CS 117

Project Report

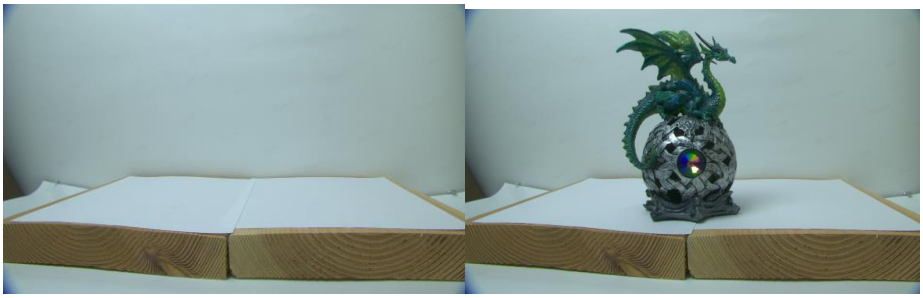
Michael Yuen

18779356

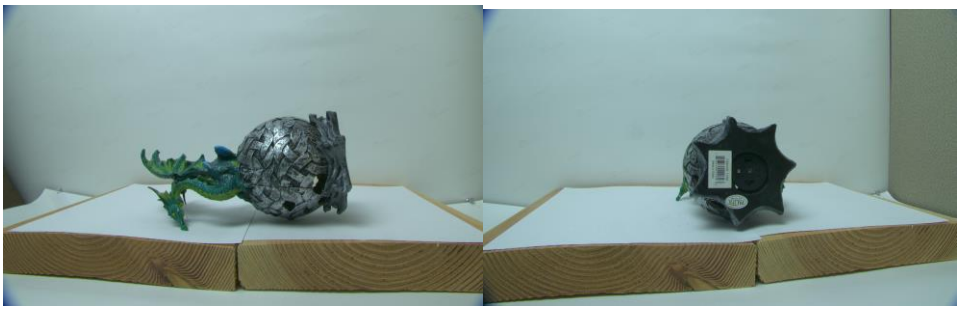
Project Overview:

The basis of this project is to create a 3d model out of scanned images of a dragon model. This will be done within 7 parts, finding the camera's intrinsic and extrinsic parameters, modifying the reconstruct function to keep color values and consider the object mask, streamlining the mesh creation pipeline, mesh smoothing, aligning corresponding points using MeshLab, Poisson surface reconstructing, and generating the final mesh. Of these 7 goals, I completed all 7, although mesh smoothing was done using a library instead of actually building it myself, and corresponding points and Poisson surface reconstruction were done using MeshLab.

Data:

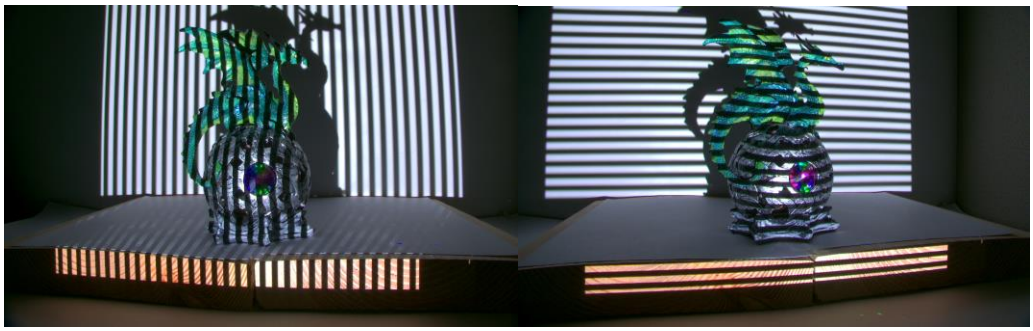


The data and scans are provided by the CS 117 staff and their setup in DBH. The item of choice is provided by Jordan Rayfield, who scanned a dragon model. Each set of scans hold 4 images that are either contain the object or contain the background, which are needed to extract color information and keep an object mask of the image. These images show off just the model itself with no extra distractions. The empty image is needed to separate the background from the model.



The dragon is then rotated and placed in 7 different angles, as shown above. These different rotations are needed to gather data from different parts of the model. This is very

similar to when a movie has a “time frozen” scene, in which there are many cameras pointing at the “time frozen” actor in order to shoot many scenes of the actor in that form. Although the process of generating the scans is very tedious and has high potential of making mistakes, it is necessary to have so many scans to generate a more accurate mesh.



The setup uses a left and right camera, which use a projector to illuminate different parts of the image. This setup creates a total of 80 images per rotation, with 40 for both the left and right cameras for horizontal illumination, and another 40 for both left and right cameras for vertical illumination. Given the number of angles and rotations, this can be a painful process, leading to about 672 images with 1200 x 1920 resolution and 2 GB of data. If even a single scan in a group of 80 images fail, that particular group of 80 images might as well not exist, making mistakes very harsh. Another scan would be needed to fix that issue.

Last but not least, the remaining data given is a set of chessboard images, which at least two are required to find decent camera parameters of the cameras used to take the images of the dragon. These are the given camera calibration chessboards used to find the camera’s intrinsic and extrinsic parameters.

Algorithms:

Calibration:

The first algorithm that is utilized is to find the camera specific intrinsic parameters, the focal length (fx, fy) and offset of a principle point (cx, cy). These intrinsic parameters are needed to generate a camera matrix similar to [[fx, 0, cx], [0, fy, cy], [0, 0, 1]], which can reduce distortion of a camera’s lens and is specific to the cameras that are used to take scans of the object. Note that there is a left and right camera that both have focal lengths: in this project the focal lengths are averaged together to find a close estimate of the focal length between both

cameras. The way to get these parameters is to gather some chessboard images and find the corners of the chessboards, which are in between multiple black and white squares of each chessboard. This will gather the 3D object points and the 2D image points in order to calibrate the camera. The calibration of the camera can then give back the focal length and offset of principle point, which are stored in a pickle file for further use.

The next step is to find the extrinsic parameters, which include how the camera was rotated and translated. The rotation matrix is a 3x3 matrix that corresponds to rotating rx degrees around the x-axis, ry degrees around the y-axis, and rz degrees around the z-axis. The translation vector is a 3x1 vector that shows how far the camera has been translated from the origin with x, y, and z coordinates. These require some sort of initial guess in order to estimate the best parameters that the camera could have taken during the image shooting. This initial guess is then thrown into an optimization function that estimates the camera's pose and minimizes the reprojection error as much as possible. The reprojection error is calculated by projecting the 3D object points found in the intrinsic parameters and then comparing that with the 2D image points to see if those are close given a set of parameters. The least squares optimization function will then optimize the parameters closer and closer to have less reprojection error, updating the camera's extrinsic parameters as the function goes. The most optimized rotation and translation is considered the camera's rotation matrix and translation vector.

Reconstruction:

Now that both the extrinsic and intrinsic parameters of both cameras are found, reconstructing the object in the image is the next step. The first step of reconstruction is to decode the given images to a mask that ignores pixels that can potentially fail in the image and to a decimal image that represents a stack of images' gray code. Each image has a projector lighting up certain surfaces of the image in order to luminate different parts of the image and create a gray code to decode into a decimal image. This stack of images is unpacked to find the gray code by checking if one image is greater than another. This gray code is then converted to binary through a loop of XORing together the images, which then is converted to decimal through traditional binary to decimal conversion. The mask is created by checking where on each image is decodable or not and constantly replaces itself until all images have been iterated over.

With decode, reconstructing the point clouds is the next goal, both the 2D and 3D points. Decode is called four times, once for each of the twenty images inside a single group of 80 images of a certain position. Decode is called once for the right camera with horizontal lighting, the left camera with horizontal lighting, the right camera with vertical lighting, and the left camera with vertical lighting. This is done within reconstruct, which takes these decoded decimal images and masks and combines them together. The left decimal images are combined, multiplying one with 1024 to convert the image into an integer image, and the right images are also combined. The left masks are combined together, along with an object mask. This object mask is calculated from taking the difference of the background and colored object image in order to find the object itself and limit what pixels can be in the mask. Again, the same operation is performed on the right masks.

Now that the masks and images are found, the next step is to find corresponding pixel coordinates in 2D in order to triangulate the respective 2D points of both cameras to find the 3D object points. The color points corresponding to each 3D point is also done here; given the object masks in color, the color points will be narrowed to just those respective 2D points and averaged among each other to find an array the same shape of the 3D object points forming the colors of each point. This reconstruction creates the 3D points of the point cloud, which basically forms the shape of the mesh.

Mesh Clean Up:

This point cloud is not quite perfect, as there are bound to be outliers in the point cloud skewing the mesh generation. Therefore, pruning on the points is needed to clean up the mesh outliers. This point cloud should be set by a bounding box that removes points outside a certain maximum and minimum x, max and min y, and max and min z. If a point were to lie outside of these set ranges, that point is a distraction to the point cloud and shall be removed. This bounding box pruning is very efficient at cornering the mesh to itself.

The other strategy utilized in pruning points is triangle pruning, which removes not only triangles that have too long of an edge within the triangle, but also points that do not attach to any particular triangle generated by the Delaunay function. The Delaunay function gets the surface mesh from a given set of 2D points, storing the mesh into a Delaunay object that contains vertices of each 3D point connected to a triangle, the triangles themselves, and neighbors to the triangles' vertices. This Delaunay object's set of vertices, called simplices, are pruned if the overall length between vertices is too long. If any vertex is removed from all triangles, the vertex is also removed from the original 3D point cloud. The trickiest part of removing the vertices is to remove the unconnected points that are removed from the point cloud afterwards; this can be done by creating a map to the indices that contain points used in the simplices of the Delaunay object, with placeholders in the points not used in the simplices, since these points are not used anymore. This will allow removal of the points that are not connected anymore and allows reupdating the simplices and 3D point cloud, staying clear of errors in indexing in the simplices.

Triangulation tries to fit every point into a triangle, connecting each vertex with an edge. The optimal goal of triangulation and calling the Delaunay function is that triangles should not be skinny and tall, as this messes with the overall mesh. Smaller and close to equilateral triangles are much better for triangulation, as these smaller triangles can form a flatter surface with more surface area. Triangulation wants to satisfy the circumcircle criterion, which emphasizes that all triangles fit inside a single circle and no triangles' points are inside the interior of the circle.

The differences in mesh generation all stem from what parameters are placed as thresholds in three different areas and what are the bounding box limits. The threshold for the color image determines what amount of the image should be taken from the mask, the threshold for decoding determines the comparison value of the combined mask, the bounding box limits force a threshold on the maximum and minimum x, y, and z values, and the threshold for the triangles determines what the max length of an edge can be in a triangle in the mesh.

Mesh Smoothing:

Now that the unneeded points are pruned, the next step is to smooth out the mesh, which reduces the mesh further. The general concept is to take each vertex and find the neighbors of the vertex. Within the Delaunay object, each simplices can get respective neighbors and move closer to the neighbors. Taking the average of each neighbor and moving each vertex closer to that average will utilize what is known as the Laplace operator, which is basically a gradient that moves each point closer to some average point. This will generate smoother meshes, rounding and cleaning up holes in the mesh. This can be done at n iterations, which is another parameter to adjust. The more iterations, the more each point would smooth out, possibly removing detail and messing with the original point cloud too much.

Note: Due to the lack of time, I did not implement this algorithm and used trimesh's implementation instead.

https://github.com/mikedh/trimesh/blob/master/trimesh/smoothing.py

Mesh Alignment:

The final step is to align each mesh with corresponding points and to fill in any large holes in the mesh. This will then generate one final 3D reconstructed model based on the original images that were first gathered in the data section. Attaching different meshes and trying to click on each meshes' corresponding points will slowly merge together different meshes together and glue each mesh one by one. This runs an algorithm akin to this: calculate correspondences between the 3D points, calculate and update alignment, and calculate error. If error exceeds some set threshold, then calculate more correspondences to calculate a better alignment. This forces the meshes to converge towards each other and to get close enough to glue together into one piece.

In this case, MeshLab was used, which does this algorithm using ply files that hold the mesh and point cloud from mesh smoothing. This makes the process of selecting points slightly easier, as correspondences are easier to put together with this application.

Poisson Surface Reconstruction:

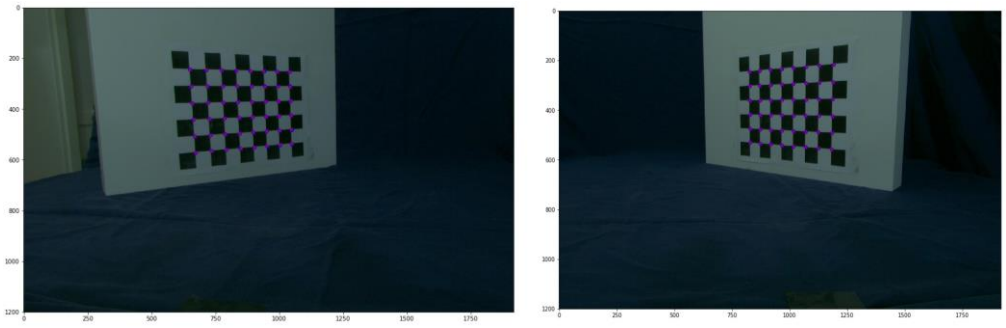
Continuing from the combined mesh, there are still holes in the mesh that cannot be fully aligned. The optimal way of dealing with these holes is to reconstruct a triangular mesh by solving a Poisson system. This algorithm takes in a set of 3D points representing surface normal estimates, which then are fed into a gradient to direct the vectors towards a combined surface. This defines the surface in between holes of the mesh, filling holes in the mesh with the reconstructed implicit function. The implicit function is found by integrating a vector field into a gradient of the function, which sometimes fails to find an overarching gradient to fit the implicit function. The final step is to minimize this gradient and vector field in order to find the best fit surface reconstruction.

Again, MeshLab simplifies this process and seals holes based on a number of parameters, such as reconstruction depth. These parameters seemed to not affect the model very much besides reconstruction depth, and running many of the parameters in higher or lower numbers crashed the program and computer.

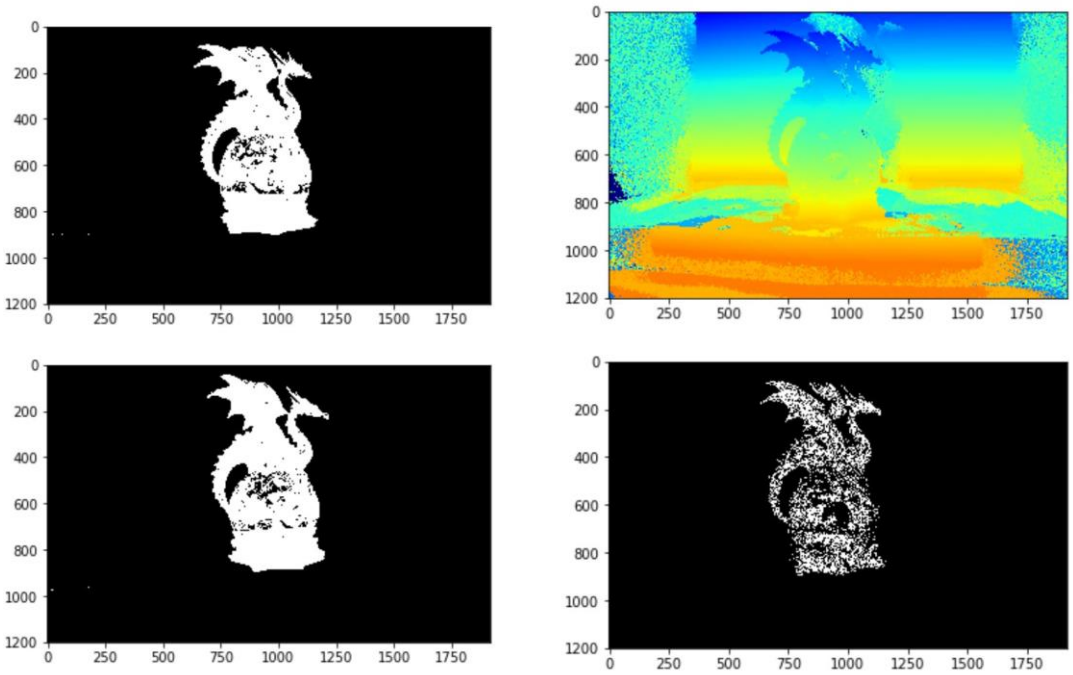
The mesh is complete.

Results:

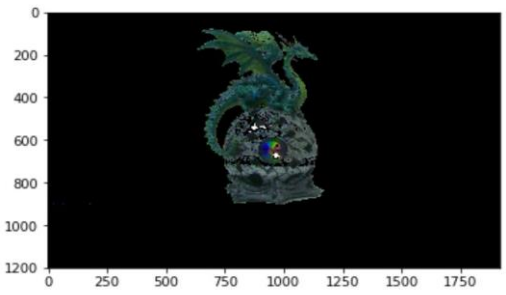
The camera calibration and parameters are found using the given calib_jpg_u files with chessboard images to find good intrinsic and extrinsic parameters. This step is done very similarly to assignment 3, in which the calibrate.py script is run and a calibrate.pickle file is generated to store the intrinsic parameters. The initial extrinsic parameters are taken from assignment 3 as well, which are the default parameters. This results in chessboards that match each corresponding point and are shown in these images below:



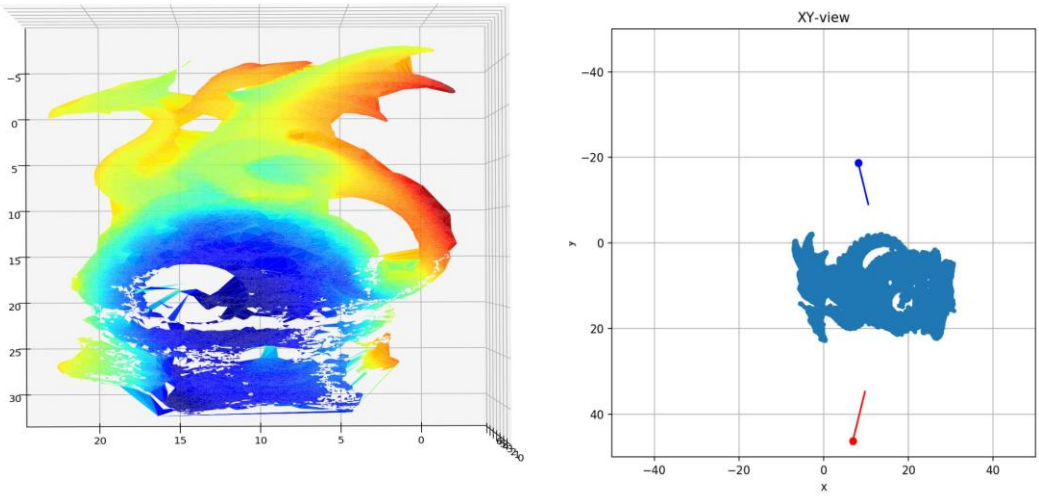
The reconstruct function uses the object mask to screen out points in the mask. This function keeps color values as proposed by the project guidelines. The object masks, left camera decoded image, and mask of the side view of the dragon are shown here.



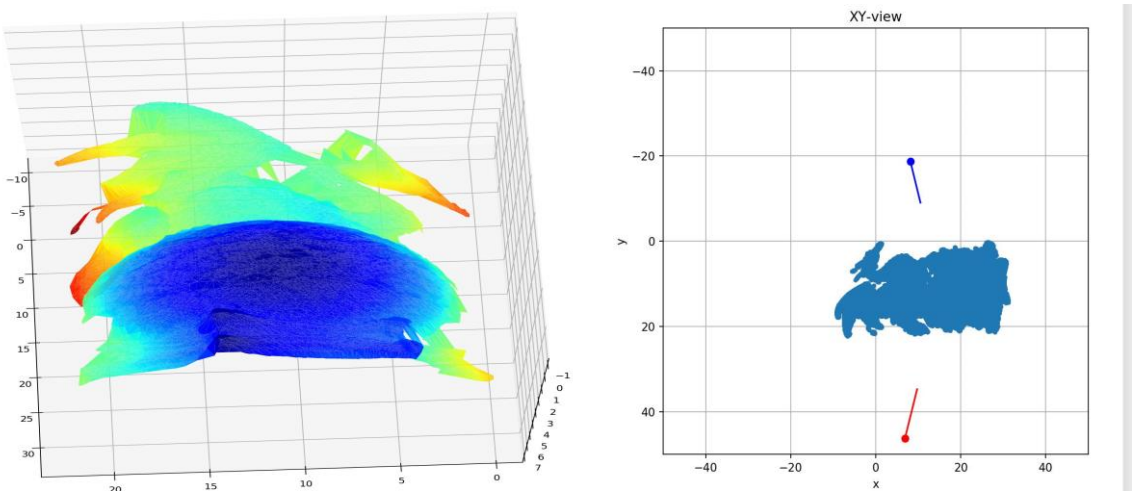
The color mask is shown here. This is done by masking the original color image with the above object masks.



The mesh creation, including bounding box pruning and triangle pruning, reconstructing using the edited reconstruct function, creating the object mask, and parameter editing, is all placed into one function. This allows for easy generation of multiple meshes from each set of images in each folder. Some point clouds and meshes that have been generated (out of the 7 that have been done) include the following.



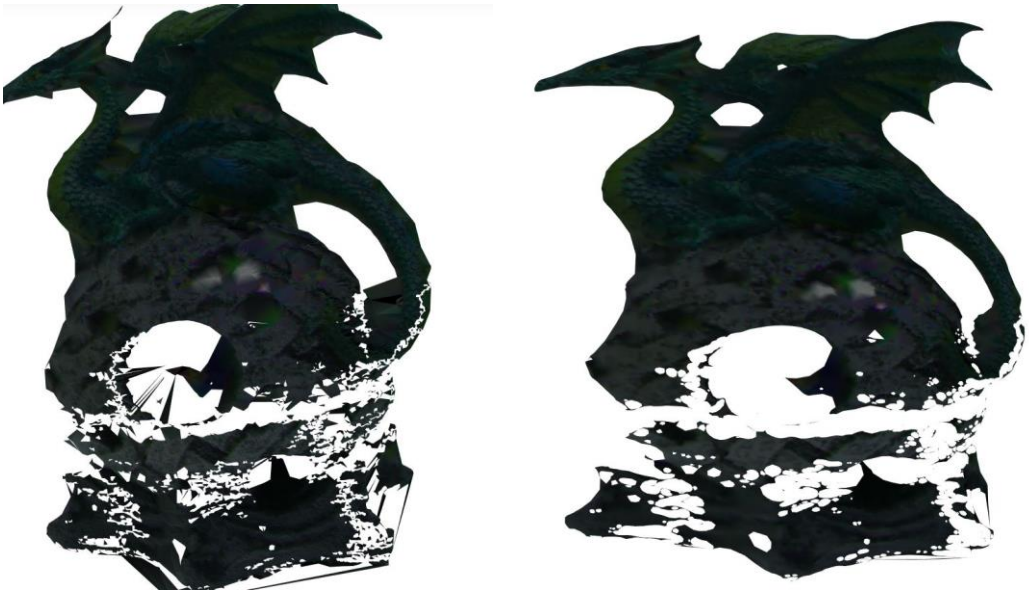
This mesh is the one generated from the object masks above. The center of the image is a bit obscured due to the circular shiny gem on the initial dragon model. The bottom half of the mesh is an issue in the initial mesh, as there are many points and triangles that are removed out of the bottom half of the mesh. However, Poisson surface reconstruction fixes this issue easily as it patches holes.



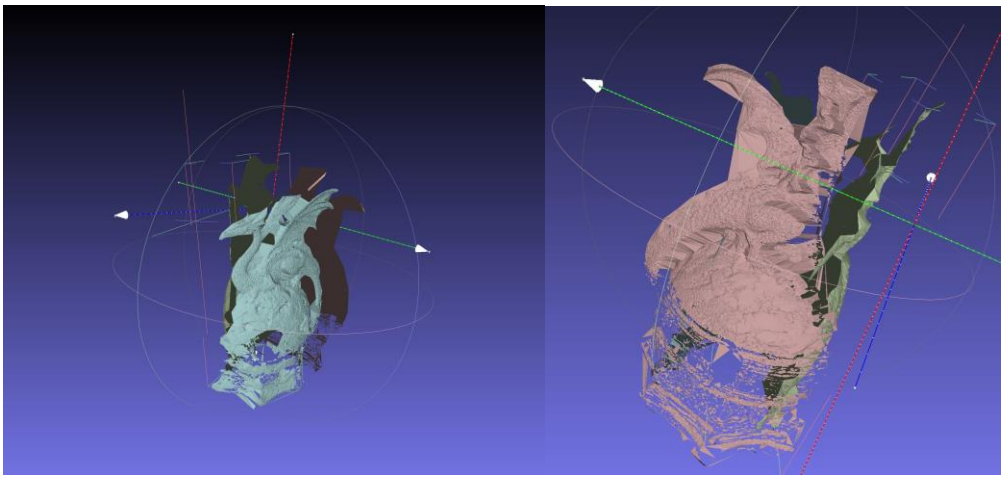
This is another mesh with the circular dome at the bottom of the model in a cleaner fashion (as it is focused on the other side of the dragon), but it is not as clean at the top, since it creates extra triangles that are unneeded.

The parameters of reconstruct, bounding box pruning, triangle pruning, and color threshold determine how accurate the mesh looks, which trisurf can generate those models. For each mesh, I have a separate triangle threshold that usually closes to 2, a decoding threshold of 0.02, a color threshold of 0.25, and box limits that are within a 50 by 50 square in the graph.

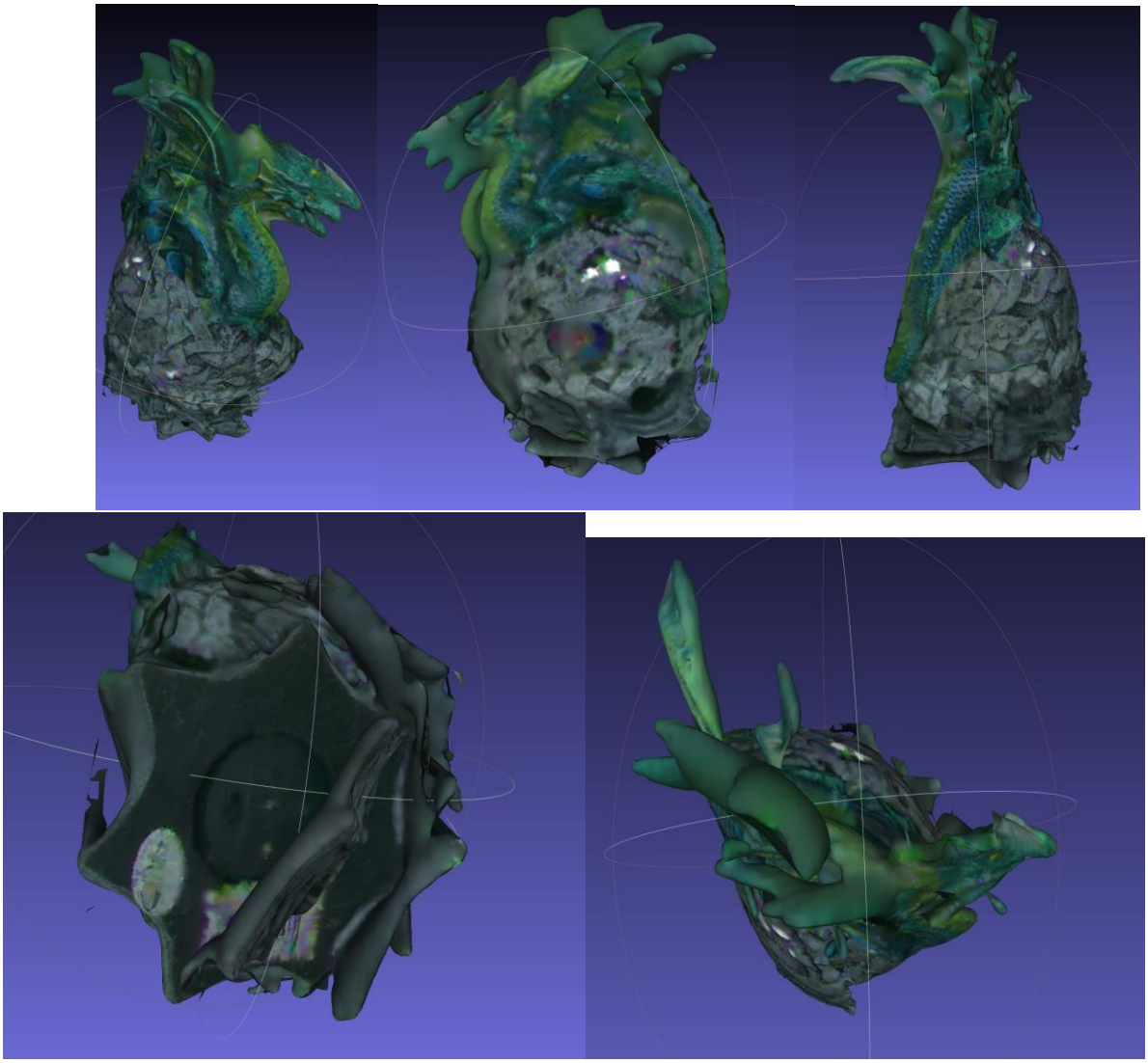
With trimesh, the models become significantly cleaner, and colors can be displayed with trimesh. In these models, the color is darker than in MeshLab.



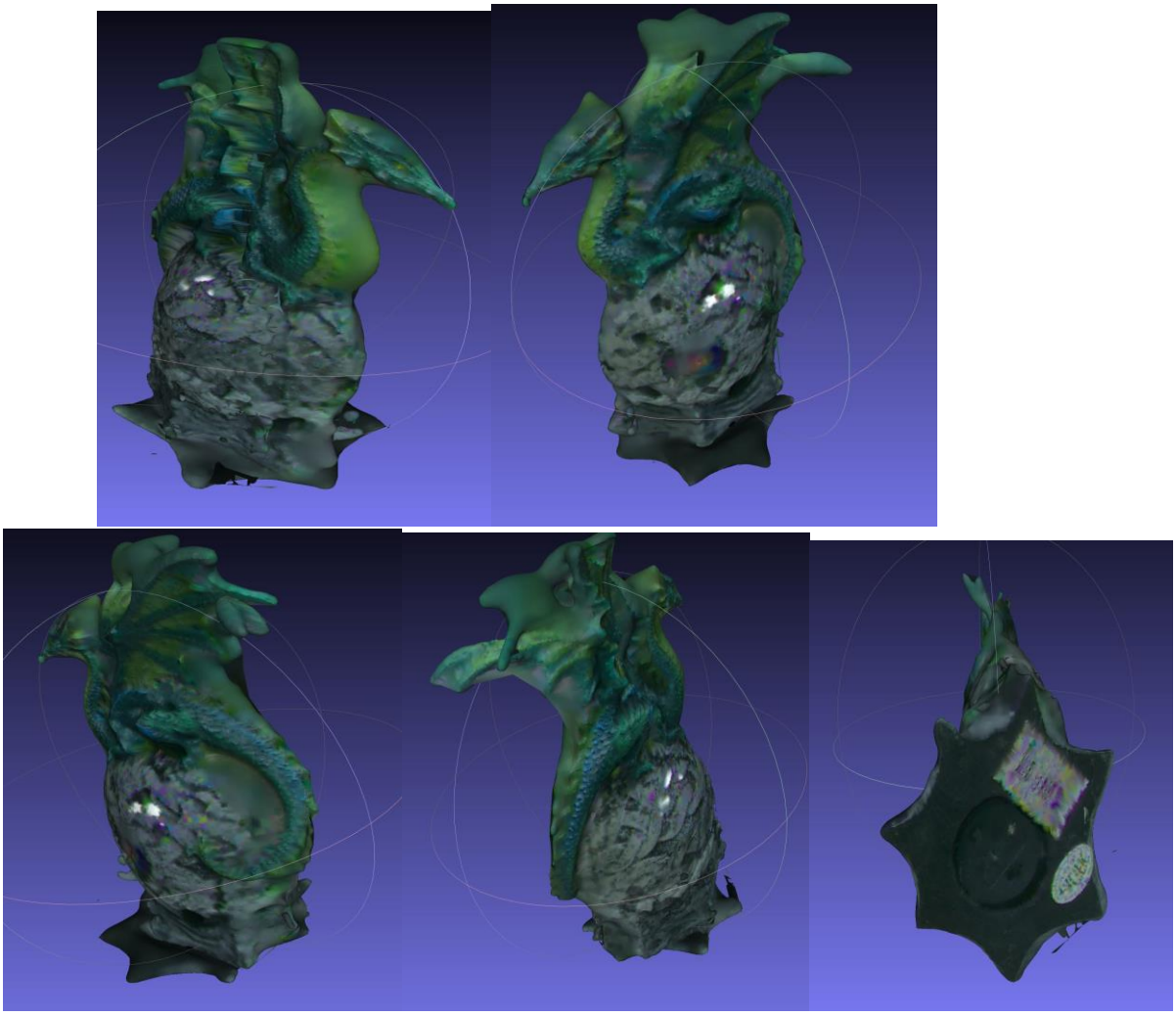
The mesh smoothing algorithm I used is from trimesh, which does Laplacian smoothing on the mesh generated from the streamlined mesh generation algorithm. The left image above does not have smoothing, but the right image does have smoothing, which makes many of the holes and fragments in the bottom of the image cleaner and smoother.



The alignment and Poisson surface reconstruction are both done using MeshLab to generate a combined mesh. This deals with the multitude of holes in some of the meshes. After this is done, the scans and data will be saved as ply files. MeshLab overall was not an intuitive tool to use and the alignment proved to be a problem, as shown above. In the end, I have two different combined meshes, one that looks more accurate, but only uses 4 meshes instead of all 7, and one that looks far less accurate, but uses all 7.

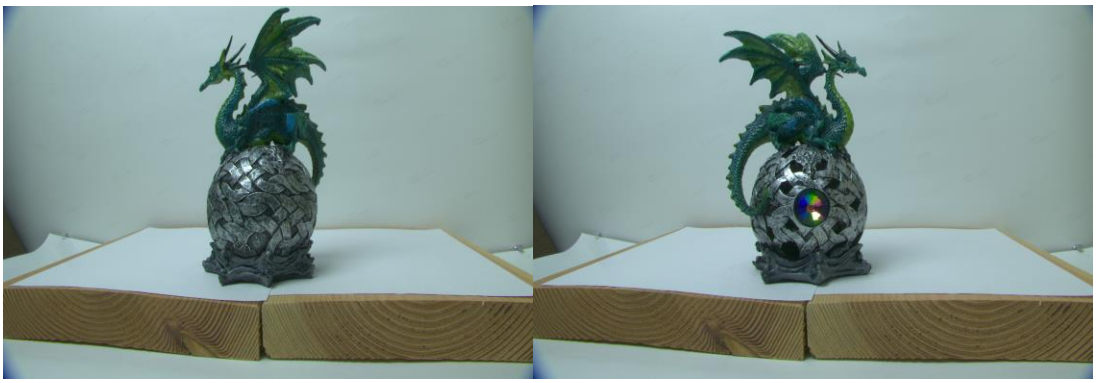


Above is my combined mesh for all 7 meshes. This combined mesh is significantly worse than my combined mesh for 4 meshes, as not only were corresponding points much harder to keep, the little details of each mesh were extremely hard to line up, especially the heads, the tails, and the base of the mesh. I also did not have a top down view of the dragon, which made the dragon's top blocky. I did not have enough views from the left of the dragon, as most views were taken from the right of the dragon to avoid the shiny gem in the middle of the left side, which lead to a blocky left side as well. However, despite all the difficulties in making the mesh within MeshLab, the mesh's circular center is quite accurate, keeping the gem and glare of the gem, and keeping smaller details like legs on the dragon. This ultimately made me realize that this combined mesh was unacceptable in quality to me, so I tried running different settings on Poisson surface reconstruction and tried using less meshes.



This combined mesh is much better, despite only using 4 meshes. The reasoning behind using only 4 meshes was that many meshes occupied the same space, either the back of the dragon facing the right side, or the right frontal view of the dragon. This made many overlapping meshes that caused the attempt at combining 7 meshes much more difficult to select good points with. However, this mesh is still not perfect, as there are locations of the mesh that are still significantly blurred out from Poisson surface reconstruction, the top is still blocky, and the base is still not aligned perfectly. The Poisson surface reconstruction in MeshLab is prone to crashing with too many iterations, which prevented me from searching for better parameters; the default parameters gave me the best results of the parameters I tried. The tools that I used to align the dragon to this level include the alignment tool, which I used point correspondences to attach meshes together, and the manipulator tool, which I used to rotate and translate meshes to fit each other cleanly. Overall, this mesh seems significantly better than the attempt at combining all 7 meshes.

In comparison, here are the original color images from both sides. I'd say the dragon model with 4 meshes is more accurate, though it is not perfect still.



Assessment and Evaluation:

Now that I have completed the project, I believe that although the project was quite time cramped, the overall project was very fun to work on and is a very interesting application of cameras.

The project was quite difficult, not because of the code, since most of the project comes from previous code, but because of the lack of computation power to run most of the algorithms, the lack of perfect scanning data, and the instability of both software and hardware. Running different parameters on trisurf took hours to get the best results of a mesh, and even then, most of my meshes were still subpar. Running trimesh was much quicker, but I did not think about trimesh or MeshLab right away when I should have. MeshLab's corresponding point alignment system was also very mediocre at getting points together, as many of my meshes were very difficult to align, or the system would place them at weird, disconnected locations. In particular, this made the head of the dragon extremely hard to align, leading to some sort of hydra that comes out of the model instead of the intended dragon. Most of my meshes were quite far apart, which made it difficult to select corresponding points. Once I got a decent combined mesh to run Poisson surface reconstruction, my desktop seemingly crashed every time I tried to run more than 20 as the reconstruction depth, which made perfect parameters extremely hard to get. I ended up using the default parameters on a smaller number of meshes due to these limitations on MeshLab, which gave me a better result than using all the meshes.

If I had more time, I would build my own mesh smoothing algorithm, look into more algorithms of smoothing and pruning, try more parameters with Google Colab and Tesla's GPU, and possibly take more scans of the object, especially of the right side with the gem, the front, and top of the dragon. This also was the biggest limit of the data, some of the scans were not perfectly aligned and missed big details of the model; the dragon model's shiny gem in the middle also skews some of the meshes, as it is harder to generate a good mesh of that side with the reflective light.

The weakest links and places I got stuck on the most were trying to generate a good mesh initially. It took many tries to run trisurf and get the best parameters for each mesh. It also took a long time to generate a good result with MeshLab. Mesh smoothing was also something I could not get working, but I didn't have enough time to fully flesh it out and decided there are more important pieces to this project, such as putting the mesh together in MeshLab.

Overall, I spent a good amount of effort to try to build a good model from the dragon, and despite the difficulties in software and hardware, the model was decently accurate. I believe that the model is quite accurate, as the scanning process of the dragon was smooth, the mesh generation and color extraction worked well, and tools like trimesh and MeshLab made it easier to recreate the model despite shortcomings in the software. This led to a solid reconstructed model that looks close to the original dragon model.

One last note that did set me back: I lost a day's progress in the middle of my project because my computer crashed mid execution of the ipython notebook. This corrupted the file completely. Luckily, I had a save in dropbox, but this would've been an absolute disaster had I not had a previous save in dropbox!

Appendix:

The code has code comments and snippets of markdown to make the ipython notebook more readable. I wrote some of the code within the mesh generation pipeline, which includes the bounding box pruning and the removing of points within the triangle pruning. I also wrote some of the visualization code, like calling different functions and printing different outputs. Lastly, I rewrote reconstruct to get the object mask, get colors from the colored images, and to fit within the mesh generation function.

Again, note that I did not write mesh smoothing, I couldn't get mesh smoothing to work properly and removed it from the code. Instead, I used the trimesh implementation of mesh smoothing.

The remaining features I did not write, and I used MeshLab to generate the combined mesh, as corresponding points took far too long to do on my own and incorporate into my code.

Credit to:

Jordan Rayfield and CS 117 staff for providing the Dragon scans and ability to scan the object in the first place.

Fowlkes for providing the project code (camutils, visutils, meshutils, calibrate).

Trimesh for doing the mesh smoothing and visualizations of some meshes.

Meshlab for doing the bulk of the correspondences, Poisson surface reconstruction, and alignment.