Model Generation from Structured Light Scans

Brandon Ha

bha3@uci.edu

June 12, 2020

Abstract

Converting two-dimensional scans to a three-dimensional model is no trivial task; however, through the meticulous experimentation and application of various algorithms and techniques, it is possible to generate satisfactory results as demonstrated in this report. The report reinforces the idea that the quality of the result is heavily dependent on every step of the process from collecting the initial scans and camera calibration to surface reconstruction and additional clean-up processes.

1 Project Overview

The problem in question revolves around the task of using a series of two-dimensional pictures to create something that is not only three-dimensional but also representative of the object in the given pictures. Therefore, the goal of the project is to take multiple scans of an object and create a three-dimensional object through a series of steps. More specifically, the majority of project entails the following:

- Reading structured light scans captured from two separate cameras
- Decoding the structured light patterns from each scan
- Generating meshes based on these decoded scans
- Cleaning up and smoothing the meshes
- Aligning and merging the generated meshes into one combined mesh
- Performing Poisson surface reconstruction on the combined mesh

This is just a short list of the numerous intricacies required to produce a good quality model. Arguably, a significant reason as to how the result is able to look as accurate as it does is due to the time and effort dedicated to tweaking the countless variables and parameters throughout the many processes whether it be calibration, surface reconstruction, decoding, object mask computation, or mesh smoothing.

2 Data

For this project, I opted to use a set of structured lights scans provided by Professor Charless Fowlkes due to the limited resources available at the time. Out of the four different objects he provided, I chose to create a model of the sculpture of a couple because I was interested in replicating the fine detail and surface texture of the object. Although the scans could be higher resolution, the resolution of the pictures at 1920 pixels by 1080 pixels prove to be adequate for our purposes.

2.1 Calibration

In order to calibrate the cameras, we were provided with 20 pictures from 20 different angles of a chessboard for both cameras. By varying the view of the chessboard, we can compute the intrinsic and extrinsic parameters of the cameras without having to create a complicated three-dimensional calibration setup.

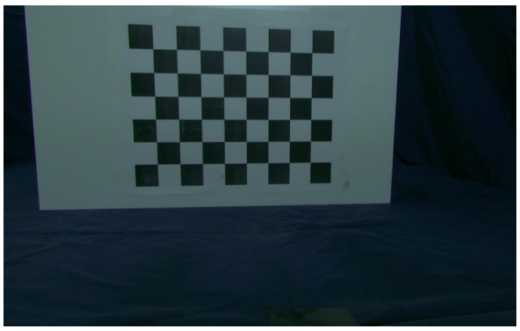


Figure 1 Left Camera - First View

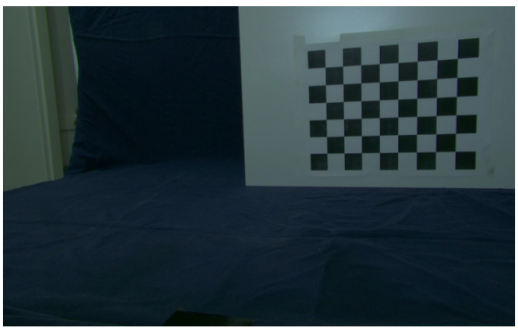


Figure 2 Right Camera - First View

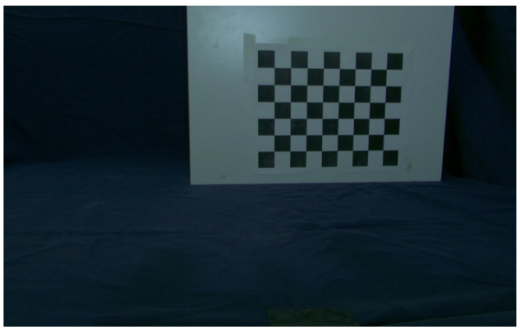


Figure 3 Left Camera - Second View

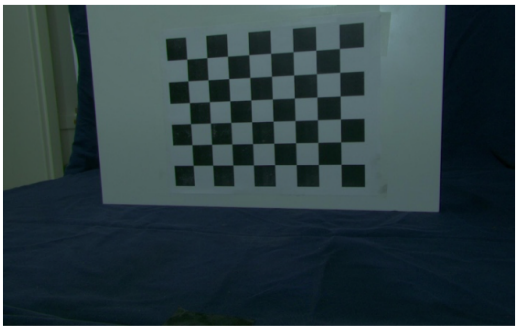


Figure 4 Right Camera - Second View

2.2 Main Dataset

The main dataset consisted of seven scans each composed of 84 images with the exception of one of the scans. Due to an error likely related to the image acquisition script used, the two images used for object mask computation, were missing. This meant that we could either disregard the scan completely or alternatively, manually create the missing image by filling where the object is with a color sufficiently different from any of the colors of the object. After alleviating this issue by masking the object completely with black pixels, we have all of the images necessary for the object masks and pixel decoding.

The seven scans are captured from different angles to maximize the coverage of the figure's surface. The scans can be broken down into two frames per camera where the scene is completely lit, one with the object and one without, and 40 frames per camera with projected structured light. By calculating the difference between the two frames, we can calculate which pixels the object occupies in the pictures for the scan. The 40 frames per camera can be further separated into projections of a binary gray code with 10 bits to project 10 different patterns, horizontally and vertically, and their respective inverses so that we can compare the corresponding frames and account for dark parts on the object. (Refer to Figures 5-12 on Page 3)

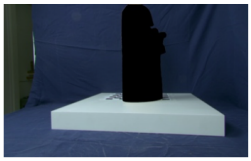


Figure 5 Manually Created Background Image

Figure 6 Example Background Image (for another scan)

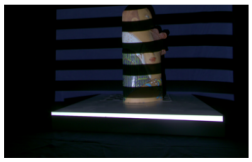


Figure 7 Horizontal Structured Light Scan (Right Camera)

Figure 8 Vertical Structured Light Scan (Left Camera)

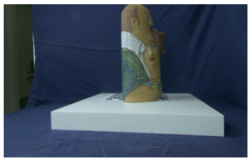


Figure 9 Corresponding Foreground Image

Figure 10 Corresponding Foreground Image (for another scan)

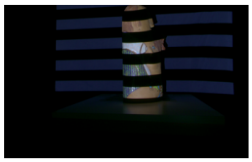


Figure 11 Inverted Horizontal Structured Light Scan (Right Camera)

Figure 12 Inverted Vertical Structured Light Scan (Left Camera)

3 Algorithms

Although I have implemented many of the following algorithms I am about to discuss, I chose to use the modules and functions provided by Professor Fowlkes because using the function provided allowed me to focus on other aspects of the project such as configuring mesh smoothing, alignment, and surface reconstruction.

3.1 Camera Calibration

The main algorithm used to calibrate the cameras remain mainly untouched from the provided files. I loaded the calibration images and ensured that the calculated calibration parameters were loaded correctly through the resulting pickle file. In terms of the methodology, I used OpenCV's ***findChessboardCorners*** function to locate the chessboard corners and estimate the cameras' intrinsic parameters, more specifically, the focal lengths and principal points of each camera. This is possible because the geometry of the chessboard in the real world is known which leaves us with a homogeneous linear system that OpenCV's ***calibrateCamera*** function solves.

After computing the intrinsic parameters, we need to estimate the extrinsic parameters of the cameras. When you move the chessboard around, the intrinsic parameters stay the same, but the extrinsic parameters (rotation and translation) for the two cameras change. With that said, the relative locations of the two cameras stays constant, so we can use an arbitrary pair of calibration scans to estimate the extrinsic parameters. For this project, I decided to use the first pair of calibration images and ran the provided ***calibratePose*** function that essentially solves a least squares problem to minimize the distance between the known two-dimensional points and the generated three-dimensional point coordinates of the chessboard points.

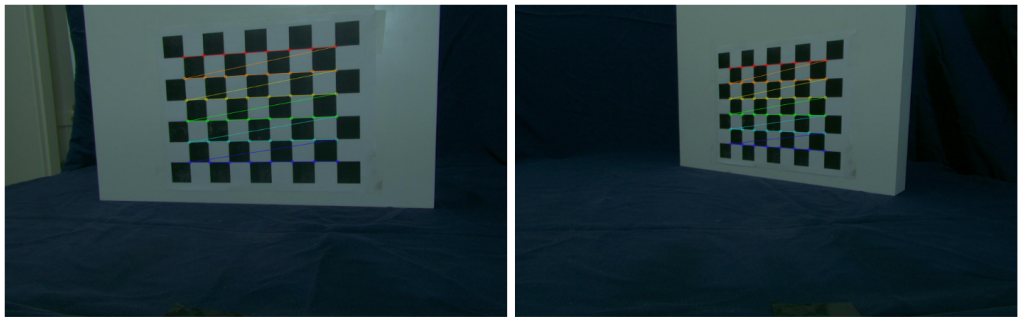


Figure 13 Calibration Visualization

3.2 Mesh Generation

Generating high quality meshes is a critical step in the formation of a 3D model. Given that the mesh generation was explored in previous assignments, I was able to modify the provided code to handle color for each 3D point. Much of the mesh generation procedure is found in the ***reconstruct*** function while some aspects such as the internal function parameters are determined outside of the function. The next six subsections detail the main portions of the mesh generation process.

3.2.1 Scan Decoding

I used the provided function responsible for decoding the projected gray code for each frame and computing the mask indicating which pixels are likely to be inaccurately decoded. The function converts the images to grayscale and float value arrays after loading them in and then recovers a bit for every pixel that has a greater value in one image versus its inverse. Meanwhile, we maintain a binary array to mark difficult-to-decode pixels that have differences between pairs of scans above a given threshold. Finally, after going through all frames, I converted the resulting gray code to standard binary and used the binary code to calculate the decimal value.

3.2.2 Object Mask Computation

In terms of object mask computation, I heavily based my code on the provided mask generation implementation in the scan decoding function. The framework of the code can be fundamentally broken down into reading the background and foreground images, converting them to float values and grayscale, and lastly, thresholding the corresponding pairs of images. We can then use the generated masks with the already-computed decoding masks.

3.2.3 Point Matching and Triangulation

Once the cameras are calibrated and necessary masks are computed, I used NumPy's ***intersect1d*** function to find intersections between the left camera and right camera scans by inputting the respective masked codes. With the calculated indices, we map the intersecting points for the two cameras as well as the three-dimensional representation of said points with the provided triangulation function.

The ***triangulate*** function takes advantage of the fact that we know the intrinsic camera properties and the 2D coordinates of each matching point. This becomes a least squares problem which can easily be solved with a call to NumPy's least squares solver. Averaging the left camera and right camera results gives us a good estimate of the three-dimensional point coordinates.

After the 2D and 3D points are computed, I made sure to pull the color values from the original foreground scan for later use when writing to the mesh files (ending in .ply).

3.2.4 Boundary Box Pruning

Depending on the chosen thresholds for object mask computation and decoding, you will get varying amounts of noise and point density. To remove clear outliers due to low thresholds, boundary box pruning is used. Extensive trial and error is needed for this process. The general approach requires 2D graphs of the cameras in relation to the triangulated points. Boundary box pruning is a relatively hands-on strategy in which the user manually selects hard limits on each coordinate plane to remove extrema. My approach involves making a mask that marks points that violate the user-provided limits and removing the offending points.

3.2.5 Delaunay Triangulation

To form the triangles for the mesh, I used SciPy's Delaunay triangulation function due to its simplicity and effectiveness. Delaunay triangulation is a very diverse algorithm for computing triangulation and is known for its ability to form relatively even triangles. This triangulation evenness is a result of the Delaunay criterion which follows the empty circumcircle criteion. According to the circumcircle criterion, no point outside of a triangle's vertices may be located within the circumcircle of its

corresponding triangle. [1] Thus, triangles with larger angles are favored over those with smaller angles. This serves as a good base for the mesh, but the algorithm can still form long triangles.

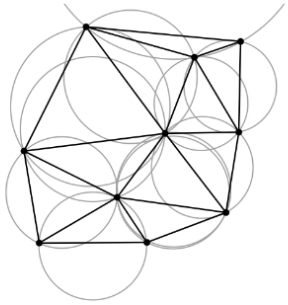


Figure 14 Delaunay Triangulation with Circumcircles [2]

3.2.6 Triangle Pruning

Triangulation can result in undesirably long triangles which can reduce mesh quality. As a result, I used code from a previous assignment to prune these triangles. I made a mask which indicated which triangles with edges shorter than a given threshold. Then, I only kept those triangles to process. In doing so, it leaves points that are not referred by any triangles.

3.2.7 Removing Unreferenced Points

Because triangle pruning can leave many unreferenced points, it is important to remove points that fall in this criteria. Following Professor Fowlkes notes, I was able to remap the remaining vertices listed in the array of triangles. The code in question can be seen below.

```
# Remapping triangles
tokeep = np.unique(tri)
remap = np.full(pts3.shape[1], -1)
remap[tokeep] = np.arange(0, tokeep.shape[0])
tri = remap[tri]

# Remove loose points
pts3 = pts3[:,tokeep]
ptsColor = ptsColor[:,tokeep]
pts2L = pts2L[:,tokeep]
pts2R = pts2R[:,tokeep]
```

3.3 Mesh Smoothing

While the mesh is fully generated at this point, the quality tends to be quite low without some form of smoothing. So, I created a function to smooth each generated mesh. The general idea is to record the neighbors of every point. With this information stored in a dictionary, I was able to index into the 3D point array and calculate the average coordinate for any given points' neighbors. This greatly improves the smoothness of the mesh and minimizes any remaining floating point clusters.

[1] https://www.mathworks.com/help/matlab/math/delaunay-triangulation.html
[2] https://en.wikipedia.org/wiki/Delaunay_triangulation

4 Results

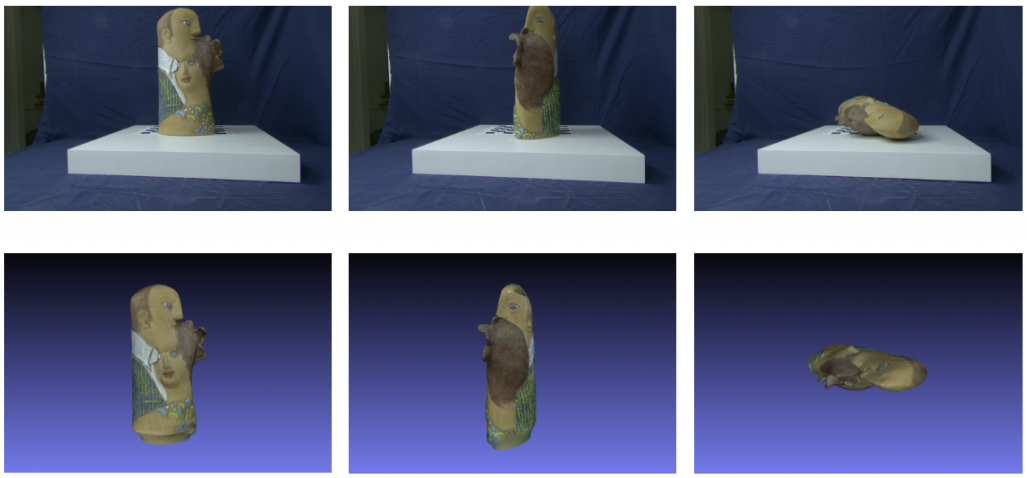


Figure 15 Original Scans versus Final Mesh

4.1 Final Model

Looking at the final result and comparing it side-by-side to the original object, we can clearly see that the generated model captures the basic geometry and texture of the original object. There are spots where the model misrepresents portions of the figure, mainly the bottom and the top of the object. The provided scans did cover most of the surface of object, but the two spots where the model has the most difficulty are the spots where the scans fail to capture. Again, near the bottom of the model, you can see that some of the patterning in the woman's clothing is somewhat blurry. With that being said, the model proves to be quite detailed and even displays dimensionality throughout as shown by the height of woman's nose and the green lines in the man's jacket.

4.2 Effects of Processing

Through my research and experimentation, I learned that reconstruction quality is heavily reliant on many different aspects of the process.

4.2.1 Boundary Box and Triangle Pruning

Figure 16 Original Mesh

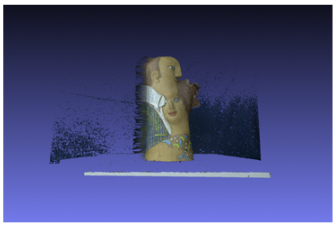


Figure 17 Mesh without Boundary Box Pruning

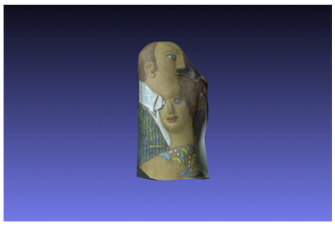


Figure 18 Mesh without Triangle Pruning

Although boundary box pruning and triangle pruning act as simple forms of processing, the effect they have on the mesh is undeniably remarkable. Boundary box pruning removes many of the points that get past the different masks. While you could change the object mask and decoding thresholds to minimize this kind of noise, it can compromise the point density of the object we are trying to resolve. Triangle pruning removes the noise that is around the object elegantly as seen in Figure 18.

4.2.2 Mesh Smoothing

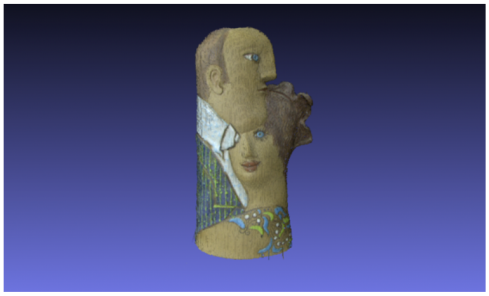


Figure 19 Mesh without Smoothing

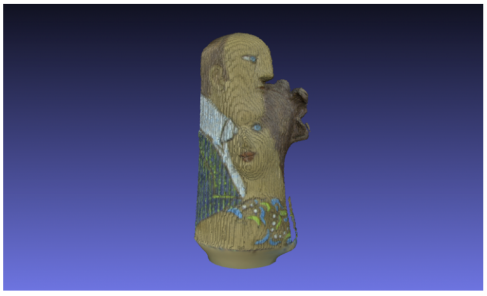


Figure 20 Aligned Model without Smoothing

Smoothing also plays an important role in the final mesh. Comparing the original mesh with the mesh without smoothing (Figures 16 and 19), you may be able to notice the graininess of the unprocessed mesh. This effect is further exacerbated once the the meshes are merged and surface reconstruction is performed.

4.2.3 Surface Reconstruction

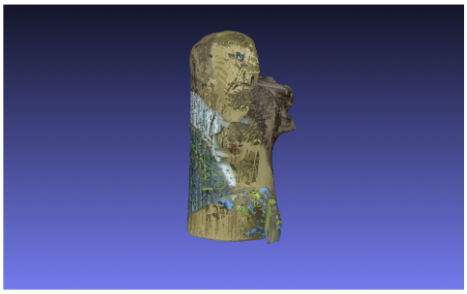


Figure 21 Merged Mesh without Surface Reconstruction

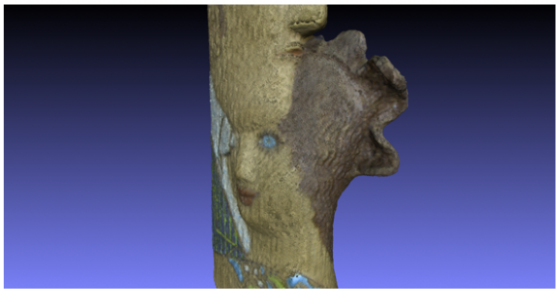


Figure 22 Poisson Surface Reconstruction with Reconstruction Depth = 12

Poisson surface reconstruction brings together the merged meshes and creates a mesh that is smoother and more refined by solving an implicit function in which the normal vectors on the surface of the object are represented by a gradient. Figure 21 shows that without surface reconstruction, the model is not closed and has texture issues because the different meshes are visible in different patches on the surface of the object. MeshLab's Poisson surface reconstruction handles this issue by taking the average color for each corresponding point in each mesh. Additionally, I went through different values for reconstruction depth which profoundly changed the smoothness of the model's surface. At a reconstruction depth of 12, there was too much artificial surface texture which is why I kept experimenting until I found an accurate result at a depth of 9. What we are left with is a relatively high-quality 3D model.

5 Assessment and Evaluation

5.1 Limitations of Approach

The approach I ended up using proved to lack much configurability despite the inclusion of the thresholds and parameters discussed in earlier sections. The final mesh still has blurring issues and does not recognize that the bottom of the figure does not extend as far the model suggests. Without more user-provided input or more sophisticated surface reconstruction methods, these issues will continue to bottleneck the system's ability to create high-quality models.

5.2 Limitations of Data

The dataset provided for this figure is lacking in that the scans do not completely cover the full surface of the original object. If the appropriate scans were provided, then the model would be more geometrically correct. Resolution seem to sufficient as detail is retained even after the mesh smoothing.

5.3 Final Thoughts

Given the resources available and significant amount of time spent on this project, I believe that the final result is more than satisfactory. The dataset provided did leave some work for the algorithms to improvise and as such, the model was unable to completely render the bottom and part of the top of the object. With that being said, the result does capture the overall essence of the original figure. The task would be considered as a relatively easy task for a computer algorithm to solve when you consider the vast range of problems that a computer algorithm *can* solve. If I had more time, I would likely use another algorithm to smooth the meshes and explore different surface reconstruction methods. It became evident that my current implementation of the mesh smoothing lacks control other than the amount of iterations it should perform. Consequently, I view it as one of the weakest links in the pipeline.

6 Appendix

The following section serves as a quick guide for the code and process used in the project. If you would like to see more of the code, please refer to the source code. The relevant functions have comments that should clarify any confusions.

6.1 Calibration

Calibration is mostly unchanged from the provided code in **calibrate.py**. Some code from Assignment 3 is used to estimate the extrinsic parameters as shown here:

Calibration

[5]:
```
if not os.path.exists("calibration.pickle"):
    import calibrate
with open("calibration.pickle", "rb") as calib_file:
    calib_data = pickle.load(calib_file)
f = (calib_data["fx"] + calib_data["fy"]) / 2
c = np.array([[calib_data["cx"]],[calib_data["cy"]]])

# load in the left and right images and find the coordinates of
# the chessboard corners using OpenCV
imgL = cv2.imread('./calib_jpg_u/frame_C0_01.jpg')
ret, cornersL = cv2.findChessboardCorners(imgL, (8,6), None)
pts2L = cornersL.squeeze().T

imgR = cv2.imread('./calib_jpg_u/frame_C1_01.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)

# Camera calibration (extrinsic parameters)
paramsL = np.array([0,0,0,0.5,0.5,-2.5])
paramsR = np.array([0,0,5,0.5,0.5,-3])

camL = camutils.Camera(f=f,c=c,R=camutils.makerotation(0,0,0),t=np.zeros((3,1)))
camR = camutils.Camera(f=f,c=c,R=camutils.makerotation(0,0,0),t=np.zeros((3,1)))

camL = camutils.calibratePose(pts3,pts2L,camL,paramsL)
camR = camutils.calibratePose(pts3,pts2R,camR,paramsR)
```

```
Estimated camera intrinsic parameter matrix K
[[1.40530791e+03 0.00000000e+00 9.62167369e+02]
 [0.00000000e+00 1.40389402e+03 5.90915957e+02]
 [0.00000000e+00 0.00000000e+00 1.00000000e+00]]
Estimated radial distortion coefficients
[[-5.23053889e-03  8.05964692e-02  2.01737110e-05 -3.90735917e-03
  -1.08096617e-01]]
Individual intrinsic parameters
fx =  1405.3079107410456
fy =  1403.8940216103824
cx =  962.167369260617
cy =  590.9159568140669
```

6.2 Mesh Generation

Mesh generation is a modified version of the my mesh generation code in Assignment 4 as well as the provided ***reconstruct*** function in **camutils.py**.

Mesh Creation Function

```
[4]: def mesh(scanFolder, camL, camR, params, output):
         """
         Mesh generation based on given scans, two cameras,
         and various parameters that allow for tweaking. Writes
         mesh to .ply file.

         Parameters
         ----------
         scanFolder : str
           folder for where the images are stored

         camL,camR : Camera
           camera parameters

         params : list of ints
           wrapper for mesh generation parameters

         output : str
           output filename

         mesh_num : int
           scan number
         """
         print("Current folder:", scanfolder)
         print("Parameters for scan:", params)
         print("Output filename:", output)
```

Modified reconstruct Function for Handling Color

```
[2]: def reconstruct(scanFolder,mask_thres,decode_thres,camL,camR):
         """
         Reconstruction based on triangulating matched pairs of points
         between to view which have been encoded with a 20bit gray code
         for colored images.

         Parameters
         ----------
         scanFolder : str
           folder for where the images are stored

         mask_thres : int
           object mask threshold

         decode_thres : float
           decodability threshold

         camL,camR : Camera
           camera parameters

         Returns
         -------
         pts2L,pts2R : 2D numpy.array (dtype=float)

         pts3 : 2D numpy.array (dtype=float)

         ptsColor : 2D numpy.array (3xN array where N = number of points)

         """
```

```
[6]: scansFolder = "./couple"
     scanCount = len(glob.glob(os.path.join(scansFolder, "*")))

     # Parameters for mesh generation
     mask_thresholds = [0.005, 0.001, 0.001, 0.005, 0.0025, 0.001, 0.005, 0.005]
     decode_thresholds = [0.005] * scanCount
     trithresh = 4
     smooth_cnt = 2
     boxLim = np.array([[-15,24,0.5,22,-27.5,-15],
                        [-12,22,4.5,17.5,-27,-10],
                        [-15,23,4.5,22,-28,-10],
                        [-12,23,0,20,-25,-12],
                        [-10,23,-100,100,-22,-10],
                        [-12,24.5,3,20,-30,-10],
                        [13.5,21,-10,22,-39,2]])

     for i in range(scanCount):
         params = [mask_thresholds[i], decode_thresholds[i], boxLim[i], trithresh, smooth_cnt]
         scanfolder = f"{scansFolder}/grab_{i}_u"
         output_fname = f"{scansFolder[2:]}_{i}.ply"
         mesh(scanfolder, camL, camR, params, output_fname)
```

6.3 Mesh Smoothing

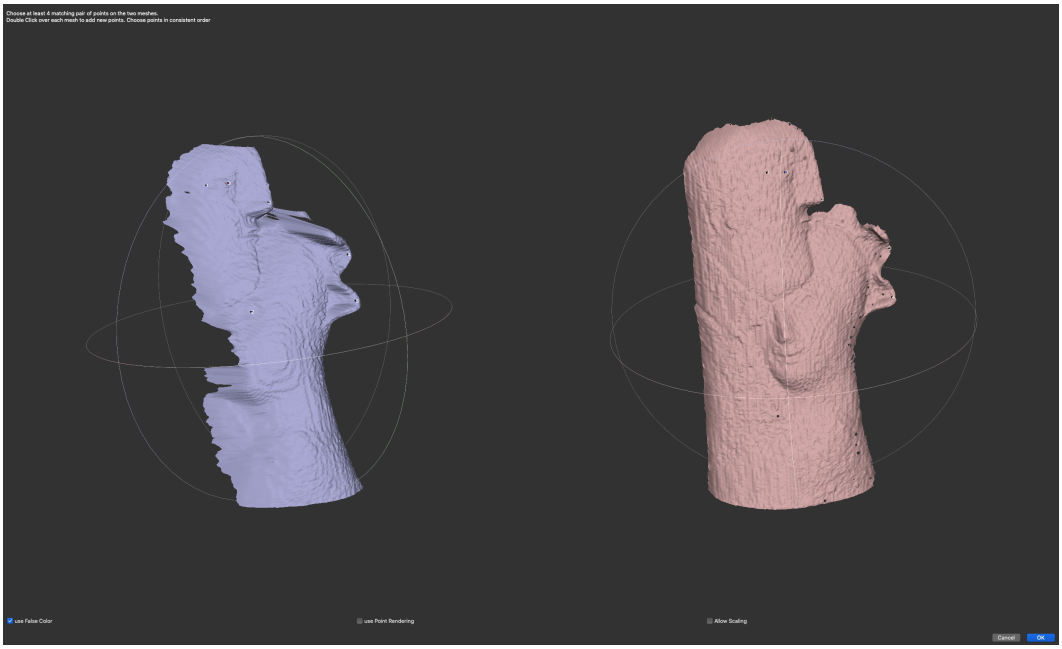
I wrote the mesh smoothing process which records each points' neighbors and changes the coordinates of each point to the average location between its neighbors.

Mesh Smoothing Function

[3]:
```python
def build_neighbor_dict(tri):
    # Auxiliary function to build dict of points and corresponding neighbors
    result = defaultdict(set)

    for triangle in tri:
        for pt_idx in triangle:
            current = {point for point in triangle if point != pt_idx}
            result[pt_idx] = result[pt_idx].union(current)

    return result

def Smooth_mesh(pts3,tri,iterCount):
    """
    Iterative mesh smoothing via averaging 1-ring neighbors

    Parameters
    ----------
    pts3 : 2D numpy.array (dtype=float)

    tri : array of triangles vertices

    iterCount : int
      count of smoothing iterations

    Returns
    -------
    pts3 : 2D numpy.array (dtype=float)

    """
    neighbor_dict = build_neighbor_dict(tri)
    pts3_t = pts3.T

    for _ in range(iterCount):
        for idx, point in enumerate(pts3_t):
            neighbors = list(neighbor_dict[idx])
            sub = np.array([pts3_t[x,:] for x in neighbors])
            pts3_t[idx,:] = np.average(sub, 0)

    return pts3_t.T
```

6.4 Mesh Alignment

I used MeshLab's mesh alignment tool to align every mesh. Then, I flattened the layers to create one merged mesh.



6.5 Poisson Surface Reconstruction

While I initially wanted to use PyPoisson to perform surface reconstruction, it failed to compile in my environment and lead to run-time issues when forced to compile with up-to-date dependencies. So, I decided to resort to MeshLab's Poisson Surface Reconstruction method.