CSE586/EE554 Computer Vision II Project 1 Report

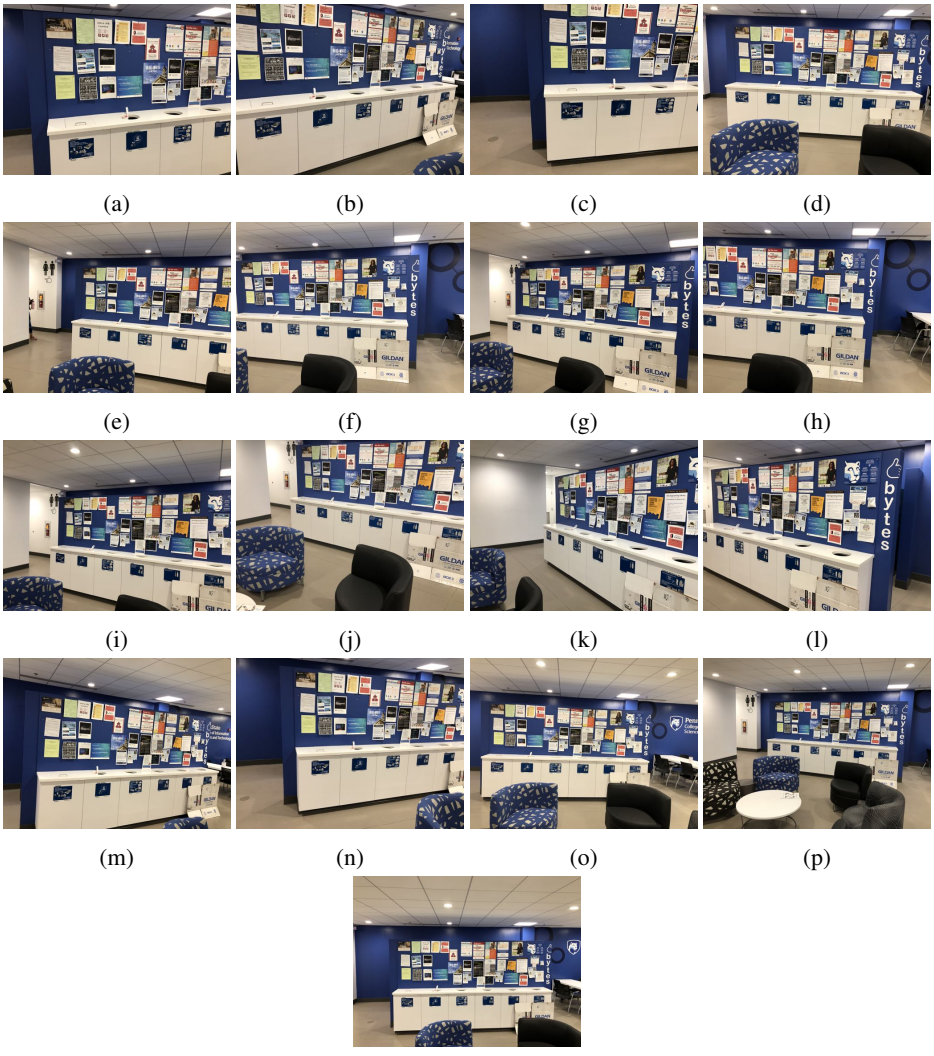
Rui Yu, Qianying Zhou

April 2019

1 Introduction

In this project, we successfully implemented an augmented reality task, which is to put an artificial object overlaid on a group of images of a bulletin board in IST college. We got a cloud of 3D points through COLMAP. Then we implemented RANSAC algorithm to find a dominant plane among the 3D points, created a 3D object on the plane, and projected the artificial object back into each of the original images. We used Matlab for programming. Due to the image size, we only submit the code in Canvas. All the input and output images along with our code can be download from:
https://drive.google.com/file/d/1qxgZ99j2nIK_AbhnBRRFbp43U3_t0JF2/view?usp=sharing

2 Step 1: Collection of images



(a) (b) (c) (d)

(e) (f) (g) (h)

(i) (j) (k) (l)

(m) (n) (o) (p)

(q)

Figure 1: Images of IST bulletin board.

In this step, we collected a set of images of the bulletin board at the first floor of IST college near the elevator. We chose this scene because there are many posters on the bulletin board, which can be easily found by COLMAP as features. The bulletin board can be regarded as the dominant planer surface in

the following steps. We also tried several other scenes, such as CSE bulletin board, but finally chose IST bulletin board in the experiment.

We took 17 photos of the bulletin board from different directions with a single camera (iPhone 8 plus) with a fixed focal length. Figure 1 shows the 17 images we collected.

3 Step 2: Get a 3D cloud of points by COLMAP

We installed COLMAP on Mac OS X. The first step is to input our set of images and create a file `database.db`. Then we run feature extraction, matching, incremental reconstruction, and bundle adjustment to get a 3D cloud of points. There are 11,751 feature points matched across images. The model is then exported as text files as well as `project.ini` so that we can import it again in the future. Figure 2 is the screenshot of the 3D cloud of points after importing the model again. We can clearly observe the dominant plane sketched by the feature points on the bulletin board.

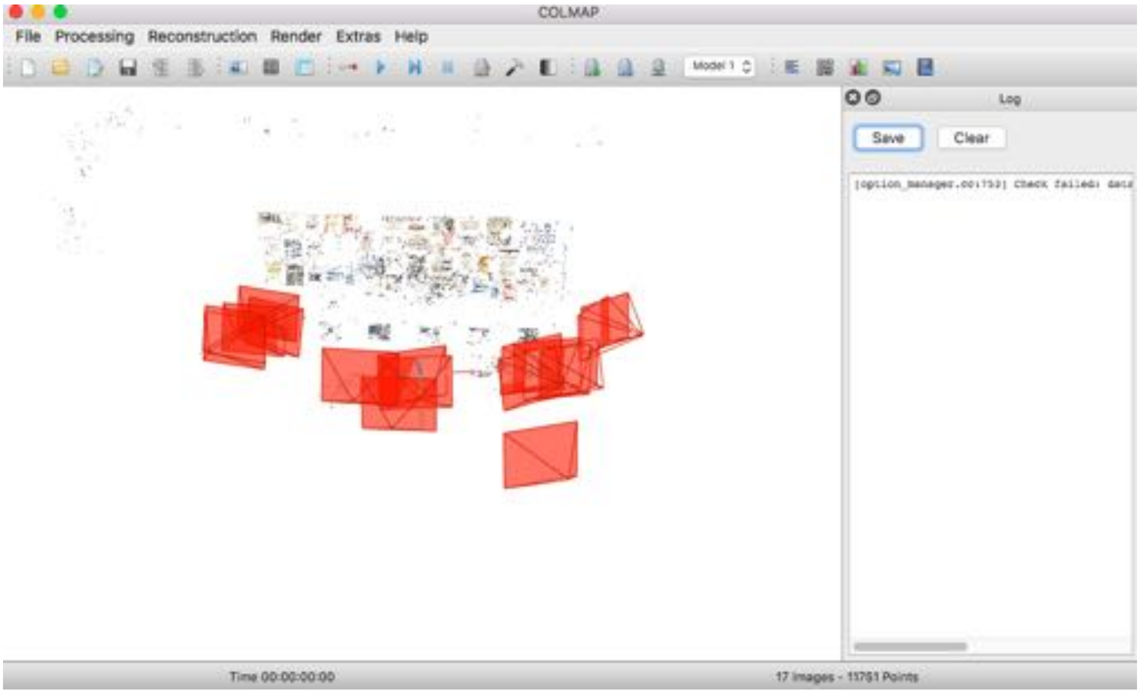


Figure 2: 3D cloud of points.

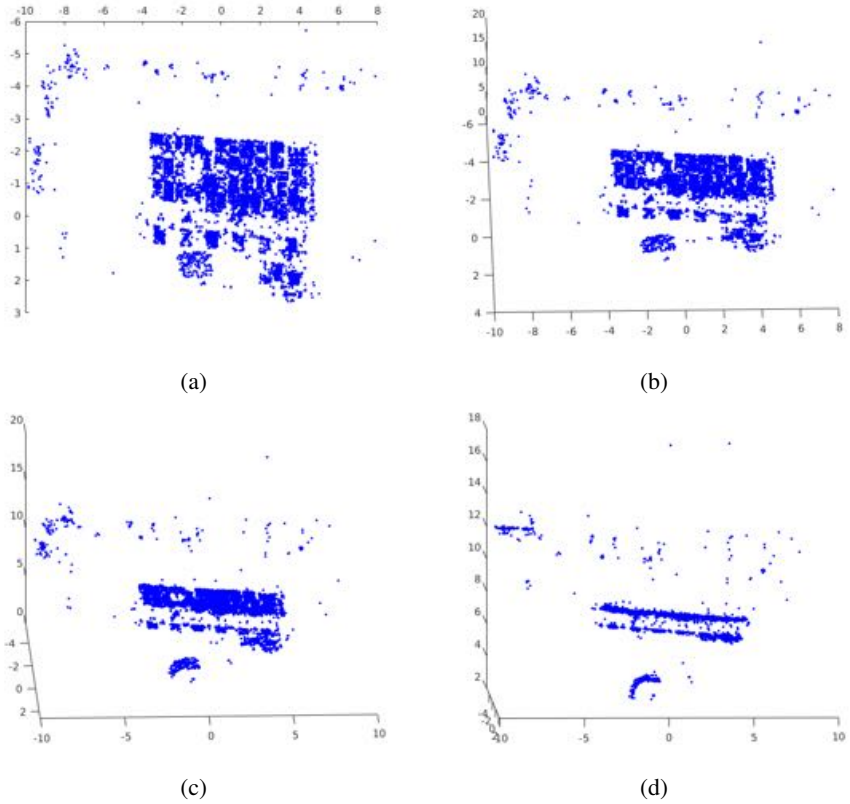
4 Step 3: Read in the 3D point cloud

We read in the X, Y, Z coordinates for each point from `points3D.txt`. The code snippet below shows how we read the text file. Figure 3 visualizes the points in the X, Y, Z coordinate.

```
point3d_file = '../data/points3D.txt';

P = [];
this_p = zeros(1,3);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Read 3D Point Coordinates to P %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
if ~exist(point3d_file, 'file')
    assert('3D points file doesn''t exist...');
end
fid = fopen(point3d_file);
%% READ FILE LINE BY LINE
tline = fgetl(fid);
while ischar(tline)
    % skip line if empty or comment
    if isempty(tline) || strcmp(strtrim(tline(1)), '#')
        tline = fgetl(fid); continue;
    end
    % split parameter into variable name and value
```

```
    C = strsplit(tline,' ');
        % get 3D point coordinate
        for i = 2:4
                this_p(i-1) = str2num(C{i});
        end
        P = [P; this_p];
    % read next line
     tline = fgetl(fid);
end
%% CLOSE FILE
fclose(fid);
```



(a) (b) (c) (d)

Figure 3: 3D cloud of points visualized by Matlab.

5 Step 4: Find the largest subset using RANSAC routine

We set the minimum number of 3D points required to fit a 3D plane as 3. We also tried different thresholds to determine the inliers or outliers, and eventually chose the threshold as 0.2.

```
no = 3; % smallest number of points required
k = 4000; % number of iterations
t = 0.2; % threshold used to id a point that fits well
d = 8000; % number of nearby points required
```

We use *randperm()* to get a list of random number, and then pick the first three numbers to be the index of the sample set.

To fit the equation of a plane to the points in a sample, we used Least Square Estimation. Assume that the plane is $Ax + By + Cz + D = 0$. First, we calculated the mean of **x**, **y** and **z**. Let $z_i - \bar{z}$ as $\hat{z}$, $x_i - \bar{x}$ as $\hat{x}$ and $y_i - \bar{y}$ as $\hat{y}$. By solving the least squares system, we get:

$$A = \frac{\sum \hat{z}_i \hat{x}_i \sum \hat{y}_i{}^2 - \sum \hat{z}_i \hat{y}_i \sum \hat{x}_i \hat{y}_i}{\sum \hat{x}_i{}^2 \sum \hat{y}_i{}^2 - (\sum \hat{x}_i \hat{y}_i)^2} \tag{1}$$

$$B = \frac{\sum \hat{x}_i{}^2 \sum \hat{z}_i \hat{x}_i - \sum \hat{z}_i \hat{x}_i \sum \hat{x}_i \hat{y}_i}{\sum \hat{x}_i{}^2 \sum \hat{y}_i{}^2 - (\sum \hat{x}_i \hat{y}_i)^2} \tag{2}$$

$$C = \bar{z} - A\bar{x} - B\bar{y} \tag{3}$$

The code snippet below shows the least square estimation.

```
function [n_est ro_est X Y Z] = LSE(p)
% INPUT:
% p - Mx3; [points x [X Y Z]]

        % Calculate mean of all points
        pbar = mean(p);
        for i = 1:length(p)
                A(:,:,i) = (p(i,:)-pbar)'*(p(i,:)-pbar);
        end
        %Sum up all entries in A
        Asum = sum(A,3);
        [V ~] = eig(Asum);
        %Calculate new normal vector
        n_est = V(:,1);
        %Calculate new ro
        ro_est = dot(n_est,pbar);
        [X, Y] = meshgrid(min(p(:,1)):max(p(:,1)),min(p(:,2)):max(p(:,2)));
        Z = (ro_est - n_est(1)*X - n_est(2).*Y)/n_est(3);
end
```

After fitting a plane to the sample set, we then calculate the distance of each of the remained points to the plane. Assume that the plane is $Ax + By + Cz + D = 0$, and the point is $[x, y, z]$, the distance of the point to the plane is equation (4):

$$Distance = \frac{|\, Ax + By + Cz + D \,|}{\sqrt{A^2 + B^2 + C^2}} \tag{4}$$

For each of the remained points, if its distance to the plane is smaller than the threshold, we add it to the inliers. After calculating the distance of all those points, we count the total number of inliers. If the number of inliers is larger than the required number we set before, we probably have found a good plane. We then fit a plane to the inliers, and then calculate the sum of the distances of these points to the fitted plane. If the sum is smaller than the current smallest sum (error), we regard this plane and these inliers as the new best surface. If not, we will just keep the current smallest error and inliers.

We repeated this process for 4000 times to find the best model and inliers candidates. This is the code of the RANSAC routine.

```
function [p_best,n_vec,ro_best,X_best,Y_best,Z_best]=ransac_func(p,no,k,t,d)
%% INPUT:
% no - smallest number of points required
% k - number of iterations
% t - threshold used to id a point that fits well
% d - number of nearby points required
%% OUTPUT:
% p_best - the points in the plane
% n_vec - the normal vector (a,b,c) of the plane
% ro_best - d in ax+by+cz-d=0
% X_best, Y_best, Z_best - representative points on the plane

        % Initialize variables
        iterations=0;
        % Until k iterations have occurrec
        while iterations < k
                ii=0;
                clear p_close dist p_new p_in p_out
                % Draw a sample of n points from the data
                perm = randperm(length(p));
                sample_in = perm(1:no);
                p_in = p(sample_in,:);
                sample_out = perm(no+1:end);
                p_out = p(sample_out,:);
                % Fit to that set of n points
```

```
                [n_est_in ro_est_in] = LSE(p_in);
                % For each data point oustide the sample
                for i=sample_out
                        dist = dot(n_est_in,p(i,:)) - ro_est_in;
                        %Test distance d to t
                        abs(dist);
                        if abs(dist) < t %If d<t, the point is close
                                ii = ii + 1;
                                p_close(ii,:) = p(i,:);
                        end
                end
                p_new = [p_in; p_close];
                % If there are d or more points close to the plane
                if length(p_new) > d
                        % Refit the plane using all these points
                        [n_est_new ro_est_new X Y Z] = LSE(p_new);
                        for iii = 1:length(p_new)
                                dist(iii) = dot(n_est_new,p_new(iii,:))
                                - ro_est_new;
                        end
                        % Use the fitting error as error criterion
                        error(iterations+1) = sum(abs(dist));
                else
                        error(iterations+1) = inf;
                end
                if iterations > 1
                        % Use the best fit from this collection
                        if error(iterations+1) <= min(error)
                                p_best = p_new;
                                n_vec = n_est_new;
                                ro_best = ro_est_new;
                                X_best = X;
                                Y_best = Y;
                                Z_best = Z;
                                error_best = error(iterations+1);
                        end
                end
                iterations = iterations + 1;
                if mod(iterations,100)==0
                        fprintf('RANSAC iteration [%d / %d] \n',iterations, k);
                end
        end
end
```

---

6 Step 5: Display the 3D point cloud and inlier points

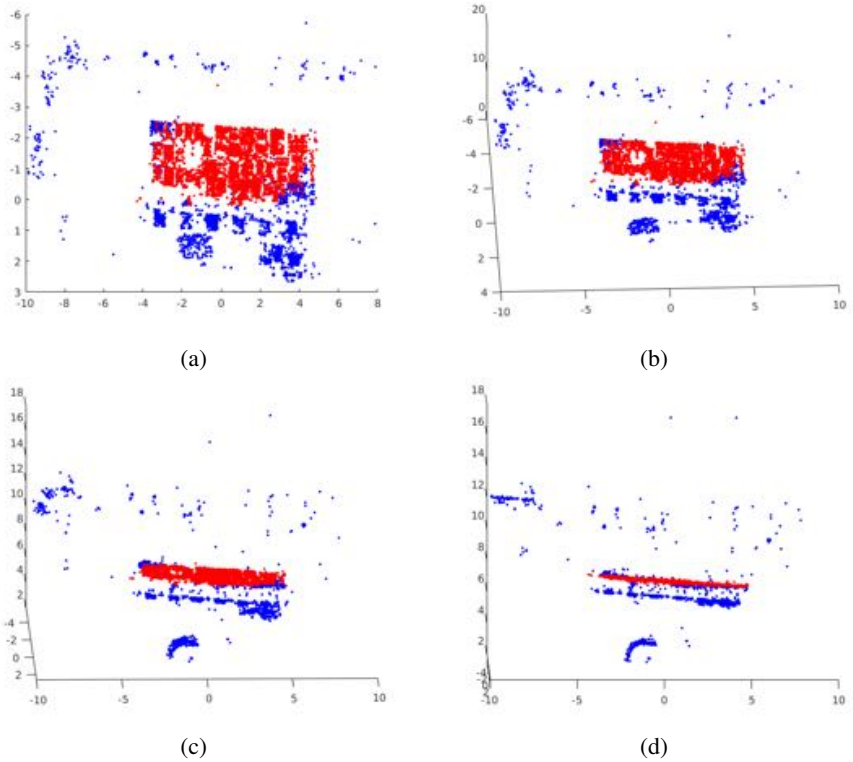
We used the *plot3* function to plot the 3D point cloud and inlier points. As Figure 4 shows, the blue points are outliers and the red points are the inliers. We can see that the inlier set looks exactly like what we expect to find.

```
figure
plot3(p_out(:,1),p_out(:,2),p_out(:,3),'.b')
hold on
plot3(p_in(:,1),p_in(:,2),p_in(:,3),'.r')
rotate3d on
```

---

7 Step 6: 3D Euclidean transformation

To get the 3D Euclidean transformation (rotation and translation) from the local coordinate system into the scene X, Y, Z coordinates, we first need to get (r1, r2, r3) three vectors which are vertical to each other. r1 and r2 are two vectors on the dominate plane we found, and r3 is the normal vector of the plane.



(a) (b) (c) (d)

Figure 4: 3D cloud of points visualized by Matlab.

First, r3 is derived from previous RANSAC step. Then, we chose an inlier point, A, roughly in the center of all inliers points to be the origin of the local coordinate system. Next, we randomly find another point B on the plane so that we can get the vectors $\overrightarrow{AB}$ as the x-axis. The corresponding orthonormal vector is r1:

$$r1 = \frac{AB}{|\,AB\,|} \tag{5}$$

According to right-hand rule, the other orthonormal vector r2 is the cross product of r3 and r1:

$$r2 = r3 \times r1 \tag{6}$$

Assume that the rotation matrix transforming (r1, r2, r3) vectors to the (x, y, z) vectors (the scene coordinates) is R_1.

$$R_1 = \begin{bmatrix} r1_x & r1_y & r1_z \\ r2_x & r2_y & r2_z \\ r3_x & r3_y & r3_z \end{bmatrix} \tag{7}$$

Let translation matrix be T and the coordinates of A be $[A_x, A_y, A_z]$. According to the *Euclidean transformation*,

$$\begin{bmatrix} r1_x & r1_y & r1_z \\ r2_x & r2_y & r2_z \\ r3_x & r3_y & r3_z \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} + T = \begin{bmatrix} A_x \\ A_y \\ A_z \end{bmatrix} \tag{8}$$

Therefore, the transformation from the local coordinate system into the scene coordinates is:

$$\begin{bmatrix} r1_x & r1_y & r1_z \\ r2_x & r2_y & r2_z \\ r3_x & r3_y & r3_z \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} + \begin{bmatrix} A_x \\ A_y \\ A_z \end{bmatrix} = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \tag{9}$$

8 Step 7: Create a virtual object to put in the scene

In this step, we created a 3D box lying in our local z=0 plane and the center of the bottom surface is x=0, y=0, z=0. Then we transformed the coordinates of its 8 corners into the scene coordinates using the Euclidean transformation we derived in Step 6.

The snippet of code below is how to create the corners of a 3D box.

```
%% Create Vertices
SL = [1;1;1]; % Length of Cuboid Side (SL - SideLength)
x = 0.5*SL(1)*[-1 1 1 -1 -1 1 1 -1]';
```

```
y = 0.5*SL(2)*[-1 -1 1 1 1 1 -1 -1]';
z = SL(3)*[0 0 0 0 -1 -1 -1 -1]';

%% Create Faces
facs = [1 2 3 4
        5 6 7 8
        4 3 6 5
        3 2 7 6
        2 1 8 7
        1 4 5 8];
```

Note that we defined r1 randomly in Step 6. To seems more real, our 3D box might need to be rotated along the z-axis for a certain angle to make the box look like vertical to the "ground". According to the general rotation matrix, where ψ, θ, ϕ are the rotation angles of three axes:

$$R_2(\psi,\theta,\phi) = \begin{bmatrix} cos\psi cos\phi - cos\theta sin\psi sin\phi & -sin\phi cos\psi - cos\theta sin\psi sin\phi & sin\theta sin\psi \\ cos\psi sin\phi + cos\theta cos\psi sin\phi & -sin\phi sin\psi + cos\psi cos\theta cos\phi & -sin\theta cos\psi \\ sin\theta sin\phi & sin\theta cos\phi & cos\theta \end{bmatrix} \tag{10}$$

Now the transformation becomes $R_1 R_2 x + T = \hat{x}$. The snippet of code below shows how to implement the transformation and plot the 3D box.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% DRAW A CUBOID ON THE PLANE %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% EA: [Yaw(Z-axis); Pitch(y-axis); Roll(x-axis)] Euler/Rotation angles [radians]
EA = [0.3, 0, 0]; % tune angle to make the cuboid vertical to ground

% Calculate Sines and Cosines
c1 = cos(EA(1));          s1 = sin(EA(1));
c2 = cos(EA(2));          s2 = sin(EA(2));
c3 = cos(EA(3));          s3 = sin(EA(3));

% Calculate Matrix
R2 = [c1*c2             -c2*s1            s2
      c3*s1+c1*s2*s3   c1*c3-s1*s2*s3   -c2*s3
      s1*s3-c1*c3*s2   c3*s1*s2+c1*s3    c2*c3]';

alph = 0.4; % transparency value of cuboid
colr = 'g'; % Color of cuboid

%% Rotate and Translate Vertices
verts = zeros(3,8);
for i = 1:8
    verts(:,i) = R1*R2*[x(i);y(i);z(i)]+T1;
end

figure
plot3(p_out(:,1),p_out(:,2),p_out(:,3),'.b')
hold on
plot3(p_in(:,1),p_in(:,2),p_in(:,3),'.r')
rotate3d on

%% Draw the faces of the cuboid
patch('Faces',facs,'Vertices',verts,'FaceColor',colr,'FaceAlpha',alph);

save_file = '../data/cuboidInfo.mat';
if ~exist(save_file, 'file')
    save(save_file, 'verts', 'facs', '-v7.3');
end
```

Now we have put a 3D box onto the surface, as figure 5 shows.

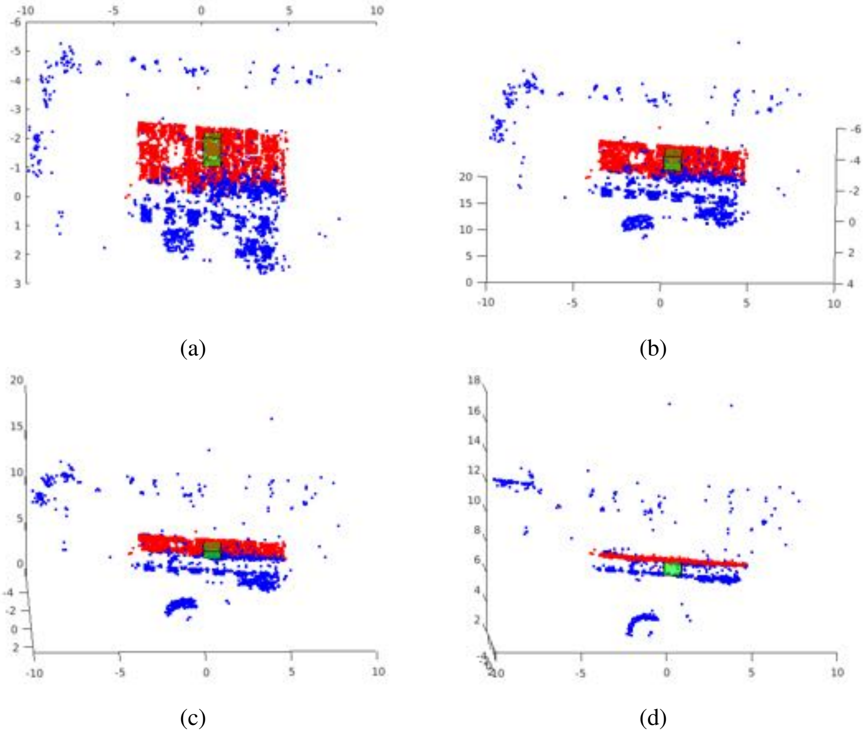


Figure 5: A 3D box on the dominant plane.

9 Step 8: Read in the camera parameters

In our .zip file, `camera.txt` and `images.txt` are under the *data* directory. The internal camera parameters can be read from `camera.txt` and the external parameters can be read from `images.txt`.

There is only one camera here using the distortion model *SIMPLE_RADIAL* with the sensor dimension as *[width: 4032 height: 3024]*. Our internal camera parameters are as followed, which means that the focal length is *3329.63* and a principal point is at pixel location *(2019.06, 1513.28)*.

$$Internal\ parameters = [f, c_x, c_y, k] = [3329.63,\ 2019.06,\ 1513.28,\ 0.0225948]$$

File `images.txt` contains the pose and keypoints of all reconstructed images in the dataset. From this file, we can get the information about the external parameters (rotation and translation), which describes how the images are projected from world to the camera coordinate system. Take the first image in the dataset for an example, its quaternion and translation vector are:

$$Quaternion = (QW,\ QX,\ QY,\ QZ) = [0.980227,\ -0.020933,\ -0.189281,\ -0.0537485]$$

$$Translation\ vector = (TX,\ TY,\ TZ) = [4.66971,\ -0.0239662,\ -1.21931]$$

In Matlab, we used *quat2rotm()* function to convert the quaternion to the 3×3 rotation matrix. The code snippet below shows how to read in the internal and external parameters from the two files.

```
internal_file = '../data/cameras.txt';
external_file = '../data/images.txt';
img_dir = '../img_input';
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Read camera internal parameter %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
inter_para = zeros(1,4); % internal parameter
if ~exist(internal_file, 'file')
    assert('Camera internal parameter file doesn''t exist...');
end
fid = fopen(internal_file);
%% READ FILE LINE BY LINE
tline = fgetl(fid);
while ischar(tline)
    % skip line if empty or comment
    if isempty(tline) || strcmp(strtrim(tline(1)), '#')
        tline = fgetl(fid); continue;
```

```
    end
    % split parameter into variable name and value
    C = strsplit(tline,' ');
        width = str2num(C{3});
        height = str2num(C{4});
        if C{2}=='SIMPLE_RADIAL'
                for i = 1:4
                        inter_para(i) = str2num(C{i+4}); % [f, cx, cy, k]
                end
        end
    % read next line
    tline = fgetl(fid);
end
fclose(fid);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Read external parameters of each image %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
num_img = 0;
ext_para = zeros(1,7); % external parameter
if ~exist(external_file, 'file')
    assert('External parameter file doesn''t exist...');
end
fid = fopen(external_file);
%% READ FILE LINE BY LINE
tline = fgetl(fid);
while ischar(tline)
    % skip line if empty or comment
    if isempty(tline) || strcmp(strtrim(tline(1)), '#')
        tline = fgetl(fid); continue;
    end
    % split parameter into variable name and value
    C = strsplit(tline,' ');
        if strcmp(C{10}(end-2:end), 'JPG')
                num_img = num_img + 1;
                for i = 2:8
                        ext_para(i-1) = str2num(C{i});
                end
                Rot{num_img} = quat2rotm(ext_para(1:4)); % external parameters
                Tran{num_img} = ext_para(5:7)';
                img_name{num_img} = C{10};
        end
    % read next line
    tline = fgetl(fid);
end
fclose(fid);
```

---

10 Step 9: Projection of 3D points into 2D pixel locations

Let the rotation matrix calculated by *quat2rotm()* function in Step 8 be R, and the translation vector be T. Different images have different R and T. If a point $p = [X_0,\ Y_0,\ Z_0]$ is in the scene coordinate system, the coordinates of the same point $p = [X,\ Y,\ Z]$ in the camera coordinate system is given by:

$$R \begin{bmatrix} X_0 \\ Y_0 \\ Z_0 \end{bmatrix} + T = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \tag{11}$$

Let location of this point on the image plane be $[u, v, f]$ where f is the focal length. According to the frontal pinhole imaging model, we can get:

$$\begin{bmatrix} u \\ v \end{bmatrix} = \frac{f}{Z} \begin{bmatrix} X \\ Y \end{bmatrix} \tag{12}$$

Then we need to transform $[u, v]$ to the pixel location on the image $[x, y]$. Because the image is a 2D plane so we do not need f here. For an image, the origin is on the up left corner and the camera is located

on the center of the image. Hence, the transformation is as followed by using the internal parameters of camera.

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} u \\ v \end{bmatrix} + \begin{bmatrix} c_x \\ c_y \end{bmatrix} \tag{13}$$

However, this is just an ideal situation. In the real world, we need to consider distortion, which is the fourth internal parameter. Here, we refer to the source code *camera_model.h* of COLMAP. Let the fourth internal parameter be k. For each point:

$$radial = k(u^2 + v^2) \tag{14}$$

Considering distortion, we can get $(\hat{u}, \hat{v})$:

$$\begin{bmatrix} \hat{u} \\ \hat{v} \end{bmatrix} = \begin{bmatrix} u \\ v \end{bmatrix} + radial \begin{bmatrix} u \\ v \end{bmatrix} \tag{15}$$

The code snippet below shows the implementation of procedures above.

```
%% load the scene coordinates of the vertices
load('../data/cuboidInfo.mat', 'verts', 'facs');
% facs: index of vertex for each face, size (6,4) (face x vertex)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Project the cuboid on each image %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
for i = 1:num_img
        % Transfer to local camera coordinate system
        verts_new{i} = Rot{i}*verts + Tran{i};
        verts_cam{i} = verts_new{i}(1:2, :)./verts_new{i}(3, :);
        % Transfer to pixel coordinate according to SimpleRadialCameraModel
        % Refer to
        https://github.com/colmap/colmap/blob/master/src/base/camera_models.h
        u = verts_cam{i}(1,:);
        v = verts_cam{i}(2,:);
        r2 = u.*u + v.*v;
        radial = inter_para(4) .* r2;
        du = u .* radial; % distortion
        dv = v .* radial; % distortion
        x = u + du;
        y = v + dv;
        x = inter_para(1).*x + inter_para(2); % transform to image coordinates
        y = inter_para(1).*y + inter_para(3); % transform to image coordinates
        verts_prj{i} = [x; y]';
        % Load the original image
        f = figure;
        imOrig = imread(fullfile(img_dir, img_name{i}));
        imshow(imOrig);
        hold on

        ...
        ...
end
```

11 Step 10: Project the 3D box into the images

To make the closer face of the 3D box occlude the farther face, we first sorted the faces by the *depth* (z-coordinate of the camera coordinate system). To estimate the depth of each face, we average the depths of the four vertices of the face. Then we draw a quadrilateral for each face onto the original image by using the projected vertices. The code snippet below shows how we get it.

```
% Sort the depth of all faces,
        and according change the order of drawing face
        depth = zeros(1,6);
        for j = 1:6
                depth(j) = mean(verts_new{i}(3, facs(j,:)));
```

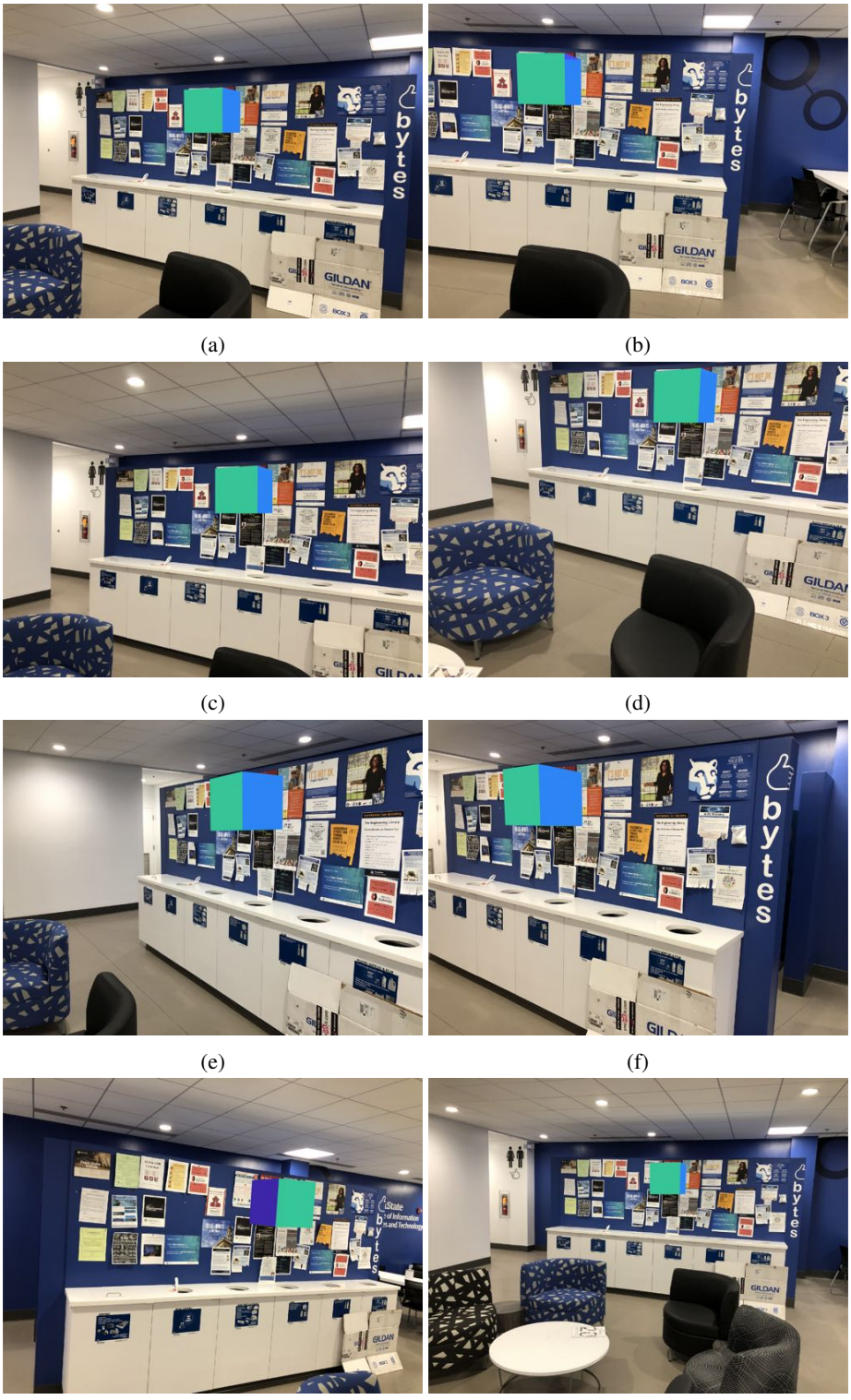
```
        % average depth of the four vertices
end
[d_sort, idx] = sort(depth, 'descend');
facs_new = facs(idx, :);
% Draw the faces
colr = [1:8]';
patch('Faces',facs_new,'Vertices',verts_prj{i},
'FaceVertexCData',colr,'FaceColor','flat','LineStyle','none',
'FaceAlpha', 0.98);
% Save the AR image
saveas(f,strcat('../img_output/result',int2str(i),'.png'))
pause(1)
```

Now we successfully finish this project. We have put a 3D box onto the dominant surface detected from the cloud of points. Figure 6, 7 and 8 shows the final result, which is the 3D virtual box object projected into each of 17 images and rendered on top of the original pixel values.

12 Contribution

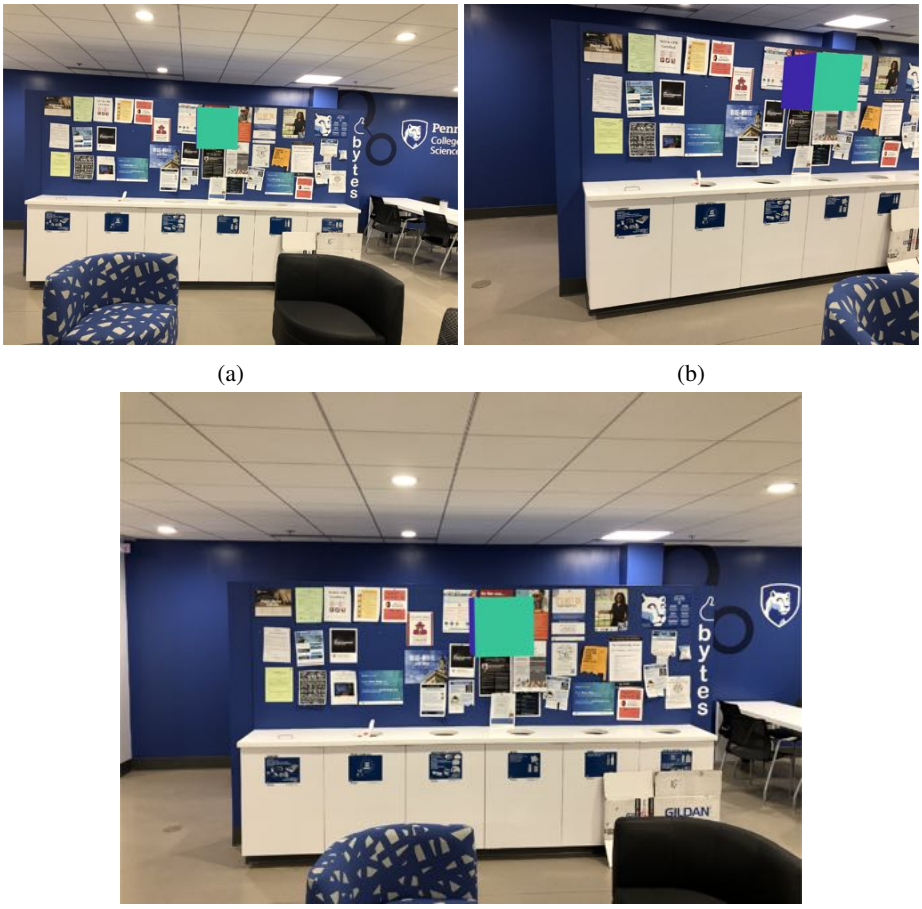
Rui Yu	Step1, 2, 3, 4, 5, 6, 7, 8, 9, 10, report
Qianying Zhou	Step1, 2, 3, 4, 5, 6, 7, 8, report

Figure 6: Final results.



(a) (b) (c) (d) (e) (f) (g) (h)

Figure 7: Final results.



(a) (b)

(c)

Figure 8: Final results.