Marte Haugestøyl Vinje

Evaluating Apple's integrated LiDAR devices for Indoor 3D Modelling

Master's thesis in Ingeniørvitenskap & IKT
Supervisor: Hongchao Fan
December 2021

Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Engineering
Department of Civil and Environmental Engineering

Norwegian University of
Science and Technology

Marte Haugestøyl Vinje

Evaluating Apple's integrated LiDAR devices for Indoor 3D Modelling

Master’s thesis in Ingeniørvitenskap & IKT
Supervisor: Hongchao Fan
December 2021

Norwegian University of Science and Technology
Faculty of Engineering
Department of Civil and Environmental Engineering

Norwegian University of
Science and Technology

Contents

1 Introduction 4

2 Background 5

3 Method 7
3.1 Experimental setup . . . 7
3.2 Getting the data . . . 9
3.2.1 Accessing data from Apple's ARKit . . . 10
3.2.2 Storing the data . . . 10
3.3 Processing the data . . . 13
3.3.1 Filtering by confidence . . . 13
3.3.2 Storing RGBD data as point clouds . . . 15
3.3.3 Matching process: Pairwise Registration . . . 16
3.3.4 Posegraph optimization . . . 17
3.3.5 Transformations and final result . . . 18
3.4 Evaluating the results . . . 19
3.4.1 Filtering/cleanup/postprocessing . . . 19
3.4.2 Alignment . . . 20
3.4.3 Distance measurements . . . 21
3.4.4 Segmentation . . . 23

4 Results 24
4.1 Scan 1: Trimble SX10 . . . 24
4.2 Scan 2: iPhone 12 Pro with own source code . . . 24
4.3 Scan A: iPhone 12 Pro with SiteScape . . . 26
4.4 Scan B: iPhone 12 Pro with Pix4DCatch . . . 27

5 Evaluation 29
5.1 Scan 1 vs. Scan 2 . . . 29
5.1.1 Full scan, no filtering . . . 29
5.1.2 Full scan, with filtering . . . 31
5.1.3 Segmented: Door . . . 33
5.1.4 Segmented: Roof . . . 34
5.1.5 Segmented: Cluttered corner . . . 36
5.2 Scan 1 vs. Scan A . . . 39

6 Discussion 42
6.1 Data . . . 42
6.1.1 Error in input RGBD-pairs, "overflow" . . . 42
6.1.2 Blurry images and laggy scan . . . 42
6.1.3 Low resolution . . . 43
6.2 Processing . . . 43
6.2.1 Simple matching structure . . . 43
6.2.2 Online vs. offline processing . . . 43
6.2.3 Reproduction problems: RANSAC . . . 43

6.2.4 Runtime 44
6.3 Evaluation 44
6.3.1 Incomplete "ground truth" 44
6.3.2 Chosen commercial applications 44
6.4 Further work 45

7 Conclusion 46

8 References 47

Glossary

RGB Term for images with the normal three values for color (red, green, blue). 15

RGBD Term for images with the normal three values for color (red, green, blue), and an additional fourth value for depth. 4, 5, 8, 13, 15, 26, 27, 33, 38, 42, 45, 46

Acronyms

AR Augmented Reality. 4, 5

C2C CloudCompare. 19–24, 29, 31–33, 36, 39, 44, 45

LiDAR Light Detection and Ranging. 4, 5, 7, 9, 10, 29, 35, 37, 38, 42, 43, 46

TLS Terrestrial Laser Scanner. 7, 8, 21, 24, 25, 36, 38–40

VR Virtual Reality. 4, 5

1 Introduction

3D scanning and modelling is the art of reconstructing real life objects and scenes into a digital 3D model. 3D modelling has been an integral part of land surveying and construction work to reconstruct land/buildings correctly. 3D modelling is also relevant in game designing and immersive application experiences with Augmented Reality (AR)/Virtual Reality (VR), and 3D sensors are used in automotive vehicles for navigation and sensing.

In recent years, a combination of laser (Light Detection and Ranging (LiDAR), depth-measurements) and photogrammetry (color-images) has shown to be a cheaper, as well as faster solution, while still providing robust and accurate 3D-models, especially in short-range areas such as indoor environments, [1]. The motivation for this work, comes from a literature review of automatic 3D modelling Indoors in 2020, [2], that showed this trend of combining LiDAR and camera. This combination is what is often called "RGBD-cameras"/depth-cameras. During the same year, Apple launched the first mobile phone with integrated LiDAR, iPhone 12 Pro. If the 3D models that are possible to create using these new mobile devices performs well, it will open up new possibilities to present 3D modelling for a wide non-expert audience.

The aim of this master thesis is to look at the opportunities for automated indoor modeling using the low-cost sensor Apple's iPhone 12 Pro, which is the first mobile phone with an integrated LiDAR sensor. The LiDAR sensor provides depth information for pixels in a color image, which can help aid 3D modeling. The range of the mobile phone is between 1-5 meters, which makes it only relevant for close object scans or indoor environments. This article presents the results of a scanning experiment executed at NTNU. The main contribution is a data processing program for processing Apple's LiDAR data. The program is written in Python 3.7, using the 3D processing library Open3D. Two additional applications has also been used to execute separate scans for comparison, SiteScape and Pix4D Catch + Cloud. The results will be evaluated against "ground truth" - a scan executed by a Trimble SX10 scanning device.

An extended goal of this thesis is to get the data format approved as a new type of sensor for the open source library used, Open3D. This way, this thesis could provide a starting point where the data from Apple's LiDAR sensor is accessible to experiment further.

2 Background

The 3D scanning and modelling field consist of two main methods to reconstruct models from reality, namely *Photogrammetry* and *LiDAR*. Photogrammetry is known as the 3D reconstruction from images (2D). The camera quality needed to reconstruct a scene is now available in mobile phones, and using photogrammetry software, a detailed representations of the real world can be achieved. One requirement is that the image capturing should be done in a particular way for good results. LiDAR is a scanning technique used for 3D scanning and modelling that directly captures depth data of a scene using laser. This method is dependent on sensors that historically has been expensive. LiDAR plays an important role in autonomous vehicles, by providing a sensor to observe the space around to avoid life-important collisions.

In the last decade, a combination of the two methods have received a good amount of interest in RGBD cameras like Azure's Kinect [1], because of it accessibility and ease to experiment with, using open source libraries like Open3D [3]. In particular, the combination of LiDAR and cameras has provided improved results from each sensor separated regarding cost vs. value [2]. The strenght of the combined sensors is the combination of direct depth information and contextual information. Professional actors like Trimble has also launched new technologies that combines both LiDAR and photogrammetry, providing cheaper options that still provide accurate 3D models.

Apple launched their two first devices with integrated LiDAR sensors, in iPad Pro 2 and iPhone 12 Pro in 2020. The launch was assumed to be to attract AR/VR developers to seek to Apple's applications, creating a development environment for new applications that combines digital and reality. A recent paper has presented use of Apple's new device to improve the modelling of as-built BIM [4]. Since the launch of Apple's LiDAR devices, it has been published some scientific evaluations of the performance [5], [6]. They use commercial applications on the market, namely SiteScape and EveryPoint. This thesis aims to do a similar type of evaluation, with an additional, new way of processing with own source code using Open3D.

The main contribution in this thesis has been to test out Apple's integrated LiDAR sensor hands on for indoor environment, and also provide a guide of how to do the reconstruction with attached source code. The aim is to test how well performance it is possible to achieve as a hobby developer since this sensor is readily accessible for anyone buying new mobile phones from the Apple Pro series. To the authors knowledge at the time of the preparation for the experiment, there existed no complete commercial software that could process input RGBD data, the closest being Pix4D Catch + Cloud, a beta version of a software using the new LiDAR sensor in Apple products. Therefore, the open 3D processing library, Open3D [3] was used to write a code to process the RGBD data.

Two commercial applications are also used as a comparison. Several applications using LiDAR has appeared in the App Store in the last year, both for entertainment as in AR/VR games, as well as applications aimed to more professional use, like 3D scanning

and modelling. One of the first applications for professional use was an application by SiteScape Inc. [7]. The free version will be mentioned in this thesis, since the pro version was launched after the execution of the experiment. The experiment was on April 30th 2021, while the pro version was launched on September 15th 2021. This pro version would be interesting to test in further work - as it proclaims to be:

> "*... the 1st iPhone LiDAR app for construction.*" [8]

As the challenge of automating 3D modelling, as-built BIM models could be made accessible even in private homes, since anyone could scan their own homes with their smartphones. Note that this will likely introduce privacy concerns, if your interior can indirectly tell discrete information about you (like that you have handicapped stairs, or where you live depending on unique things only bought in local regions). The benefits of such as-built modelling could for example be to monitor the need for renovation or prepare for how the texture and interior indoors would influence the spreading of flooding or a fire. It is likely that the low cost sensor provided from Apple does not yet provide accurate enough data for professional use like as-built BIM. It is, however, intriguing to test out if access to new technology can shape the development of a field such as indoor 3D modelling.

3 Method

In this chapter, the methods used in this work will be presented as detailed as possible for anyone to reproduce the results. In Section 3.1, the experimental setup is presented, with an overview of the scanned scene and the different scans executed. Section 3.2 explains the steps to get the data from Apple's LiDAR sensor with Swift code, and Section 3.3 explains how this data is processed using Python code to end up with the point cloud for the scan later referred to as "Scan 2". Section 3.4 then presents how the different scans have been evaluated.

Later, the results from each scan are presented in Chapter 4, and evaluated in Chapter 5.

3.1 Experimental setup

To evaluate the quality of the iPhone scans, a scan done by Trimble SX10 Scanner [9] has been used as "ground truth", with millimeter accuracy and 25 minutes of scanning time. The chosen scene to scan and reconstruct was a corner of a corridor at NTNU Geomatics, providing a room with different wall depths and a complicated roof, as well as bookshelves and doors along the walls. The chosen room layout is meant to provide a scene that has enough of clear(visible) walls, roof and floor to check the accuracy of the scan on a big scale, while also having smaller objects that can be inspected closer for detailed accuracy - and some occlusion to show the performance of the mobile phones mobility - meaning that it can get into smaller spaces. The scene is shown in Figure 1, where one can also see the Trimble SX10 Terrestrial Laser Scanner (TLS).

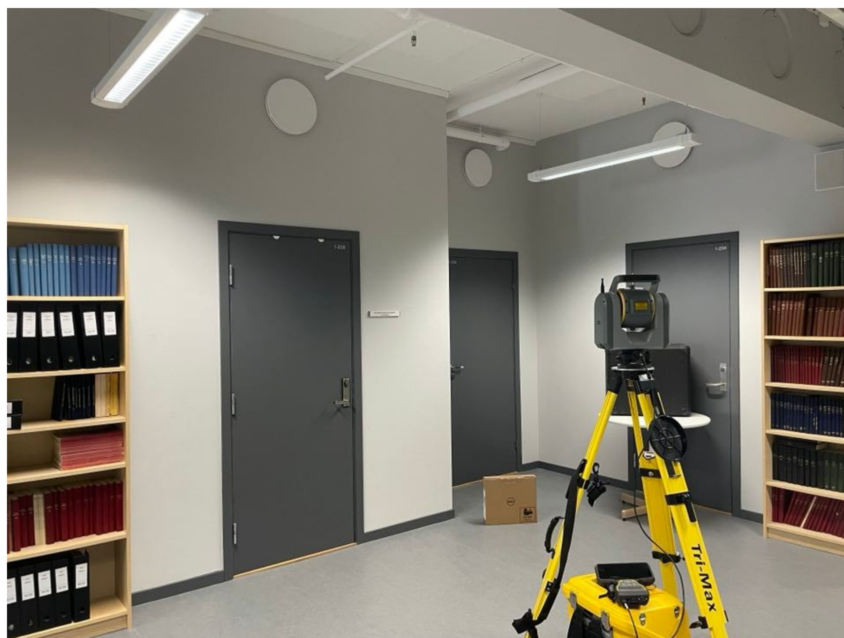
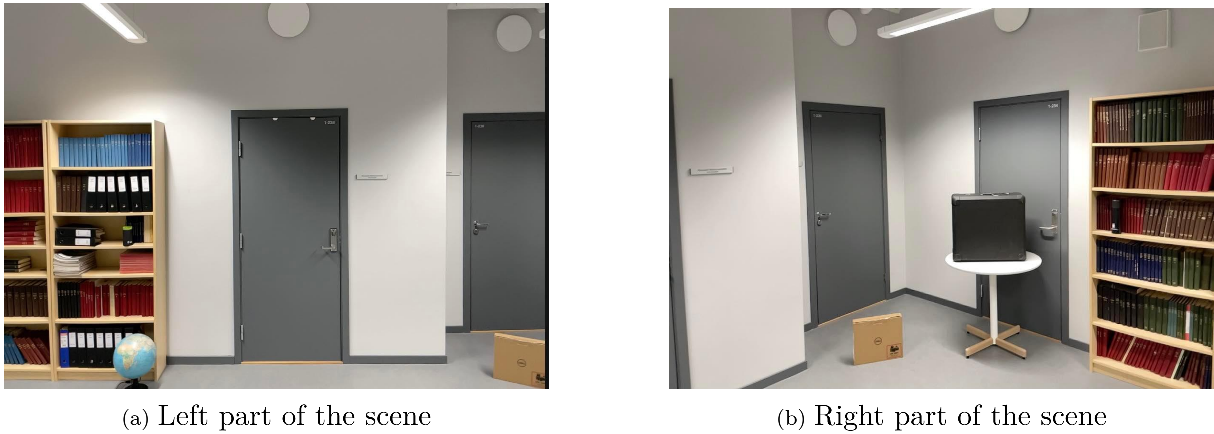


Figure 1: Room with Trimble SX10

In addition to the main structure of the room, some objects were added to increase the complexity of the scene. The objects were:

- A globe
- A brown paper box
- The black casing to the TLS scanner, on top of a round table
- Two coffee mugs

Figure 2 shows all the objects placed around the room. Figure 2a shows the globe on the floor and one of the coffee mugs (on the fourth bookshelf from the floor), and Figure 2b shows the black casing on a table and the other coffee mug (one the fourth bookshelf). The brown box shown in both figures is the same.



(a) Left part of the scene

(b) Right part of the scene

Figure 2: Objects placed in the room

Table 1 shows an overview of the four scans executed of the room shown earlier (see Figure 1). Scan 1 and 2 are the main focus of the thesis, as Scan 1 is used as "ground truth" to compares Scan 2 which is executed with the low cost sensor iPhone 12 Pro that has been processed using own source code.

Scan	Sensor	$\text{Cost}_{\text{sensor}}$	Software	$\text{Cost}_{\text{software}}$
1	Trimble SX10	$57.000	None	None
2	iPhone 12 Pro	$1.300	Swift script for accessing data, python script to process	None
A	iPhone 12 Pro	$1.300	App Store: SiteScape	None
B	iPhone 12 Pro	$1.300	App Store: Pix4D Catch	$159/month*

Table 1: Overview of the different executed scans with RGBD sensors from iPhone 12 Pro and Trimble SX10. * = a free 15 day trial was used.

The scans were all captured on April 30th, all four within one hour. In order to provide the best comparison foundation, the three scans executed with iPhone 12 Pro went around the room following a similar path. Starting from the left, the room was gathered strip-wise, with each new strip overlapping the last strip. The direction of each strip was horizontal. The scan followed the general rules for image acquisition for photogrammetry, guide from Pix4D [10].

Scan 1 is done using a stationary Trimble SX10 3D scanner. For simplicity, the Trimble scanner only scanned the room once. This way, there was no need to register the different point clouds together. The one scan took 25 minutes. The estimated accuracy is 1 mm + 1.5 ppm Prism / 2.0 mm + 1.5 DR [9]. This scan is considered "ground truth" and is the comparison base used to evaluate the other scans.

Scan 2 is the main contribution in this thesis. How this data was required and processed will be explained in the next subsections (3.2 and 3.3). The scan was executed in about 5 minutes, scanning the room strip by strip - achieving as much overlap as possible. An additional note is that the range of the iPhone 12 Pro is up to 5 meters, and the phone was kept between 1 and 4 meters from the objects.

The two prices shown in Table 1 are taken from i) a publication from Inside GNSS in 2016 [11], since the price is not easily accessible elsewhere, and ii) the bought price of iPhone 12 Pro (128GB), January 18th 2021. After the launch of the newer generation, iPhone 13, Apple does no longer sell the iPhone 12 Pro, so the price is taken from own expenses.

In addition to the two main scans, two other scans - **Scan A** and **Scan B** - where added to show how other applications on the market performed using the same iPhone 12 Pro as used in Scan 2. The two applications chosen were free and downloaded April 2021. These scans are presented, and briefly commented and evaluated. **Scan A** represents the market for free mobile applications (through the app SiteScape [7]), while **Scan B** is presented as the commercial product, through Pix4D Catch [12]. The data processing through Pix4D Cloud requires subscription $159/month [13]. Note, a free 15 day trial was used for this work.

3.2 Getting the data

This section presents Swift code of how to get the data from Apple's LiDAR device, and then how this data can be stored for later processing. The inserted Source Code in this section can all be found in Attachments (see Inspera), in folder **METHOD_CODE > swift_code**.

3.2.1 Accessing data from Apple's ARKit

Apple's ARKit is where the LiDAR data (such as depthMap) is accessible through. Note: at the time of the experiment execution (January-April 2021), ARKit 4.0 was the latest version. Since then (at the time of writing, October 2021), ARKit 5.0 has been released. This new release could provide different data and provide a better result, which would require a new experiment for another time.

To access the iPhone data, Apple's Xcode project "*Displaying a Point Cloud using Scene Depth*" [14] was used as foundation. With this Swift code, required steps such as setting up an **ARSession** was already done. From there, it was possible to the relevant data like:

- Color image: ARSession.ARFrame.capturedImage
- Depth image: ARSession.ARFrame.ARDepthData.depthMap
- Confidence image: ARSession.ARFrame.ARDepthData.confidenceMap
- Intrinsic matrix: ARSession.ARFrame.ARCamera.intrinsics

Apple's public Xcode projects all focused on *online* processing using the new LiDAR data. "Online" meaning instant, that the processing is done simultaneously as the app is running. The opposite is *offline*, meaning that the processing happens after the data has been captured. Instead of implementing such an online application, which would have a high requirement regarding processing time, it was decided to rather store the data, to rather do offline processing with a python script (see Section 3.3 for details of how this processing was done).

3.2.2 Storing the data

Within the Swift code ViewController.swift, the function for storing the data was triggered using a timer, every 2 seconds.

See Source Code 1 for how the **timer** implementation was set up. This code is inside the viewDidLoad() function in the ViewController.swift file, which is called continually while the app is running. Most importantly is the function called on line 7, **captureRGBD-pair()**.

Source Code 1: Timer setup for automatic function call, in **ViewController.swift**

```
1 // within ViewController.swift
2     override func viewDidLoad() {
3     ...
```

```
4            let timer = Timer.scheduledTimer(withTimeInterval: 2, repeats: true) {
5                (timer) in
6                print("Frame number: ", self.capturedFrames)
7                self.renderer.captureRGBDpair(session: self.session,
8                    frameNumber: self.capturedFrames)
9                self.capturedFrames+=1
10
11           }
12       }
```

Source Code 2 shows the main function **captureRGBDpair()** that captures ARKit-data *capturedImage* (captureImage(), line 9), *intrinsicMatrix* (captureInfo(), line 11), *depthMap* (captureDepth(), line 13) and *confidenceMap* (captureConfidence(), line 15). All of this data was stored with a unique ID that correlated to a specific scan, and numbered with an index, "i", which correlated so that every data-object with the same ID and same index would belong together.

An example of the file-naming syntax is: **"123ABC456dybde_0.json"**., where ID=123ABC456 and index i=0.

Source Code 2: Function captureRGBDpair, in **Renderer.swift**

```
1       func captureRGBDpair(session: ARSession, frameNumber: Int) {
2           guard let currentFrame = session.currentFrame,
3                 let imagePixelBuffer = session.currentFrame?.capturedImage,
4                 let depthMap = session.currentFrame?.sceneDepth?.depthMap,
5                 let confidenceMap = session.currentFrame?.sceneDepth?.confidenceMap
6           else {
7               return
8           }
9           captureImage(imagePixelBuffer: imagePixelBuffer,
10              frameNumber: frameNumber)
11          captureInfo(currentFrame: ARFrame,
12              frameNumber: frameNumber)
13          captureDepth(depthMap: depthMap,
14              frameNumber: frameNumber)
15          captureConfidence(confidenceMap: confidenceMap,
16              frameNumber: frameNumber)
17      }
```

Looking at the example of depthMap, Source Code 3 presents the function called on line 13 in Source Code 2, **captureDepth()**. The input is the **depthMap** captured from ARKit (see line 4 in Source Code 2), which is of type CVPixelBuffer, and the **frameNumber.** Line 4 converts the depthMap into JSON format, which is then stored in line 15 as a JSON-file.

Source Code 3: Function captureDepth, in **Renderer.swift**

```
    func captureDepth(depthMap: CVPixelBuffer,
                frameNumber: Int) {
        let jsonDict: [String : Any] = [
            "depth_data" : convertDepthData(depthMap: depthMap)
        ]
        let jsonStringData = try! JSONSerialization.data(
            withJSONObject: jsonDict,
            options: .prettyPrinted
        )
        let paths = FileManager.default.urls(for: .documentDirectory,
            in: .userDomainMask)
        let documentsDirectory = paths[0]
        let fileDepth = documentsDirectory.appendingPathComponent(
            "\(ID)dybde_\(frameNumber).json")
        try? jsonStringData.write(to: fileDepth)
    }
```

Source Code 4 shows how to access and store the data from the format CVPixelFormat to **JSON**. JSON format is chosen since it is easy to structure however required, has a small storage size, as well as it is easily readable.

Source Code 4: Function convertDepthData, in **Renderer.swift**

```
 func convertDepthData(depthMap: CVPixelBuffer) -> [[Float32]] {
        let width = CVPixelBufferGetWidth(depthMap)
        let height = CVPixelBufferGetHeight(depthMap)
        var convertedDepthMap: [[Float32]] = Array(
            repeating: Array(repeating: 0, count: width),
            count: height
        )
        CVPixelBufferLockBaseAddress(depthMap, CVPixelBufferLockFlags(rawValue: 2))
        let floatBuffer = unsafeBitCast(
            CVPixelBufferGetBaseAddress(depthMap),
            to: UnsafeMutablePointer<Float32>.self
        )
        for row in 0 ..< height {
            for col in 0 ..< width {
                convertedDepthMap[row][col] = floatBuffer[width * row + col]
            }
        }
        CVPixelBufferUnlockBaseAddress(depthMap, CVPixelBufferLockFlags(rawValue: 2))
        return convertedDepthMap
    }
```

3.3 Processing the data

Processing the data is done to end up with a point cloud. The general steps are

1. Confidence-filtering depth-images
2. Pair each RGB-image with the corresponding confidence-filtered depth-images as .pcd-files (Point cloud). The reference for each point cloud is the camera position.
3. Matching process: For each pcd, i, iterate pairwise through:
   (a) Between each point cloud pair (i'th and i+1'th), a best fitted transformation is found by first using RANSAC.
   (b) then Colored-ICP on the result from the RANSAC step. The best fitted transformation for each point cloud, p_i, is stored in the **PoseGraph**, edge e_i
4. Posegraph optimization
5. Apply the transformation from each pose, e_i, on the corresponding point cloud, p_i and add the transformed point cloud into *combined pcd*.

These five steps will be further explained in Subsection 3.3.1-3.3.5. The method is based off of Open3D Reconstruction System [15], and in particular the registration/matching method (step 3 above) is based on Open3D **Pairwise Global Registration**. The Python Source Code presented in this section can all be found in Attachments (see Inspera), in folder **METHOD_CODE > python_code**.

In addition to these steps, an improvement-step before the matching procedure by applying IMU transformations (IMU information gathered from ARKit) as an initial transformation to align the RGBD-frames was tried. However, this sometimes provided worse results, so it was not implemented in the end.

3.3.1 Filtering by confidence

The depth-values are first filtered with the data from ARKit's confidenceMap (ref theory). By filtering on only confidence-values of type "HIGH" (==2), this is meant as an improvement step. The reconstruction method can also be used without this step, then one can use the depth-images directly (see folder "**preprocessing/output/depth**") Note: the filtering most times only removes a small percentage of the images, and the confidence-filtering does not guarantee that every depth-pixel is correct.

Source Code 5 presents the main functionality from **confidence_filtering.py**. Line 1-7 consist of the packages used. Listed is a brief explanation of the packages.

- "**os**" and "**sys**" is used to access hardware information and functionality on things like the local directory and filepaths.
- "**json**" is a package to used to load .json-files.
- "**numpy**" is a package to handle arrays and matrices in python.
- "**cv2**" and "**PIL**" are image handling/processing packages.

After accessing the .json-file containing the confidence, every value is checked in the double for-loop on line 13-14. If the value is 2 (considered "high", the highest confidence), the confidence-data is set to "1" (line 16), else it is set to 10000000 (line 18). Line 20 is where the depth-data is filtered using the updated confidence-data, where the confidence and depth data is multiplied element-wise. The last two steps before storing the new depth-image is to multiply by 1000 on line 21 (changing the unit from meter [m] to millimeter [mm] (since Open3D expects the depth-data in millimeter), and then resize the image to the set image scale (640, 480) on line 22.

Source Code 5: Python code for filtering, **confidence_filtering.py**

```
1  import cv2
2  import os
3  import json
4  import numpy as np
5  from PIL import Image
6  import sys
7
8  def save_confidence_depth():
9      ...
10     for instance in range(93):
11         ...
12         for i in range(192):
13             for j in range(256):
14                 if data['confidence_data'][i][j] == 2: # confidence level HIGH
15                     data['confidence_data'][i][j] = 1
16                 else: # confidence level MEDIUM or LOW
17                     data['confidence_data'][i][j] = 100000000
18         ...
19         filtered_depth = np.multiply(confidence, depth)
20         filtered_depth_mm = filtered_depth*1000
21         filtered_depth_mm_resize = cv2.resize(filtered_depth_mm, (640, 480))
22         ...
```

In theory, the "else" value was intended to be set to "0", in order to filter out anything that is "medium" or "low" confidence when multiplying the two arrays. However, this resulted

in a point cloud where all depth-values of "medium" or "low" confidence was floating mid-air. Therefore it was set to an arbitrary high value (10000000). In the code-snippet in later Subsections (like 3.3.2, Source Code 6, there is a cap of 5000 for values recognized in Open3D methods. Therefore, the values that are above this (like the depth-values multiplied by 10000000) are disregarded.

3.3.2 Storing RGBD data as point clouds

After the filtering step, we now have a list of RGBD pairs - RGB image and a filtered depth image paired together by index *i*. These pairs are now stored as *point clouds*, using Open3D functionality. Point clouds will be referred to as "pcd"/"pcds". Given that we have chosen image resolution of (640, 480), the number of points per point cloud will be 307.200. Note that the resolution of the depth images are originally 256x192, and resolution much higher than the original scale will not benefit from more depth-information.

In Source Code 6, the for loop on line 5 iterates through every instance of color_image and depth_image (i=93) and saves the point clouds (on line 19) by first creating an **RGBDImage** (line 10-15) and the corresponding *intrinsicMatrix* (line 17), and then storing each point cloud (see line 19). Two important inputs line 13-14 is: (see Open3D doc for full description)

- Depth scale: 1000.0 (scale of depth-values are given in millimeter)
- Depth trunc: 5.0 (maximum depth value 5.0 meter)

Each point cloud, p_i, is then saved with the command on line 22.

Source Code 6: Python code for storing pcd with filtered depth-data, **confidence_filtering.py**

```
1  import open3d as o3d
2  ...
3  def save_confidence_pcds():
4      ...
5      for instance in range(93): #93 is number of images
6          ...
7          #save_confidence_pcd(input_dir, instance):
8          color_image = o3d.io.read_image(dir_color)
9          depth_image = o3d.io.read_image(dir_depth)
10         rgbd_image_i = o3d.geometry.RGBDImage.create_from_color_and_depth(
11             color_image,
12             depth_image,
13             1000.0,
14             5.0,
```

```
15                  False)
16              ...
17              pinhole_camera_intrinsic_i = o3d.io.read_pinhole_camera_intrinsic(
18                      dir_intr)
19              pcd_i = o3d.geometry.PointCloud.create_from_rgbd_image(
20                      rgbd_image_i, pinhole_camera_intrinsic_i)
21              ...
22              o3d.io.write_point_cloud(save_loc, pcd_i, write_ascii=False,
23                  compressed=False, print_progress=False)
```

3.3.3 Matching process: Pairwise Registration

In Source Code 7, the main matching process is presented. First, the point clouds from the last step (Subsection 3.3.2) is down-sampled to improve runtime, without it reducing the final product (see how final transformations are applied to the full point clouds in Subsection 3.3.5).

Then, a for loop iterates through the downsampled point clouds once, where every pair of pcds (source, target) is used as input to find the best fitted transformation between them. To iterate neighbour-wise is one of the simplest methods to guarantee that the requirement of overlap is overheld, which gives the best starting point for a unique transformation match. To find each transformation between a pair, a preprocessing step (see line 13-14) is applied. This preprocess-method calculates the normals and FPFH Fast Point Feature Histograms information of the downsampled point clouds. These results are then used in the Fast alignment RANSAC step on line 17, and this result is again used as input in a (final) refined alignment, using Colored ICP (line 21).

The three methods **preprocess_point_cloud(...)**, **execute_global_registration(...)** and **refine_color_icp_registration(...)** are all copied from Open3d Register Fragments. The difference regarding implementation of the source code used to generate Scan 2 and Open3D is that the matching is only done the pairwise point clouds, and is therefore not the global registration done in Open3D. In addition, the transformation in the method used for Scan 2 is a combination of two matching-methods used in Open3d, RANSAC –> Colored ICP.

Source Code 7: Python code for the matching process, **global_registration.py**

```
1 import open3d as o3d
2 import numpy as np
3 ...
4 voxel_size = 0.05
5 pcds_down = load_down_point_clouds(dir, voxel_size=0.05)
6 n_pcds = len(pcds_down) #93
7 ...
8 for source_id in range(n_pcds-1):
```

```
9       #setup and preprocessing
10      source = pcds_down[source_id]
11      target = pcds_down[target_id]
12      ...
13      source_down, source_fpfh = preprocess_point_cloud(source, voxel_size)
14      target_down, target_fpfh = preprocess_point_cloud(target, voxel_size)
15
16      #fast/rough alignment using RANSAC
17      result_ransac = execute_global_registration(source_down, target_down,
18                                                  source_fpfh, target_fpfh,
19                                                  voxel_size)
20      #refined (final) alignment, improving on result_ransac
21      refined = refine_color_icp_registration(source, target, result_ransac)
22      final_transformation = refined.transformation
23      final_information =
24          o3d.pipelines.registration.get_information_matrix_from_point_clouds(
25              source, target, max_correspondence_distance_fine,
26              final_transformation)
27      ...
28      #storing the final_transformation to the corresponding node
29      odometry = np.dot(final_transformation, odometry)
30      pose_graph.nodes.append(o3d.pipelines.registration.PoseGraphNode(
31                  np.linalg.inv(odometry)))
32      pose_graph.edges.append(o3d.pipelines.registration.PoseGraphEdge(
33                  source_id,
34                  target_id,
35                  final_transformation,
36                  final_information,
37                  uncertain=False))
38  ...
```

Lastly, each transformation is stored in the PoseGraph in edge i, e_i, as showed on line 29-37.

NOTE: the matching methods returns values for how well the matching fits. However, there is no check in the code to filters on this. That is an improvement step that would remove bad matches. Open3D Reconstruction System has implemented steps to filter on "success"-mathces, and should be implemented in further work.

3.3.4 Posegraph optimization

After every transformation has been calculated in Subsection 3.3.3, a optimization step is applied to adjust transformations that does not fit with the rest of the poses stored in

the stored posegraph. This PoseGraph Optimization is showed in Source Code 8 and is copied from Open3D Multiway Registration

Source Code 8: Python code for the PoseGraph Optimization from Open3D, **global_registration.py**

```
...
        print("Optimizing PoseGraph ...")
        option = o3d.pipelines.registration.GlobalOptimizationOption(
            max_correspondence_distance=max_correspondence_distance_fine,
            edge_prune_threshold=0.25,
            reference_node=0)
        with o3d.utility.VerbosityContextManager(
                o3d.utility.VerbosityLevel.Debug) as cm:
            o3d.pipelines.registration.global_optimization(
                pose_graph,
                o3d.pipelines.registration.GlobalOptimizationLevenbergMarquardt(),
                o3d.pipelines.registration.GlobalOptimizationConvergenceCriteria(),
                option)
...
```

3.3.5 Transformations and final result

Lastly, Source Code 9 shows the code that transforms every point cloud with the corresponding transformation matrix. Line 4 loads the point clouds, now without downsampling. Line 8-13 transforms each point cloud with its corresponding transformation and adds it one combined, final point cloud.

Source Code 9: Python code for applying the final transformation for each point cloud, **global_registration.py**

```
from datetime import datetime
...
        print("Transform points and store combined point cloud in ouput-folder")
        pcds = load_point_clouds(dir) #fetching the full point clouds
        #defining the combined point cloud to store the complete result.
        #Initially, this is empty
        pcd_combined = o3d.geometry.PointCloud()
        for point_id in range(n_pcds):
            #transforming the relevant point cloud with its corresponding
            #transformation stored in the pose graph.
            pcds[point_id].transform(pose_graph.nodes[point_id].pose)
            #adding the new pointcloud to the final result
            pcd_combined += pcds[point_id]

```

```
15          #storing the final point cloud
16          date = datetime.now()
17          d1 = date.strftime("%H-%M-%S-%b-%d-%Y")
18          o3d.io.write_point_cloud("output\\global_registration_"+d1+".ply",
19              pcd_combined)
```

Line 16-19 stores the combined point cloud (pcd_combined) in a designated ouput-folder, with the current time and date stored in the name. The time and date is calculated using the package datetime (line 1).

3.4 Evaluating the results

Now for evaluating these results, the Trimble SX10 scan will be provided as ground truth. This section explains what types of evaluation methods that is used to compare the two results (Scan 1 (Trimble) vs. Scan 2 (iPhone with open source code)).

The comparison software used is CloudCompare (C2C) [16], using *Absolute distance*-function, which calculates the distance between the closest point in two point clouds. For the actual evaluation, see Chapter 5.

3.4.1 Filtering/cleanup/postprocessing

Before any evaluation can be done, a post-processing step to clean-up the point cloud in Scan 2 is executed. Using C2C and the method "**Tools –> Clean –> SOR-filter**". SOR stands for Statistical Outlier Removal (see C2C wiki: SOR-filter), and the degree of strictness is decided by this equatuion in Figure 3

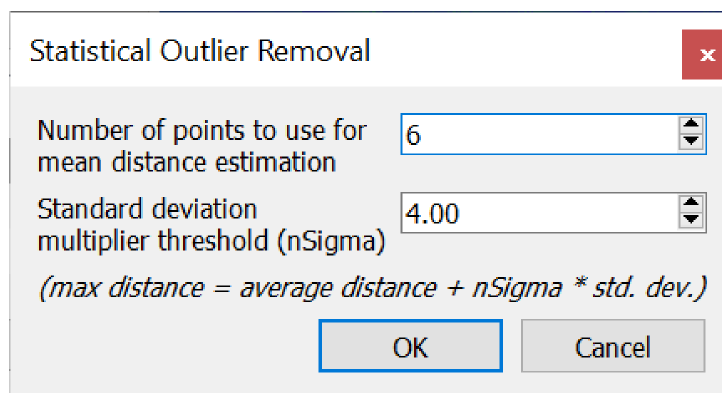


Figure 3: SOR-filter in C2C

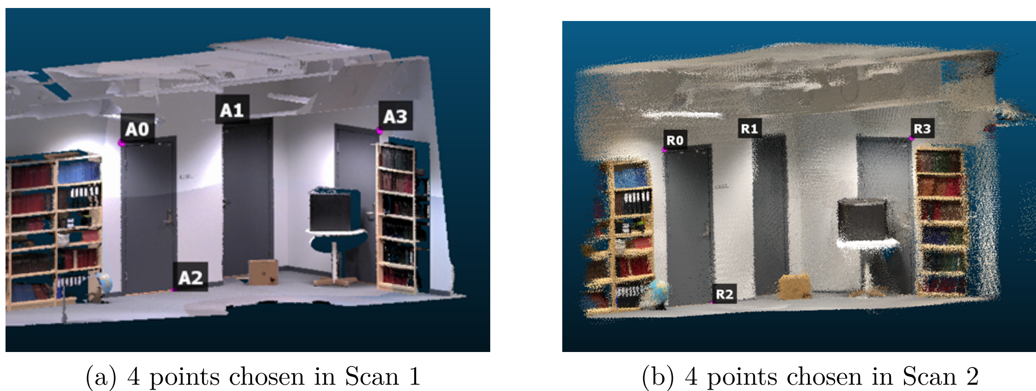
This filtering step was done once, with the *nSigma* value is set to 1.0, which is considered a strict value. This value influence the max distance that each point is checked against

to evaluate if it is an outlier to be removed or not. The result is a filtered point cloud, and this will be presented and evaluate in Chapter 5, against the non-filtered result.

This step would be a part of the python-script in the future. For the sake of this thesis, the program I am using for evaluating already has is implemented, and was therefore used for the filtering as well.

3.4.2 Alignment

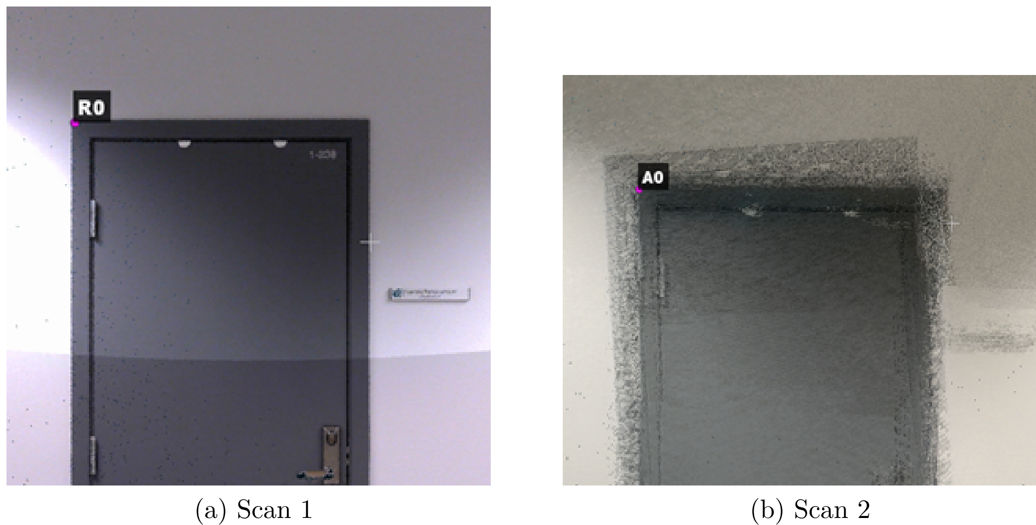
First, the two point clouds are **roughly** aligned using C2C Align(point pairs picking). 4 points were used, at different plane locations (different door corners), see Figure 4.



(a) 4 points chosen in Scan 1 (b) 4 points chosen in Scan 2

Figure 4: Point pairs for C2C rough alignment

Note that some of the point pairs used in the rough alignment might be inaccurate, due to the noise in Scan 2. See Figure 5. To improve this potential faulty alignment, another alignment step is applied.



(a) Scan 1 (b) Scan 2

Figure 5: Example of point pair nr. 0 used in the rough alignment

A second, finer alignment is done using C2C Iteratively Closest point (ICP) with default settings.

3.4.3 Distance measurements

When the alignment step was done, distance measurements is now executed. This is the metrics that will be used to evaluate the accuracy/performance of Scan 2.

First, Absolute distance is found using "Tools > Distance > Cloud to Cloud dist." is chosen. Following the same steps as in the C2C wiki, we first choose the Trimble TLS scan as the reference (see Figure 6). This sets the chosen **reference** as the "ground truth". This means that it is each point of the **compared** point cloud that the function calculates distance to closes other point from.

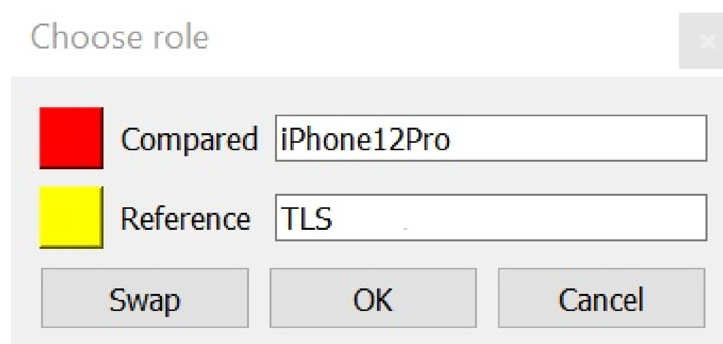
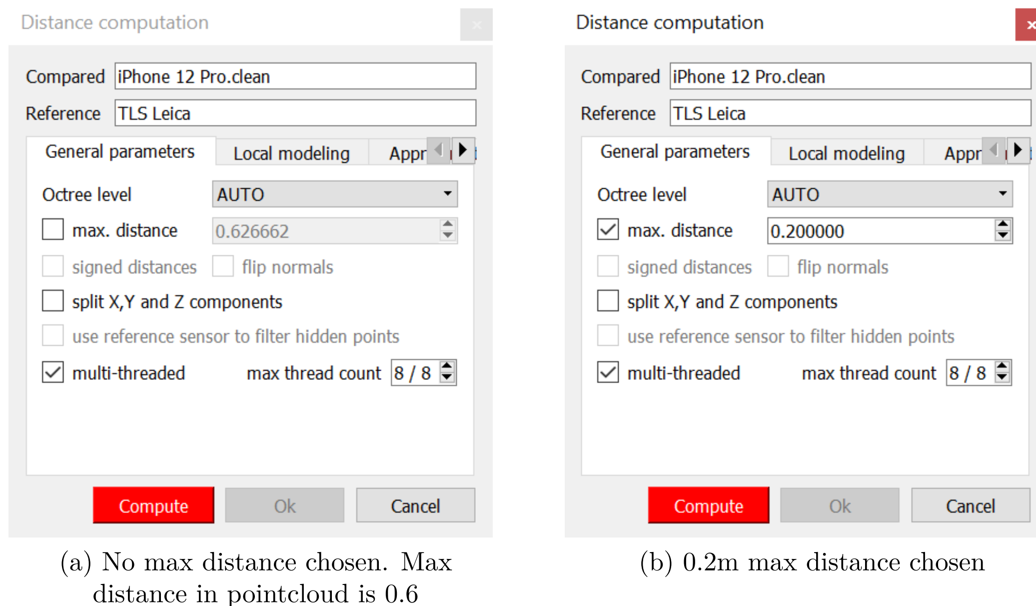


Figure 6: Choose role in C2C. Reference is the "ground truth"

Secondly, one can decide to put a maximum distance for the calculation. To get statistics for the whole point cloud, a distance was first calculated with no maximum distance.

Afterwards, a max distance of 0.2m was chosen. This was done to have a scalar point cloud with more range of colors, making it easier to evaluate by only looking at it. The different distances are shown in Chapter 5.



(a) No max distance chosen. Max distance in pointcloud is 0.6

(b) 0.2m max distance chosen

Figure 7: Distance computation in C2C

After computing the distance, the **compared** point cloud now contains data of distance to closest other point for each point. For a visualization of this, choose "Scalar field" as **Color** in the **compared** point cloud Properties pane (see Figure 8).

Additionally, a **histogram** and a **Gauss distribution fitting** can be shown and stored. This is where statistics like **mean distance** and **standard deviation** can be found.

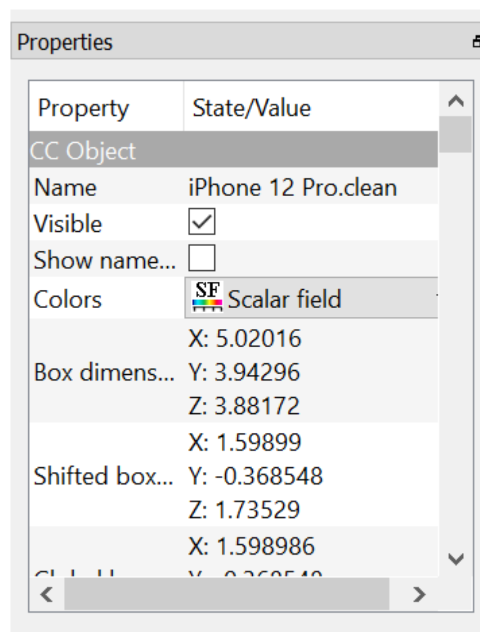


Figure 8: How to show the distance measurements as scalar fields in the point cloud. Choose Scalar Field as Color

3.4.4 Segmentation

For another evaluation, the point cloud is *segmented* into smaller sections. In particular, Section 5.1.3-5.1.5 shows a closer inspection of the *door*, the *roof* and the *cluttered corner*. To include only the relevant part of the point cloud, C2C Cross Section is used. A 3D boundary box is manually adjusted to fit around the relevant areas. See Figure 9 for example. The cross-section can be stored as a new, separate point cloud.

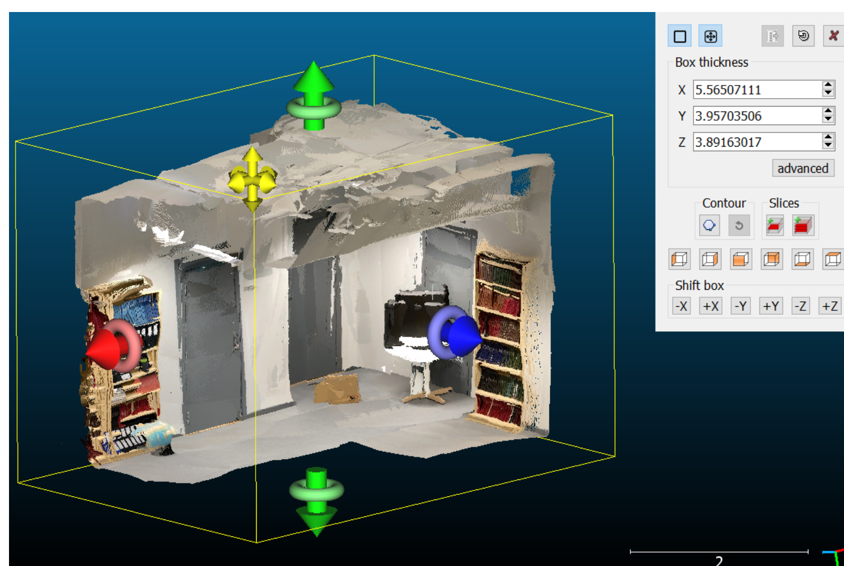


Figure 9: 3D boundary box for applying C2C Cross-Section

4 Results

This chapter presents the results from each scan presented previously in Section 3.1. Section 4.1 presents the scan executed by Trimble SX10 (Scan 1), Section 4.2 presents the iPhone scan processed with own source code (see Section 3.3 for detailed method), Section 4.3 presents the scan processed with the commercial application SiteScape, and Section 4.4 presents the scan processed with the commercial application Pix4D Catch + Cloud.

This section is meant to give a brief overview of the performance of each scan, while a more in depth evaluation is done in Chapter 5. Each scan is shown as a snapshot from a .pcd file viewed in C2C, from the "front" view. For a closer inspection of these datasets, please see the attached .pcd-files in folder **RESULTS_pcds**.

4.1 Scan 1: Trimble SX10

Figure 10 presents the scan captured with the Trimble SX10 TLS. This is considered the "ground truth", i.e. the reference that will be used to measure the performance of the different scans later in Chapter 5.

This point cloud is the result of one scan executed by the Trimble TLS. Notice the blank spaces in the scan (the blue strip between the doors, the vertical strip in the upper left corner and the see-through side of the bookshelf on the right). This is parts of the space that was outside the field of vision of the Trimble TLS, by obfuscating objects such as a wall, boxes, pipes or lights.

4.2 Scan 2: iPhone 12 Pro with own source code

Figure 11 presents the point cloud that has been captured with an iPhone 12 Pro. The data was captured in a mobile application launched via swift code (see Section 3.2), and the data has been processed by a python script, applying methods from the 3D processing library Open3D (see Section 3.3).

What one can see in Figure 11 is that the rough structure of the room is well reconstructed, with flat surfaces staying straight, and the room has kept its rectangular outline. Objects and distinct areas are also clearly bundled in their correct areas (meaning that they are placed relatively correct).

One can also see clear signs of mismatching, mainly visible with the color of the points. For example, one can see that the left door suffers from a smudged and skewed upper left corner, making it appear slighlty tilted to the left. Other places, like the bookshelf on the left, the bookshelves themselves do not align correctly, with some matches presenting

Figure 10: Front of pointcloud from TLS

shelves in the middle of two actual shelves.

Another sign of mismatching is the fact that flat areas such as the floor has been matched wrong, making the plane appear much thicker than it is in reality.

This scan is able to cover areas that Trimble missed, since the iPhone has no limitations as to where it can be moved when traversing through the room. However, even with this additional flexibility to move, this scan also has missing spots. These missing parts likely caused by insufficient scanning base - see above the light, behind the black box etc.

In the area mentioned earlier where Trimble missed to cover one whole wall (the one between first and second door from the left), there was clear signs of what can referred to as "overflowing" depth-points.

Lastly, the objects that was placed around the room to be closer inspected when evaluating the finished point cloud is briefly commented. These objects are

- Globe
- Black box
- Brown box
- 2 coffee-mugs (try to find them on your own ;) HINT: they are in the bookshelves)

The globe has mostly been covered on one side, and some of it has indentation, hinting

Figure 11: Front of pointcloud from own pipeline

towards error in RGBD-input. The black box (and the white table it stands on), is one of the objects in the room with the most overflowing points, as well as bad matching, making points "bleed out" far away from where it actually is, and missing points behind it due to bad data fundament. The brown box is also "bleeding out", and some of the areas behind it is missing. The two coffee mugs are barely visible in the bookshelf.

4.3 Scan A: iPhone 12 Pro with SiteScape

Figure 12 shows the point clod using the commercial software SiteScape. SiteScape is a free mobile app found in App Store and provided a commercial solution that lets one scan a room and store the point cloud. While scanning, the application provide visual feedback that shows what part of the room has been captured with the iPhone sensors. Due to the better visual guidance while scanning, Scan A was able to cover more corners of the room than Scan 2. The scan and storing took below 5 minutes.

SiteScape also provides the option to export the file as an E57 file, which can be uploaded to several 3D softwares (like Sketchup, Autodeks Recap etc.) Their point cloud scans had the capacity to reach 12 million points at the time of scanning.

In general, the results from online processing using a free application provides an overall good representation of the room, but details are not represented clearly. As one can see, the walls/floor/roof reconstruction is kept a rectangular shape, with a slight bend some places (watch the floor and the right wall). The scan has however covered areas the Trimble scanner could not access, and the cluttered areas has not suffered from visible drift, mismatching and overflow. The other difference from the two other scans, is that

Figure 12: Front of pointcloud from SiteScape

this scan is less dense, as one can see with the walls being slightly see-through with blue dots.

4.4 Scan B: iPhone 12 Pro with Pix4DCatch

Pix4D Catch provided a versatile tool to capture the RGBD data from Apple. The scan took under 5 minutes, with processing of unknown time (there was no notification or info of start and stop). The way to process the data captured with Pix4D Catch data was through Pix4D Cloud (beta), a cloud solution. This software was used to generate the point cloud of Scan B. Note: to get access to the cloud-processing, free trial was used. It would require a subscription if used regularly. The results from Scan B can be seen in Figure 13.

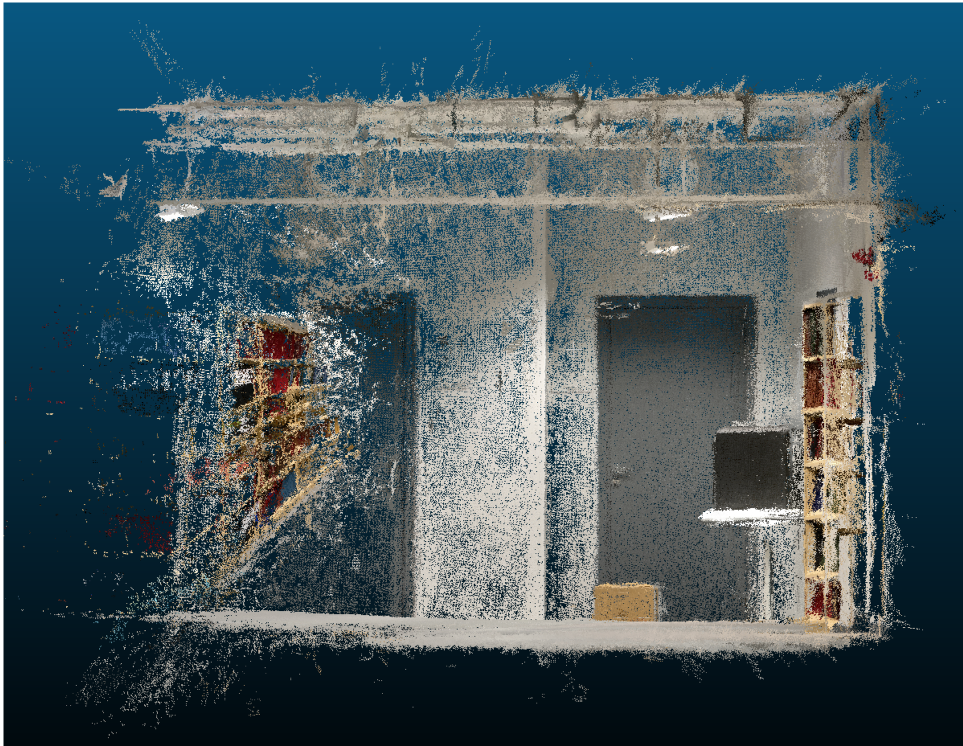


Figure 13: Front of pointcloud from Pix4D cloud (beta) processing

This result contains large areas of unmatched points, and is much noisier than the previous scans. Some areas of the scan has been well reconstructed, but some problems occured on the left side of the pointcloud, which is hard to know why. Some part of this scan could be used for evaluation, but since the current state did not provide a reliable point cloud using, it was disregarded in the rest of the work. The reason for the poor performance could be because of a bug in the beta version, or fault in the scanning. This was however not further explored after the initial scan.

5 Evaluation

In this chapter, the scans from Chapter 4 are compared to see how well iPhone 12 Pro performs for indoor 3D modelling. Section 5.1 compares the ground truth (Scan 1, see Section 4.1) with the scan executed by iPhone and processed with own source code (Scan 2, see Section 4.2). Section 5.2 compares Scan 1 with the scan executed by iPhone and SiteScape (Scan A, see Section 4.3). Scan B (see Section 4.4) has not been evaluated due to the poor performance.

Section 5.1 contains the most extensive evaluation. The idea is to form a foundation for people to compare against in further work, since this code is open source. Section 5.2 is included to form a reference to how well commercial applications perform with the integrated LiDAR sensor in Apple's devices.

All the data presented in this chapter can be found in Attachments (see Inspera), folder **..>EVALUATION_data**.

5.1 Scan 1 vs. Scan 2

This section begins with two evaluatiions of the full Scan 2, in Section 5.1.1 and Section 5.1.2. These two evaluations are intended to give a general idea of the performance of Scan 2, and the influence of filtering. Section 5.1.3 and 5.1.4 continues to evaluates the data from Scan 1 and Scan 2, now on a smaller scale to get another perspective.

All the data in this section can be found in subfolder **..>scan1vs2**.

5.1.1 Full scan, no filtering

To get a general picture of how good the scan executed with iPhone is compared to Trimble, a comparison of the "full scan" is presented here. The data can be accesed in subfolder **without_filter**

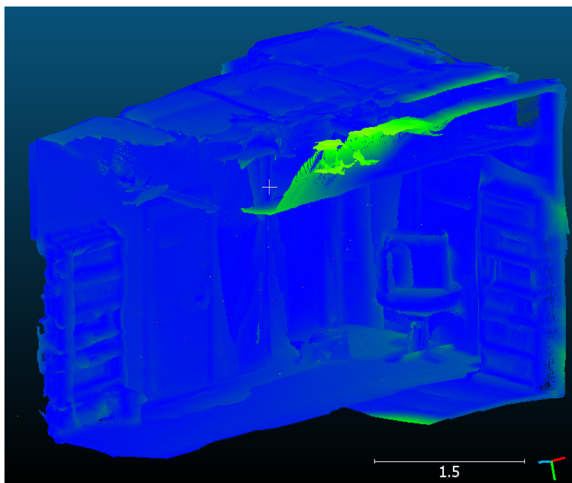
After C2C computation is done, Scan 2 now has an attached scalar field visualizing how each point performs.

First, Figure 14 shows some figures from the scan *without any filtering* - meaning it contains every point stored from the processing step. Figure 14a is the scalar field with no cap, meaning that the scale goes from 0 to the max distance measured, which in this case is 1.187234m. This provides a scalar field that is hard to see nuances in, since most of the measurements are in the left 20 percent of the graph and only shows up as blue points.

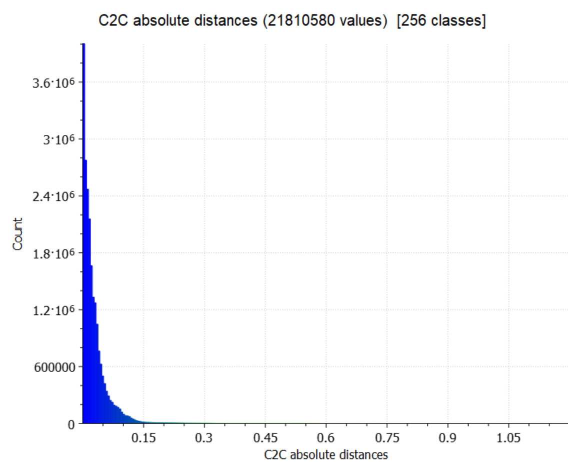
In order to give a more nuanced picture, the distance computation has been capped at

0.2m (meaning that every point over this cap is only stored as the value of the cap). This new, capped distance measurement is shown in Figure 14c. This gives a better picture of the performance of the point cloud. Figure 14d shows the corresponding histogram. As shown in the histogram in Figure 14d, red point values in Figure 14c represents points with the largest distance from the ground truth, while blue points are closest to its true position.

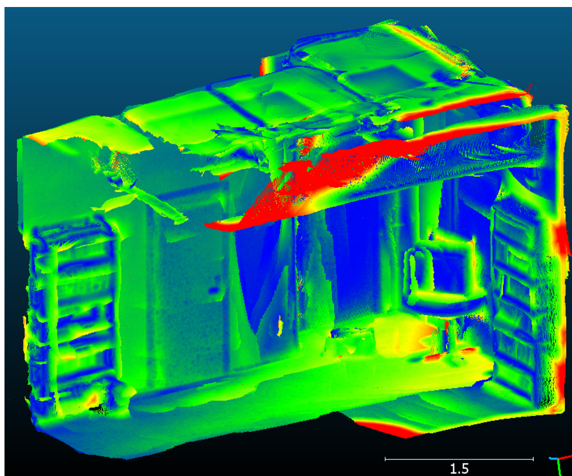
In general, the areas with the best performance is the walls, bookcases and parts of the black box, areas with worse performance is the floor and roof - and the worst areas are a big section of part of the roof.



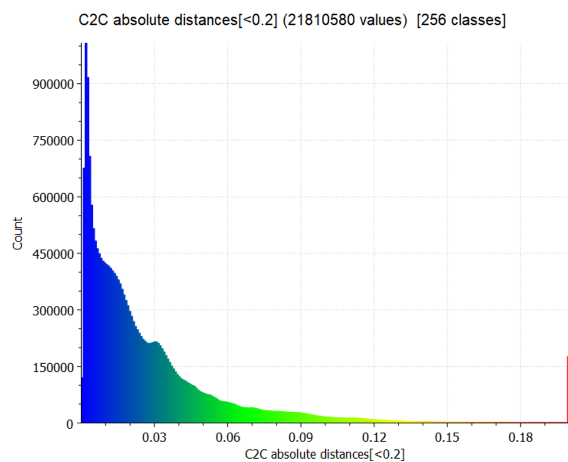
(a) Full scan scalar field of Scan 2, no filter and no cap



(b) Corresponding histogram



(c) Full scan scalar field of Scan 2, no filter and 0.2m cap



(d) Corresponding histogram

Figure 14: Full scan 1, no filtering

Table 2 shows the statistics from the distance computations corresponding to the data shown in Figure 14. The statistic is the mean distance and the standard deviation (SD).

The median distance of the full scan, with no cap (line 1), is 2.87 cm, with a standard deviation of 3.94 cm. The additional statistic in Table 2 (line 2) shows what the mean and standard deviation would be with the cap of 0.2m. The mean is then 2.77 cm, with a standard deviation of 3.17 cm. This is added to give an idea of how much the outliers influence these values, especially the standard deviation. Note that it is the value with *no cap* that is used as performance evaluation metrics, since it considers the whole dataset.

Full scan 1 (no filtering)	Mean	SD	Max dist.
Cloud-to-Cloud distance	0.028704	0.039387	1.187234
Cloud-to-Cloud distance [<0.2m]	0.027744	0.31741	0.2

Table 2: Mean distance and standard deviation (SD) for C2C Cloud-to-Cloud distance done on the full scan, no filtering. Shown is both values for max distance, as well as a cap on 0.2m. The unit is given in meters

.

5.1.2 Full scan, with filtering

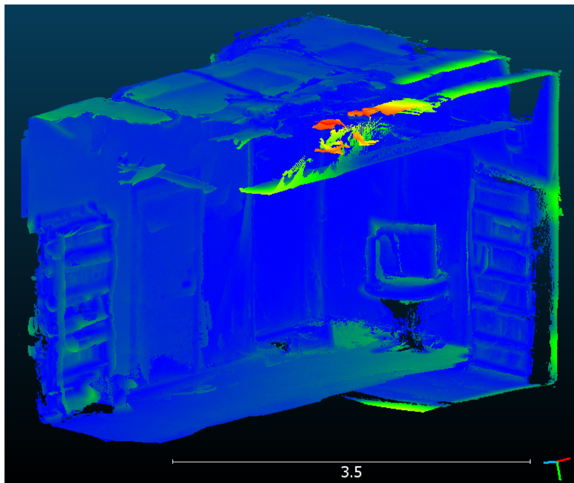
After the filtering step in Section 3.4.1, we have a filtered scan, which is considered as the actual result. The filtering step was done to eliminate floating points with a large distance to corresponding points. The filtered point cloud can be seen in Figure 15. The data in thsi section can be found in subfolder **..>with_filter**

In general the two figures show the same tendencies with good and bad regions as presented in the previous section (Section 5.1.1). The difference is that about 2.8 million points have been removed by the SOR-filter, corresponding to 13 percent decrease from the unfiltered point cloud in Figure 14 (previous section). Sparse areas and floating points has been removed as a consequence. Another thing that can be seen in Figure 15 is that the scale of Figure 15a and 15b has been reduced (because outliers with the largest distances are removed), providing a more nuanced scalar field without any cap than what was observed in Figure 14a and 14b. This point cloud is also represented with a cap of 0.2m for the best visualization of nuances (see Figure 15c and 15d).

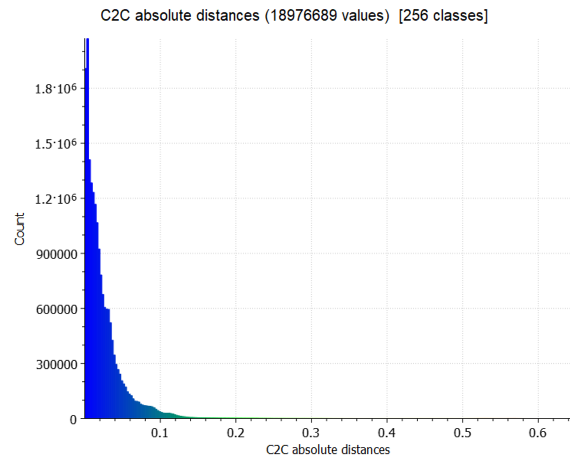
Table 3 shows that the median distance of the filtered full scan is 2.58 cm, with a standard deviation of 3.64 cm.

Full scan 2 (with filtering)	Mean	SD	Max dist.
Cloud-to-Cloud distance	0.025805	0.036387	0.636144
Cloud-to-Cloud distance [<0.2m]	0.024975	0.28908	0.2

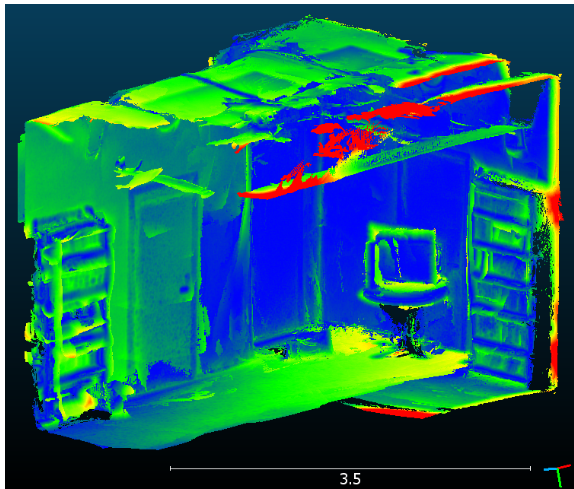
Table 3: Mean distance and standard deviation (SD) for C2C Cloud-to-Cloud distance done on the full scan, with filtering. Shown is both values for max distance, as well as a cap on 0.2m. The unit is given in meters



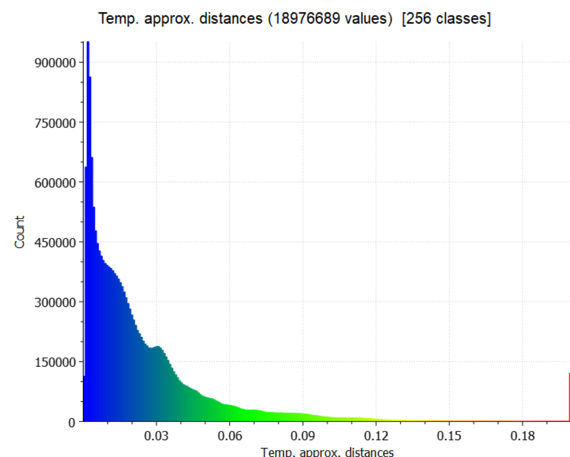
(a) Full scan scalar field of Scan 2, with filter and no cap



(b) Corresponding histogram



(c) Full scan scalar field of Scan 2, with filter and 0.2m cap



(d) Corresponding histogram

Figure 15: Full scan 2, with filtering

Table 4 shows the table values from Section 5.1.1 and 5.1.2, without cap, and the resulting **difference** between the two measurements. The filtering does improve the mean distance and standard deviation by 3 mm. The max distance has been reduced to 63.61cm.

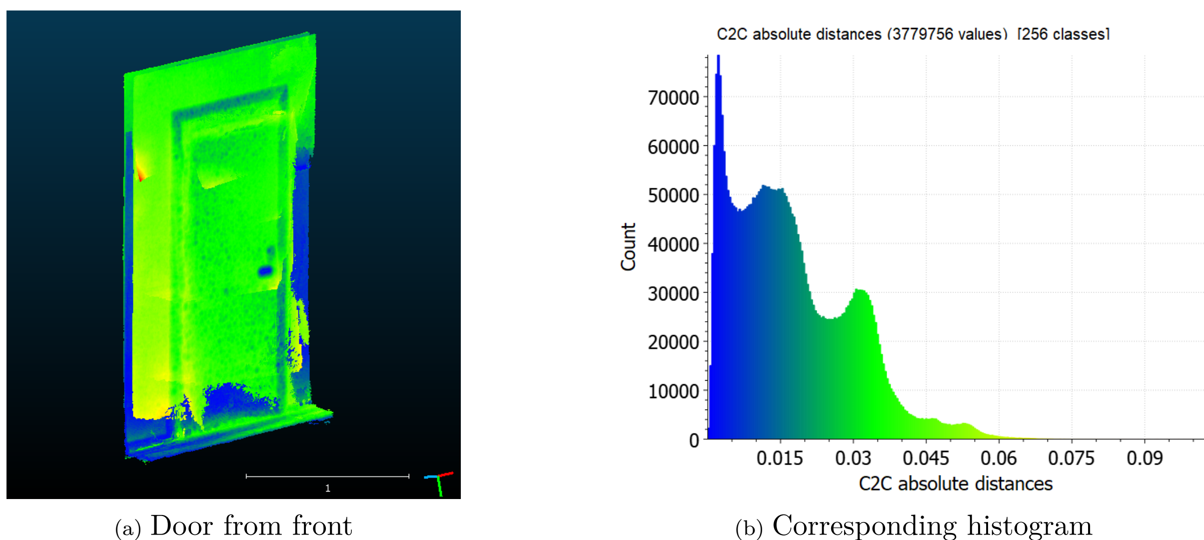
Full scan	Mean	SD	Max dist.
Cloud-to-Cloud distance, no filter	0.028704	0.039387	1.187234
Cloud-to-Cloud distance, with filter	0.025805	0.036387	0.636144
Difference	**0.002899**	**0.003000**	**0.551090**

Table 4: Summary of mean distance and standard deviation (SD) for C2C Cloud-to-Cloud distance done on the full scan with and without filter. Shown is both values for max distance, and the resulting difference. The unit is given in meters

5.1.3 Segmented: Door

To get a closer understanding of how certain areas of the scan performed, a closeup of the leftmost door is segmented from the filtered point cloud (from Section 5.1.2). Figure 16 presents the segmented door, with its corresponding histogram. The data can be found in subfolder **..>segmented>door**.

Looking at the scalar field at a smaller scale, it is possible to detect new trends. By tilting the wall to the right (Figure 16a), it is possible to see that the door/wall actually consists of several *layers*. These layers are visible in the histogram as well, as three distinct peeks (see Figure 16b).



(a) Door from front

(b) Corresponding histogram

Figure 16: Door segmentation, front

The results from distance computation on the door segment can be seen in Table 5. In this smaller section of the filtered Scan 2, the mean is 1.45 cm, and a standard deviation of 1.36 cm. The max distance in this area is at 10.32 cm, and can be seen in the red spot visible to the left in Figure 16a.

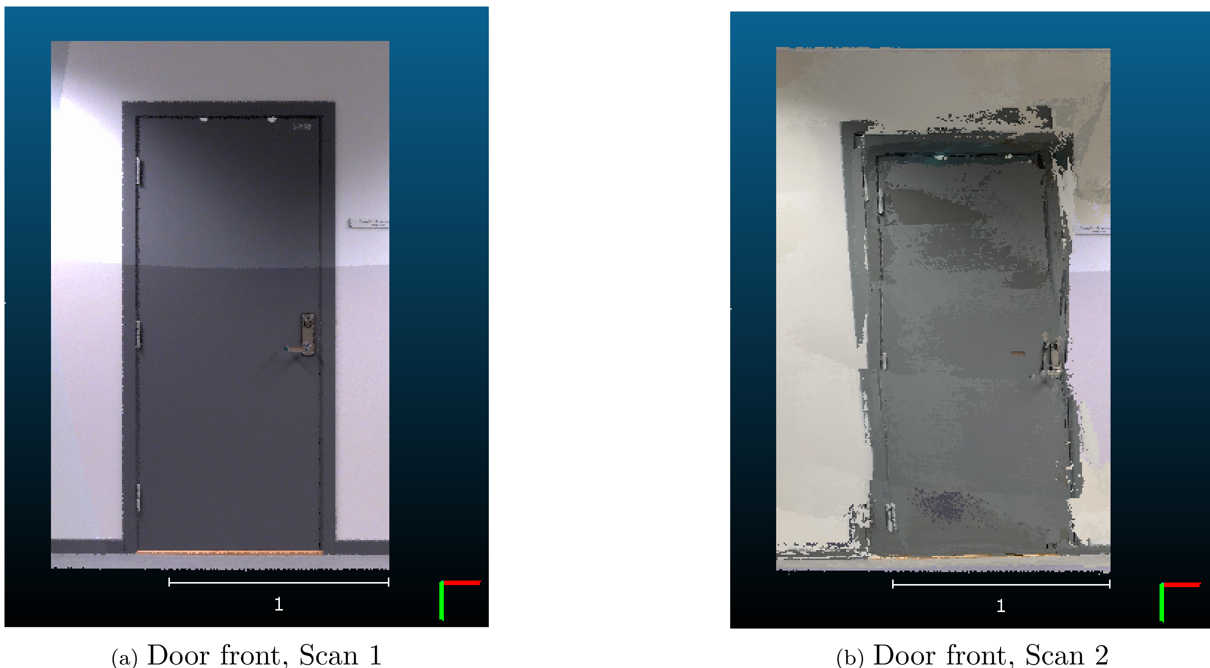
Overlap	Mean	SD	Max dist.
Cloud-to-Cloud distance	0.014521	0.013553	0.103184

Table 5: Mean distance and standard deviation (SD) for C2C Cloud-to-Cloud distance done on the door in Figure 16

These results highlight one of the clearest vulnerabilities of the matching process used to produce Scan 2. Since each RGBD-frame processed is only matched to its neighbor, "almost perfect" transformations will transform into drift over time at the camera moves. With method like this, more data/scans will provide a risk of more noise and more drift.

Additionally, similar to the full scan, the trend here is that the average distance from ground truth is below 2 cm, which is a good result. The backside is the relative high standard deviation, telling of a low reliability that each point will be correct.

Lastly, let us inspect how the actual color/RGB-values look at this segment. Figure 17 shows the two scans - Scan 1 (Figure 17a) and Scan 2 (Figure 17b) side by side. What can be seen is that the Scan 2 door has several areas that are wrongly matched - with some part of the door being tilted to the left. However, this tilted door is not detectable with the evaluation metric used in this chapter, since the depth-values of the wall/door are still the same.



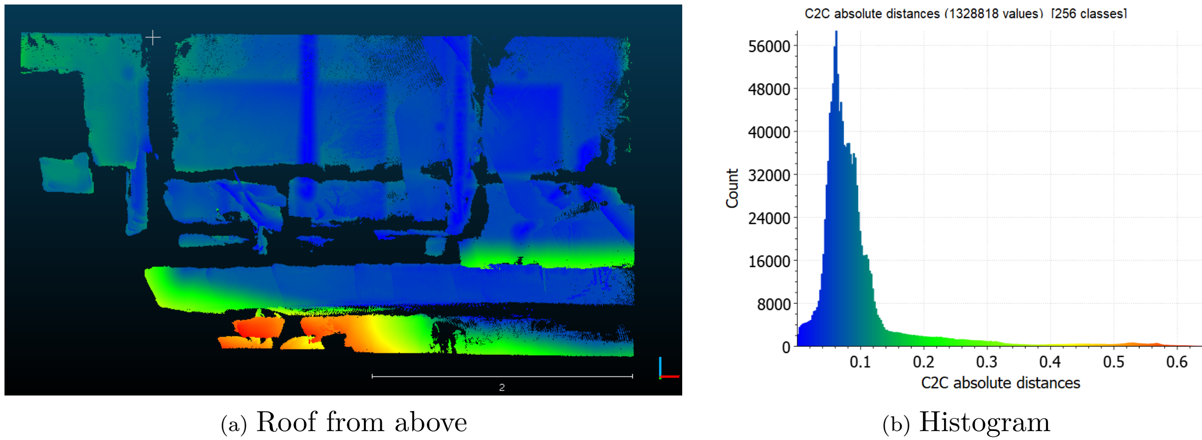
(a) Door front, Scan 1

(b) Door front, Scan 2

Figure 17: Door front, RGB comparison

5.1.4 Segmented: Roof

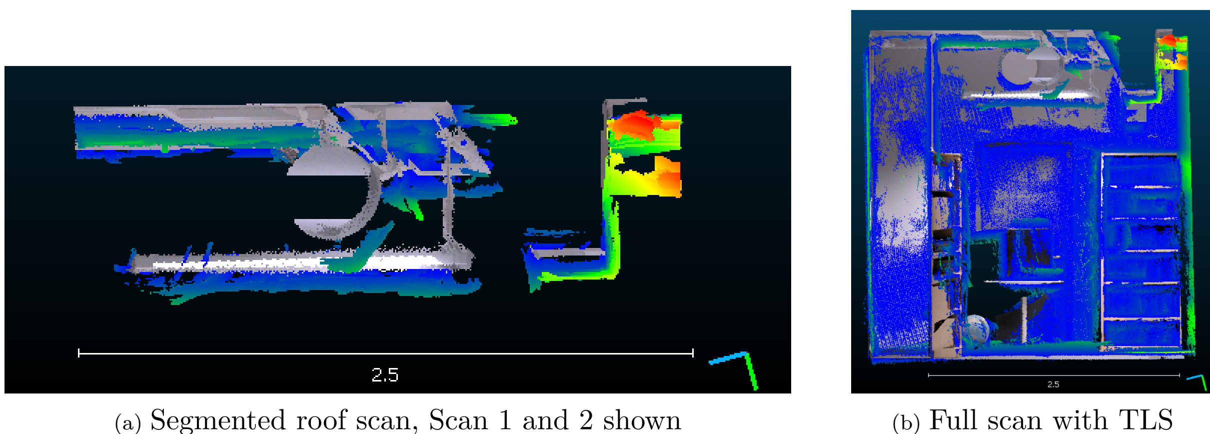
The segmented point cloud of the roof can be seen in Figure 18. This segment was chosen to be closer inspected because it is the area with the largest amount of error in Figure 15. The data can be found in subfolder **..>segmented>roof**



(a) Roof from above (b) Histogram

Figure 18: Roof from above with corresponding histogram

The general performance of the roof is worse than the full scan, with most points are at a distance just below 10 cm (see Figure 20b). When inspecting the roof from its side (see Figure 19a), we can observe a general trend that the whole roof is *shifted downwards*. One explanation for this could be that the two point clouds has been wrongly aligned in the vertical direction. However, when inspecting the full scan and - the interesting part - the floor, it can be seen that the floor is not shifted too far down at all (see Figure 19b). Therefore, these distance measurements should be considered as a correct evaluation. Some reasons for this shift in accuracy could be explained by the fact that the roof includes direct light-sources, which is known to disrupt LiDAR measurements. Another explanation could be that the roof was scanned with less frames than the rest. This should be further tested, to understand if it is a fault in the scanning procedure, a fault in the matching process or a fault in the sensor.



(a) Segmented roof scan, Scan 1 and 2 shown (b) Full scan with TLS

Figure 19: Left sideview of segmented roof and full scan, Scan 1 and 2 shown

Note, the horizontal displacement can also be explained that a fault in pairwise matching resulted in large portions of the roof layers ends up shifted too far down. This can be backed up by the fact that there is also blue areas of layers/points that are further up.

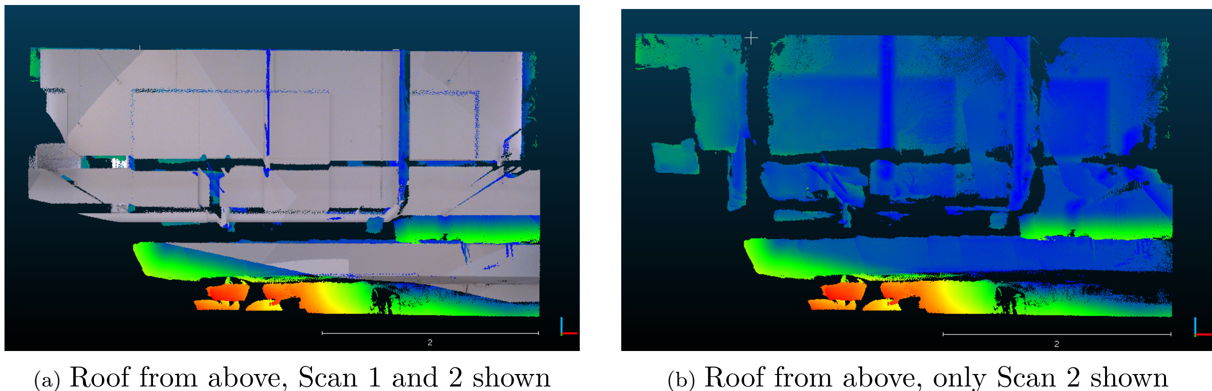
The results from C2C Cloud-to-Cloud distance on the scan in Figure 18 is shown in Table 6. The mean distance is at 9.81 cm, with a standard deviation of 8.58 cm. The max distance in this segment matches the max distance in the full scan, meaning that the local maximal distance in this segment is the global maximal distance.

(NOTE: the local max distance is actually higher than the global max distance (given in Table 3). This can be caused by the fact that these "max. distance" measurements are given as estimates before processing the distance.)

Overlap	Mean	SD	Max dist.
Cloud-to-Cloud distance	0.098072	0.085784	0.642753

Table 6: Mean distance and standard deviation (SD) for C2C Cloud-to-Cloud distance done on the roof in Figure 18

The explanation for this large error section becomes clear when comparing with the TLS scan. In Figure 20 we see that the area where the distance begins to worsen, is the same area that TLS does not have any direct corresponding points. This means that every distance of the points in this area from Scan 2 is calculated with reference to the closest part where Scan 1 stops.



(a) Roof from above, Scan 1 and 2 shown

(b) Roof from above, only Scan 2 shown

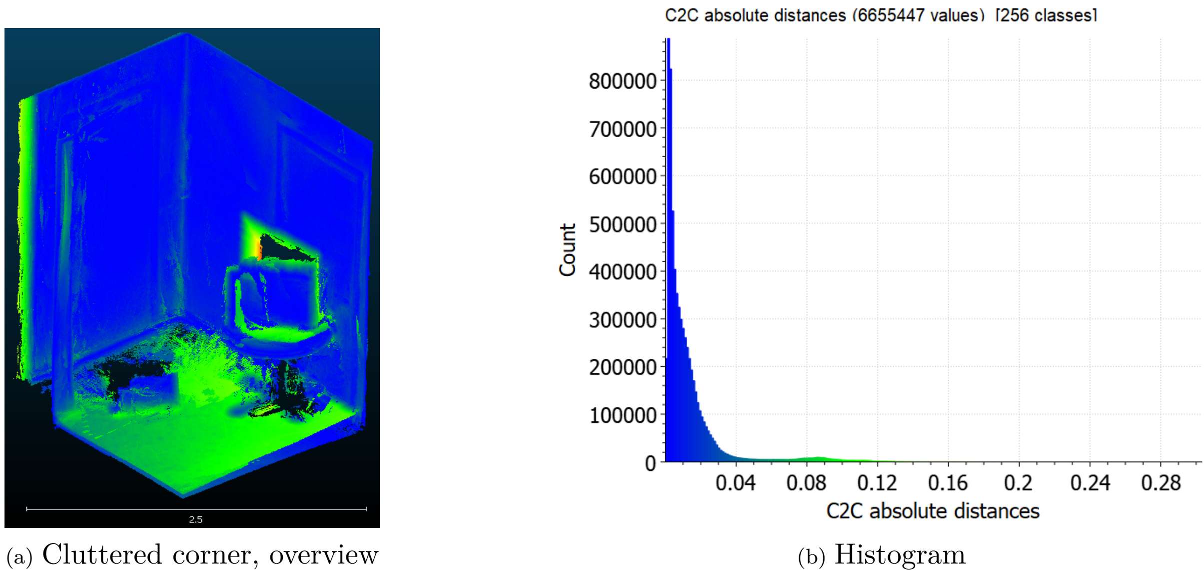
Figure 20: Roof from above with and without Scan 1

Since this part of the roof in practice has no base for comparison, these parts would ideally be cut out of the evaluation. This however has not been prioritized in this work. By removing these points, the mean and standard deviation is expected to improve slightly.

5.1.5 Segmented: Cluttered corner

For a closer inspection of the area of the room that has the most occlusion, we will look at the "cluttered" corner. Figure 21 shows an overview of the corner as a scalar, filtered point cloud, with its corresponding histogram. The data can be found in subfolder **..>segmented>clutteredcorner**

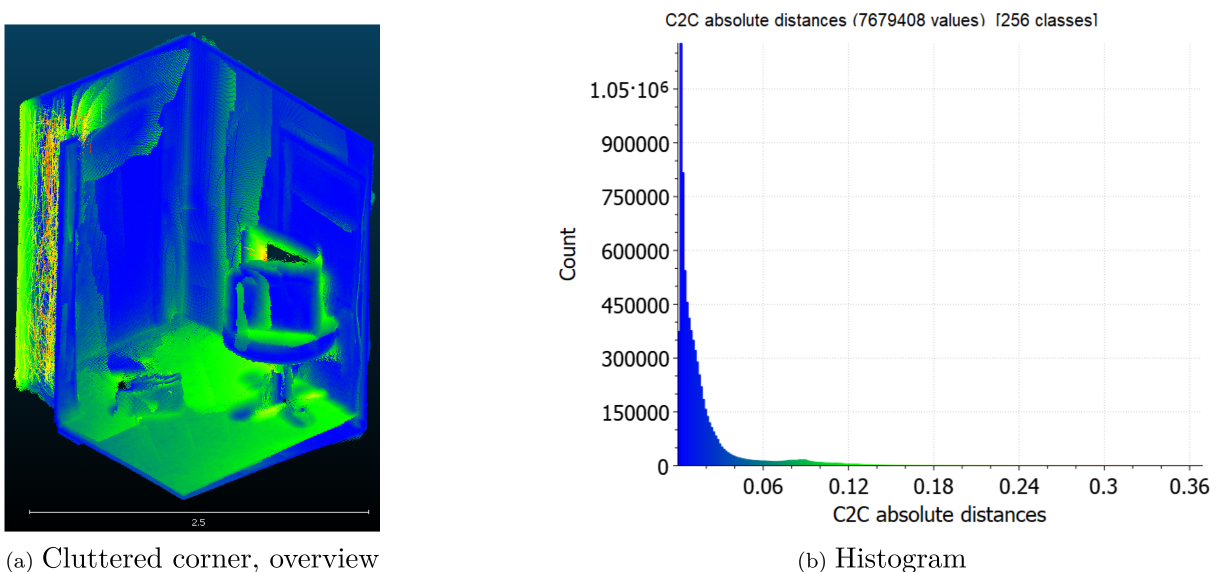
General remarks is that the point cloud has good performance on the walls - except for a missing wall to the left, a bad performing floor, and some errors and missing parts around the two occluding objects.



(a) Cluttered corner, overview (b) Histogram

Figure 21: Cluttered corner from front

Figure 22 shows an overview of the corner as a scalar, non-filtered point cloud, with its corresponding histogram. The reason why presenting this area non-filtered as well is to shed light on an occuring error due to the LiDAR sensor in the iPhone 12 Pro. This error will be called "*overflow*".

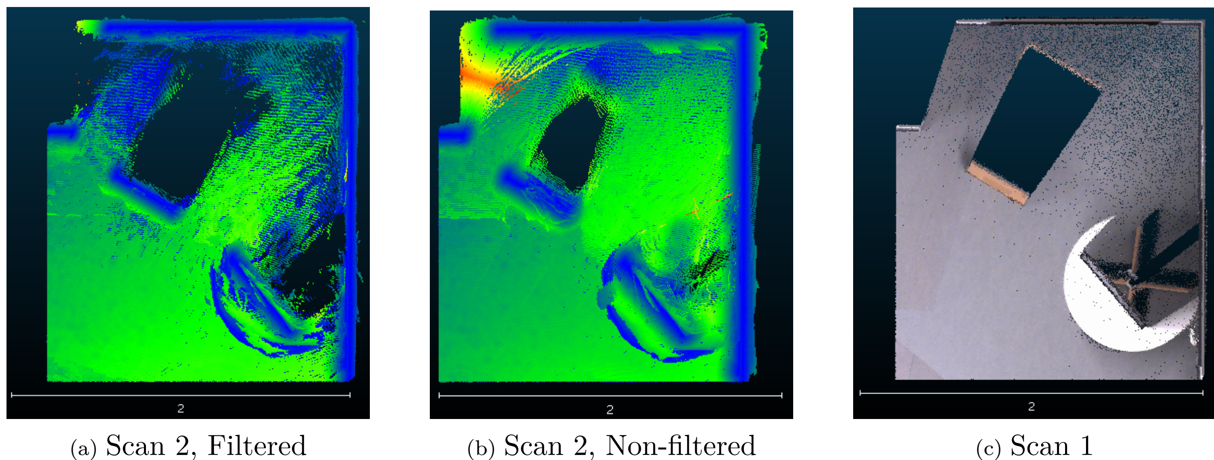


(a) Cluttered corner, overview (b) Histogram

Figure 22: Cluttered corner, non-filtered

The error of "overflow" is best explained looking at the two point clouds (filtered vs. non-

filtered) side by side from above - see Figure 23. Note that this trend follows the whole vertical axis up the wall, as one can see in Figure 22a. The interesting area of the scan is the red/yellow colored points in the top left part of the Figure 23b. Here we can see that *before* filtering, the scan suffered from a large portion of points being wrongly placed in floating air. This seems to not be a problem with matching, but rather a *problem in the input data*. When the iPhone camera and LiDAR sensor is aimed at objects and structure with a sharp edge, the depth-measurements in the area between two depth-levels (closest wall and the wall further away) tend to smear out - looking as "overflowing" water similarly as in a waterfall. The explanation for this is difficult to pinpoint, and should be taken into consideration when using the LiDAR depth-data captured by iPhone 12 Pro. Such faulty depth-measurements also negatively influences matching algorithms. Ideally these faults should be filtered out *before* processing. It might prove difficult to filter the input correctly all the time, but one promising beginning would be to do similar filtering as the SOR-filtering, to remove points that are too far away large areas of the RGBD-frame.



(a) Scan 2, Filtered (b) Scan 2, Non-filtered (c) Scan 1

Figure 23: Cluttered corner from above, filtered vs. non-filtered

Another trend in the point clouds viewed in Figure 23 is that there is missing areas behind the objects placed on the floor. Before executing the scan, it was expected that the mobility of mobile phones would make the scan executed by iPhone more complete in areas where TLS had missing areas (see Figure 23c) - since it is possible to walk around and behind objects, to capture every angle. Some of this extra captured data can be seen in Figure 23b, but dissappears after the filtering process, as one can see in Figure 23a. This is likely due to the fact of "overflow" happening a lot around objects, as well as an incomplete scan.

The cluttered area therefore suffers from noise like "overflow" which disappears after filtering, as well as direct missing scanned areas, as well as the objects not being very well reconstructed. This area is also the part where the floor has the highest misalignment.

Table 7 shows the evaluation metrics for the cluttered corner, with and without filtering - as well as the difference between the two. Mean distance in filtered is 1.37 cm and 2.02 cm SD, and a max distance of 30.80 cm. Mean distance in non-filtered is 1.80 cm and

2.75 cm, and a max distance of 36.54 cm. Considering the faulty points of "overflow", as well as missing ground truth comparison base, these results are good. The main problems are that the cluttered portion of the room is badly reconstructed by having large missing areas and tilted floor.

The difference of 4.2 mm (mean) and 7.23 mm (SD) shows that this cluttered section of Scan 2 benefitted more by the filtering than the full scan (by comparing the difference in Table 7 with Table 4)

Overlap	Mean	SD	Max dist.
Cloud-to-Cloud distance, with filter	0.013772	0.020230	0.307983
Cloud-to-Cloud distance, without filter	0.017972	0.027495	0.365369
Difference	**0.004200**	**0.007265**	**0.057386**

Table 7: Mean distance and standard deviation (SD) for C2C Cloud-to-Cloud distance done on the roof in Figure 21 and 22

5.2 Scan 1 vs. Scan A

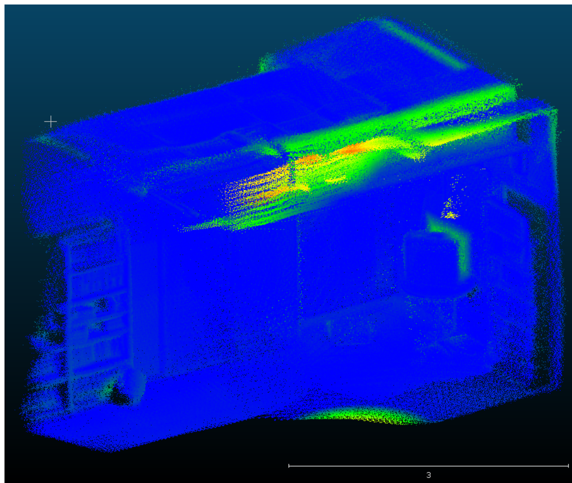
Lastly, a brief comparison between TLS and the point cloud captured by the free mobile app SiteScape (found in App Store on iOS) will be presented in this section. Figure 24 shows the scalar field of SiteScape, with no cap (Figure 24a and 24b), and with 0.2 m cap (Figure 24c and 24d). The data can be found in subfolder **..>scan1vsA**.

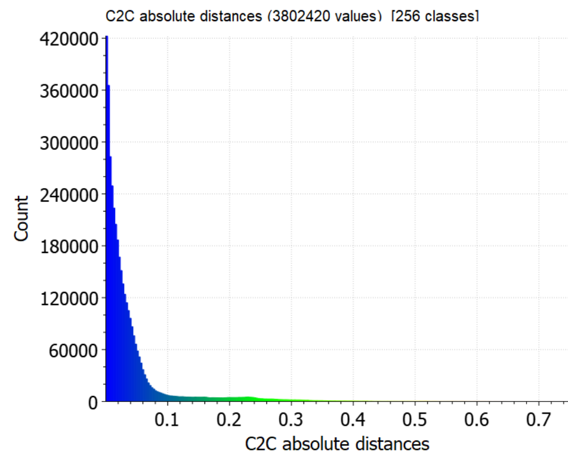
Table 8 shows the metrics for SiteScape. The evaluation done on the full scan with no cap gives a mean distance of 4.09 cm, SD of 6.55 cm and max distance of 79.60 cm. When capping on 0.2m, the evaluation is mean distance of 3.70 cm and SD of 4.90 cm. This data is significantly worse than the iPhone result evaluated in Section 5.1.2, which was 2.58 cm mean and 3.64 cm SD. However, inspecting the point cloud shows that this result is largely influenced by the gaps in Scan 1 - which the SiteScape scan has covered better.

Overlap	Mean	SD	Max dist.
Cloud-to-Cloud distance	0.040912	0.065546	0.795966
Cloud-to-Cloud distance [<0.2]	0.036956	0.048888	0.2

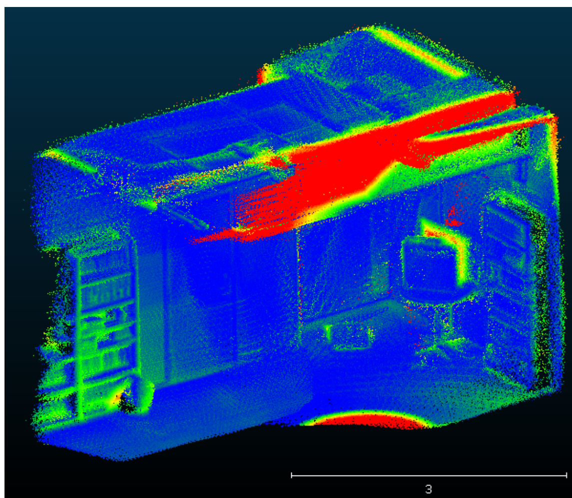
Table 8: Mean distance and standard deviation (SD) for C2C Cloud-to-Cloud distance done on the full scan by SiteScape, Figure 24

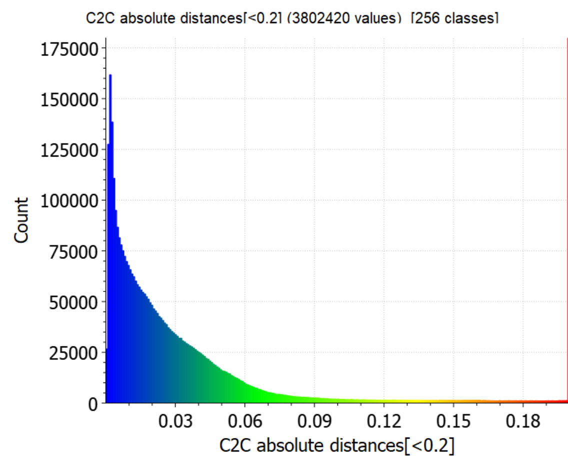
Figure 25 shows a great example of how the worst performing areas of the SiteScape scan is actually good. Figure 25a shows the segmented corner from above with both scans on top of each other, Figure 25b shows the same view with only TLS, and Figure 25c is with only SiteScape. The worst performing areas (marked with red/yellow in Figure 25a and 25c) is actually a result of the same argumented mentioned in Section 5.1.4 - that this


(a) SiteScape scalar field, no cap



(b) Corresponding histogram


(c) SiteScape scalar field, 0.2m cap



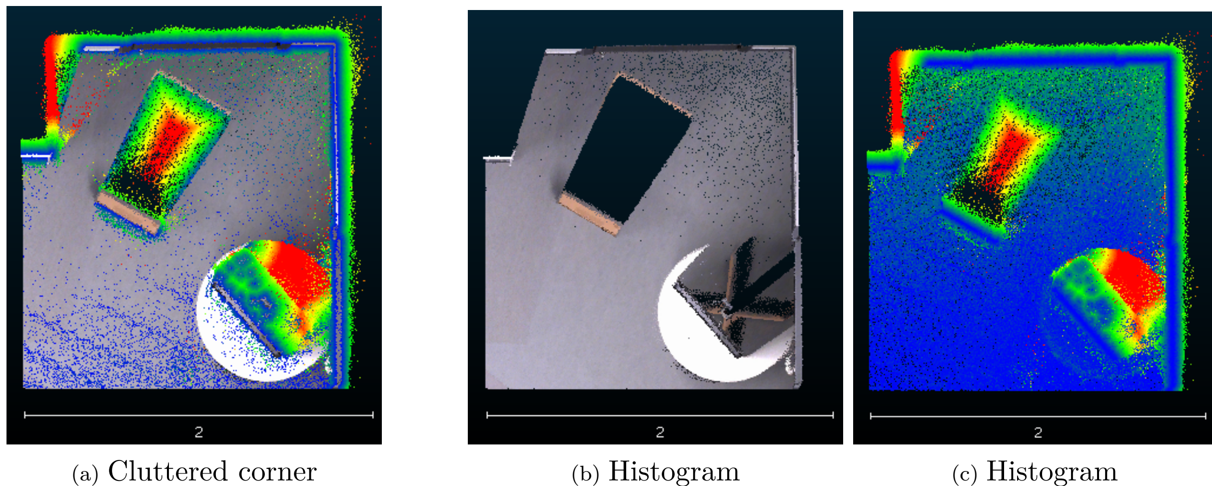
(d) Corresponding histogram

Figure 24: Full scan by SiteScape, with and without cap

is due to missing data in the TLS scan. In reality, these red areas are well placed. This SiteScape scan is the only on of the three scans evaluated that has completely captured the white table and the black box - as well as complete walls and floor. The only sign of holes in the scan is behind the brown box on the floor. NOTE: Inspecting the rest of the SiteScape scalar field, this same trend goes for the other significant red areas in Figure 24c, like on the floor, the hidden wall, the roof and the area behind the table and boxes. Some red points are correctly marked as errors, but that is sparse, floating areas with few points which would not be causing the big spike in Figure 24d. This close inspection of every red/yellow area is not presented thoroughly in this thesis, so it is up to the reader to check this out for themselves. Although the evaluation metrics of the SiteScape scan performs worse than Scan 2, the conclusion is that SiteScape scan performs better - since the bad performing areas are actually areas where SiteScape captured points neither Scan 1 or Scan 2 was able to capture. It is difficult to say what the mean distance and standard deviation actually is due to the missing areas in the "ground truth" (Scan 1),

but it is assumed that by doing another evaluation on a complete "ground truth" scan, the evaluation metrics would be better than 2.58 cm mean distance and 3.64 cm in SD.

The most clear drawbacks of this scan is that the scan has thick walls, a result of noisy depth measurements. and some signs of "overflow" and outliers. NOTE: the SiteScape scan has not been filtered with the SOR-filter used in Section 5.1.2, since it is assumed that commercial applications found in App Store provides a "complete/finished" solution that should not require additional processing.



(a) Cluttered corner (b) Histogram (c) Histogram

Figure 25: Cluttered corner, non-filtered

6 Discussion

This chapter will cover three main topics (same as from Chapter 3), namely section 6.1 to talk about factors contributing to the *data* quality, then section 6.2 to discuss what could be done differently with the *processing* step, and lastly section 6.3 to discuss difficulties and faults with the *evaluation* method

6.1 Data

6.1.1 Error in input RGBD-pairs, "overflow"

With the provided depth data captured with Apple's LiDAR sensor, the depth values were not reliably correct. One method to minimize this was to apply Apple's «confidence map»-values as a filter to the depth maps (i.e. only using depth-values with HIGH confidence), but this still didn't remove important inaccuracies like the inaccuracies of "overflow" mentioned in the evaluation chapter.

By using the RGBD-matching method applied in this thesis, these inaccurate depth-values led to bad transformation-matrix solutions between two subsequently frames. It would be interesting for another time to figure out what leads to the obfuscation of the depth values in each frame. Some inaccuracies can be expected to be correlated with the distance from the camera to the object, however, this would not account for the inaccuracies at surfaces with a similar distance from the camera. It is difficult to improve these initial matches, since there is little knowledge to understand what part of the depth map can be "trusted" more than other parts.

6.1.2 Blurry images and laggy scan

One problem that was difficult to do anything about was that the shutter speed could not be adjusted through Apple's ARKit. This becomes a problem since the shutter speed correlates to how blurry the captured images are if the camera is moved during the capture moment. This means that, worst case, images of the same object might not be recognizable as a match. When the scan was executed too hastily or with too much movement between frames, the images became blurry and could result in bad image matches. To partly solve this, the scan was executed very carefully, with a very slow movement between frames. Since a realistic scan consist of movement around the room and around objects, this has a high risk of often storing blurry images.

The Xcode-project used to capture the data from each scan stored the data in a time-costly way, and when this was done, the frame froze in the application. This left the scan feeling "laggy", and made the scanning experience bad.

6.1.3 Low resolution

The resolution of the depth data from Apple's ARKit is 256x192, and therefor limits how much depth information can be represented per frame. The further away the camera is from the object, the fewer details can be represented in the 3D model. (since the distance between two objects in an image is further the farther away). Given this, the closer the camera would be to the scanned object, the better. This however is a problem when it comes to the actual image-matching since the closer the camera is, the less distinct features can be captured in one image, which can lead to bad matches since an image pair might have more than one unique accepted transformation.

6.2 Processing

6.2.1 Simple matching structure

Scan 2 produced relatively good results of the indoor environment, but this could likely be due to a carefully executed scan with little movement, and good overlap. The simple method used to produce Scan 2 will be vulnerable to accumulated errors if image pairs has bad transformations that are not filtered out. Some examples of how this was seen in Scan was the *thick walls* or *skewed planes* which was seen on the floor. As mentioned, the inaccuracies from the input images negatively influenced the result, by providing potential bad transformation results between each image pair. Note that Open3D has a robust method already implemented for Azure Kinect and Intel RealSense. When trying to adjust this code with the input data from Apple, I was unable to produce any good results at all, and a simple implementation of their code was used instead.

6.2.2 Online vs. offline processing

Before working on this project, the idea of a mobile phone with online reconstruction seemed very appealing, due to the instant feedback of what part of the scan that had been scanned, as well as the flexibility to move around through cluttered areas. However, the aim of the article was shifted towards processing done after the scan, in Python. The idea was that offline processing could utilize the full potential of both the LiDAR data and the color images, with no limitation of how long it would take.

6.2.3 Reproduction problems: RANSAC

When running the code documented in Section 3.3.3, the output point cloud sometimes differ. A reason for this could be that these matching methods consist of a *random* matching step - RANSAC. The points it uses to find a match is random, and can therefore

be varying from run to run.

6.2.4 Runtime

The processing method relies on two matching methods, RANSAC and ICP. RANSAC is a fast matching method, while ICP is considered complex, with potential n square running time, where n is the number of points in each point cloud. This was not of importance for this experiment, since the dataset was small (93 frames). If the dataset is increased, the runtime will grow squared.

6.3 Evaluation

6.3.1 Incomplete "ground truth"

The main evaluation method in Chapter 5 was C2C Absolute distance, which calculates the distance between points in two overlapping point clouds. The problem was when the comparison scan (Scan 1) had holes and had not covered all areas of the room, and the absolute distance did still calculate distance in these missing areas. This means that the evaluation metrics most likely is a bit off the real metrics. However, since this challenge would lead to an *increased* calculated distance, so the evaluated performance is likely to be a *worse* estimate than the actual performance. One solution to to provide a more correct picture of the evaluation, segments of the scan was inspected as well. An ideal evaluation would have excluded areas where two scans did not have enough in common.

6.3.2 Chosen commercial applications

As mentioned in Chapter 4, the Pix4D scans were not included in the evaluation Chapter 3.4 due to its apparent flaws and noise. This result was processed without any influence, and was therefore difficult to understand how it could be improved. Since this software (Pix4D cloud) still was in BETA mode at the time of executing the experiment, it is understandable that the results are not stable. Considering it's been 8 months since the experiments was executed, a new evaluation should be done with newer versions, before concluding anything.

Commercial apps like SiteScape provides a quick and easy way to scan and export a processed pointcloud/other 3D models without much experience. However, these applications are often not accepted for more professional processing. Professionals who is looking to elevate their 3D reconstruction results, would likely not be convinced by the results that are now at the market - at least if you do not have any insight in how things are being done. This is one reason why open source could be an important way to go, since it provides experts possibilites to adjust parameters how they see fit.

6.4 Further work

For anyone going to work further on such a project as this, some final notes are presented here.

For improvement of the processing method, note that Open3D has a open source repository ([17]) that can be used as well. When trying to produce results with my data and Open3Ds repo, the results proved worse/very bad compared to the source code used on Scan 2. Therefore I found it easier to make my own, simpler implementation of the rough idea of Open3D's registration. For further work, an implementation of Open3D's full reconstruction method should be implemented for best result both in scaleability and accuracy.

Some explanation for why the full implementation did not work could be:

- different configurations. Open3D's repo is created fitted to specific RGBD cameras. Note, one can of course adjust this, but that requires a good understanding of the influence of each parameter in CONFIG-file, which I was not able to solve.
- Assumably Open3Ds method, which is more complex, depends on a sensor with higher framerate (leading to more overlap), as well as higher data dimensions - with less noise.

The filtering step would also be better fitted to be included directly in the processing step. As mentioned in the work, the filtering step was done using C2C, and should be possible to implement in Python. The filtering step could be done before or after the matching process, or a combination of both.

Another improvement would be to find an evaluation method that is not influenced as much by false distance measurements in areas where the Trimble SX10 scan/"ground truth" had missing data. This means that the data used to evaluate the accuracy will be slightly wrong, due to correct points being labelled as incorrect. To provide a better evaluation, these areas should be filtered out in the comparison, or a new scan should be done with the Trimble SX10 to provide a complete "ground truth" scan. The reason why the evaluation was still done on the incomplete "ground truth" was because it provided a good way to get an overview of the general performance without extra processing steps to filter out portions of the scans.

7 Conclusion

This article has presented the current best result by using the new iPhone 12 Pro utilizing the integrated LiDAR sensor to generate point clouds. Since the release one year ago, several applications have been launched to apply the integrated LiDAR sensor for indoor reconstruction.

The open source processing solution (Scan2) gave a 2,58cm offset and 3,64cm standard deviation after filtering. This result was produced with a matching procedure with óne iteration through the dataset, comparing and matching each neighboring RGBD pair, and is expected to produce even better results if a full matching procedure (like the one from Open3D) is implemented.

A commercial application found in App Store (SiteScape, Scan A), was evaluated as a benchmark for the performance of the market. This scan performed better than the open source implementation, which was to be expected. Note that due to problems with the base for evaluation, accurate metrics for the performance of this scan was not established, but it is assumed to be better than the performance of own implementation.

For use of small, indoor 3D modelling, low-cost equipment such as the LiDAR-sensor in the newest iPhone 12 Pro, provides very promising results. Commercial solutions found in the App Store provides a great way for non-experts to scan their surroundings. For people interested in testing out Apple's LiDAR data with their own code, open source documentation of ARKit and Open3D provides quick results with simple methods. The open source implementation in this thesis provided good performance in the scanned corner, but needs to be further worked on to be robust for larger areas, more movement in occluded areas and the possibilities for real-time feedback.

8 References

[1] Michael Zollhöfer **andothers**. "State of the Art on 3D Reconstruction with RGB-D Cameras". **in**: *Computer graphics forum*. **volume** 37. 2. Wiley Online Library. 2018, **pages** 625–652.

[2] Marte Haugestøyl Vinje. *Literature Review: 3D Scanning Techniques for Indoor Environments*. Not published, but delivered as my project thesis through NTNU Inspera. **december** 2020.

[3] Qian-Yi Zhou, Jaesik Park **and** Vladlen Koltun. "Open3D: A Modern Library for 3D Data Processing". **in**: *arXiv:1801.09847* (2018).

[4] Thomas Klauer **and** Bastian Plaß. "Point Cloud Capturing and AI-based Classification for as-built BIM using Augmented Reality". **in**: (2021).

[5] A Spreafico **andothers**. "The Ipad Pro Built-In LIDAR Sensor: 3d Rapid Mapping Tests and Quality Assessment". **in**: *The International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences* 43 (2021), **pages** 63–69.

[6] Gregor Luetzenburg, Aart Kroon **and** Anders A Bjørk. "Evaluation of the Apple iPhone 12 Pro LiDAR for an Application in Geosciences". **in**: *Scientific reports* 11.1 (2021), **pages** 1–9.

[7] SiteScape. *SiteScape - LiDAR 3D Scanning*. https://apps.apple.com/us/app/sitescape-lidar-3d-scanner/id1524700432. [Online; accessed 23-April-2021]. 2020.

[8] SiteScape. *INTRODUCING SITESCAPE PRO, LIDAR APP FOR CONSTRUCTION*. https://lidarnews.com/press-releases/introducing-sitescape-pro-lidar-app-for-construction/. [Online; accessed 20-December-2021]. 2021.

[9] Geospatial. *Trimble SX10, product*. https://geospatial.trimble.com/products-and-solutions/trimble-sx10. [Online; accessed 20-December-2021]. 2016.

[10] Pix4D. *Pix4D's Image acquisition instructions*. https://support.pix4d.com/hc/en-us/articles/115002471546-Image-acquisition. [Online; accessed 15-December-2021]. 2021.

[11] InsideGNSS. *Trimble SX10, price*. https://insidegnss.com/trimble-rolls-out-sx10-scanning-total-station/. [Online; accessed 20-December-2021]. 2016.

[12] Pix4D. *Pix4D Catch*. https://www.pix4d.com/product/pix4dcatch. [Online; accessed 15-December-2021]. 2021.

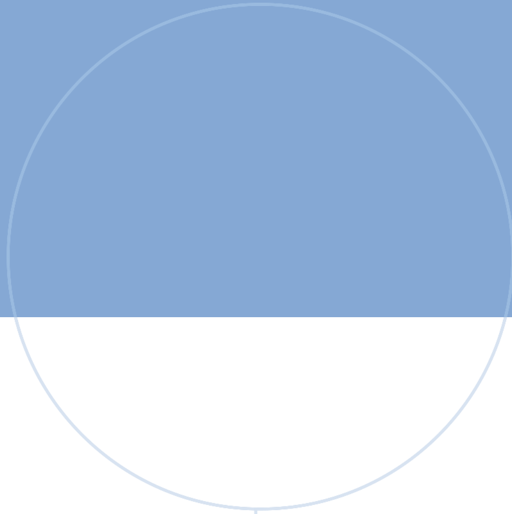
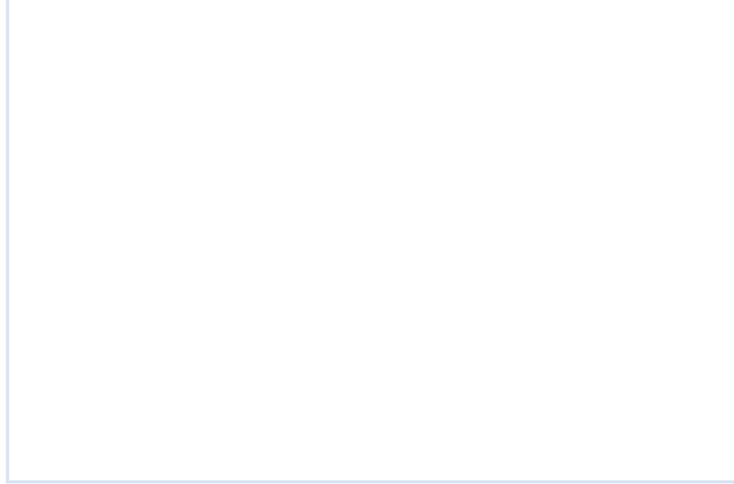
[13] Pix4D. *Pix4D Cloud pricing*. https://www.pix4d.com/pricing/pix4dcloud. [Online; accessed 15-December-2021]. 2021.

[14] Apple. *ARKit Documentation: Displaying A Point Cloud Using Scene Depth*. https://developer.apple.com/documentation/arkit/environmental_analysis/displaying_a_point_cloud_using_scene_depth. [Online; accessed 15-December-2021]. 2021.

[15] Open3D. *Open3D Tutorial - Reconstruction System*. http://www.open3d.org/docs/release/tutorial/reconstruction_system/index.html. [Online; accessed 15-December-2021]. 2021.

[16] CloudCompare. *CloudCompare - 3D point cloud and mesh processing software*. https://cloudcompare.org/. [Online; accessed 15-December-2021]. 2021.

[17] Open3D. *Open3D GitHub*. https://github.com/isl-org/Open3D. [Online; accessed 15-December-2021]. 2021.

NTNU

Norwegian University of
Science and Technology