Rendering Models with Structured Light Scanning

By Gautam Banuru

June 14, 2019

Table of Contents

Project Overview

Structured Light Scanning is the process of projecting a set of patterns onto an object in order to retrieve the distance of each pixel relative to the camera using triangulation. This information can be used to create a point cloud of the object's pixels in 3D space and meshing these points together will create the object model. This project uses images projected with structured light in order to produce a 3D model of the object in the image. In order to do this, the following goals have to be met:

1. Calibrate cameras to gather intrinsic and extrinsic parameters of the camera
2. Apply box and color masks to separate the object from the background
3. Triangulate non-masked points to create a point cloud of the object
4. Correlate pixel color data with each point in point cloud
5. Generate mesh using cleanup tools (mesh smoothing and bounding box/triangle pruning)
6. Align meshes and perform poisson surface reconstruction with MeshLab
7. Generate final model renderings

Data

This project uses 8 sets of images. All images have the resolution 1200x1920 pixels. The first set includes 20 image pairs used to calibrate the cameras. The image pairs display a checkerboard pattern taken at varying angles where each image pair represents a left and right angle of the camera (see figure 1). The next 7 sets of images include 40 image pairs of the object taken from a left and right angle in which each image pair is projected with a different barcode pattern of structured light to be used for reconstructing the image points in 3D space (see figure 2). Each of these 7 sets show different angles of the object (ie. top view, side view, etc.) and include 2 additional image pairs of left and right angles with one image pair including the object and the other image pair discarding the object. This is used to mask the background from the object as well as provide color data of the object for the final model rendering.

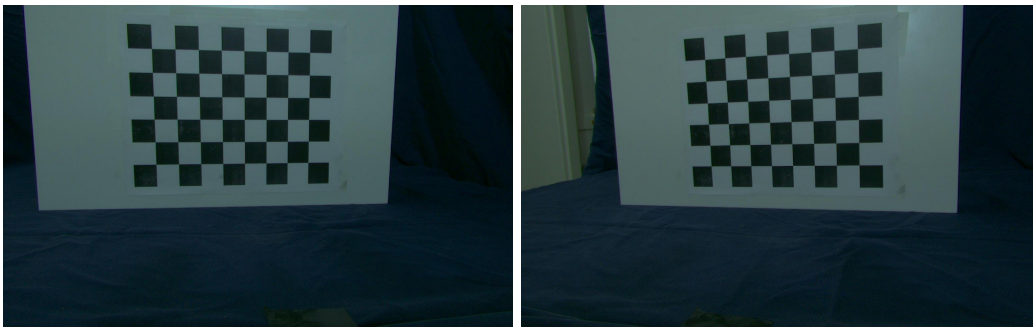

Figure 1: Pair of Checkerboard Calibration Images from Left and Right Angles

Figure 2: Two Pairs of Structured Light Object Scans from Left and Right Angles

Algorithms

There are several main algorithms at play to transform a set of images into a model of an object. These algorithms focus on calibrating the camera, triangulating points from a pair of left and right images, masking unwanted features, and creating a mesh from a point cloud. These algorithms will be described here in detail with images and pseudocode. All algorithms not from a library (cv2, numpy, etc.) have been coded by myself with guidance from Charles Folkes's previous assignments unless stated otherwise.

To start off we must distinguish between the global coordinate system and camera coordinate system. Global coordinates constitute the physical location of an object in 3D space. When we take a photo of an object from two different viewpoints, we are retrieving the coordinates of the object relative to the camera. Since photos are a 2D rendering of a 3D space we must figure out which pixels in both photos are the same in order to triangulate points together. The first step is to find our camera locations relative to the global coordinate system.

There will be two cameras called camL and camR from which photos from the left and right angles are taken respectively. The camera class takes in four parameters: the camera focal length, offset of principal point, camera rotation, and camera translation. The focal length allows us to transform physical coordinates to pixel coordinates. The camera coordinate system sits at the principal point. The focal length and principal point are intrinsic parameters. Camera rotation determines which direction the camera is pointing. Camera translation is the distance between the camera coordinate system and the origin of the global coordinate system. Camera rotation and translation are extrinsic parameters.

Intrinsic Camera parameters: cv2.calibrateCamera()

The intrinsic parameters of the camera are found using cv2.calibrateCamera(). Before this function is called, each image of the chessboard is fed into cv2.findChessboardCorners() which return a list of 2d chessboard corner coordinates in the image plane (see figure 3 for example). These corner coordinates are mapped to a grid of points representing 3d points in real world space. The 2d and 3d points are compiled into separate lists which cv2.calibrateCamera uses to determine the intrinsic parameters of both cameras.

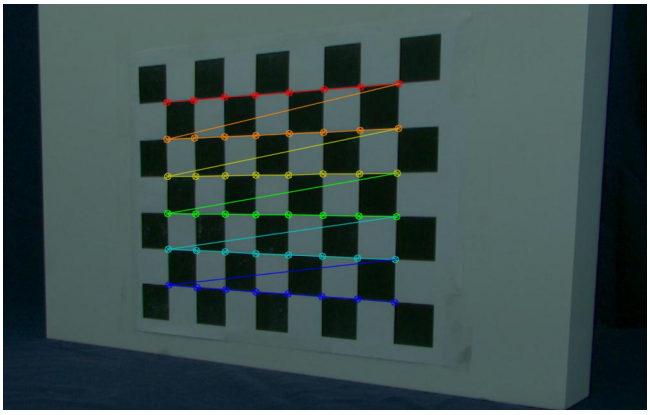

Figure 3: Corners found by cv2.findChessboardCorners() visualized

Extrinsic Camera parameters: calibratePose()

The extrinsic parameters of the camera are found using calibratePose(pts3,pts2,cam,params_init). This function takes a grid of 3d points representing the chessboard corners as pts3, a list of chessboard corners found using cv2.findChessboardCorners() as pts2, the camera updated with the intrinsic parameters, and an array of the initial estimate of extrinsic parameters. calibratePose() calls a multitude of functions with separate jobs. residuals(pts3,pts2,cam,t) computes the difference between the projection of 3D points by the camera with the given parameters and the observed 2D locations. To do this, after updating the extrinsic parameters with t, cam.project(pts3) is called which projects the given 3d points in world coordinates into the camera to produce a set of 2d coordinates as if the camera took a photo of the scene. This is done with the following code:

```
#project() function
# self.R=rotation, self.t=translation, self.f=focal length, self.c=offset of principal point
# get point location relative to camera
pcam = self.R.transpose() @ (pts3 - self.t)
# project
p = self.f * (pcam / pcam[2:])
# offset principal point
pts2 = p[0:2,:] + self.c
return pts2
```

By changing the value of t in residuals() with a lambda function we can compute a residual function. To get the optimal extrinsic parameters we will find the least squares solution between our residual function and our estimated extrinsic parameters using scipy.optimize.leastsq(func_residual, params_init). This provides updated extrinsic parameters of the camera so that pts3 projects as close as possible to pts2. Whenever the extrinsic parameters are updated the camera translation is set and the camera rotation is updated using make_rotation(). make_rotation() sets up 3 rotation matrices representing the degrees to rotate the x, y, and z axis respectively. The dot product of these matrices results in the final rotation matrix. We can confirm we have the right intrinsic and extrinsic parameters by projecting pts3 with cam.project() and mapping its 2d points onto the chessboard.

Create Masks: decode()

The purpose of decode(path,start,threshold,cthreshold=0.6,frame="",color="") is to create masks that can be applied on the input image to remove unnecessary detail such as the background since we only want to make a model of the object. decode() will use the images with projected light patterns to create a box mask. Images are converted to grayscale and read in pairs where the projected light pattern is swapped (see figure 5). Each pair of images will gradually increase the number of lines projected onto the object. Each pixel coordinate in the first image that is brighter than its corresponding pixel in the second image will append a 1 (or if not brighter a 0) to its coordinate in a 2d array of the same size. This will create a box code image in binary coded decimal that will be converted to decimal values (0-1023) for each pixel with the function graytobcd(). The box code image is what we will apply our masks to. We will create a box mask and a color mask. To create the box mask we will

take the absolute value of the difference between the first and second image in a pair and see if each pixel value is above a certain threshold. Pixels that do not satisfy difference threshold are placed on the box mask which starts out as a an empty 2d array of the same image size. This threshold determines how much of the image will be masked out. The box code and box mask will be modified with each pair of images to create the final 2d arrays. The color mask is made by thresholding the difference of a color image of the object with a color image of the background to determine what pixel values have not changed dramatically which will constitute the background and thus be masked out. Because each pixel has the three values blue, green, and red we take the square root of the difference of the images and sum each pixel's colors to get a single pixel value in order to use the threshold comparison to make the color mask. See figure 6 for the box code, box mask, and color mask.

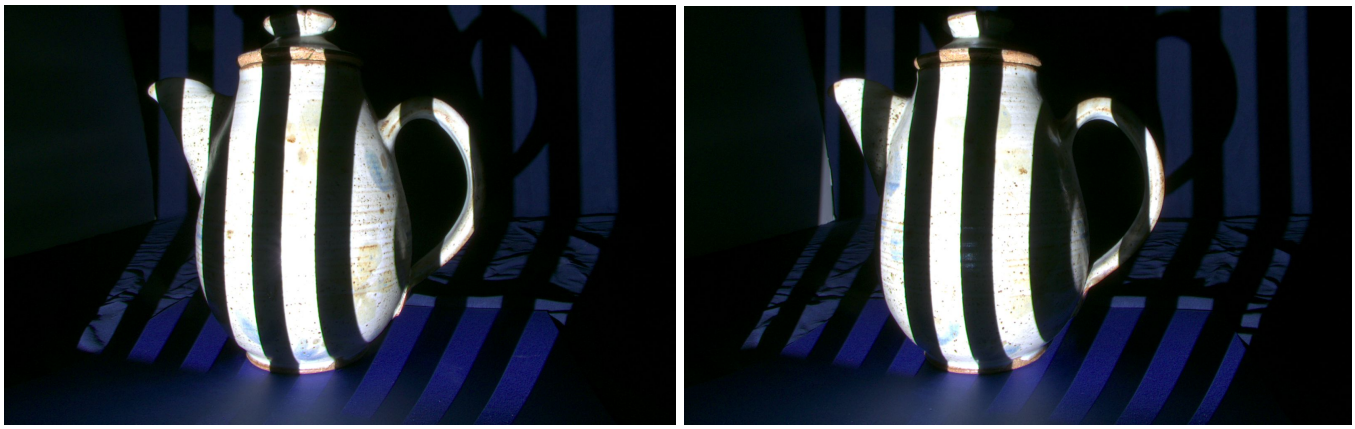

Figure 5: Images are read in pairs to determine what pixels will be masked out

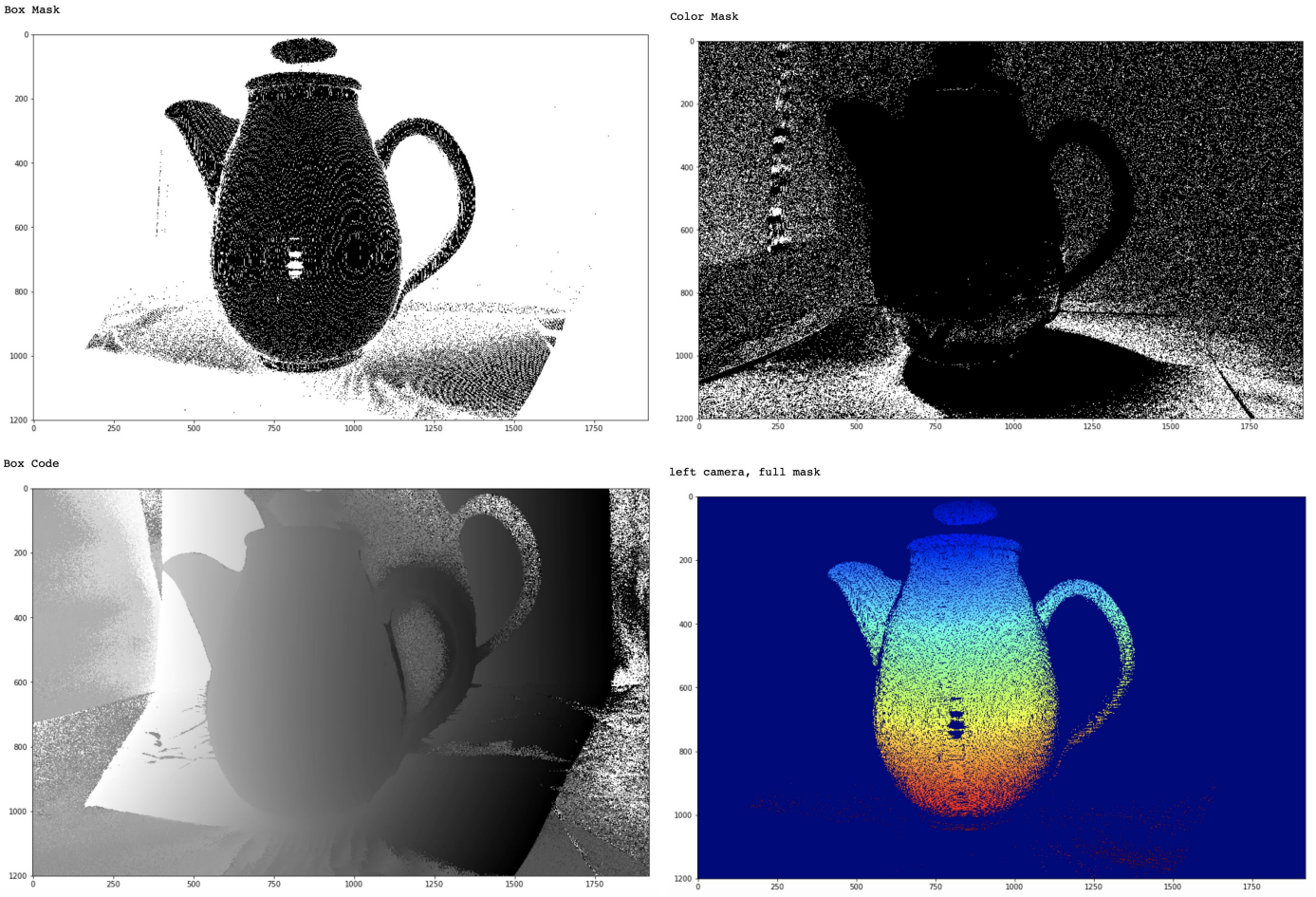

Figure 6: White pixels in the box/color masks will mask corresponding pixels in the box code. The box code with masks applied is the bottom right image (notice only points of the teapot are seen).

Find points in 3D space: triangulate()

triangulate(pts2L,camL,pts2R,camR) takes the set of points seen from the left (pts2L) and right (pts2R) masked images from their corresponding cameras (camL and camR) and returns the 3D coordinates of the points in the images relative to the global coordinate system. We are finding points from both images that match (see figure 7). First we need to get the z values of each pixel in pts2L/pts2R relative to each camera. To do this we will use the formula in figure 7 where qL/qR are 3x1 matrices in the format [focal_length*Point_x, focal_length*Point_y, focal_length]. t is the translation difference between camR.t and camL.t. We can use the linear least squares algorithm from np.linalg.lstsq() to find the values of z from the left and right cameras that correlate to a point match in 3D space as seen from each camera of known orientation in the global coordinate system. To match these two points to with respect to the global coordinate system we will multiply each point with its respective camera rotation matrix and add its respective camera translation matrix. The values of the two points should be the same so we will take the average of the two points to take into account slight variations to get the point in 3D space. This is done for each pair of points from each camera and the 3D points are compiled into a 3xN array.

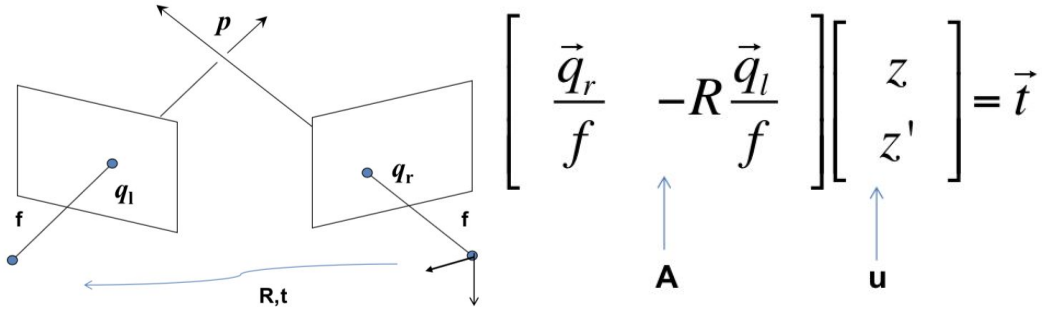

Figure 7: To find a matching point p we must find point p's z value relative to each camera as shown in the formula with respect to the diagram.

Generate Point Cloud: reconstruct()

reconstruct(imprefix,threshold,cthreshold,camL,camR) completed the pipeline of matching and triangulating points seen from two cameras to produce a point cloud as seen in figure 8. To do this the masks for each image set is retrieved from decode(). There is a set of photos with horizontal bars of structured light and vertical bars of structured light for the left and right cameras respectively. This produces 8 masks (4 color masks and 4 box masks) which are applied to their respective left and right images. With these masks applied we can use np.intersect1d to find the indices of pixels in the left and right code image that have matching pixel values. These indices are fed into a grid of points which will become pts2L and pts2R that is taken by triangulate() to produce the array of coordinates for the 3D points pts3. reconstruct() also takes the color values for the left and right images that become matched with the points in pts3 to be used for applying color to our model in the alignment phase where all the models from different scans are combined.

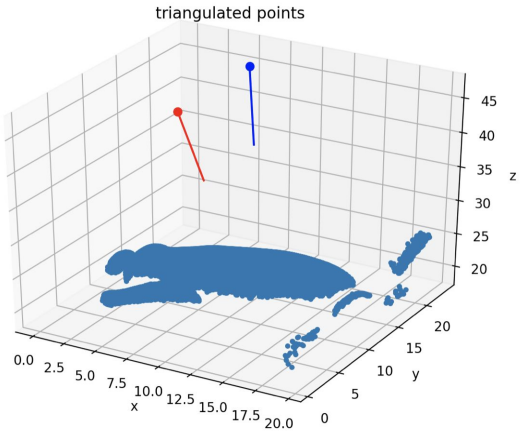

Figure 8: Triangulated point cloud of teapot created from reconstruct()

Generate Mesh: make_mesh()

make_mesh(path,threshold,cthreshold,boxlimits,store,camL,camR) creates a mesh using the point cloud pts3 from reconstruct(). Once the points are retrieved we apply two procedures called bounding box pruning and triangle pruning to remove extraneous points that are not part of the object. Bounding box pruning sets limits on what points are allowed within a certain volume of space. We specify boxlimits=[xmin, xmax, ymin, ymax, zmin, zmax]. Any point in pts3 not in the ranges specified in boxlimits is removed from pts3. The corresponding point is also removed from pts2L/pts2R and from the color array used for applying color to the final object mesh. Then we use the Delaunay() function from scipy.spatial to triangulate the 2D points to get the surface mesh saved as tri. This is a triangle mesh. Tri.simplices returns an array of points that constitute the corners of each triangle made. To perform triangle pruning we measure the edges of each triangle by calculating the distance between the corners of each triangle in 3D space. Any triangle with an edge larger than the specified threshold has its points removed from pts3, pts2L/pts2R, and the color array. Mesh smoothing involves averaging a set of local points to produce a smoother mesh. We will save this data (pts3,color,tri.simplices) as a .ply file with writeply() (written by Charles Folkes) to be used in meshlab to create align different meshes together. Because the color matrix must have values between 0 and 1, we will divide the color matrix by 255 before calling writeply(). An example of a created mesh can be seen in figure 9.

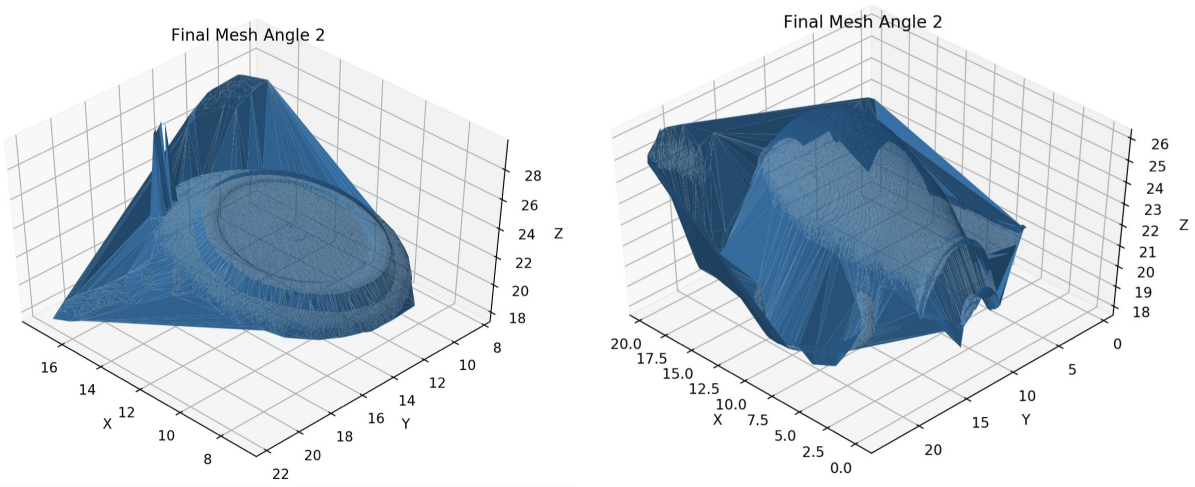

Figure 9: Teapot Bottom and Side Mesh created using make_mesh()

Clean Up Aligned Meshes: Poisson Surface Reconstruction

Once we have generated the meshes from the 7 sets of object photos and different angles we can align the meshes using meshlab (meshlab is an application used for mesh manipulation). Alignment may still leave some gaps in the mesh which can be filled up using Poisson Surface Reconstruction (PSR) via meshlab. PSR works by using the mesh point cloud. These points are divided into an oct-tree used for the least squares problem to solve for the indicator function from which extract iso-values are extracted to be placed in the marching cubes algorithm to finally extract the surface mesh. These details can be abstracted away by using the PSR tool in meshlab.

Results

The final results can be seen in the images below. There are several things to note about these results. These results do not implement color mapping on the object and mesh smoothing. Color mapping was attempted and mesh smoothing was not. As seen in figure 10 all six meshes have been aligned using meshlab. Each mesh is a different color for viewing purposes. There are some holes in the meshes partly due to the cleanup of unwanted triangles using meshlab's "delete faces" tool. These holes can be filled with Poisson Surface Reconstruction as shown in figure 11. An interesting circumstance of this is that the PSR leaves an additional paper like mesh around the teapot. While this mesh can be trimmed with the "delete faces" tool it is a bit tedious.

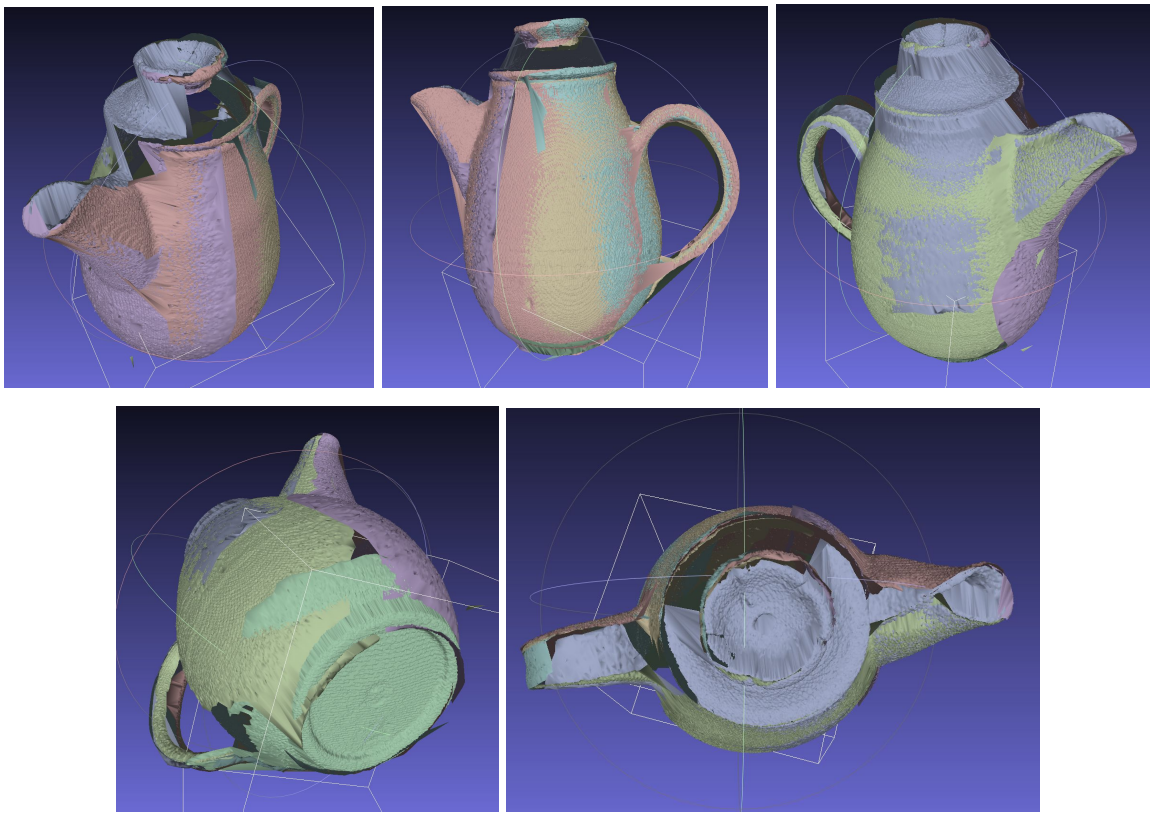

Figure 10: Final Teapot Mesh Reconstruction from Various Angles

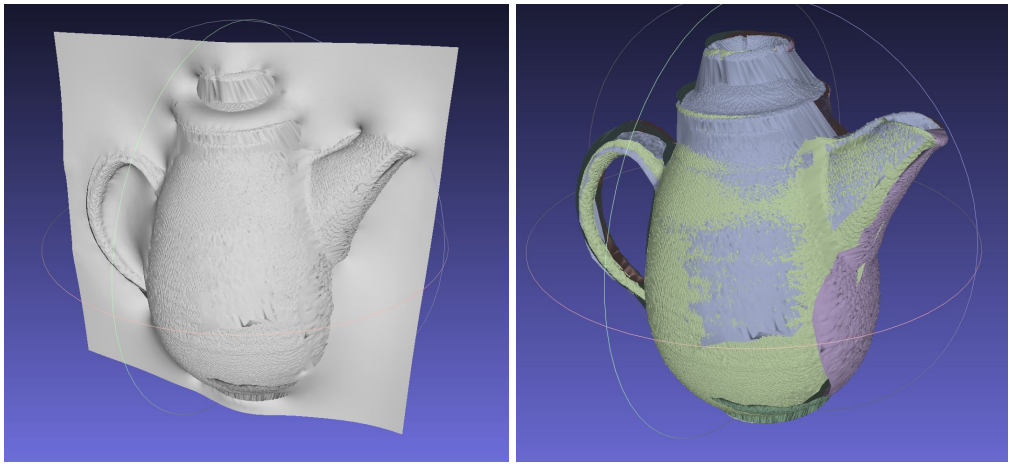

Figure 11: Teapot Mesh with and without Poisson Surface Reconstruction

There were several steps as part of the scanning pipeline in order to produce the final mesh. Once I retrieved the box and color masks from the decode() function I was able to apply those masks onto the left and right images of the object in order to mask the background as shown in figure 12. These points are what become triangulated to produce a point cloud of the object as part of the reconstruct() function shown in figure 8. An important part of creating the meshes for each set of images is to observe where to set the box limits for bounding box pruning as part of the make_mesh() function. An example of creating such meshes is in figure 9. These meshes had extraneous points that caused triangles to be created that far exceeded the threshold for triangle pruning in make_mesh() which was set at length 2 (possibly due to a code error). A workaround for this was to use the "delete faces" tool in meshlab.

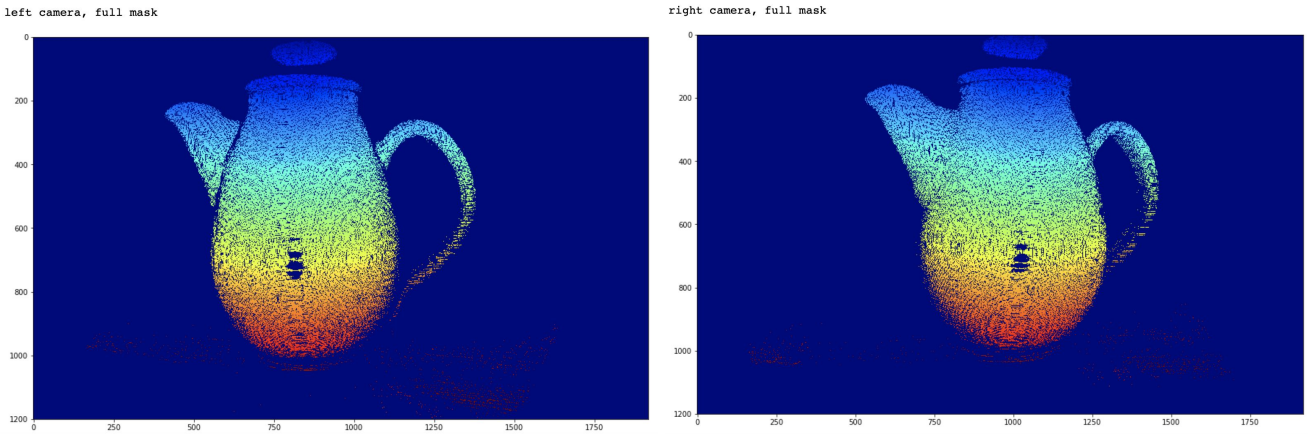

Figure 12: Masked Teapot from Left and Right Angles

Assessment and Evaluation

Much of this project consisted of writing the documentation. The results of the final mesh do not implement color mapping and mesh smoothing. For color mapping I implemented all the code that involves converting the color image point coordinates from Nx3 to 3xN so that the pixel values for each color can be accurately measured when using the color threshold to determine the color mask. The color mask is moderately successful because it does identify the background pixels with a little fuzziness that becomes a non-issue when the box mask is applied onto the image. I correlated the color matrix with pts3 and pts2L/pts2R when points were removed due to pruning so that the matrices would have the same number of points. Even after making sure the color matrix satisfies the constraints for writeply(), on meshlab the user-defined color is white when Poisson Surface Reconstruction is applied. This must be an error in the code I overlooked. Mesh smoothing was not attempted due to a lack of time. In regards to the accuracy of the meshes in the final model it looks like there was an error in triangle pruning as there are several large triangles in the final model that have not been removed. This is evident in the images of figure 9 but was slightly remediated meshlab's "delete faces" tool.

The task of stitching images together to create a 3D model is very complex even for a computer algorithm because there are several steps to follow from calibrating the camera to creating image masks to triangulating points and aligning meshes. The limitations of my approach is the need to project several barcode patterns onto the object from several viewpoints with two cameras. Lots of image data is needed. My successes included following the pipeline to create the aligned meshes shown in figure 10. While the final model could have been cleaner I am proud that I can recognize the model is a teapot. The sides of the teapot do not have a smooth texture which is where mesh smoothing really would have helped. I would implement mesh smoothing given more time. The lid of the teapot does not have a good mesh reconstruction. My assumption is that since the lid is bulbed shape, this casts a shadow on the base of the lid which gives decode() a hard time recognizing that the base of the lid is still part of the teapot to be excluded from the mask. The bulb shape of the lid also prevents the bottom of the bulb to be seen from the camera during a top down view. The lack of data results in an empty spot in the mesh generation that is poorly filled up with large triangles.

The data was presented well with 6 different orientations of the teapot however more photos of the teapot lid at an angle would have been beneficial. The weakest link in the pipeline would have to be the mesh creation with make_mesh() given that mesh smoothing is not implemented and triangle pruning seems to allow triangles to be created larger than the threshold. This could be because Delaunay() is called immediately after triangle pruning is completed and needs to be investigated further. Because much of the algorithms implemented are part of previous works the newest algorithm that gave me the most trouble was creating the color array for color mapping. This is because I tested the code to make sure it works on small batches of data before deploying it in the decode() and make_mesh() algorithms yet it still did not function as intended in the grand scheme of things. The Poisson Surface Reconstruction model is white when I apply the color matrix onto it so it the matrix must only have values of 0.

Appendix

makerotation(rx,ry,rz): Generates a rotation matrix and returns a rotation matrix of size 3x3.

Camera.init(self,f,c,R,t): Initializes the camera focal length, principal point offset, rotation, and translation.

Camera.project(self, pts3): Projects the given 3D points in world coordinates into the specified camera. Returns image coordinates of N points stored in an array of shape (2,N).

Camera.update_extrinsics(self,params): Updates the rotation and translation of the camera.

residuals(pts3,pts2,cam,params): Computes the difference between the projection of 3D points by the camera with the given parameters and the observed 2D locations. Returns vector of residual 2D projection errors of size 2*N

calibratePose(pts3,pts2,cam,params_init): Calibrate the provided camera by updating R,t so that pts3 projects as close as possible to pts2. Returns refined estimate of camera with updated R,t parameters.

triangulate(pts2L,camL,pts2R,camR): Triangulates the set of points seen at location pts2L / pts2R in the corresponding pair of cameras. Return the 3D coordinates relative to the global coordinate system.

dist3d(pt1, pt2): Finds distance between two points in 3d space.

bool_to_int(v): Converts boolean values to integers.

graytobcd(gray3d): Converts graycode to decimal. Returns integer.

decode(path,start,threshold,cthreshold=0.6,frame="",color=""): Given a sequence of 20 images of a scene showing projected 10 bit gray code, decodes the binary sequence into a decimal value in (0,1023) for each pixel. Marks those pixels whose code is likely to be incorrect based on the user provided threshold. Images are to be named "imageprefixN.png" where N is a 2 digit index (e.g., "img00.png,img01.png,img02.png..."). Also computes and returns the box code, box mask, color mask, and color image of shape 3xN.

reconstruct(imprefix,threshold,cthreshold,camL,camR): Performs matching and triangulation of points on the surface using structured illumination. This function decodes the binary graycode patterns, matches pixels with corresponding codes, and triangulates the result. This is where masks are applied to the box code image. The returned arrays include 2D and 3D coordinates of only those pixels which were triangulated where pts3[:,i] is the 3D coordinate produced by triangulating pts2L[:,i] and pts2R[:,i].

make_mesh(path,threshold,cthreshold,boxlimits,store,camL,camR): Creates a mesh of the points in the point cloud and uses bounding box/triangle pruning to remove extraneous points from the mesh. Returns pts3 and the mesh array tri.simplices. make_mesh() saves the mesh data as a .ply file to be used in meshlab. This function uses reconstruct(), Delaunay(), and writeply(). make_mesh() saves the color data of each triangulated point.

Delaunay(pts2): Takes the coordinates of 2D points and returns the triangle mesh tri.

writeply(X,color,tri,filename): Saves out a triangulated mesh to a ply file.

cv2.calibrateCamera(): Used to calibrate the camera.

cv2.findChessboardCorners(): Used to find chessboard corners for camera calibration.