Project 1
Augmented Reality

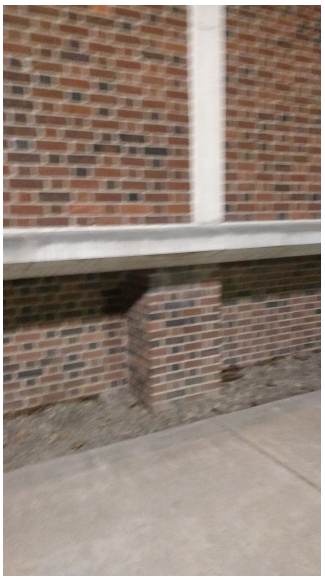
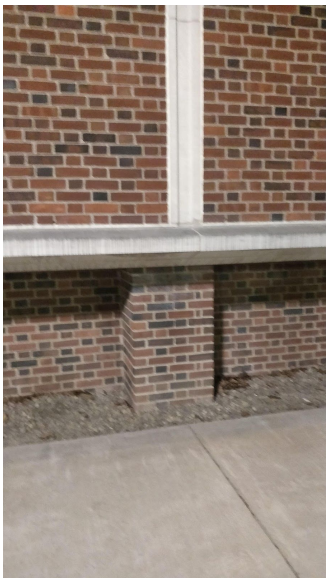
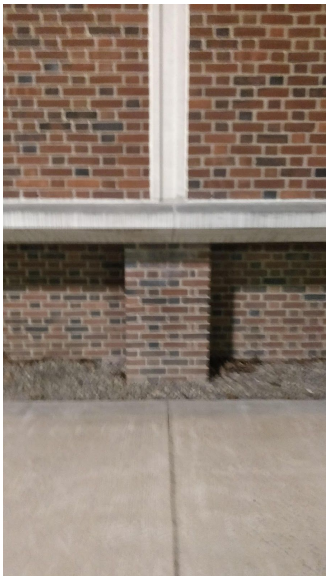
Submitted by: Niranjan Thirusangu, Pakhi Agarwal, Tsung - Yu Hsieh, Visweshwar Srinivasan

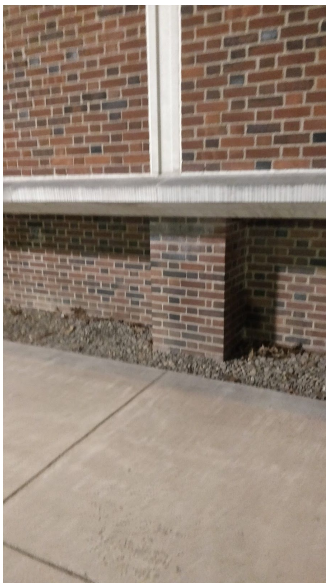
1) Collect a set of images that show the same area of a 3D scene. Your life will be easier if you take these yourself, with a single camera (cell phone camera is OK) at a single focal length (don't zoom in or out between shots) because then the internal camera parameters will be the same. Make sure your "scene" has a dominant planar surface in it, since that is where you are going to put your virtual object. This surface should have a lot of texture to make it easy for COLMAP to find and match features. Ideas for "good" surfaces with lots of corner-like textures include bulletin boards, building facades, wood tabletops, granite countertops, etc. To make your scene more interesting, try to also include some 3D structure besides the dominant plane, that is, don't just take a close up of a single planar surface. I've included an example below of images I took of a little work area in the corner of a room. The posters on the right side wall contain lots of features for matching, and that wall ends up becoming the dominant plane in my reconstructed scene.

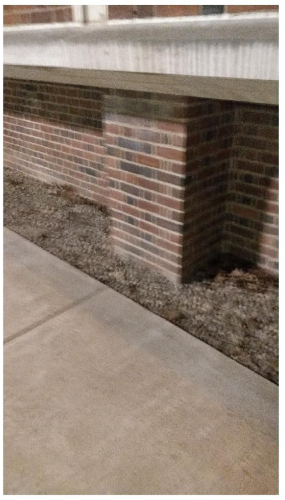
Sol. We collected a set of images having dominant planar surface. The scene consisted of lots of corner like textures.
*this image set is collected from https://colmap.github.io/datasets.html
Figure below shows few images extracted (Fig. 1):

Fig. 1 Input Images

We tried this on another set of images, that we obtained using our camera:



2) Get a working version of COLMAP from https://colmap.github.io/. There are compiled binary versions for Windows, Mac OS X, and Linux at (https://colmap.github.io/install.html). There are also a lot of other resources available on the main site, including documentation and tutorial videos. Play around with COLMAP, and look at the tutorials and docs to figure out how to do the following:

a) input your set of images from part 1,

Sol. We set up COLMAP by creating a New Project (File -> New Project) while creating the new project, it prompted to select the location where we wanted to save the database as well as to select the input images for the COLMAP (Fig. 2).

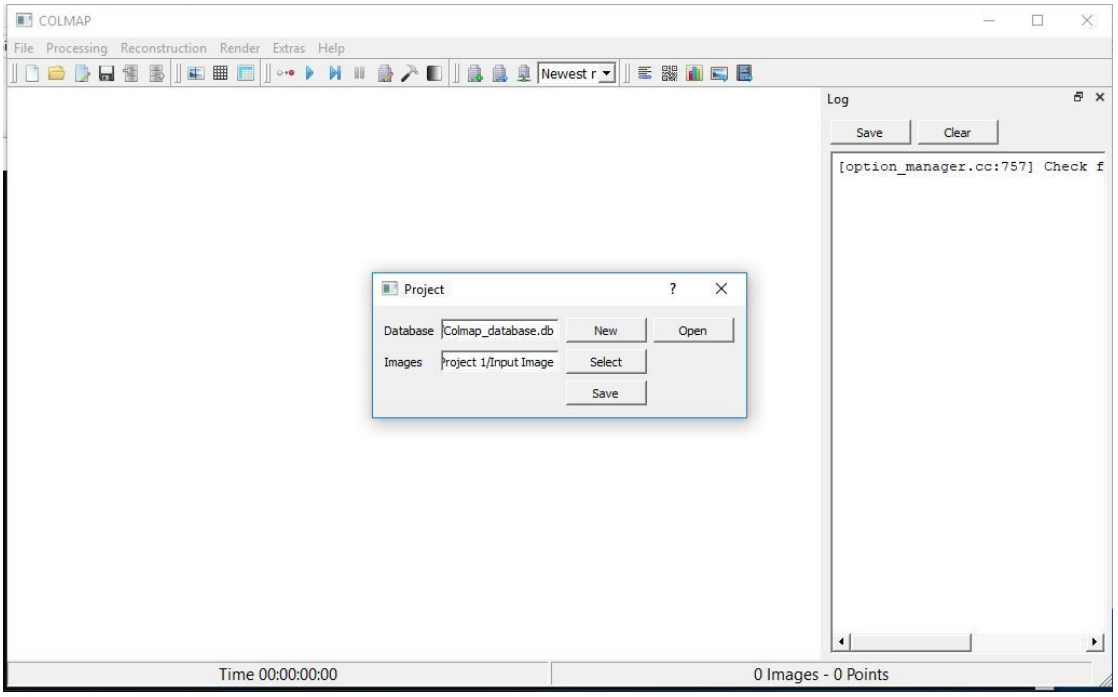


Fig. 2 COLMAP Setup

The whole project was saved as col.ini to retrieve the same configuration settings every time we reopen the project.

b) run feature extraction, matching, incremental reconstruction and bundle adjustment to get a "sparse" reconstruction, which will be a 3D cloud of points. There are a lot of user settable parameters, but it is likely that the default settings will work OK (I would try that first), with the exception of: if you followed my advice and took all the pictures with the same camera, click the box "Shared for all images" for the camera model specification under feature extraction (see figure below). This will tell COLMAP to fit a single set of internal camera parameters using data from all the images, instead of using a different camera model for every image. On the other hand, if you just downloaded a set of images taken by different cameras from the internet, you should not check the box, therefore allowing COLMAP to know each image is from a different camera.

Sol. Feature Extraction -> For Feature extraction, from the menu, we chose Processing -> Extract Features, so as to find sparse feature points in the image which describes their appearance using some kind of numerical descriptors. As we extracted features from a single camera having identical zoom factor, so we shared intrinsics between all images and selected on the box saying "Shared for all images" (Fig. 5), which makes COLMAP fit a single set of internal camera parameters using data from all images (Fig. 3).

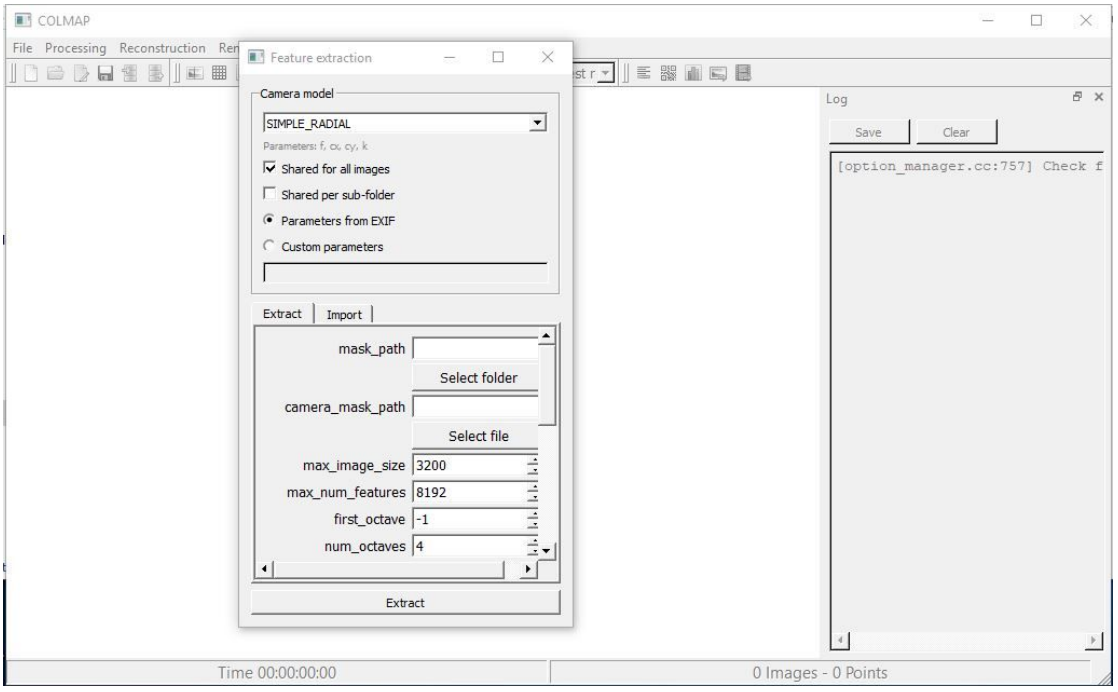


Fig. 3 Feature Extraction

To run Feature extraction successfully we came across many challenges as the program exited every time because of variation in image size or because of some difference in focal length (one of the reasons for change in focal length could be due to movement of hand or change in position while moving camera and taking a couple of images) (Fig. 4).

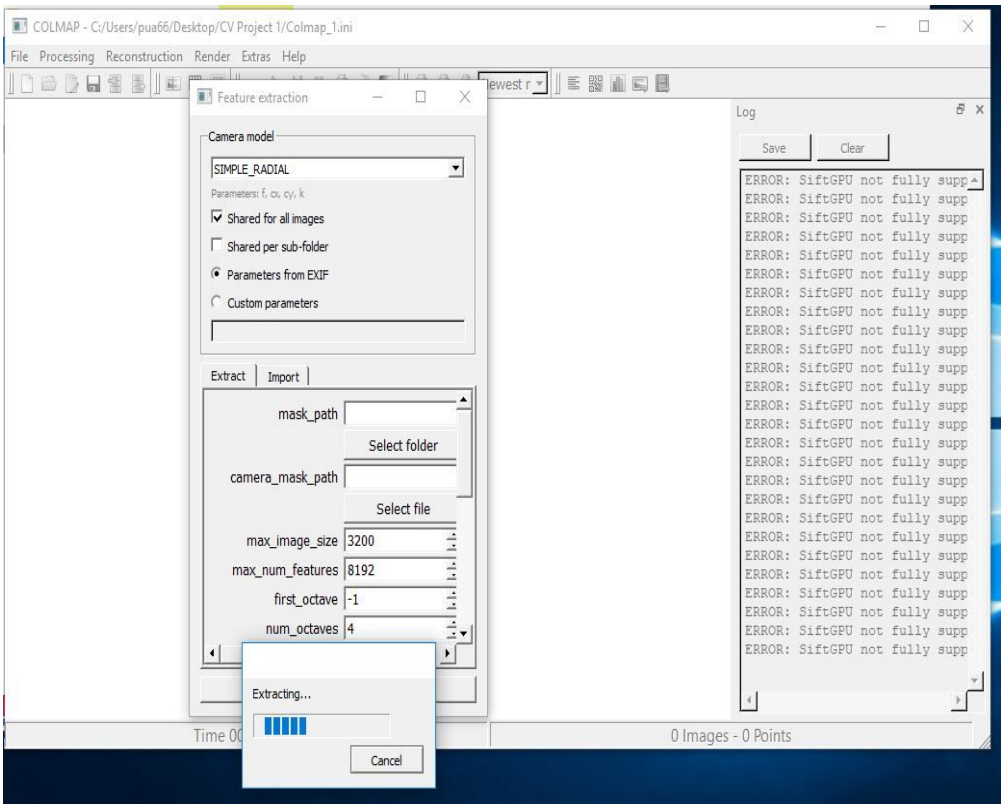


Fig. 4 Extracting Features

We can either detect or extract new features from the image or import existing features from a text file. Once we have done all the settings for feature extraction (Fig. 5) we have to choose Extract to extract all features from the image. We also observed that if we cancel extracting features from the image for the same project the COLMAP automatically continues from where it left off. All the extracted data will be stored in the database file.

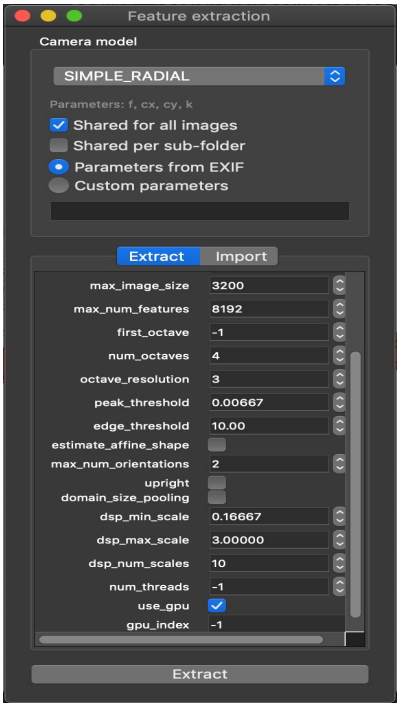


Fig. 5 Settings for Feature Extraction

Feature Matching -> For Feature matching goto Processing -> Feature Matching. There are various types of feature matching, like Exhaustive Matching, Sequential Matching, Vocabulary tree matching, Spatial Matching, Transitive Matching, Custom Matching.

Exhaustive matching is used when the number of images in the dataset is relatively low, this matching is fast enough and leads to the best reconstruction results.

Sequential matching is used when the images are acquired in sequential order, e.g., by a video camera. Here the consecutive images are matched against each other.

Vocabulary Tree matching is used when every image is matched against its virtual nearest neighbour using a vocabulary tree with spatial re-ranking.

Spatial Matching matches every image against its spatial nearest neighbours.

Transitive matching uses transitive relations of already existing feature matches to produce a more complete matching graph.

Custom matching allows specifying individual image pairs for matching or to import individual feature matches.

For this work, we chose Exhaustive Feature Matching (Fig. 7).

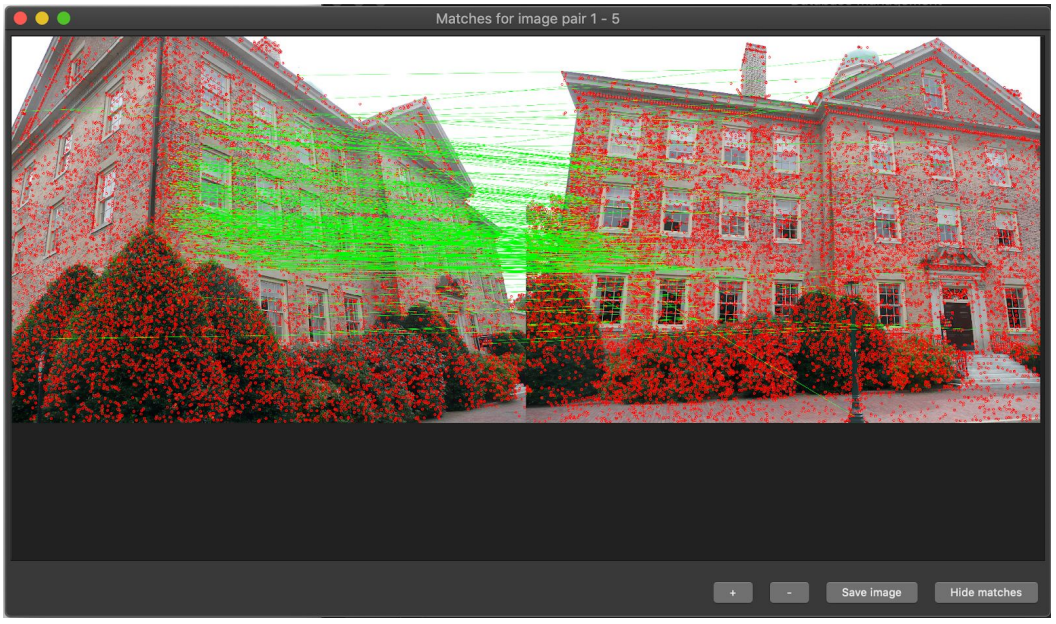


Fig. 6 Feature Matching

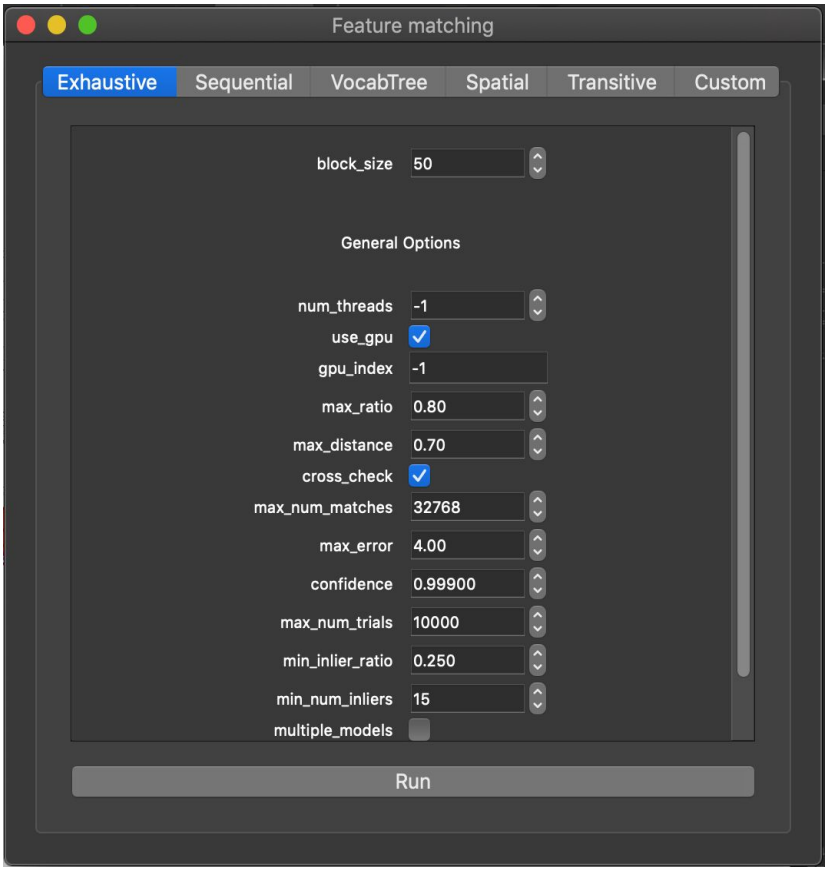
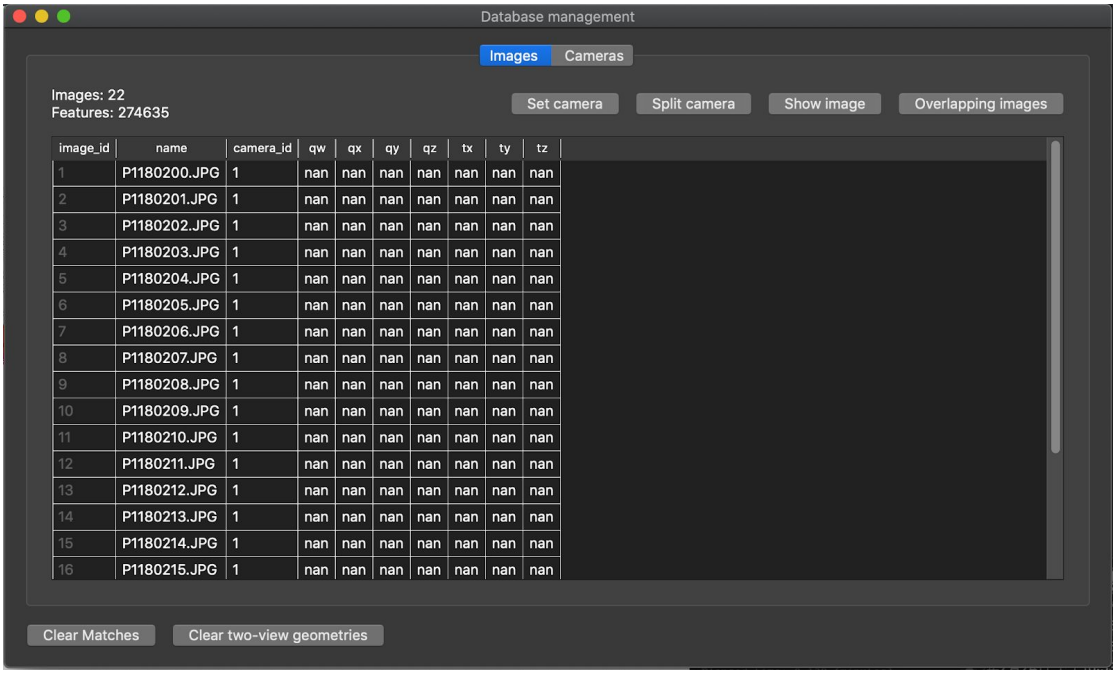


Fig. 7 Settings for Feature Mapping



image_id	name	camera_id	qw	qx	qy	qz	tx	ty	tz
1	P1180200.JPG	1	nan	nan	nan	nan	nan	nan	nan
2	P1180201.JPG	1	nan	nan	nan	nan	nan	nan	nan
3	P1180202.JPG	1	nan	nan	nan	nan	nan	nan	nan
4	P1180203.JPG	1	nan	nan	nan	nan	nan	nan	nan
5	P1180204.JPG	1	nan	nan	nan	nan	nan	nan	nan
6	P1180205.JPG	1	nan	nan	nan	nan	nan	nan	nan
7	P1180206.JPG	1	nan	nan	nan	nan	nan	nan	nan
8	P1180207.JPG	1	nan	nan	nan	nan	nan	nan	nan
9	P1180208.JPG	1	nan	nan	nan	nan	nan	nan	nan
10	P1180209.JPG	1	nan	nan	nan	nan	nan	nan	nan
11	P1180210.JPG	1	nan	nan	nan	nan	nan	nan	nan
12	P1180211.JPG	1	nan	nan	nan	nan	nan	nan	nan
13	P1180212.JPG	1	nan	nan	nan	nan	nan	nan	nan
14	P1180213.JPG	1	nan	nan	nan	nan	nan	nan	nan
15	P1180214.JPG	1	nan	nan	nan	nan	nan	nan	nan
16	P1180215.JPG	1	nan	nan	nan	nan	nan	nan	nan

Fig. 8 Database Management

If we select Processing -> Database Management, we can see the image_id along with image _name, camera_id and a set of x,y,z coordinates to the image (Fig. 8).
We can review and manage the imported cameras, images, and feature matches in the database management tool.

Incremental and Sparse reconstruction -> COLMAP first loads all extracted data from the database into memory and seeds the reconstruction from an initial image pair. Then, the scene is incrementally extended by registering new images and triangulating new points. The results are visualized in "real-time" during this reconstruction process (Fig. 9).

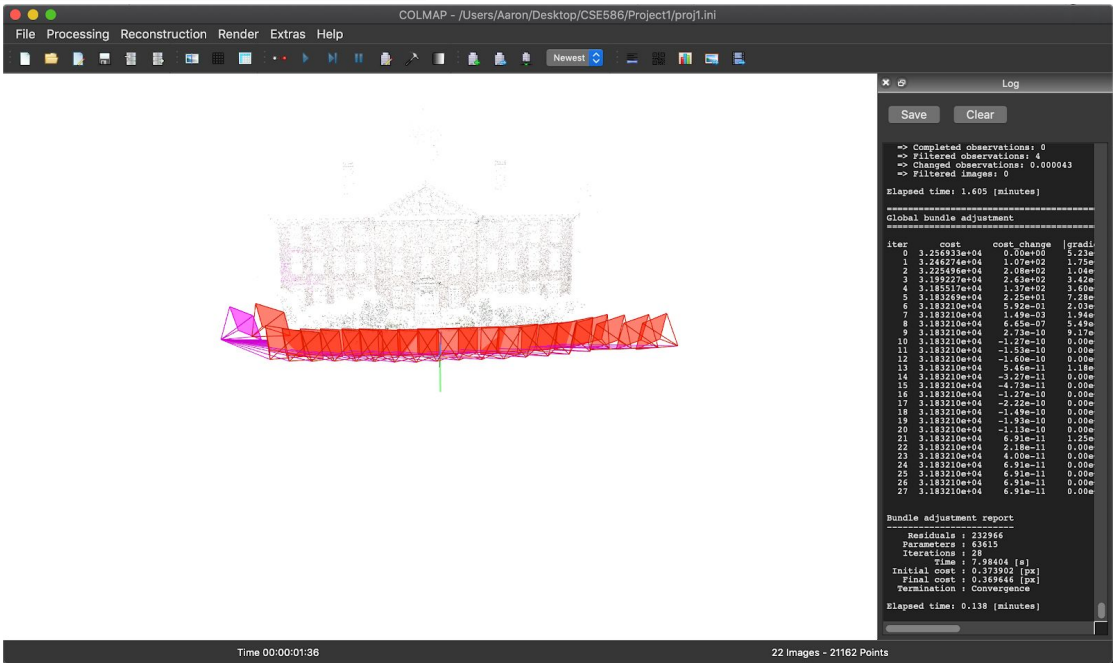


Fig. 9 Reconstruction

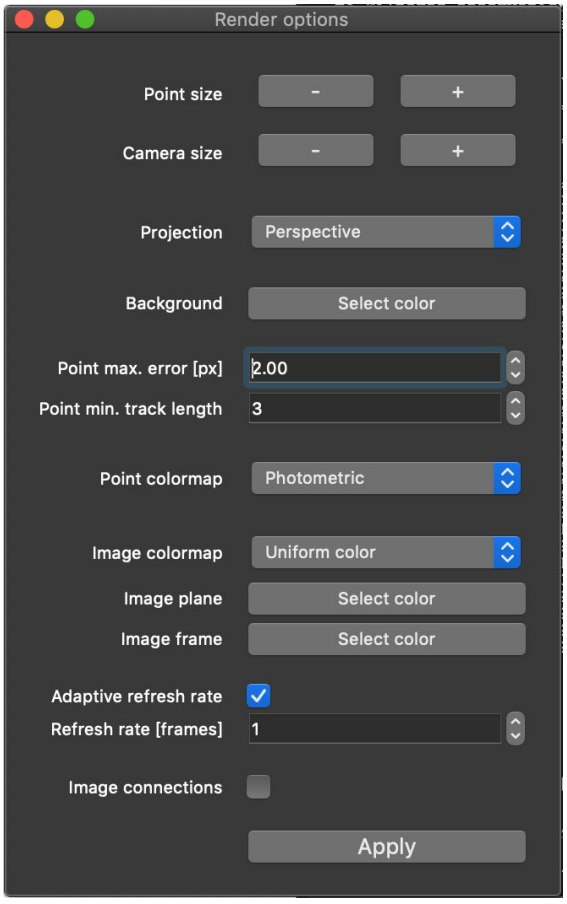


Fig. 10 Render Options

c) export the sparse 3D point model COLMAP produces as a set of text files (see menu item selected in the figure below – by the way, these screenshots are from the Mac binary, and menus for windows or linux might differ slightly). This exporting will output a set of 3 text files: cameras.txt, images.txt, and points3D.txt . These are what you are going to use for the rest of the project. The format of the text files is documented at https://colmap.github.io/format.html

Sol. COLMAP provides several export options for further processing. When exporting in COLMAP's data format, we can re-import the reconstruction for later visualization, image undistortion, or continue an existing reconstruction from where it left off.
For exporting the sparse 3D point model, we need to select File-> Export model as text. After exporting the file, we get 3 text files: cameras.txt, images.txt, and points3D.txt.

3) Read in the 3D point cloud stored in points3D.txt . We only need the X,Y,Z coordinates for each point for what we want to do, and can ignore other fields in the text file.

Sol. COLMAP is a general-purpose Structure-from-Motion (SfM) and Multi-View Stereo (MVS) pipeline with a graphical and command-line interface. It offers a wide range of features for reconstruction of ordered and unordered image collections. As followed in the previous few steps, COLMAP gives 3D image data that it obtains from processing image information. It gives a 3D point cloud that represents the 3D representation of what's on the image.

The point3D.txt file was converted to a .csv extension so as to read the data with more ease. We took only the x, y and z coordinate data into a variable called **X** for each point. The code to do this looks as follows, and we've attached the data contained in the variable below:

```
X = csvread('point3D.csv');
```

Link: This link consists of data contained in the variable X:

drive.google.com/file/d/1obZPPoAtTYzd5YNjOL4tZH4wpP8GGrvC/view?usp=sharing

4) Here is the first legitimate piece of code you need to write: make a RANSAC routine to find the largest subset of 3D points that can be described by the equation of a 3D plane. To do this you need to ask yourself: What is the minimum number of 3D points needed to fit a 3D plane? How to randomly sample a minimal set of points? How to fit the equation of a plane to the points in a sample? How to determine which other points in the model lie close enough to that plane to be considered as "inliers" that should be counted when voting for the current plane fit? How to keep track of the inliers that have voted for the largest coplanar set found so far? After a large number of RANSAC interactions (e.g. several thousand), you will hopefully end up with a large set of inliers that represents the "dominant" plane in the scene, where dominant in this case means the plane that contains the most scene points.

```
%MAIN_METHOD:

clear all
close all
X = csvread('point3D.csv');

f1 = figure;

plot3(X(:,1),X(:,2),X(:,3),'bo');
hold on

[normal,basis,inliers] =
RANSAC_fittingplane(X,'threshold',0.001,'maxIter',50000);
center = mean(X(inliers,:),1);
plot3(X(inliers,1),X(inliers,2),X(inliers,3),'ro');
plot3(center(1),center(2),center(3),'rx');
length = 0.1;
vertices = findVertices(normal,basis,center,length);

plotSquare(vertices,f1);

cameraFile = csvread('camera.csv');
width = cameraFile(1,2);
height = cameraFile(1,3);
focalLength = cameraFile(1,4);
pixelCenter = cameraFile(1,[5,6]);

imgFile = csvread('images.csv');
numImg = size(imgFile,1);
for img = 1:numImg
    imgName = ['P' num2str(imgFile(img,10)) '.JPG'];
    quat = imgFile(img,2:5);
    Trans = imgFile(img,6:8);
    rotM = quat2rotm(quat);
```

```
    f = figure;
    img_temp = imread(imgName);
    imshow(img_temp);
    hold on

plot2Img(f,vertices,rotM,Trans,pixelCenter,focalLength,width,
height);
saveas(f,strcat('final_output_image_',int2str(img)),'jpg');
end
```

Sol. First of all, the minimum number of points needed to describe/fit a 3D plane would be 3. We've written a function here to perform RANSAC routine to identify the biggest subset of 3D points that can be described by the equation of a 3D plane.

Before we delve into this, we would like to explain what the RANSAC algorithm is and how it works. RANSAC stands for Random Sample Consensus. This algorithm is mainly used to perform fitting of lines, curves, or planes using the data that's available, or in other words, to find the "inliners". This is basically done to remove the outliers, so that the least square fitting, which is quite sensitive to outliers. We draw samples out as "batches", see if these are collinear using a simple collinearity rule (written in the function named "isColinear()"), if collinear, we find the normal and distance and store it in an array variable. We test the distance of each point to a minimal distance, we call "delta", using the least square algorithm. These points, that we get are called the inliers.

The number of points to choose comes from the following equation:

$$1 - (1 - (1 - e)^S)^N = p$$

Where:
e = probability that a point is an outlier s = number of points in a sample N = number of samples (we want to compute this) p = desired probability that we get a good sample

Although this could be done, we went ahead choosing the number of iterations as 5000 (several thousand iterations). Thus, using this algorithm, the "out of plane coefficients" and "in-plane batches" are found and passed on to the variables B, P and inliers.

Function Call:

```
[Normal,Basis,inliers] =
RANSAC_fittingplane(X,'threshold',.001,'maxIter',50000);
```

Function:

```
function [Normal,Basis,inliers] =
```

```
RANSAC_fittingplane(X,varargin)

% This function performs RANSAC plane fitting task.
Input X is a N-by-3 matrix, where N is the number of 3D
points. Optional parameters include: 'threshold', which
sets the threshold for selecting inliers, and 'maxIter',
which determines the number of iterations.
% The outputs:
% Normal: the unit normal (column) vector of the planar
surface
% Basis: a 3x2 matrix, where the two column vectors form
an orthonormal basis for the planar surface
% Inliers: a Nx1 indicator vector, where entries with
value 1 indicate inlier samples, and 0 denotes outliers.

numSamp = size(X,1);
if numSamp < 3
    error('Not enough points to fit a plane');
end
% Initialize function parameters
delta = 1;
maxIter = 5000;
% read in user-defined parameters
for i = 1:2:size(varargin,2)
    if(strcmp(varargin{i}, 'threshold'))
        delta = varargin{i+1};
    elseif(strcmp(varargin{i}, 'maxIter'))
        maxIter = varargin{i+1};
    end
end

max_num_inlier = 0;
inlier_list = zeros(numSamp,1);
for iter = 1:maxIter
    % At each iteration, randomly select 3 points to
form the batch, and skip this iteration if the 3 points
are co-linear.
    batch = X(randsample(numSamp,3),:);
    if isColinear(batch)
        continue
    end
    % Find the plane determined by the 3 selected
points.
    plane_coeff = fitplane(batch);

    % Compute the distance between all sample points to
the surface, and identify the inliers.
    dist = (X*plane_coeff(1:3) - plane_coeff(4)).^2;
```

```
    inlier = dist < delta;
    % Update the best candidate if the number of inliers
at the current iteration exceeds the previous one.
    if sum(inlier == 1) > max_num_inlier
        max_num_inlier = sum(inlier == 1);
        inlier_list = inlier;
    end
end

% With the inlier list, find the best fitting plane that
minimizes the squared error.
inliers = inlier_list;
[Normal,Basis,~] = affine_fit(X(inliers,:));


function P = fitplane(X)
% This function fits the plane on the given 3 3D points.
% To do that, we compute two vectors from the 3 points,
and compute the cross product of the two vectors, which
will be the normal vector to the plane determined by the
3 points. Then, plug in any of the three points to find
the constant that uniquely defines the plane.


normal = cross(X(2,:) - X(1,:), X(3,:) - X(1,:));
normal = normal'/norm(normal);

d = X(1,:)*normal;
P = [normal;d];

function r = isColinear(X)

tempX = [X(2,:)-X(1,:);X(3,:)-X(1,:)];
r = rank(tempX) < 2;


function [n,V,p] = affine_fit(X)
% This function finds the best-fitting plane given the
set of inliers in terms of the squared error criterion.
Reference:
https://www.mathworks.com/matlabcentral/fileexchange/433
05-plane-fit

p = mean(X,1);

R = bsxfun(@minus,X,p);

[V,~] = eig(R'*R);
```

```
n = V(:,1);
V = V(:,2:end);
```

5) Display the 3D point cloud produced by COLMAP and denote the inlier points in some way (for example you could draw them in a different color). Make sure your inlier set / dominant plane looks like what you would expect to find. Also include a picture in your project report.
Aside: If you are using Matlab, plot3 is a good function to use to plot a set of 3D points. After plotting, "rotate3D on" is also cool, as it allows you to interactively rotate the view so you can look at it from different angles.

Sol. The data produced in the COLMAP process is a 3D point cloud, this data is further, as described in the above step, converted into inliers and outliers using the RANSAC algorithm. Further, this data is displayed using the following codes.

This code, as it reads, uses a function called "RANSAC_fittingplane", this basically takes the data in variable "X" with a few other key words as seen, such as the threshold, which is taken to be 0.001, and iterations, which is the number of iterations, passed to be 50000.

We figuratively display the 3D point cloud produced by the COLMAP by with the following code.

Function:

```
f1 = figure;
plot3(X(:,1),X(:,2),X(:,3),'bo');
hold on
[B,P,inliers] =
RANSAC_fittingplane(X,'threshold',1,'maxIter',50000);
center = mean(X(inliers,:),1);
plot3(X(inliers,1),X(inliers,2),X(inliers,3),'ro');
legend('Outliers','Inliers')
```

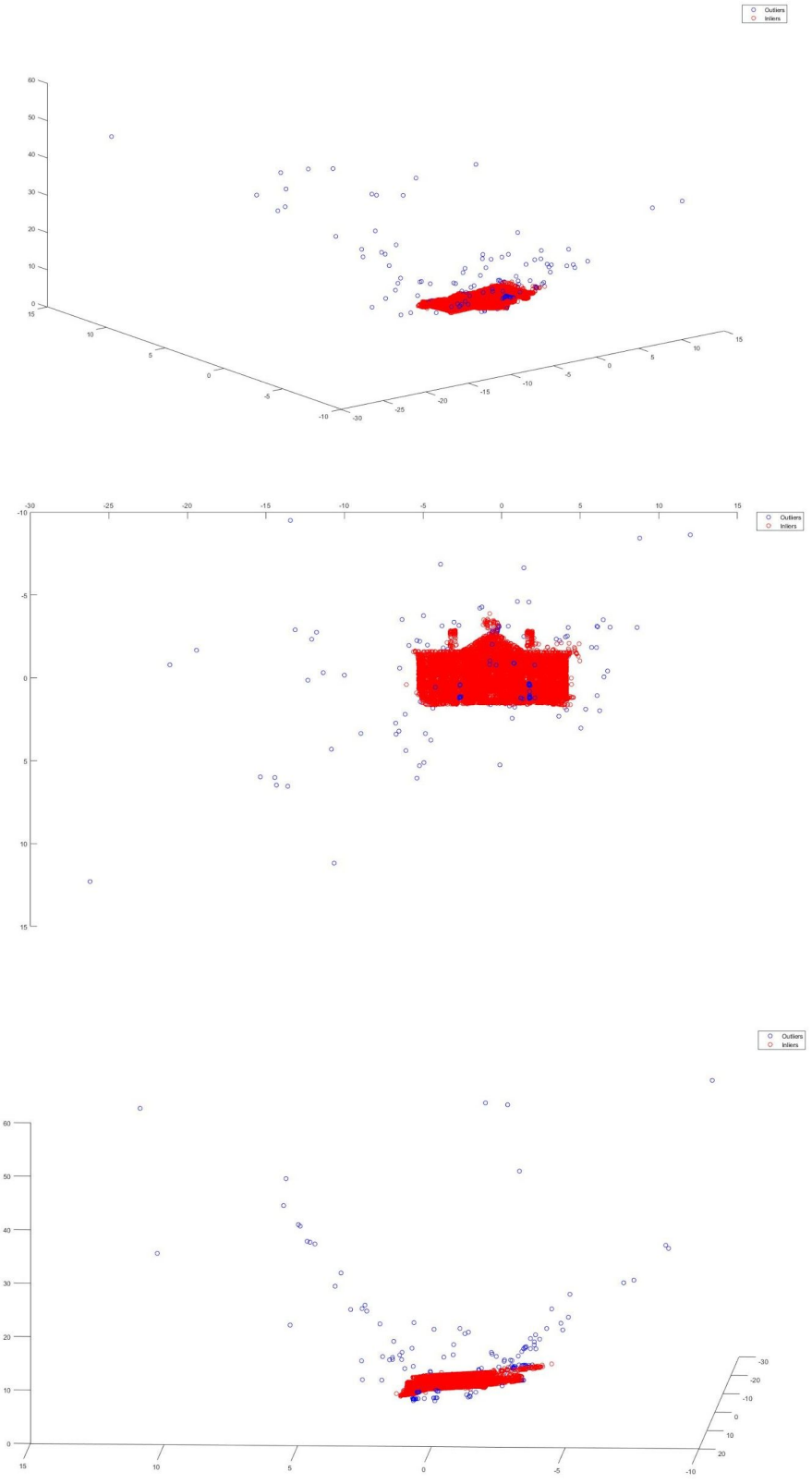


Fig. 11 3D points cloud from COLMAP, outliers are represented as the blue dots and the inliers are represented as the red dots

6) This next part isn't rocket science but it could cause some difficulties if you aren't familiar with coordinate systems and geometric transformations. I'd like you to form a local x-y-z 3D coordinate system where the dominant plane you found above becomes the z=0 plane. Choose the local origin of this coordinate system so that x=0, y=0 lies roughly in the "middle" of the set of inlier points that lie on the dominant plane. Figure out the a 3D Euclidean transformation (rotation and translation) that maps an x,y,z point in your local coordinate system into scene X,Y,Z coordinates.

Sol.

This was quite interesting, we chose the local origin to be the mean of all inliers on the dominant plane. Further, this point was chosen to be x = 0, y = 0 and z = 0. We wrote a function called "findVertices()", which took inputs such as Out of plane coefficients, and In-Plane Coefficients, compute the new coordinates and give the outputs in the variable called "coordinate", which are the coordinates in the new local coordinate system. Hence, this function returns the vertices in a variable named "Vertices", which contains the new coordinates of the points that are passed into the function.

Function Call: `vertices = findVertices(B,P,center,length);`

Function:

```
function coordinate =
findVertices(normal,basis,center,length)
% This function finds the vertices of the cube in
real-world coordinate.
% Inputs: normal, is the normal (column) vector of the
planar surface
% basis, is a 3x2 matrix, where the 2 column vectors
form an orthonormal basis of the plane
% center, is a column vector where the origin of the
local coordinate locates.
% length, is a scalar that determines the length of the
cube.

% Outputs: coordinate, is a 6x3 matrix, where each row
is the 3D coordinate of a vertex of the cube.

% First, determine the 3 unit directional vector of the
local coordinate in terms of real-world coordinate.
u_z = normal';
u_x = basis(:,1)';
% u_x = u_x/norm(u_x);
u_y = basis(:,2)';
% u_y = u_y/norm(u_y);

% Find the coordinate of the vertices in local
```

```
coordinate.
vertices = [
    length,length,0;
    length,-length,0;
    -length,-length,0;
    -length,length,0;
    length,length,2*length;
    length,-length,2*length;
    -length,-length,2*length;
    -length,length,2*length;
    ];
coordinate = zeros(8,3);

% transform the vertices to real-world coordinate, which
is done by summing up the proper multiple of the
orthonormal basis vectors and translating according to
the center.
for i = 1:8
    coordinate(i,:) = center + vertices(i,1)*u_x +
vertices(i,2)*u_y + vertices(i,3)*u_z;
end
```

7) Create a virtual object to put in the scene. For simplicity, you can just make it a 3D box. Define a 3D box with bottom surface lying in your local z=0 plane, so that the center of that rectangular bottom surface is centered at x=0, y=0,z=0. Form the 8 corners of the box in x,y,z coordinates, and convert them into scene X,Y,Z coordinates using the transformation you defined in part 6. At this point, it might be helpful to visualize the box in 3D, to make sure all is well. In Matlab, you could do this by drawing the rectangular surfaces of the box in the same window where you have drawn your 3D points. This web page might be helpful: http://www.matrixlab-examples.com/3d-polygon.html . Interactively rotate the view to see if the box is where you hoped it would be (and debug if not).
At this point we have placed a virtual object within a 3D scene coordinate system that was created from a set of images. We are now going to project that virtual object into each of those images, and overlay it on the original pixel values. This is the "augmented reality" part!

Sol.

This was the most interesting part of the entire project, we got to learn how to realize augmented reality using this. This part needs us to plot a cube, we all know that cubes are 3 Dimensional. We define a box whose bottom surface lies on the local z = 0 plane. The bottom surface is centered at the local origin (x = y = z = 0. The 8 corners/vertices of the cube were formed using the "plotSquare" function, where we have used transformations of the last question, to convert one coordinate system to another. We draw 6 different rectangular surfaces,

which are connected by the vertices, this enables us to form a cube with the required side length, located on the origin of the local coordinate system.

Thus, we were successfully able to place the cube within a 3D scene coordinate system, with the set of images.

Function Call: `plotSquare(vertices,f1);`

Function:

```
function plotSquare(vertices,f)
% This function plots the cube in real-world coordinate.
% Inputs: vertices, is a 6x3 matrix, where each row
contains one vertex
% f1, is the figure to plot in

figure(f)
hold on

p1 = vertices(1,:);
p2 = vertices(2,:);
p3 = vertices(3,:);
p4 = vertices(4,:);

x = [p1(1) p2(1) p3(1) p4(1)];
y = [p1(2) p2(2) p3(2) p4(2)];
z = [p1(3) p2(3) p3(3) p4(3)];

fill3(x,y,z,1);

p1 = vertices(1,:);
p2 = vertices(2,:);
p3 = vertices(6,:);
p4 = vertices(5,:);

x = [p1(1) p2(1) p3(1) p4(1)];
y = [p1(2) p2(2) p3(2) p4(2)];
z = [p1(3) p2(3) p3(3) p4(3)];

fill3(x,y,z,1);

p1 = vertices(1,:);
p2 = vertices(5,:);
p3 = vertices(8,:);
p4 = vertices(4,:);

x = [p1(1) p2(1) p3(1) p4(1)];
y = [p1(2) p2(2) p3(2) p4(2)];
```

```
z = [p1(3) p2(3) p3(3) p4(3)];

fill3(x,y,z,1);

p1 = vertices(2,:);
p2 = vertices(6,:);
p3 = vertices(7,:);
p4 = vertices(3,:);

x = [p1(1) p2(1) p3(1) p4(1)];
y = [p1(2) p2(2) p3(2) p4(2)];
z = [p1(3) p2(3) p3(3) p4(3)];

fill3(x,y,z,1);

p1 = vertices(4,:);
p2 = vertices(3,:);
p3 = vertices(7,:);
p4 = vertices(8,:);

x = [p1(1) p2(1) p3(1) p4(1)];
y = [p1(2) p2(2) p3(2) p4(2)];
z = [p1(3) p2(3) p3(3) p4(3)];

fill3(x,y,z,1);

p1 = vertices(5,:);
p2 = vertices(6,:);
p3 = vertices(7,:);
p4 = vertices(8,:);

x = [p1(1) p2(1) p3(1) p4(1)];
y = [p1(2) p2(2) p3(2) p4(2)];
z = [p1(3) p2(3) p3(3) p4(3)];

fill3(x,y,z,1);
```

NOTE: Although we've attached more pictures for this after considering Step 10, we have attached one of the many pictures here Fig. 12, for reference.

Fig. 12 One of the images on which a virtual cube is placed

8) Read in the external and internal camera parameters from files cameras.txt and images.txt . The internal parameters are in cameras.txt , and if you followed my suggestion of having COLMAP estimate just a single set of internal parameters across all images, then there is only one set of internal parameters in the file. Refer to https://colmap.github.io/format.html to see what parameters are specified and in what order. The external parameters (rotation and translation) describing camera pose for each image are in images.txt . There are a lot of other numbers in that file that we don't care about, so refer to the documentation to see which lines of the file you want to read. Note that rotation is represented as a unit quaternion. It is OK if you use a predefined routine to convert from quaternion to 3x3 rotation matrix. In Matlab, that function is quat2rotm.

Sol.

As we've been asked, we read the 'camera.txt' (in our case, we used a camera.csv file for easiness), and the parameters in the file were saved into different variables such as "width", "height", "focalLength" and "pixelCenter". We did this with the help of the reference put on the link specified. The external parameters (rotation and translation) describing camera pose for each image, as we all know, were saved in "images.txt" file (we further changed the saved extension to "images.csv" for easiness), This file is used to get information about rotation and translation, and the documentation is used to read those parameters. We further use inbuilt predefined routines to convert the quaternion to 3x3 rotation matrix, since we've used Matlab, the function we used is "quat2rotm".

The code we did for this is what is pasted below:

Codes:

```
cameraFile = csvread('camera.csv');
width = cameraFile(1,2);
height = cameraFile(1,3);
focalLength = cameraFile(1,4);
pixelCenter = cameraFile(1,[5,6]);

imgFile = csvread('images.csv');
numImg = size(imgFile,1);
imgName = ['P' num2str(imgFile(img,10)) '.JPG'];
quat = imgFile(img,2:5);
Trans = imgFile(img,6:8);
rotM = quat2rotm(quat);
```

9) This is the last significant piece of vision-related code you will write. Refer to our lectures on camera models and associated reference material to write your own routine that takes external and internal camera parameters and projects a given set of 3D (X,Y,Z) points into a set of 2D (col,row) pixel locations. DO NOT USE OpenCV or other vision code to do this, I want you to write it yourself. However, feel free to look at other source code if you need some guidance on how other people implement camera projection. In fact, it might be a good idea to find the camera projection function in COLMAP that matches your kind of camera model (e.g. SIMPLE_RADIAL) since that will exactly define what the 3D to 2D projection model is for the set of parameters you have.

Sol.

We used the lecture slides (Lectures 12, 13, 14 and 15 from CV - 1) as reference and wrote our own routine that took external and internal camera parameters and projects a given set of 3D (X, Y and Z) points into a set of 2D (col, row) pixel locations.

The following code that we've attached basically plots the 3D cube that we placed in one of the few steps before, as a 2D object.

Function Call:

```
for img = 1:numImg
    imgName = ['P' num2str(imgFile(img,10)) '.JPG'];
    quat = imgFile(img,2:5);
    Trans = imgFile(img,6:8);
    rotM = quat2rotm(quat);
```

```
    f = figure;
    img_temp = imread(imgName);
    imshow(img_temp);
    hold on
plot2Img(f,vertices,rotM,Trans,pixelCenter,focalLength,w
idth,height);

end
```

Function:

```
function
plot2Img(fig,vertices,rotM,trans,pixelCenter,focalLength
,width,height)

% This function plots the cube to the original image.
% Inputs: fig, is the figure to plot in;
% vertices, is a 6x3 matrix, where each row contains a
vertex;
% rotM, is a 3x3 rotation matrix for this image;
% trans, is a column translation vector for this image
% pixelCenter, is a row vector denoting the pixel index
of the image center;
% focalLength, is a scalar value denoting the focal
length of this camera model;
% width and length, are scalar values that denote the
number of pixels in width and in length.


vertices_pix = zeros(8,2);
depth_vertex = zeros(8,1);
for i = 1:size(vertices,1)
    % First transform the real-world coordinate to
camera coordinate, and record the depth of the vertex
    vertices_c = (-rotM*vertices(i,:)' - trans')';
    depth_vertex(i) = vertices_c(3);
    % transform the camera coordinate to film coordinate
    vertices_f = focalLength*(vertices_c/vertices_c(3));
    % transform film coordinate to pixel coordinate
    vertices_pix(i,:) = pixelCenter + vertices_f(1,1:2);
end
% determine the depth of each surface
depth_surf = zeros(6,1);
surf = [
    1 2 3 4;
    1 2 6 5;
    1 5 8 4;
```

```
    2 6 7 3;
    4 3 7 8;
    5 6 7 8;
  ];
for i = 1:6
    depth_surf(i) =
min([depth_vertex(surf(i,1)),depth_vertex(surf(i,2)),dep
th_vertex(surf(i,3)),depth_vertex(surf(i,4))]);
end
% sort the depth of each surface and draw the surfaces
in deep to shallow order.
[~,depth_surf_order] = sort(depth_surf,'descend');
figure(fig)
for i = 1:6
    p1 = vertices_pix(surf(depth_surf_order(i),1),:);
    p2 = vertices_pix(surf(depth_surf_order(i),2),:);
    p3 = vertices_pix(surf(depth_surf_order(i),3),:);
    p4 = vertices_pix(surf(depth_surf_order(i),4),:);

    x = [p1(1) p2(1) p3(1) p4(1)];
    y = [p1(2) p2(2) p3(2) p4(2)];
%     z = [p1(3) p2(3) p3(3) p4(3)];

    fill(x,y,1);
end
```

10) Finally, for each of your original images, project the corners of your 3D virtualbox object into the image and draw it overlaid over the original pixels values. You should at least draw a wireframe representing the edges of the box. However, the visualization will be more compelling if you draw filled polygons for each projected surface of the box, so that pixels representing parts of the scene that are behind the box from a particular camera viewpoint will be hidden from view. It's actually a little nontrivial to do this, since you also want faces of the box that are closer to you to hide faces of the box that are farther away. I believe that for a nice convex shape like a box, you can achieve this effect by drawing the faces of the box in order from farthest away to closest, determined by figuring out distance along the principle axis OF THE CAMERA (aka depth) to each of the corner points of the box, and ordering the faces by min depth to any of the 4 points bounding each face. This strategy is related to the concept of Z-ordering in computer graphics. The reason to do this more complicated kind of drawing instead of a wireframe is that it gives a more realistic look to your virtual object – the box appears "solid" because it occludes the scene behind it.

Be aware that, depending on the structure of your particular 3D scene, any scene features IN FRONT of the virtual box from a particular camera view will not appear correct – they will look like the box is in front of them, hiding them from view, instead of occluding the box as they would if it had been a real object in the scene. It is OK if this happens to you, because it is a very hard problem to solve without segmenting out surfaces in the scene and computing their depths. Even "professional" SDKs like ARKit for iOS do NOT currently attempt to do this.

Sol.

We finally project the corners of our 3D Virtualbox object into the image and draw it overlaid over the original pixel values, for each of our original images. The edges of the box were represented by using projected surfaces of polygons that were filled up. We drew the boxes in nice convex shape like a box by drawing the faces of the box in order from farthest away to closest. We did this using the inbuilt sort function, in descending order.

Fig. 13a Represents a subset of images with a virtual cube

Fig. 13b Represents a subset of images with a virtual cube

Fig. 13c Represents a subset of images with a virtual cube

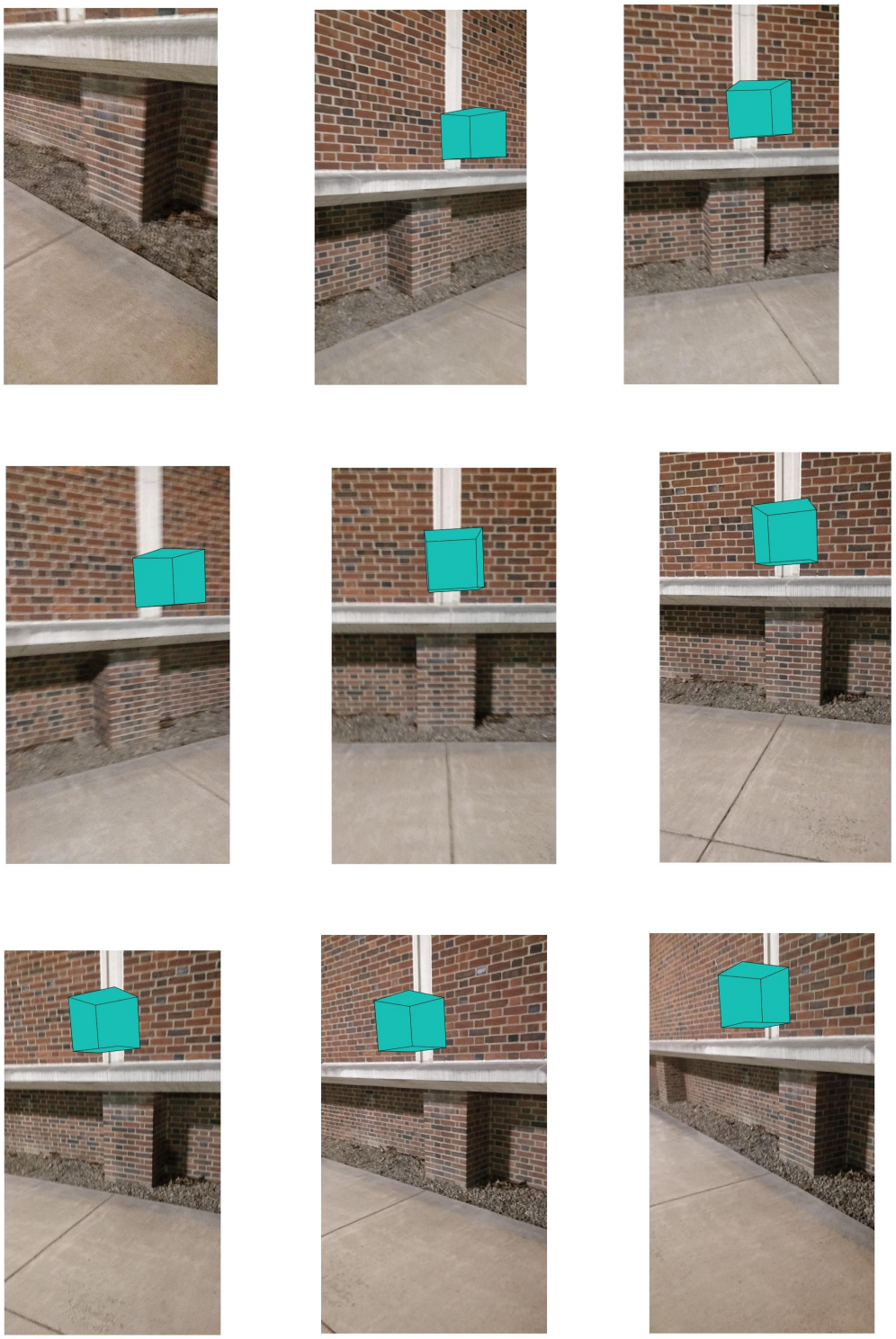


Fig. 14 This set of images are those we tried with inputs from a phone camera, as we can see, due to smaller amounts of complexities compared to previous ones, we can find that the cube is formed perfectly well.

Team Contributions:

Niranjan Thirusangu	**Tasks 1, 2,3, 4, 5, report**
Pakhi Agarwal	**Tasks 1, 2,3, 4, 5, report**
Tsung - Yu Hsieh	**Tasks 6, 7, 8, 9, 10, report**
Visweshwar Srinivasan	**Tasks 6, 7, 8, 9, 10, report**