Lydia Qiao
ID: 45837573

Final Project Report

Project Overview

For this final project, the problem I am trying to solve is reconstructing a 3D teapot using the provided scanned images. To do that, I need to calibrate the camera, prune out the photo's background, form meshes for different views, and combine all the meshes to get the 3D teapot. My goal is to form a 3D teapot mesh that looks the same as the one in the real world.

Data

I used all images in the calib_jpg_u and teapot folder. In calib_jpg_u, there are 40 images with resolution 1920x1200; Left and right camera each takes 20. In the teapot folder, there are seven different views of the teapot. For each aspect, each camera provides 40 pictures for binary encoding, one colorful image, and one background image; All those pictures also have 1920x1200 resolution.

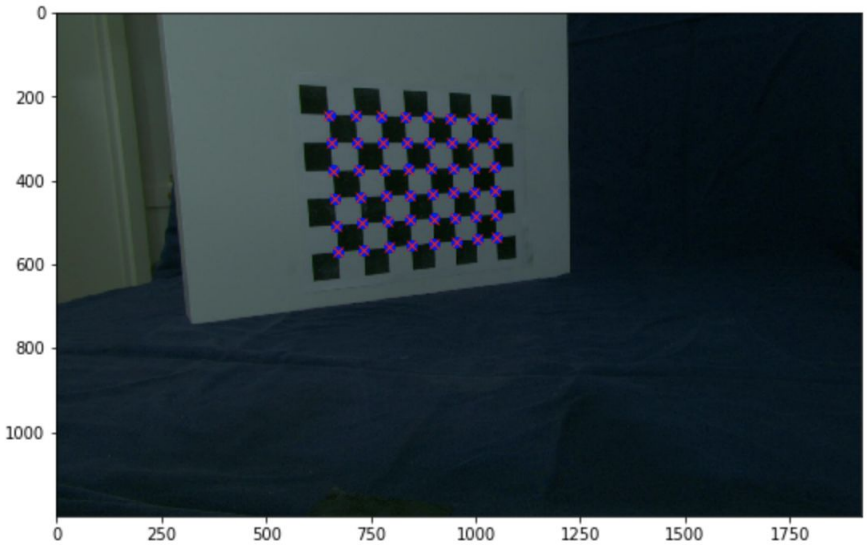
Algorithms

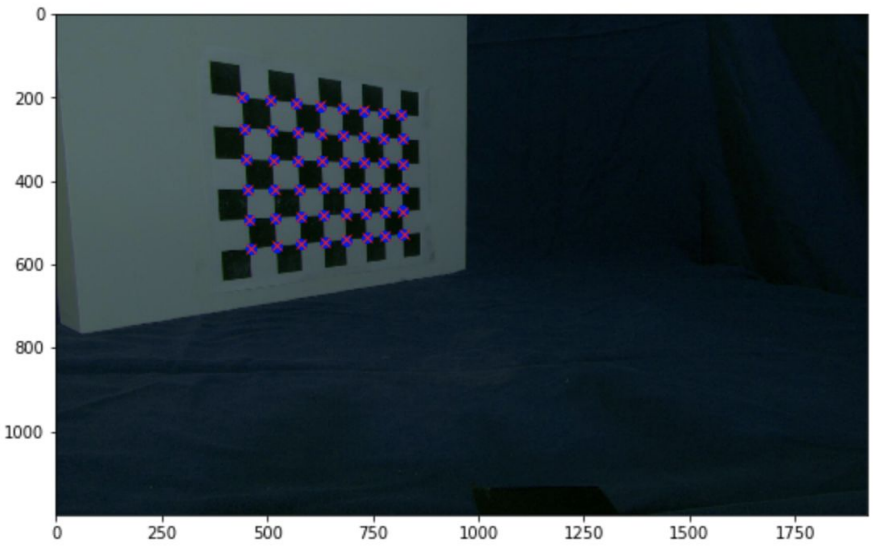
I implemented four principle algorithms, which are camera calibration, color decoding and mask, triangulation, and pruning.

Camera calibration

To calibrate the camera, I first want to get the intrinsic parameter for the left and right cameras. The provided calibrate.py produces a pickle file that stores the intrinsic parameter fx,fy,cx,and cy after importing it. After importing that python file and getting the parameters, I use fx+fy to get an overall focal length f, so that I can create two cameras using those intrinsic parameters. Now, we have two cameras with a default parameter. To find out whether that calibration is correct, I called cv2.findChessboardCorners with frame_C0_21.jpg and frame_C1_21.jpg, and triangulated two cameras to visualize the result. As the images show below, the red crosses and blue dots overlay with each other, which proves the calibration is correct.

```
intrinsic_param = pickle.load(open('/Users/ooo/Documents/CS117/Project/calibration.pickle','rb'))
fc= (intrinsic_param['fx']+intrinsic_param['fy'])/2

# create Camera objects representing the left and right cameras
# use the known intrinsic parameters you loaded in.
camL = Camera(f=fc,c=np.array([[intrinsic_param['cx'],intrinsic_param['cy']]]).T,
              t=np.array([[0,0,0]]).T, R=makerotation(0,0,0))
camR = Camera(f=fc,c=np.array([[intrinsic_param['cx'],intrinsic_param['cy']]]).T,
              t=np.array([[1,0,0]]).T, R=makerotation(0,0,0))
# load in the left and right images and find the coordinates of
# the chessboard corners using OpenCV
imgL = plt.imread('calib_jpg_u/frame_C0_21.jpg')
ret, cornersL = cv2.findChessboardCorners(imgL, (8,6), None)
pts2L = cornersL.squeeze().T

imgR = plt.imread('calib_jpg_u/frame_C1_21.jpg')
ret, cornersR = cv2.findChessboardCorners(imgR, (8,6), None)
pts2R = cornersR.squeeze().T

# generate the known 3D point coordinates of points on the checkerboard in cm
pts3 = np.zeros((3,6*8))
yy,xx = np.meshgrid(np.arange(8),np.arange(6))
pts3[0,:] = 2.8*xx.reshape(1,-1)
pts3[1,:] = 2.8*yy.reshape(1,-1)

# Caliberation for 2 cameras

params_init = np.array([0,0,0,0,0,-2])

camL = calibratePose(pts3,pts2L,camL,params_init)
camR = calibratePose(pts3,pts2R,camR,params_init)

print("Left",camL)
print("Right",camR)

# As a final test, triangulate the corners of the checkerboard to get back there 3D locations
pts3r = triangulate(pts2L,camL,pts2R,camR)
```





Binary decoding and color mask

Since the decode and reconstruction function were provided, I decided to create a new function called color_decode to keep things separate and organized. This function produces the left and right camera's color mask to erase the background and a 3N array that stores the RGB values of each scanned point.

```
def color_decode(pts2L,pts2R,imprefix,start,threshold):
    mask_t=[]
    colorL = plt.imread(imprefix+"C0_00.png")
    colorFL= plt.imread(imprefix+"C0_01.png")
    colorR= plt.imread(imprefix+"C1_00.png")
    colorFR= plt.imread(imprefix+"C1_01.png")
    cL_list=[]
    cR_list=[]
    for i in range(pts2L.shape[1]):
        cL_list.append(colorFL[pts2L[1][i]][pts2L[0][i]])
        cR_list.append(colorFR[pts2R[1][i]][pts2R[0][i]])
    cL=np.array(cL_list).T
    cR=np.array(cR_list).T
    rbg=(cL+cR)/2
    colorL= colorL-colorFL
    colorR= colorR-colorFR
    mask_t= np.asarray(mask_t)
    mask_t=mask_t.astype(int)
    maskL= np.zeros((colorL.shape[0],colorL.shape[1]))
    maskR= np.zeros((colorR.shape[0],colorR.shape[1]))
    undecodableL=np.where(np.abs(colorL) >threshold )
    undecodableR=np.where(np.abs(colorR) >threshold )
    for i in range(undecodableL[1].size):
        maskL[undecodableL[0][i]][undecodableL[1][i]]=1
    for i in range(undecodableR[1].size):
        maskR[undecodableR[0][i]][undecodableR[1][i]]=1

    return rbg,maskL,maskR
```

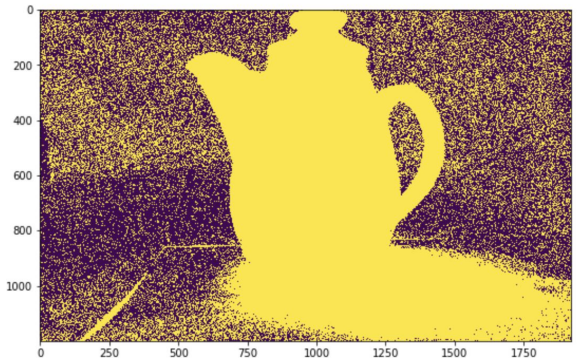


Figure 1: Left color mask

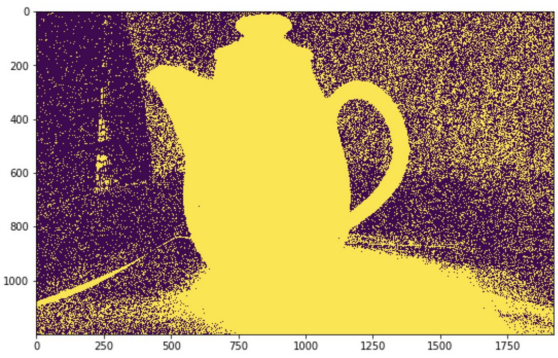
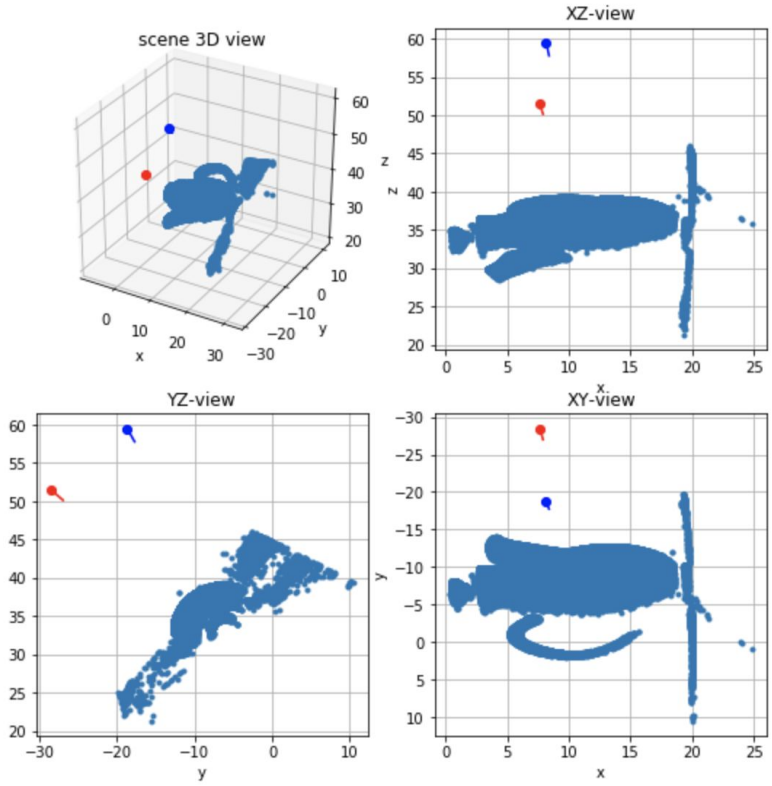


Figure 2: Right color mask

After reconstructing the teapot and mask out the background, I visualized each aspect to see whether they are correct. The following are views for aspect 0. You can see they are redundant points near the bottom of the mesh. The functions can not mask out those points because they are the shadow of the teapot. Pruning will handle those excess points.



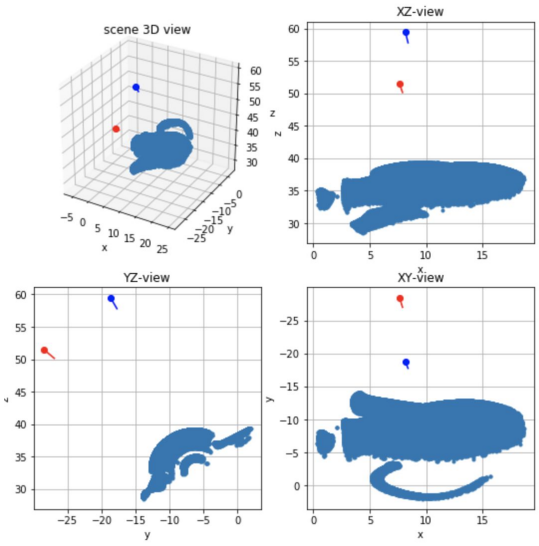
Pruning

This pruning algorithm is similar to assignment 4's pruning method. It prunes the points based on the bounding box and triangle. First, I determined a proper limitation and then dropped points in pts3, pts2L, pts2R, and the RBG that are outside that range. After doing that, I called Delaunay.simplices to get the corresponding points for triangulation. Then, I removed triangles

from the surface mesh that include edges that are longer than an input threshold. That threshold varies according to the performance of different aspect's mesh. If some points are no-longer connected to any neighbor in the mesh, I delete that point too. Finally, I remapped all the indices so that the triangle array can make correct reference after all those points are deleted.

```
def dis(a,b,c,tri):
    return np.sqrt(np.sum(np.power((pts3.T[tri[a][b]]-pts3.T[tri[a][c]]),2), axis=0))
def clean_up(boxlimits,trithresh,pts2L,pts2R,pts3,color):
    #
    # bounding box pruning
    #
    drop=[]
    for i in range(pts3.shape[1]):
        if (pts3[0][i]<boxlimits[0] or pts3[0][i]>boxlimits[1] or
            pts3[1][i]<boxlimits[2] or pts3[1][i]>boxlimits[3] or
            pts3[2][i]<boxlimits[4] or pts3[2][i]>boxlimits[5]):
            drop.append(i)
    pts3=np.delete(pts3, drop, axis=1)
    pts2L=np.delete(pts2L, drop, axis=1)
    pts2R=np.delete(pts2R, drop, axis=1)
    color=np.delete(color, drop, axis=1)
    #
    # triangulate the 2D points to get the surface mesh
    #
    tri= Delaunay(pts2L.T).simplices
    #
    # triangle pruning
    #
    drop_tri=[]
    for i in range(tri.shape[0]):
        if (dis(i,0,1,tri)>trithresh
        or dis(i,0,2,tri)>trithresh or dis(i,1,2,tri)>trithresh):
            drop_tri.append(i)

    tri=np.delete(tri,drop_tri,axis=0)
    tokeep=np.unique(tri)
    map = np.zeros(pts3.shape[1])
    pts3=pts3[:,tokeep]
    color = color[:,tokeep]

    map[tokeep] = np.arange(0,tokeep.shape[0])
    tri=map[tri]

    return pts2L,pts2R,pts3,color, tri
```

After cleaning up the redundant points, I used the writeply function to generate the mesh and store it into the ply file. The following picture is aspect 0's points after the pruning. We can see all the points are reasonable now. Repeating the previous steps on other aspects to get 7 meshes, I import them into the Meshlab so that I can align and smooth them to generate the final mesh.

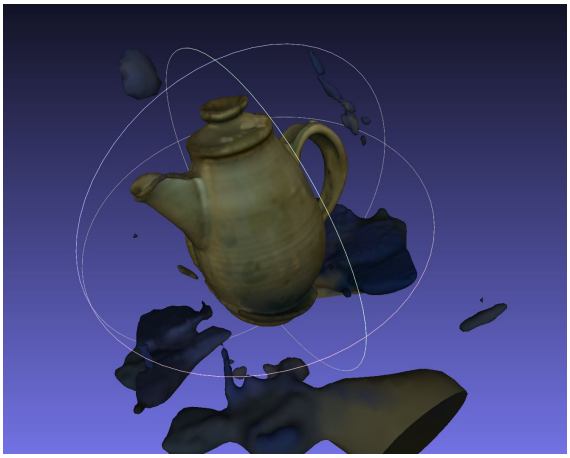


Results

The following is the final 3D teapot mesh that I successfully reconstructed using Screen Reconstruction: Screened Poisson and Smooth Face normal function in Meshlab and above algorithms.

During the assembling of multiple scans, I find out that the function Screen Reconstruction: Screened Poisson can effectively prune out the noisy point. In my opinion, if I provide more points, the final mesh will be more accurate. Thus, I experimented about whether it will do a better job if I do not prune any meshes at all.

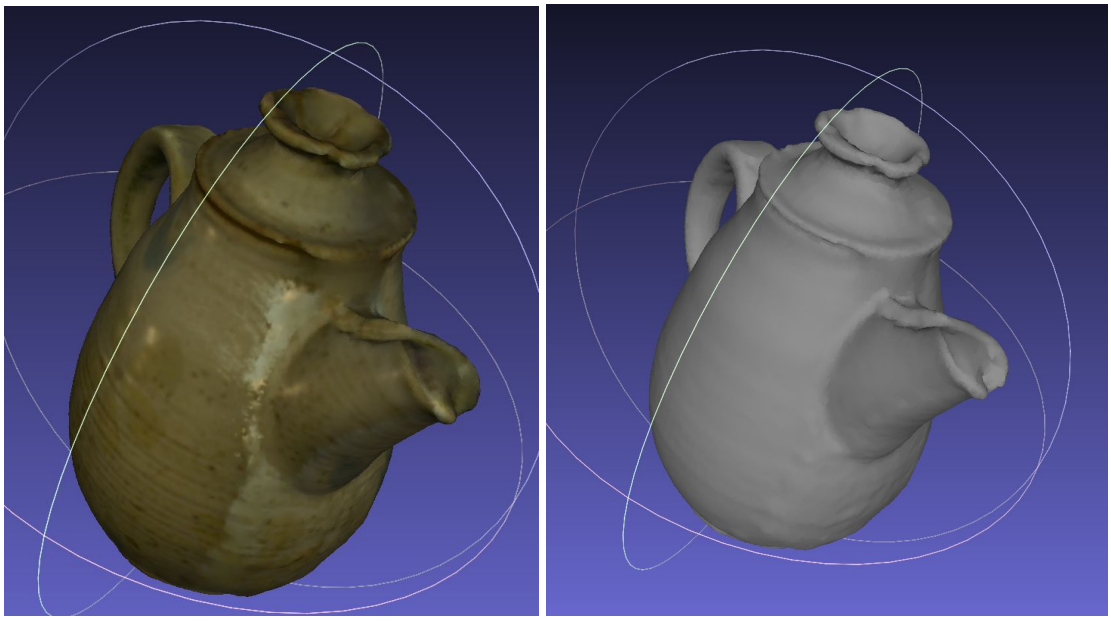
The following mesh is formed without pruning beforehand. As we can see, it has some objects other than the teapot itself. The function considers those noisy points, mostly from shadows, as a surface. After reconstructing them, it creates additional meshes. In conclusion, pruning each aspect's mesh before reconstruction is a better choice.



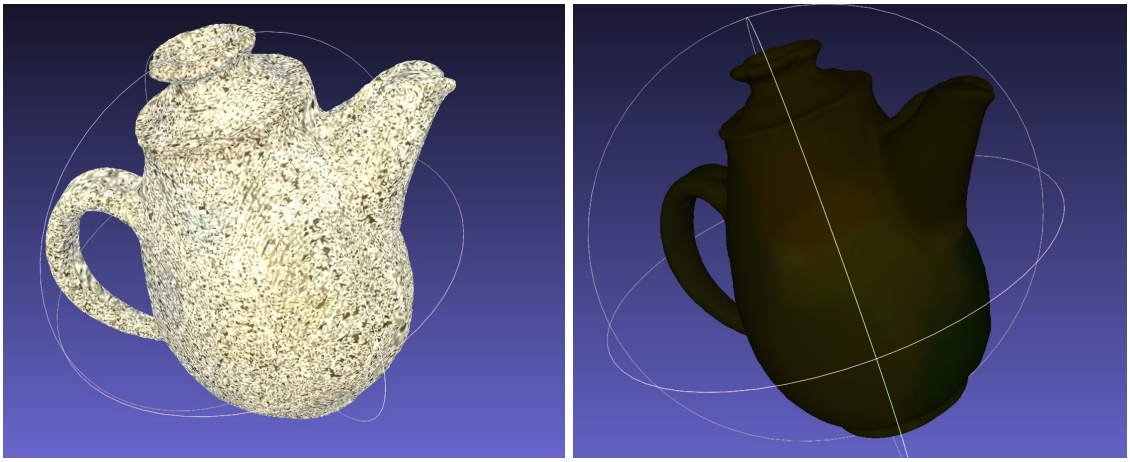
Assessment and Evaluation

I think it is practicable to use a computer algorithm to solve everything other than combining different teapot aspects. When using Meshlab to stitch two aspects' meshes together, I realized that it is hard to use the point-based gluing function to align them perfectly, because I was not able to determine which points are the same in those two meshes. The principle shape of the teapot is similar to a sphere, and the only significant structures are the handle and the spout. Thus, there are not many points that I can tell are the same just by looking at it. Besides, I can't see the texture of the meshes during Align tool mode, which also increases the difficulty of locating points. Due to all those reasons, the function came up with an inaccurate alignment caused by the poorly paired points. Thus, I have to align the meshes to form a good looking one manually.

One issue of my mesh is that the colors of different aspect mesh can not transform smoothly. You can see there is a clear boundary in the left picture, but the uncolored mesh in the middle looks smooth.



Then, I tried two ways to smooth the color by vertex and my face, but both did not work. The left image shows the effect of smoothing color by vertices, which creates a piebald look. The right image shows the effect of toning color by faces, which unifies the color too much and makes the texture disappear.



After spending plenty of time on this problem, I went back and checked the scan images again. Then, I realized that some aspects have unique lighting; they might be taken from a different time. To build a better system if I have time, I would like to put the object into a room that has no window. If all the lights come from the lamp or the lighting setup, the color of each aspect will be consistent since there is no brightness variation.

This project successfully reached the final goal and produced a decent final mesh. I am satisfied with the overall outcome.

Appendix

All the functions in my project are either provided by the professor or written entirely from scratch by myself. To ensure the correctness of all the details, I used many provided features such as camera class, makerotation, triangulate, residuals, calibratePose, reconstruct, vis_scene,and writeply.