Advanced Vision Practical

Xiaoyang Wang
s1839441

Yanting Zeng
s1829742

1 Introduction

With the advent of cheap RGB-D sensors like Kinect sensor, depth images are easier to obtain and tasks relying on such sensors such as 3D reconstruction have become much more feasible than before. In fact, the main theme of this report is to reconstruct Prof. Bob's indoor office based on the provided 40 consecutive frames of point clouds. Specifically, transformation matrices that link those frames should be estimated and works need to be done to fuse them together under an unified coordinate system. Therefore, the rest of this report will go through the details of each step and hopefully fully explain the methods we've used so far.

2 Data Preprocessing

The raw point clouds have a lot of noises that we do not want to retain, so we apply steps in the following order to remove them:

2.1 Remove Voxels Corresponding to Image Edges

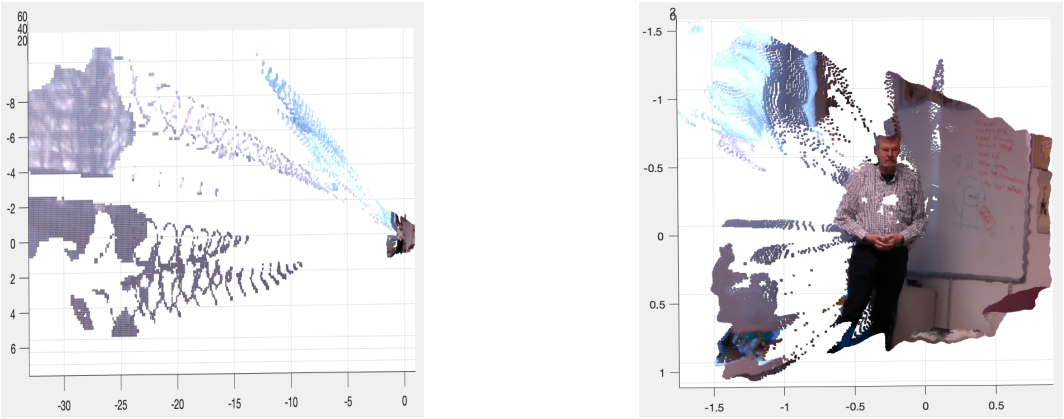
There are problems with the 3D sensing algorithms at image boundaries due to lens distortion, resulting in bad data values (e.g. bad colours, bad 3D point positions). Removing the pixels and associated 3D points from the edge of the image is a simple way to avoid any bad data that might be there. Example effect on the right edge of the zoomed-in image is shown in Figure 1.

(a) Zoomed-in image 27 before removing edge pixels (b) Zoomed-in image 27 after removing edge pixels

Figure 1: Comparison of zoomed-in image 27 before and after removing edge pixels

2.2 Thresholding Z Coordinate

In some frames, points coming from outside the window serve no purpose for our reconstruction task and hence need to be removed from the original point clouds. Our method for dealing with this kind of noise is to clip off all points which lie beyond a certain distance on Z axis ($Z > 4$), as illustrated in Figure 2.



(a) Frame 27 before thresholding Z

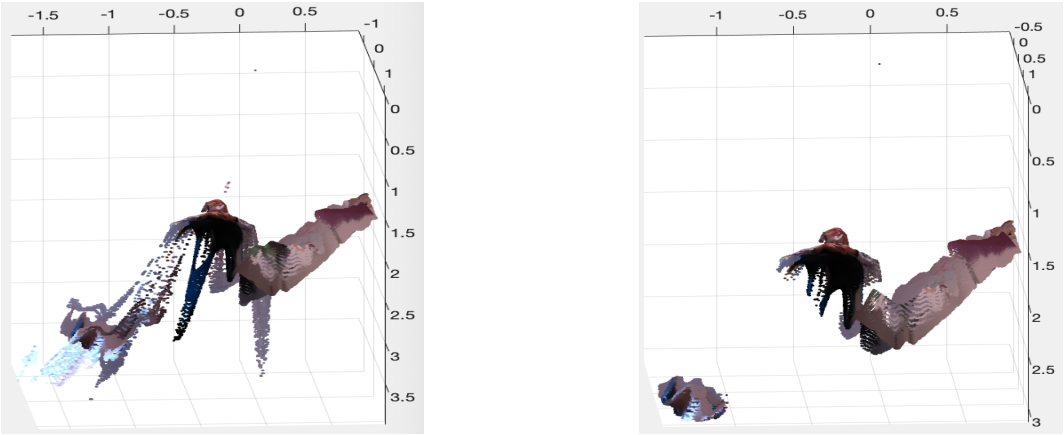
(b) Frame 27 after thresholding Z

Figure 2: Comparison of frame 27 before and after thresholding Z

2.3 Remove Flying Voxels

There are some flying voxels in the point clouds that lie in the middle empty space between foreground objects and background, which can interfere our reconstruction task. To identify and remove them, we count the number of neighboring points around each point within a pre-specified radius and remove those who do not have enough neighbors. To accelerate the speed of this algorithm, we apply K-D tree (1) to reduce the number of candidate points to be considered. Example of the effect of this algorithm can be found in Figure 3.

After reviewing David Strömbäck and Xingyu Jin's report, we found that their method of removing flying voxels shows comparable performance with ours but is much faster. Therefore, we decide to migrate to their method later for the demo because of the 30 minutes time limit. Briefly speaking, their method removes outliers in a 11×11 neighborhood around each point. Since the search space is constrained to such a small region around each point, the speed is greatly boosted compared to global nearest neighbor search.



(a) Frame 27 before removing flying voxels

(b) Frame 27 after removing flying voxels

Figure 3: Comparison of frame 27 before and after removing flying voxels

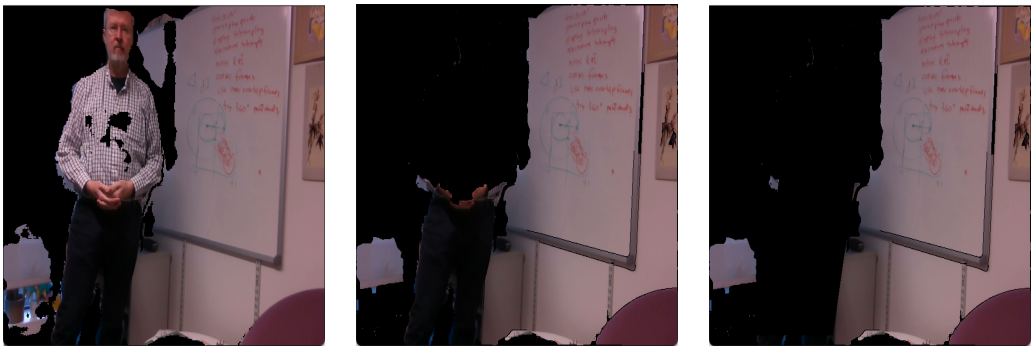
2.4 Dig Out Prof. Bob's Body

Frame 27 contains Prof. Bob's body, which does not belong to the original office and serves as an obstacle. Therefore, we apply image processing techniques as well as a manually selected bounding box to remove his body from the scene.

For the upper body, we first use canny edge detector to outline the edges of Bob's upper body. Then we dilate these edges and fill the holes of the image to get the Bob's full upper body. Lastly, we erode the image a little bit to slightly shrink the size of the upper body and put it in our mask.

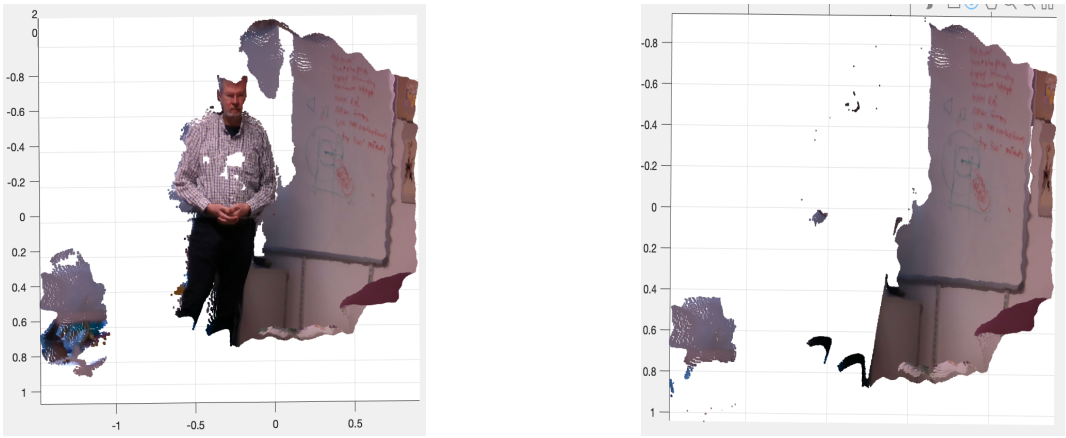
For the lower body, since Bob's pant has no texture at all, it's hard to apply the above techniques to distinguish Bob's legs from the background. Therefore, we manually draw a bounding box around Bob's legs and put this box in our mask.

The demonstration of the results of digging out Bob's body can be found in Figure 4 and 5.



(a) The whole body in the scene (b) Upper body removed (c) Lower body removed

Figure 4: Comparison of image 27 before and after digging out Bob



(a) Frame 27 before removing Bob (b) Frame 27 after removing Bob

Figure 5: Comparison of frame 27 before and after digging out Bob

3 Transformation Estimation

This section explains the process of firstly finding all sift point matches, then filtering out mismatches and finally applying RANSAC and ICP to obtain the transformation for consecutive frames.

3.1 SIFT Points detection and Mismatch Filtering

We first detect all sift points in two consecutive frames by sift detector in *vl_feat* library (2). Then we search for possible matches by comparing sift descriptors. However, there are numerous mismatch pair which can not be detected only by setting a single distance threshold. The mismatches tend to be one-to-many mapping which can be shown by Fig.6. The geometric feature is quite similar for each fence, which makes it difficult to find a right match for the fence top in the left image. In our task, the same situation happens in most frames. For example, the geometric shape of books can be quite similar. The one-to-many mismatch confuses our algorithm used to calculate transformation.

Algorithm 1 SIFT points detection and mismatch filtering

Input: $pc1$, $pc2$: PointCloud1 and PointCLoud2 to be processed; $siftDistTh$: Threshold distance to determine sift points match; $validDistRatio$: Threshold distance to check if sift match is valid;

Output: $validMatch$: Valid SIFT matches

1: Extract RGB images $img1$ and $img2$ from $pc1$ and $pc2$;
2: Apply sift detector in $img1$ and $img2$ to collect sift points $siftSet1$ and $siftSet2$;
3:
4: // Find all possible sift pairs
5: **for** each $point1$ in $siftSet1$ **do**
6: **for** each $point2$ in $siftSet2$ **do**
7: Calculate L2 distance $dist$ between descriptors of $point1$ and $point2$;
8: **if** $dist < siftDistTh$ **then**
9: ($point1$, $point2$) is a candidate match and being appended in $allMatch$;
10: **end if**
11: **end for**
12: **end for**
13:
14: // Filtering out mismatches
15: **for** ($pt1$, $pt2$) in $allMatch$ **do**
16: Find the closest pair involving $pt1$ in $allMatch$ and obtain their distance $dist1$;
17: Find the second closest pair involving $pt1$ in $allMatch$ and obtain distance $dist2$;
18: **if** $dist1/dist2 <= validDistRatio$ **then**
19: Keep the closet pair involving $pt1$ and append it in $validMatch$;
20: **end if**
21: Delete all pairs involving $pt1$ (i.e. ($pt1$, x) $\forall x \in siftSet2$) in $allMatch$.
22:
23: Find the closest pair involving $pt2$ in $allMatch$ and obtain their distance $dist1$;
24: Find the second closest pair involving $pt2$ in $allMatch$ and obtain distance $dist2$;
25: **if** $dist1/dist2 <= validDistRatio$ **then**
26: Keep the closet pair involving $pt2$ and append it in $validMatch$;
27: **end if**
28: Delete all pairs involving $pt2$ (i.e. (x, $pt2$) $\forall x \in siftSet1$) in $allMatch$;
29:
30: **end for**
31: **return** $vlidMatch$

To solve the problem of mismatch, we apply a algorithm to compare, for each sift point, its best match and the second best match. We consider the best match as a valid match if the best match is much better than the second best one by comparing their distances. Otherwise, we abandon all pairs containing the target sift point. For example, we have a sift point A in the first frame, which pairs with two sift points B (best match) and C (second best match) in the second frame under some threshold. Then, for the two pairs A-B and A-C, what we do is to compare the distances of A-B and A-C. If A-B is significantly better that A-C. which means A-B has a much higher probability of being the correct match for point A. Then, we can keep A-B and throw away A-C. However, if A-B and A-C are equally good match according to their distance measure, we have to abandon them both because we cannot actually find a reasonable match for A. The whole process is shown from line 13 to 26 in Algorithm 1.

Figure 6: An example for feature point mismatch from IVC course material

Fig.7 shows the relation of distance ratio and probability density of both correct and incorrect matches. It points out ratio around 0.7 can lead us to find most correct points without including many mismatch. Therefore, we set 0.7 as a threshold in our algorithm.

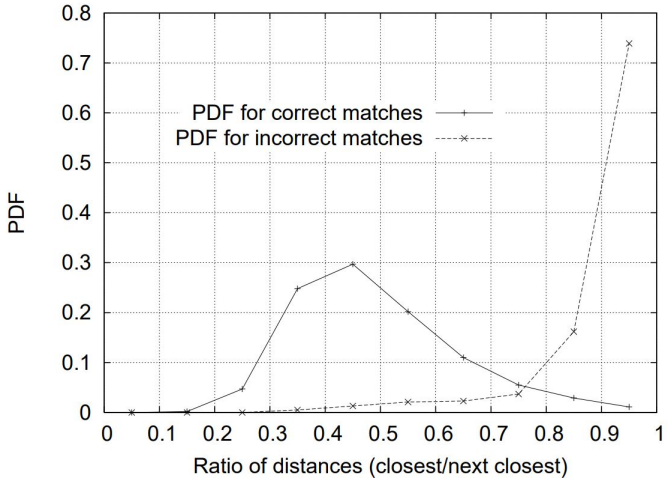


Figure 7: Relation between distance ratio and PDF for feature point matches. Figure cited from (3)

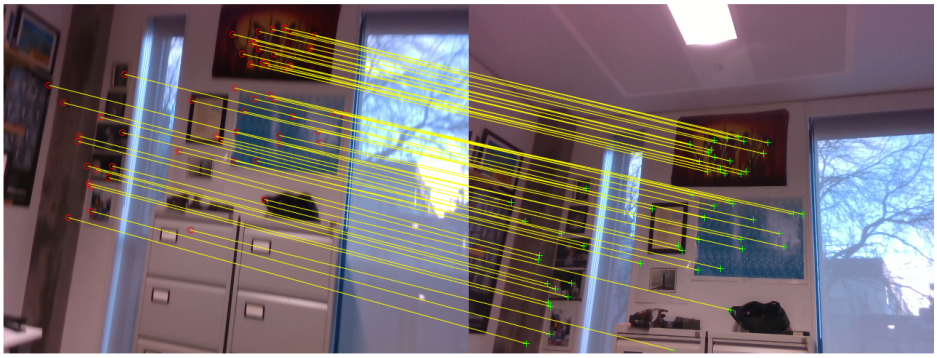
3.2 RANSAC

This section is to obtain the best transformation matrix using our filtered sift matches. This is done in an iterative process by RANSAC algorithm as shwon in Algorithm 2. In each iteration, we randomly sample some sift pairs and input them to ICP algorithm to get a transformation matrix. Then, all the sift points from the second frame is transformed by the matrix to the first frame. Then, we count number of inliers within the distance threshold. The final model is determined as the one with most inliers. One example is shown in Fig.8 that (a) is all the valid match in frame 19 and 20 while (b) shows some outliers by RANSAC.

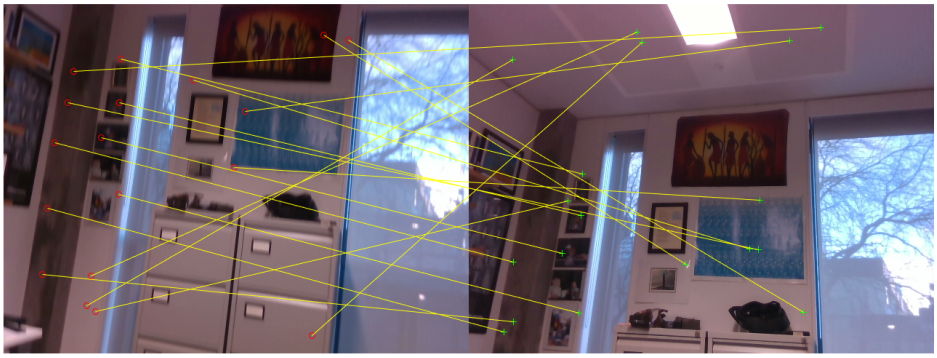
Algorithm 2 RANSAC algorithm with ICP

Input: $validMatch$: All the valid sift pairs indexed back to point cloud locations; $inlierDistTh$: Distance threshold for inlier detection; N_i: number of iterations; N_s: sample size;
Output: $MatT$: Transformation matrix obtained from RANSAC

$I \leftarrow 0$;
while $I < N_i$ **do**
    Randomly sample N_s sift pairs from $validMatch$;
    Apply ICP to calculate transformation matrix Mat based on sampled sift pairs;
    Transform all sift points from one frame to another based on Mat;
    Calculate all the distance between each pairs.
    Count number of inliers with the criterion $inlierDistTh$.
    $I \leftarrow I + 1$
end while
Set $MatT$ to Mat which has most inliers;
return $MatT$



(a) Valid sift matches illustration



(b) Sift mismatch illustration

Figure 8: Illustration for correct sift pairs and some mismatches

4 Frame Fusion

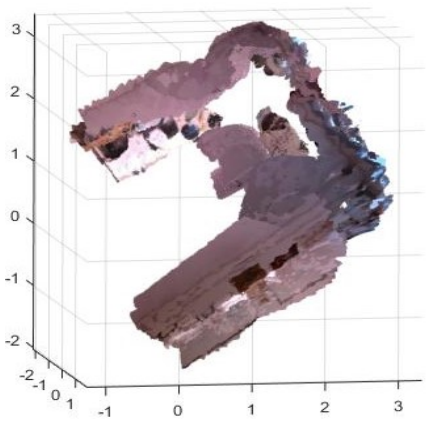
4.1 Algorithmic Procedure

Once all transformation matrices $MatTs$ have been found, the next step is to transform all point clouds to the coordinate system of the first frame and then fuse them together. The relevant pseudocode is given in Algorithm 3.

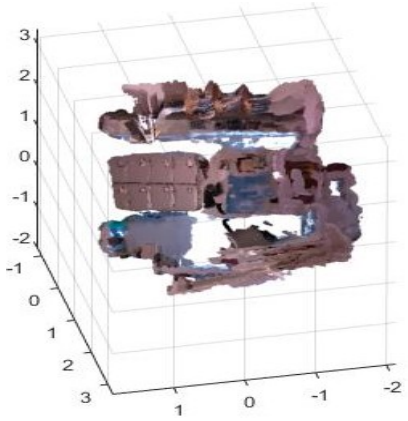
Algorithm 3 Fuse all frames with transformation matrices obtained previously

Input: $MatTs$: List of 39 transformation matrices obtained from RANSAC previously; pcs: List of all point clouds indexed from frame 1 to 40;
Output: pc_{merged}: The resulting point cloud with all frames merged;

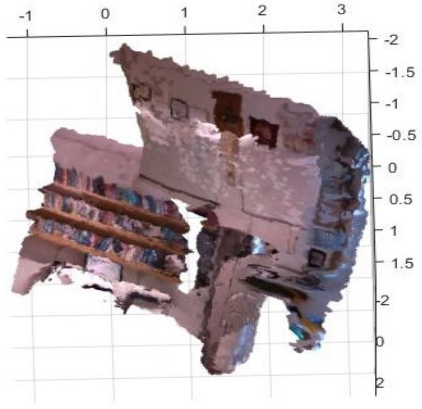
$pc_{merged} = pcs[1]$;
$I \leftarrow 40$;
while $I > 1$ **do**
    $pc_I \leftarrow pcs[I]$;
    for $i = I - 1 \rightarrow 1$ **do**
        pc_I = homogeneous_transformation($MatTs[i], pc_I$)
    end for
    pc_{merged} = pcmerge(pc_{merged}, pc_I)
end while
return pc_{merged}



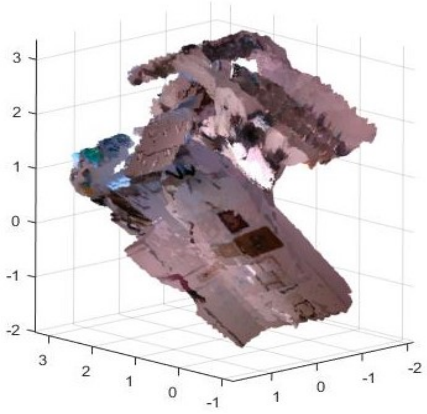
(a) The top view of the reconstructed office



(b) The middle view of the reconstructed office



(c) The left view of the reconstructed office



(d) The right view of the reconstructed office

Figure 9: The exhibition of our final result

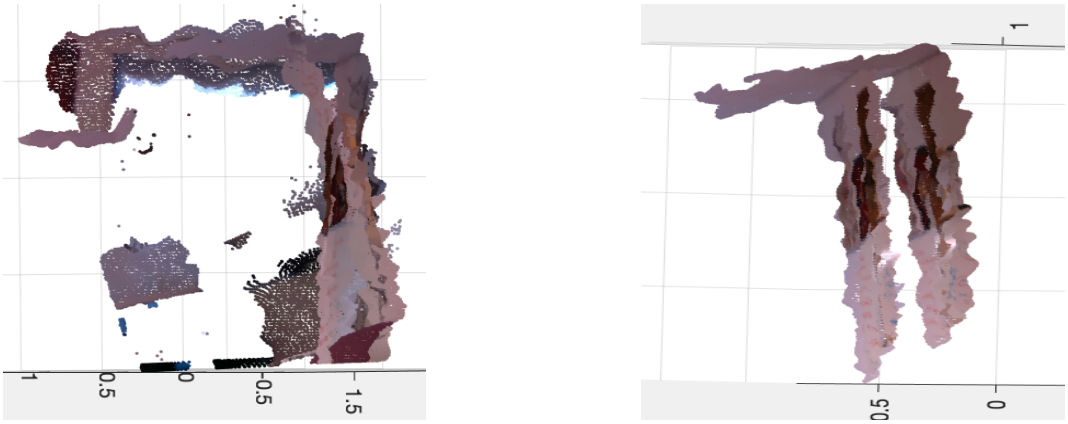
As shown in Algorithm 3, each frame between 2-40, depending on their relative distances to frame 1, will undergo a series of transformations in order to reach the coordinate system of the first frame. Once all frames are situated in the same coordinate system, the fusion becomes much more straightforward.

4.2 Final Result

Our final result of the reconstructed office is shown in Figure 9. Frames before 27 are fused smoothly and elegantly. However, due to the disturbance induced by Bob's body, the frames between 27-32 cause some troubles for our reconstruction task, and the quality of the fusion of the right wall is not as good as expected. In fact, there are some noticeable deviations between the frames of the right wall.

4.3 Example Failures

Two examples of defective fusions of the right wall are exhibited in Figure 10. One thing in common between these two cases is that the frames do not completely overlap each other. Especially for the second example, two frames are parallel and do not touch each other at all.



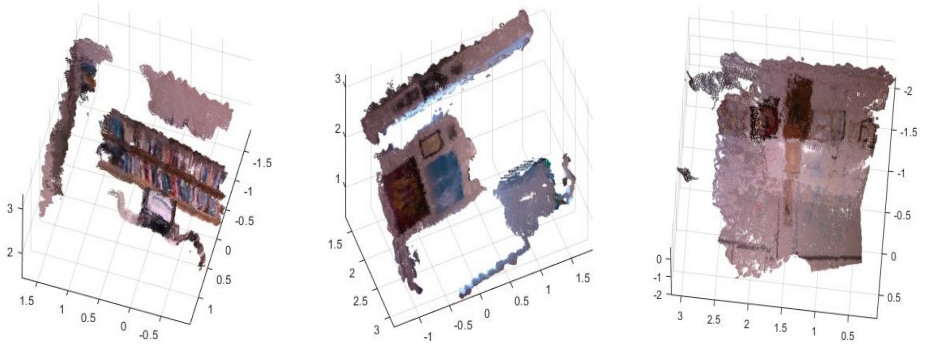
(a) The imperfect fusion between frame 27 and 28 (b) The imperfect fusion between frame 31 and 32

Figure 10: Two examples of failed fusions

5 Evaluation

This section is to evaluate the constructed 3D model by measuring wall distances and wall angles. The Matlab funtion *pcfitplane* is applied to threshold the three walls of the office. Three separted walls are shown in Fig.11. The fitting algorithm extracts parts of the point cloud in the fitting, resulting in a incomplete wall illustration. The angles are calculate based on the normal vectors returned by *pcfitplane* and results are listed in Table.1. The left wall is perfectly perpendicular to the end wall while it is not parallel with the right wall, with a error of $32.4°$. The end wall is nearly perpendicular to the right wall with a error of $4°$.

To calculate the distance between left wall and the right wall, we first reconstruct two walls by their plane equations. This is to avoid significant shift of mass center on uneven original walls. Fig.12 (b) shows the reconstructed walls. The location of mass center are calculated by averaging all coordinates and the distance result is 3.47.

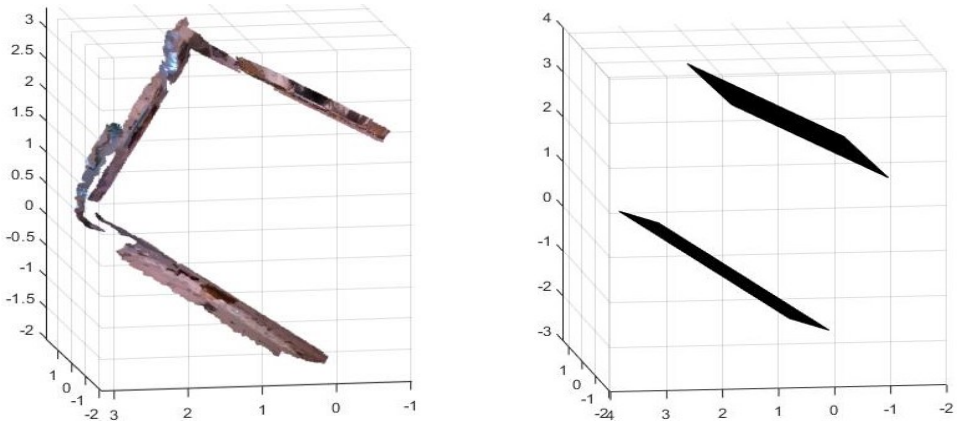


(a) Left Wall (b) End Wall (c) Right Wall

Figure 11: Illustration for separate walls. Left wall is fitted through frame 1 to frame

	Left to End	Left to Right	End to Right
Angle	90.0°	32.4°	94.0°

Table 1: Angles between two walls.



(a) Combined three walls without ceiling (b) Reconstruction of left and right walls

Figure 12: Illustration of combined walls and reconstruction from plane equation. It is clear to see there is a angle between the left and the right walls.

References

[1] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Commun. ACM*, vol. 18, no. 9, pp. 509–517, Sep. 1975. [Online]. Available: http://doi.acm.org/10.1145/361002.361007

[2] A. Vedaldi and B. Fulkerson, "VLFeat: An open and portable library of computer vision algorithms," http://www.vlfeat.org/, 2008.

[3] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *Int. J. Comput. Vision*, vol. 60, no. 2, pp. 91–110, Nov. 2004. [Online]. Available: https://doi.org/10.1023/B:VISI.0000029664.99615.94

Appendices

Listing 1: The pipeline

```
%% Set seed and add paths
rng(0);

addpath('./1_preprocessing');
addpath('./2_valid_sift_match');
addpath('./3_ransac_solve');
addpath('./4_pc_fusion');
addpath('./5_evaluation');
addpath('./dataset');
addpath('./results');
addpath('./utils');

%% Load trainig data
office = load('dataset/office1.mat');
pcs = office.pcl_train;
%pcs(23:end) = [];

%% Preprocessing data

X = 640;
Y = 480;

% Exclude boundary points in image:
idx_upper = [];
idx_lower = [];

for i = 1:Y
    idx_upper = [idx_upper; 1 + (i-1)*X];
    idx_lower = [idx_lower; X + (i-1)*X];
end
idx_left = [2:X-1]';
idx_right = [X*Y-X+2:X*Y-1]';
idx = cat(1, idx_upper, idx_lower, idx_left, idx_right);

masks = {};

for i = 1:size(pcs, 2)
    xyzpc = pcs{i}.Location;
    xpc = xyzpc(:,1);
    ypc = xyzpc(:,2);
    zpc = xyzpc(:,3);

    % Exclude points with z > 4 in Location:
    idx1 = find(zpc > 4);

    % Exclude NaN points in Location:
    idx2 = find(isnan(xpc));
    idx3 = find(isnan(ypc));
    idx4 = find(isnan(zpc));

    %{
    % Remove flying points slowly but with high quality:
    radius = 0.3;
    min_neighbors = 6000;
    MdlKDT = KDTreeSearcher(xyzpc);

    idx5 = [];
    for j = 1:size(xyzpc, 1)
```

```
        Y = xyzpc(j,:);

        IdxKDT = rangesearch(MdlKDT,Y,radius);

        if size(IdxKDT{1}, 2) < min_neighbors
            idx5 = [idx5; j];
        end
    end
    %}

    % Remove flying points fastly:
    idx5 = remove_flying_pixels(xyzpc, 0.15, 100, X);

    idx_final = cat(1, idx, idx1, idx2, idx3, idx4, idx5);
    masks{i} = unique(idx_final);
end

%{
% Exclude Bob's body in Color:
idx_27 = masks{27};

pc = pcs{27};
colorpc = pc.Color;
colorpc(idx_27, :) = 0;
colorpc = reshape(colorpc, 640, 480, 3);

grayimg = rgb2gray(colorpc);
edcanny = edge(grayimg, 'Canny', 0.285);

SE = strel ('square', 4);
dilation = imdilate(edcanny, SE);

filledbob = imfill(dilation, 'holes');

SE2 = strel('square', 4);
body_mask=imerode(filledbob, SE2);

idx_body = find(body_mask == 1);

% Exclude Bob's legs in Color:
idx_2d = [];

start_x = 141;
start_y = 245;

width = 150;
height = 220;

for h = 0:220
    if mod(h, 5) == 0
        start_x = start_x - 1;
    end

    for w = 0:150
        idx_2d = [idx_2d; start_x + w, start_y + h];
    end
end

idx_legs = [];
for i = 1:size(idx_2d, 1)
    idx_legs = [idx_legs; idx_convert_2d_to_1d(idx_2d(i,:))];
end

% Exclude Bob as a whole from the scene:
idx_27 = cat(1, idx_27, idx_body, idx_legs);
```

```
masks{27} = unique(idx_27);
%}

% Save masks to mat file:
save('results/mask_collection.mat', 'masks');

%% Estimate transformations
mask_collection = load('results/mask_collection.mat');
masks = mask_collection.masks;

min_frame = 1; % transform all frames after min_frame to ...
    min_frame's coordinate system
max_frame = size(pcs, 2); % the last frame to be transformed
models = cell(1,max_frame-1); % store up to 39 transformation ...
    matrices:

% Setting Hyper parameters
sift_dist_th = 60; % the less the stricter
area_ratio_th = 0.9; % the more the stricter
best_2nd_ratio = 0.7; % the less the stricter
ransac_param.sample_size = 20; % the more the stricter
ransac_param.th_dist = 0.1; % the less the stricter
ransac_param.itr_num = 100; % number of iteration
ransac_param.inl_ratio = 0.2; % the more the stricter

for frame = max_frame:-1:(min_frame+1)
    frame1 = frame;
    frame2 = frame-1;

    pc1 = pcs{frame1};
    pc2 = pcs{frame2};

    mask1 = mask_convert_1d_to_2d(masks{frame1});
    mask2 = mask_convert_1d_to_2d(masks{frame2});

    rgb_img1 = imag2d(pc1.Color);
    rgb_img2 = imag2d(pc2.Color);

    %Find valid sift match
    sift_pairs = valid_sift(rgb_img1, mask1, rgb_img2, mask2, ...
        sift_dist_th, area_ratio_th, best_2nd_ratio);
    [A, B] = get_depth(pc1, pc2, sift_pairs);
    [model, pt_idx] = ransac_icp(A, B, ransac_param);

    models{frame2} = model;
end
save('results/model_collection.mat', 'models');

%% Transform frames based on models
model_collection = load('results/model_collection.mat');
models = model_collection.models;
transformed_pcs = cell(1,max_frame); % store up to 40 transformed ...
    point clouds

% Transform point clouds [min_frame+1, max_frame] toward min_frame
for frame = max_frame:-1:min_frame
    pc = pcs{frame};
    mask = masks{frame};

    color_pc = pc.Color;
    color_pc(mask,:) = [];

    xyz_pc = pc.Location;
    xyz_pc(mask,:) = [];
    xyz_pc = cat(2, xyz_pc, ones(size(xyz_pc, 1), 1));
```

```
    xyz_pc = xyz_pc';

    model_idx = frame - 1;
    while model_idx ≥ min_frame
        model = models{model_idx};
        xyz_pc = model*xyz_pc;
        model_idx = model_idx - 1;
    end

    xyz_pc = xyz_pc';
    xyz_pc = xyz_pc(:,1:3);
    pc_t = pointCloud(xyz_pc, 'Color', color_pc);
    transformed_pcs{frame} = pc_t;
end
end
save('results/new_office.mat', 'transformed_pcs');

%% Fuse frames
min_frame = 1;
max_frame = size(pcs, 2);
new_office = load('results/new_office.mat');
transformed_pcs = new_office.transformed_pcs;

% Merge point clouds from min_frame to max_frame
pc_merged = transformed_pcs{min_frame};
for frame = (min_frame+1):max_frame
    pc_merged = pcmerge(pc_merged, transformed_pcs{frame}, 0.015);
end

%% Evaluation
angle, wall_dist = planefit(transformed_pcs);
```

Listing 2: Find Valid Sift Pairs

```
function sift_pairs = valid_sift(img1, mask1, img2, mask2, ...
    dist_th, area_ratio_th, best_2nd_ratio)
% This function is to filter and then find valid sift matches
% Args:
%   img1, img2: RGB images
%   mask1, mask2: Masks indicating valid area for SIFT
%   dist_th: Distance threshold for sift matching.
% Returns:
%   sift_pairs: Matrix contatining index of sift pairs (in depth ...
    image).

img1_gray = single(rgb2gray(img1));
img2_gray = single(rgb2gray(img2));
[f1, d1] = vl_sift(img1_gray);
[f2, d2] = vl_sift(img2_gray);
pt_num1 = size(d1, 2);
pt_num2 = size(d2, 2);

% Filtered by mask
loc1 = floor(f1(1:2,:));
loc2 = floor(f2(1:2,:));

val_loc1 = [];
val_loc2 = [];
val_d1 = [];
val_d2 = [];

for i=1:1:pt_num1
    loc = loc1(:,i);
    x_min = max(loc(1)-8, 1);
```

```
    x_max = min(loc(1)+8, 640);
    y_min = max(loc(2)-8, 1);
    y_max = min(loc(2)+8, 480);
    patch = mask1(x_min:x_max, y_min:y_max);
    [h, w] = size(patch);
    ratio = sum(patch, 'all') / (h * w);

    if ratio ≥ area_ratio_th
        if mask1(loc(1), loc(2)) == 1
            val_loc1 = [val_loc1, loc];
            val_d1 = [val_d1, d1(:,i)];
        elseif mask1(loc(1)+1, loc(2)) == 1
            loc(1) = loc(1) + 1;
            val_loc1 = [val_loc1, loc];
            val_d1 = [val_d1, d1(:,i)];
        elseif mask1(loc(1), loc(2)+1) == 1
            loc(2) = loc(2) + 1;
            val_loc1 = [val_loc1, loc];
            val_d1 = [val_d1, d1(:,i)];
        elseif mask1(loc(1)+1, loc(2)+1) == 1
            loc(1) = loc(1) + 1;
            loc(2) = loc(2) + 1;
            val_loc1 = [val_loc1, loc];
            val_d1 = [val_d1, d1(:,i)];
        end
    end
end

for i=1:1:pt_num2
    loc = loc2(:,i);
    x_min = max(loc(1)-8, 1);
    x_max = min(loc(1)+8, 640);
    y_min = max(loc(2)-8, 1);
    y_max = min(loc(2)+8, 480);
    patch = mask2(x_min:x_max, y_min:y_max);
    ratio = sum(patch, 'all') / area_ratio_th;

    if ratio ≥ area_ratio_th
        if mask2(loc(1), loc(2)) == 1
            val_loc2 = [val_loc2, loc];
            val_d2 = [val_d2, d2(:,i)];
        elseif mask2(loc(1)+1, loc(2)) == 1
            loc(1) = loc(1) + 1;
            val_loc2 = [val_loc2, loc];
            val_d2 = [val_d2, d2(:,i)];
        elseif mask2(loc(1), loc(2)+1) == 1
            loc(2) = loc(2) + 1;
            val_loc2 = [val_loc2, loc];
            val_d2 = [val_d2, d2(:,i)];
        elseif mask2(loc(1)+1, loc(2)+1) == 1
            loc(1) = loc(1) + 1;
            loc(2) = loc(2) + 1;
            val_loc2 = [val_loc2, loc];
            val_d2 = [val_d2, d2(:,i)];
        end
    end
end

num_val_sift1 = size(val_loc1, 2);
num_val_sift2 = size(val_loc2, 2);

% Find matched pair
sift_pairs = [];
distances = [];

```

```
for i=1:1:num_val_sift1
    for j=1:1:num_val_sift2
        dist = dist_cal(val_d1(:,i), val_d2(:,j));
        %dist = dist_chi_sq(val_d1(:,i), val_d2(:,j));
        if dist < dist_th
            % Convert pixel location to depth coord
            idx1 = idx_convert_2d_to_1d(val_loc1(:,i));
            idx2 = idx_convert_2d_to_1d(val_loc2(:,j));
            sift_pairs = [sift_pairs; idx1,idx2];
            distances = [distances; dist];
        end
    end
end

sift_pairs = clean_pairs(sift_pairs, distances, best_2nd_ratio);
sift_pairs = sift_pairs';

return
```

Listing 3: Sift mismatch clean

```
function cleaned_pairs = clean_pairs(pairs, distances, best_2nd_ratio)
% Check the ratio between distances of best match and second best ...
    match
% Args:
%   pairs: Index of SIFT pairs in point cloud. [x1, y1; x2, y2; ...]
%   distances: Corresponding distances
%   best_2nd_ratio: Threshold ratio to judge if it is a valid match
% Returns:
%   cleaned_pairs: Filtered SIFT pairs

    % left cleaning:
    left = pairs(:,1);
    uniq = unique(left);

    % pair indexes to be eliminated
    idx_cumulative = [];

    for i = 1:size(uniq, 1)
        x = uniq(i);

        idx = find(left==x);

        % if this point on the left involves only one pair, then ...
            keep it
        % and continue
        if length(idx) == 1
            continue;
        end

        dist_dup = distances(idx);

        [dist_smallest, idx_smallest] = min(dist_dup);
        dist_dup(idx_smallest) = Inf;
        dist_2nd_smallest = min(dist_dup);

        % If the best pair passes ratio test, then keep it and
        % discard the rest. Otherwise elminate all of them.
        if dist_smallest / dist_2nd_smallest < best_2nd_ratio
            idx(idx_smallest) = [];
        end

        idx_cumulative = unique([idx_cumulative; idx]);
```

```
    end

    pairs(idx_cumulative,:) = [];
    distances(idx_cumulative,:) = [];

    % right cleaning:
    right = pairs(:,2);
    uniq = unique(right);

    idx_cumulative = [];

    for i = 1:size(uniq, 1)
        x = uniq(i);

        idx = find(right==x);

        % if this point on the right involves only one pair, then ...
            keep it
        % and continue
        if length(idx) == 1
            continue;
        end

        dist_dup = distances(idx);

        [dist_smallest, idx_smallest] = min(dist_dup);
        dist_dup(idx_smallest) = Inf;
        dist_2nd_smallest = min(dist_dup);

        % If the best pair passes ratio test, then keep it and
        % discard the rest. Otherwise elminate all of them.
        if dist_smallest / dist_2nd_smallest < best_2nd_ratio
            idx(idx_smallest) = [];
        end

        idx_cumulative = unique([idx_cumulative; idx]);
    end

    pairs(idx_cumulative,:) = [];
    distances(idx_cumulative,:) = [];

    cleaned_pairs = pairs;
end
```

Listing 4: Ransac with ICP

```
function [best_model, pt_idx] = ransac_icp(A, B, ransac_param)
% Args:
%   A, B: Point set A and B each with size n x m. m points with n
%         dimensions. [x, y, z]' for example.
%   ransac_param: Parameter setting for ransac algorithm
% Returns:
%   model: Matrix containing Rotation and Translation matrix. [R, ...
    T; 0, 1]
%   pt_idx: vector of matched point indices

sample_size = ransac_param.sample_size; % number of sample pairs ...
    to use
th_dist = ransac_param.th_dist; % distance threshold
itr_num = ransac_param.itr_num; % number of iteration
inl_ratio = ransac_param.inl_ratio;% inlier ratio

if sample_size < 3
```

```
    fprintf('Need more sample pairs to fit !\n');
    return
end

% transform to homogenous coord
match_num = size(A, 2);
A_homo = [A; ones(1, match_num)];
B_homo = [B; ones(1, match_num)];

if match_num < sample_size
    fprintf('Total pairs not enough in the first place !\n')
    return
end

inl_th = round(match_num*inl_ratio);
best_inl_num = inl_th;
best_model = [];

for i=1:1:itr_num
    %{
    indexes = [1:match_num];

    idx = randsample(indexes, 1);
    indexes(find(indexes==idx)) = [];

    sample_A = A(:,idx);
    sample_B = B(:,idx);
    while size(sample_A, 2) < sample_size
        if size(indexes, 2) == 0
            fprintf('Total pairs not enough after removing ...
                repetitive pairs !\n')
            return
        end
        idx = randsample(indexes, 1);
        indexes(find(indexes==idx)) = [];

        a = A(:,idx);
        b = B(:,idx);
        sample_Ax = sample_A(1,:);
        if ismember(a(1), sample_Ax) == 0
            sample_A = [sample_A, a];
            sample_B = [sample_B, b];
        end
    end
    %}
    idx = rand_idx(match_num, sample_size);
    sample_A = A(:,idx);
    sample_B = B(:,idx);
    %%%%%%%%%%%%%%%%%%%% ICP %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    pc_A = pointCloud(sample_A');
    pc_B = pointCloud(sample_B');
    F = pcregistericp(pc_B,pc_A,'Extrapolate',true);
    F = F.T;
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    dist = dist_cal(F*A_homo, B_homo);
    inl_num = length(find(dist<th_dist));
    if inl_num ≤ best_inl_num
        continue;
    end
    best_inl_num = inl_num;
    best_model = F;
end

if isempty(best_model)
```

```
    fprintf('Mo model meets success criteria !\n')
    return
end

dist = dist_cal(best_model*A_homo, B_homo);
pt_idx = find(dist < th_dist);

return
```

Listing 5: Fitting Plane

```
function [angle_mat, distance] = planefit(transformed_pcs)

    % left wall with bookshelf
    lw_frame_min = 1;
    lw_frame_max = 13;
    % frontal wall with window
    fw_frame_min = 14;
    fw_frame_max = 23;
    % right wall with white board
    rw_frame_min = 24;
    rf_frame_max = 40;

    wall_sec = [lw_frame_min fw_frame_min rw_frame_min;
                lw_frame_max fw_frame_max rf_frame_max];

    % result of plane fitting.
    % First row contains point cloud of wall
    % Second row contains normal vector
    wall_result = cell(2,3);
    %%
    dist_th = 0.07;
    for i=1:3
        min_frame = wall_sec(1,i);
        max_frame = wall_sec(2,i);
        pc_merged = transformed_pcs{min_frame};
        for frame = (min_frame+1):max_frame
            pc_merged = pcmerge(pc_merged, transformed_pcs{frame}, ...
                0.015);
        end
        [model, inlier, ¬] = pcfitplane(pc_merged, dist_th);
        plane = select(pc_merged, inlier);
        wall_result{1,i} = plane;
        wall_result{2,i} = model.Normal;
    end

    % Calculate wall center of mass based on raw data
    wall_left_loc = wall_result{1,1}.Location;
    wall_right_loc = wall_result{1,3}.Location;

    % Function to calculate angle between vectors
    vec_angle = @(x,y) atan2(norm(cross(x,y)), dot(x,y));
    angle_mat = zeros(1,3);
    angle_mat(1,1) = vec_angle(wall_result{2,1}, ...
        wall_result{2,2}); % Angle between left and end walls
    angle_mat(1,2) = vec_angle(wall_result{2,1}, ...
        wall_result{2,3}); % Angle between left and right walls
    angle_mat(1,3) = vec_angle(wall_result{2,2}, ...
        wall_result{2,3}); % Angel between end and right walls

    %% Reconstruct even plane by equations
    normal_vec_left = wall_result{2,1};
    normal_vec_right = wall_result{2,3};
```

```
49
50      min_x_left = min(wall_left_loc(:,1));
51      max_x_left = max(wall_left_loc(:,1));
52      min_y_left = min(wall_left_loc(:,2));
53      max_y_left = max(wall_left_loc(:,2));
54
55      min_x_right = min(wall_right_loc(:,1));
56      max_x_right = max(wall_right_loc(:,1));
57      min_y_right = min(wall_right_loc(:,2));
58      max_y_right = max(wall_right_loc(:,2));
59
60      %% Get plane parameters
61      D_left = -dot(wall_left_loc(1,:),normal_vec_left);
62      D_right = -dot(wall_right_loc(1,:),normal_vec_right);
63
64      A_left = normal_vec_left(1);
65      B_left = normal_vec_left(2);
66      C_left = normal_vec_left(3);
67
68      A_right = normal_vec_right(1);
69      B_right = normal_vec_right(2);
70      C_right = normal_vec_right(3);
71
72      x_interval_left = (max_x_left - min_x_left)/50;
73      y_interval_left = (max_y_left - min_y_left)/50;
74
75      x_interval_right = (max_x_right - min_x_right)/50;
76      y_interval_right = (max_y_right - min_y_right)/50;
77
78      %% Plot fitted plane and even one
79      figure;
80      pcshow(wall_result{1,1})
81      hold on
82      pcshow(wall_result{1,2})
83      hold on
84      pcshow(wall_result{1,3})
85
86      figure;
87      [x1, y1] = meshgrid(min_x_left:x_interval_left:max_x_left, ...
            min_y_left:y_interval_left:max_y_left); % Generate x and y ...
            data
88      z1 = -1/C_left*(A_left*x1 + B_left*y1 + D_left); % Solve for z ...
            data
89      surf(x1,y1,z1) %Plot the surface
90      hold on
91      [x2, y2] = meshgrid(min_x_right:x_interval_right:max_x_right, ...
            min_y_right:y_interval_right:max_y_right); % Generate x ...
            and y data
92      z2 = -1/C_right*(A_right*x2 + B_right*y2 + D_right); % Solve ...
            for z data
93      surf(x2,y2,z2) %Plot the surface
94
95      %% Calculate wall center of mass based on even plane
96      new_center_left = [sum(x1,'all'), sum(y1,'all'), ...
            sum(z1,'all')]/(size(x1,2)^2);
97      new_center_right = [sum(x2,'all'), sum(y2,'all'), ...
            sum(z2,'all')]/(size(x2,2)^2);
98      distance = pdist([new_center_left;new_center_right], 'Euclidean');
99
100 return
```