CS 117 Final Project

Daniel Binoy
University of California, Irvine
Irvine, CA 92697
dbinoy@uci.edu

Overview

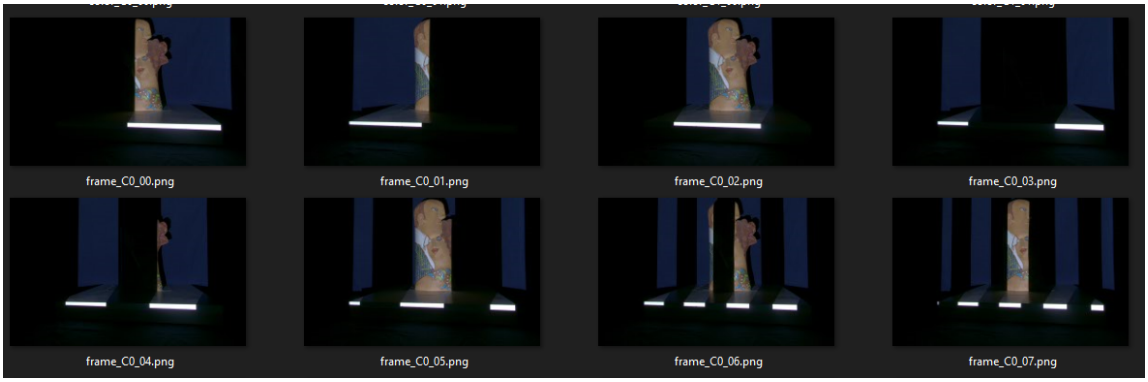
The goal of this project is to produce a high quality 3D reconstruction of an object from a collection of structured light scans. Specifically, a series of images of a small sculpture dubbed “couple” are processed in order to create the 3D reconstruction using tools such as MeshLab for and Blender. In addition, images of a checkerboard are used for calibration of the cameras.

1. Data

This project requires a large amount of data. There are two main sets of input data given to the algorithms: camera calibration images and object images. Every image has dimensions of 1920x1200 pixels. The camera calibration images are composed of 10 pairs of images of a checkerboard placed at different locations in a scene (Figure 1.1). This results in a total of 20 camera calibration images.

Figure 1.1 : Camera calibration images

The object images are divided into 6 “grabs”. Each grab contains images of the object taken in some specific rotation(ex. Looking straight at the couple, with the couple’s top facing the camera, looking at the back of the couple, etc.) Each camera takes 40 light scans(Figure 1.2) and 2 normal images(Figure 1.3). The normal images consist of 1 image with the object and 1 image without the object. These are used to create a color mask. The light scans consist of a pattern of alternating horizontal and vertical lines. Starting with 2 lines, and steadily increasing the line count, these images are used to later create a unique “binary” code for each spot on the image. This will be described further in the algorithms section.



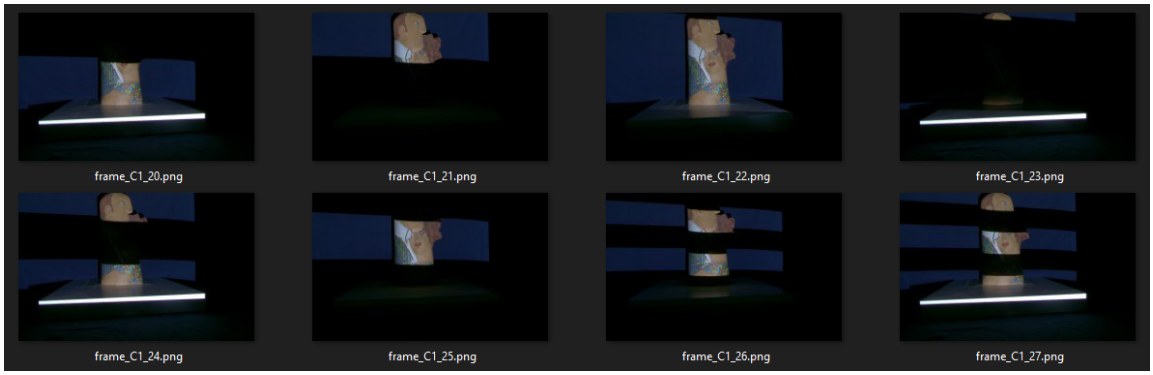


Figure 1.2 : Structured light scans

Figure 1.3 : Color images

2. Algorithms

This project required the implementation and tuning of a number of algorithms. The general algorithm consists of the following steps:

1. Calibrate the cameras using camera calibration images
2. Decode a 10bit gray code pattern from light scan images
3. Use gray codes to triangulate and reconstruct 3d points from 2d point matrices
4. Apply bounding box to 3d points, and perform Delaunay triangulation to generate triangle mesh
5. Perform mesh smoothing and triangle pruning on mesh to clean up mesh
6. Align scans in Meshlab using corresponding 3d points
7. Perform minor postprocessing in Blender to remove remaining unwanted vertices
8. Perform Poisson surface reconstruction in Meshlab
9. Perform final postprocessing in Blender to generate renders

2.1 Camera Calibration

The camera calibration is done in the **calibrate.py** file. For each file in the **calib_jpg_u** directory, the **cv2** library function **findChessboardCorners** is used to find the chessboard corners . This is later used in the **cv2** library function **calibrateCamera** to get the various camera intrinsic parameters. This is written to a pickle file for usage later.

2.2 Decoding

The decoding is done using the function **decode** in **camutils.py**. This function loops through pairs of light scan images with inverse patterns. It subtracts the paired patterns pixel values and compares against a threshold to create a mask. It also creates a matrix of "codes" with the corresponding binary code at each pixel location. The decode function also takes in the color images and returns a **colorMask** that is created by subtracting the image with the object from the image without the object. The pixels with a substantial difference in comparison with a threshold are given 1s in the mask and the others are 0. The 1s are foreground(pixels that contain the object), the 0s are background.

2.3 Reconstruction

The 3d point reconstruction is done by the **reconstruct** function in **camutils.py**. The function uses the **decode** function on the horizontal and vertical light scans from both the left and right cameras. This results in 4 code matrices and 4 masks, as well as 4 color masks, of which we use one from each camera. The masks and codes are combined together, resulting in **CL, CR, maskL,** and **maskR.** From there, **np.intersect1d** is used to get points that occur in both **CL** and **CR**, and that are within the corresponding masks. Subsequently**, pts2L** and **pts2R** are generated using a variety of numpy functions. The colors are also recorded for each 2d point by getting the corresponding pixel from the color pixels. Finally, the **triangulate** function is called with **pts2L, pts2R**, as well as the 2 camera objects from the calibration step. This returns **pts3**, which is our final 3d point cloud.

2.4 Bounding Box/Delaunay Triangulation

Next, **pts3**, **pts2L, pts2R,** and **colors** are passed through to the **generateMesh** function in **camutils.py.** The first part of **generateMesh** deals with bounding box pruning, as well as Delaunay triangulation. Using an array called **boxlimits**, each point is removed from **pts3, pts2L, pts2R**, and **pts2L** if it is outside the bounding box. Next, **pts2L** is passed to the **scipy.spatial.Delaunay** function which performs Delaunay Triangulation on the point set. This produces a set of triangles that contain all the points, avoiding extremely thin triangles.

2.5 Mesh Smoothing/Triangle Pruning

Next, within **generateMesh**, mesh smoothing and triangle pruning are applied to the set of triangles. The mesh smoothing works by setting each point to the average location of its neighbors. This is done a total of 4 times, resulting in 4 passes of the **smooth** function. Each time it is called, the overall mesh becomes more smooth. For triangle pruning, all the triangles who have any edge longer than a specified **trithresh** are removed. For this case, **trithresh** is set to 2.0. Finally, all points that aren't referenced in any triangles are removed. The function returns **pts3, simp**(the triangles)**,** and **colors**. Finally, the provided **writeply** function from **meshutils.py** is used to create a ply file compatible with Blender and Meshlab from that data.

2.6 Meshlab/Blender Processing

The various ply files(one for each grab) are then loaded into Meshlab. When loaded into Meshlab, the scans are quite rough and are rotated in various directions. In order to be combined into one useful mesh, they must be aligned.

Figure 2.6.1 : Ply files loaded into Meshlab

The various scans are aligned using Meshlab's built in point based aligning tool. This tool works by selecting corresponding points in 2 scans and aligning them by trying to match up those points as closely as possible through transformations of one of the scans. Below we see two scans with points being selected as well as the resultant aligned meshes.

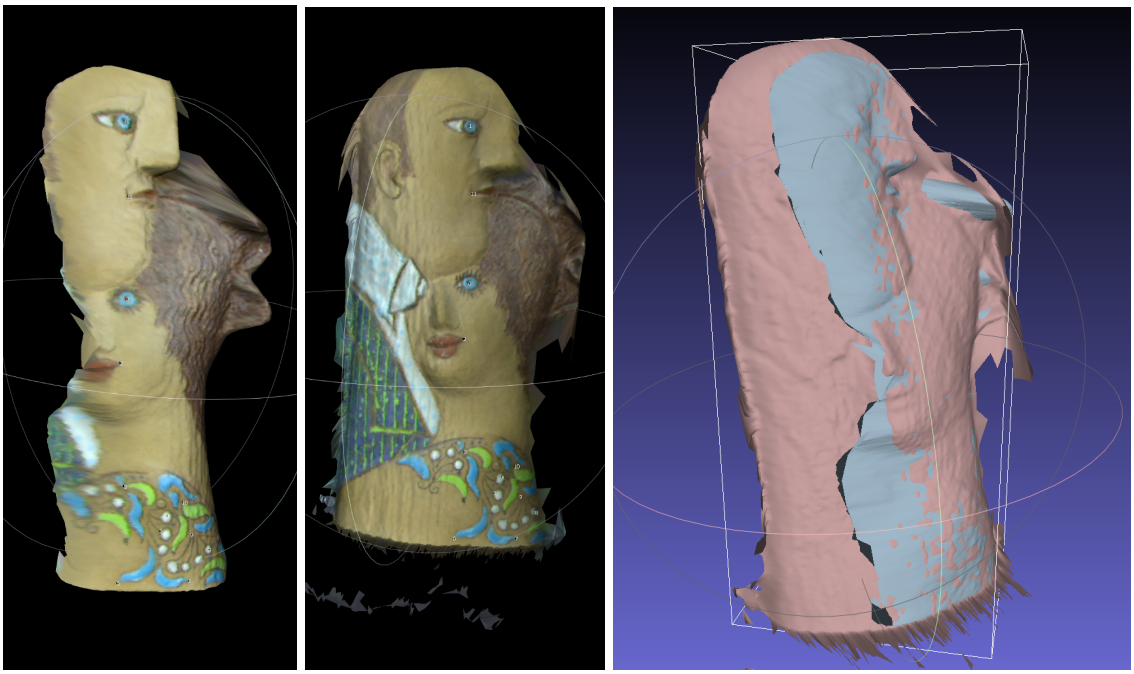


Figure 2.6.2 : Meshlab point alignment

Once all the scans have been aligned, they are flattened into one final mesh. To get a nicer output, this combined mesh is brought into Blender and some outlier vertices are removed. On this final mesh, we apply Meshlab's Poisson Reconstruction. Poisson Reconstruction simplifies the mesh and removes vertices that aren't attached to other vertices and constructs one solid 3d object.

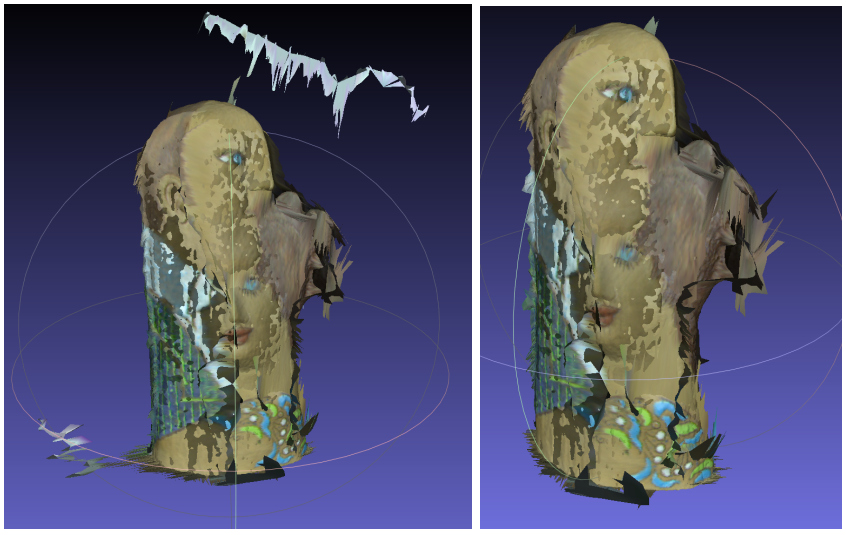


Figure 2.6.3: Before and after removing unwanted vertices

Below we see what is essentially the complete final output mesh. This final output mesh is then brought back into Blender to get the final renders with lighting and material effects applied, as seen in the Results section below.

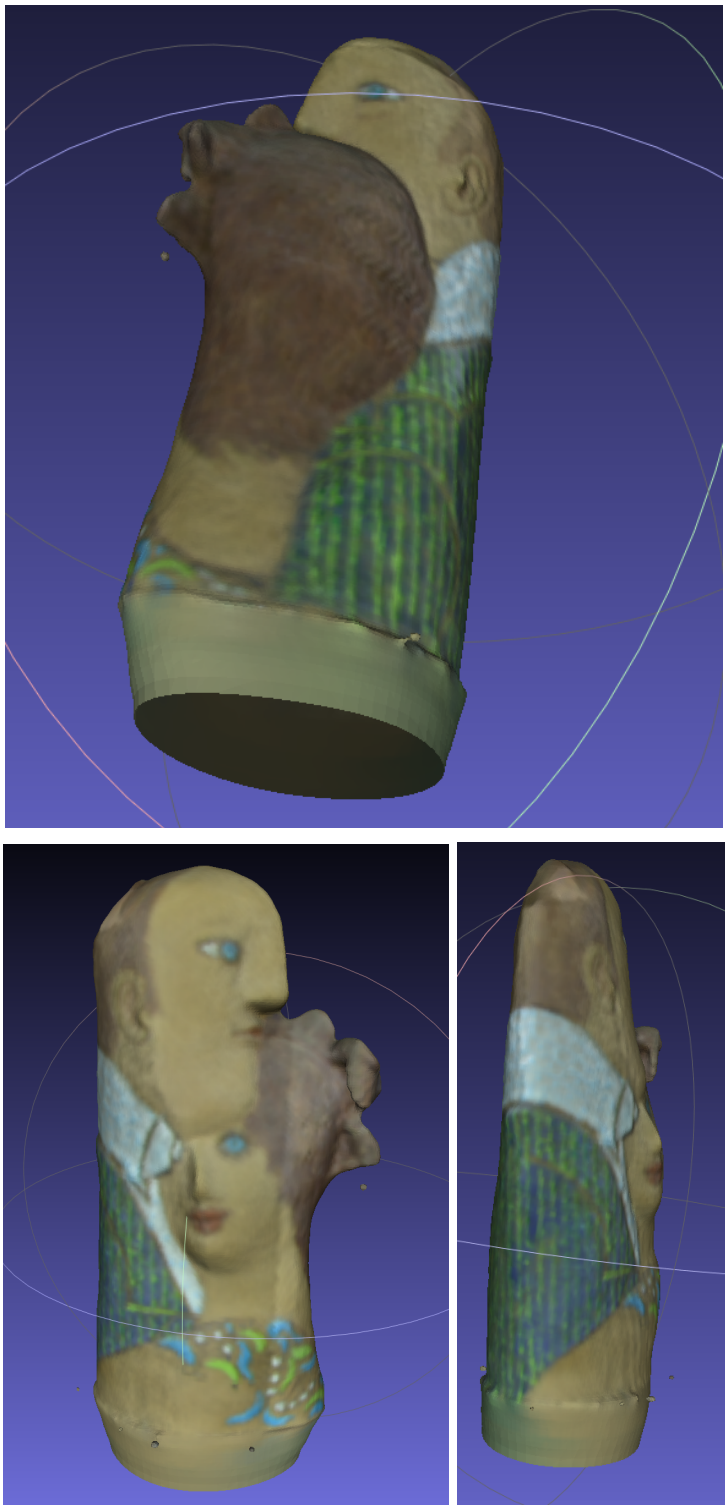


Figure 2.6.4: After Poisson Surface Reconstruction

3. Results

Below are the final results, as seen in Blender. These images were generated by converting the vertex colors into a texture(done in Meshlab) and then applying that texture to the model.

Figure 3.1 : Final results in Blender

Next, we can compare the final output model to the initial color images in order to better visualize the accuracy.

Figure 3.2 : Comparing final results to input images

4. Assessment

4.1 Difficulty

Having completed the project, I'd consider the task difficult without the plethora of resources and background that this class provided. Writing the entirety of the code from scratch would certainly have taken a lot more time. For example, algorithms like Delaunay triangulation and Poisson Surface Reconstruction would be difficult to implement from scratch. However, having Meshlab, Blender, scipy, numpy, and cv2 helped immensely and made the project much more approachable. The majority of the code used for the project was already written in **camutils.py**, saving a significant amount of time.

4.2 Limitations

There are many limitations to the methodology for this project. For one, the structured light scans were not in my control. A better system would perhaps be fully integrated. This would allow the cameras' intrinsic parameters to be fully controlled and known. It would also allow for more accurate calibration at various points. In addition, for this project, we used 10bit gray codes decode from 20 images from each camera for each grab. To increase the detail level, we could use more images and go deeper with the light scans. An 11 bit gray code would be twice as detailed and would require only 2 more images from each camera for each grab. The weakest link of my system was probably my mesh generation code. It ultimately failed to prune many vertices that should not have been included in the final meshes. Luckily, I was able to use Blender to remove some of the more egregious results, and the Poisson Surface Reconstruction done by Meshlab did an excellent job of removing unnecessary triangles.

4.3 Successes

Ultimately, however, I am quite pleased with the final results. As seen in the Results section, the final model is quite similar to the original grab images when compared from the same angles. While the actual vertex colors are not anywhere near as fine detailed as the original image, they are generally well matched. The general structure of the model comes through very well. Given the number of grabs and general lack of control over the input images, the final output is rather impressive. To be fair, this accuracy is mostly possible due to hours spent aligning the images in Meshlab and deleting unwanted vertices in Blender. I am certain that with less time spent on those steps, the output would be significantly less accurate. I am also certain that those steps could also be automated given more time and knowledge of numpy.

5. Appendix

The following is an appendix of all the code used for this project. The vast majority of the code was provided by the instructor to this course, Charless Fowlkes. This appendix will be separated by files and functions. Only functions written or modified by me will be described in detail. The full project code and resources, as well as output files can be found at https://github.com/binoy-d/cs117-final-project.
This repository is private, so access must be requested.

5.1 camutils.py

- makerotation(rx, ry, rz)
  - Provided by instructor
- Camera.__init__(self,f,c,R,t)
  - Provided by instructor
- Camera.__str__(self)
  - Provided by instructor
- Camera.project(self, pts3)
  - Provided by instructor
- Camera.update_extrinsics(self, params)
  - Provided by instructor
- triangulate(pts2L,camL,pts2R,camR)
  - Provided by instructor
- residuals(pts3,pts2,cam,params)
  - Provided by instructor
- decode(imprefix,start,threshold, imprefixC, thresholdC)
  - Modified by me
  - This function decodes the 10bit gray code pattern with the difference **threshold**.
  - My modification was adding **imprefixC** and **thresholdC**, which are used to create and return a **colorMask** by subtracting foreground and background color images
- reconstruct(imprefixL,imprefixR,threshold,camL,camR, imprefixLC, imprefixRC, thresholdC)
  - Modified by me
  - This function performs a reconstruction using the triangulate function, It also returns a 3xN numpy array of RGB color values.
  - My modification was adding **imprefixLC, imprefixRC,** and **thresholdC** and using the **colorMask** from the modified **decode** function. I used the left and right color images along with **pts2L** and **pts2R** to extract the corresponding colors for each point.
  - colorsL = imgL[pts2L[1], pts2L[0]].T
  - colorsR = imgR[pts2R[1], pts2R[0]].T

    - colors = (colorsR+colorsL)/2
- generateMesh(boxlimits = np.array([-40,50,-40,20,-40,20]), \
  trithresh = 2, \
  pts3 = np.array([]), \
  pts2L = np.array([]), \
  pts2R = np.array([]), \
  colors = np.array([]))
    - Created by me
    - This function does 4 things: bounding box pruning, delaunay triangulation, mesh smoothing, and triangle pruning
    - The bounding box pruning is simple, it just removes all points which have any coordinate outside of the bounding box(specified as an array)
    - Delaunay triangulation is done using **scipy.spatial.Delaunay** and **pts2L**
    - Mesh smoothing is done with 2 helper functions: **neighbors** and **smooth**
    - **Neighbors** provides the indices of the neighbors of each triangle
    - **Smooth** repeatedly sets each point to the average location of its neighbors using **np.mean**
    - Calling smooth(n) performs n smoothing iterations
    - Smooth(4) is called
    - Triangle pruning is done with a simple **distance** function that returns the distance of an edge of each triangle
    - Any edge with a length above the threshold(2.0 by default) is removed
    - All updates are done to **pts3, pts2L, pts2R**, **colors,** and **simp**(triangle.simplices)
    - The function returns **pts3, simp, colors**

5.2 main.py

- performCameraCalibration()
    - Modified version of calibrate.py
    - Performs camera calibration based on calibration folders and returns the calibrated cameras
- generatePoints(camL, camR, imprefix)
    - Created by me
    - Calls reconstruct given a particular grab directory **imprefix**
    - Returns **data** dictionary with **pts2L, pts2R, pts3, colors**
- generatePlyFiles(directory)
    - Performs camera calibration and then loops through each grab in the directory
    - For each grab directory, it calls **generatePoints** and then calls **generateMesh** using the output of **generatePoints**.
    - It uses **generatePlyFiles** with the output of **generateMesh** to generate each .ply

5.3 meshutils.py

- writeply(X,color,tri,filename)
    - Provided by instructor

5.4 calibrate.py

- calibrate(calibimgfiles:str, resultfile:str)
    - Modified by me
    - Originally, this performed camera calibration within the script. I converted this to a function, which I later turned into **performCameraCalibration** in **main.py**
    - This file is essentially not used