A. E. Ichim, F. Tombari, R. B. Rusu, and S. Behnke. Registration with the point cloud library: A modular framework for aligning in 3-d. *IEEE Robotics & Automation Magazine*, 22(4):110–124, 2015.

[13] J. Hyvärinen et al. Surface reconstruction of point clouds captured with microsoft kinect. 2012.

[14] M. Kazhdan and H. Hoppe. Screened poisson surface reconstruction. *ACM Transactions on Graphics (TOG)*, 32(3):29, 2013.

[15] C. Levcopoulos and A. Lingas. Fast algorithms for greedy triangulation. *BIT Numerical Mathematics*, 32(2):280–296, 1992.

[16] R. Li, L. Liu, L. Phan, S. Abeysinghe, C. Grimm, and T. Ju. Polygonizing extremal surfaces with manifold guarantees. In *Proceedings of the 14th ACM Symposium on Solid and Physical Modeling*, pages 189–194. ACM, 2010.

[17] L. Ma, T. Whelan, E. Bondarev, P. H. de With, and J. McDonald. Planar simplification and texturing of dense point cloud maps. In *Mobile Robots (ECMR), 2013 European Conference on*, pages 164–171. IEEE, 2013.

[18] Y. Ma, Y. Guo, J. Zhao, M. Lu, J. Zhang, and J. Wan. Fast and accurate registration of structured point clouds with small overlaps.