Jesse Allas

Charless Fowlkes

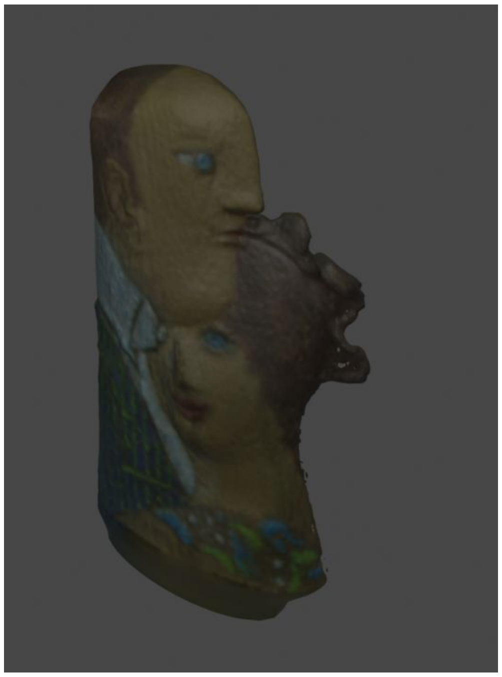
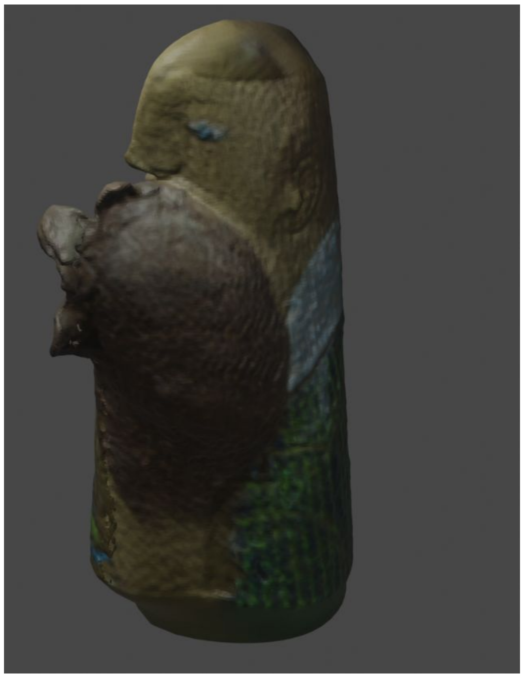
CS 117

6/8/2020

CS 117 3D Model Reconstruction

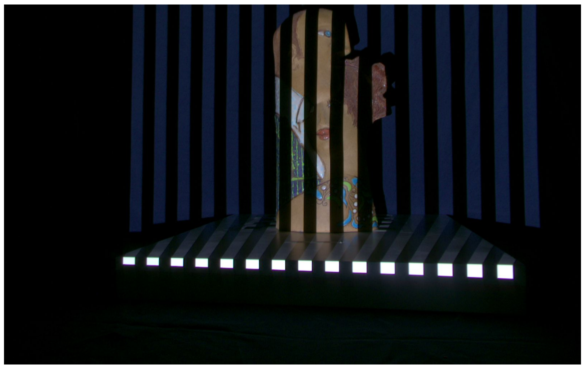
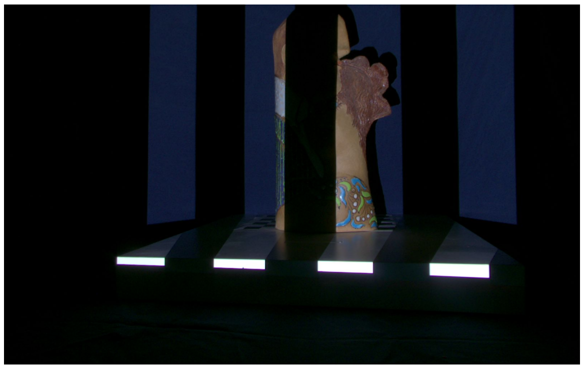
Project Overview:

The goal of this project (as defined in class) was to build a high quality 3D model of an object from a selection of four. My process was divided into three parts; camera setup, mesh generation, and alignment and rendering. The first goal was to calibrate the cameras used for the scanning and calculating the intrinsic and extrinsic parameters for both of them. Intrinsic and extrinsic parameters are the information about the camera, such as position in the world, rotation, focal length and principal point. After my cameras were calibrated, next was to generate the meshes from each of the scans, there were six in total. To do this, I used a reconstruction method to calculate 3D points for the meshes in each scan, and then I used a built-in function to create triangles that made up the mesh. After I made the triangles, I smoothed out the mesh by setting each point in the mesh to the average of its neighboring points. Last was to take all my generated meshes and align them together, and then use poisson reconstruction to finally recreate the object. I imported all my created meshes into MeshLab and used their alignment tools to align my meshes into the shape of the original object. After I got them aligned correctly, I used MeshLab's Poisson Reconstruction tool to clean up the mesh. Once the mesh was fully built, I imported it to Blender and created some renders for presentation.

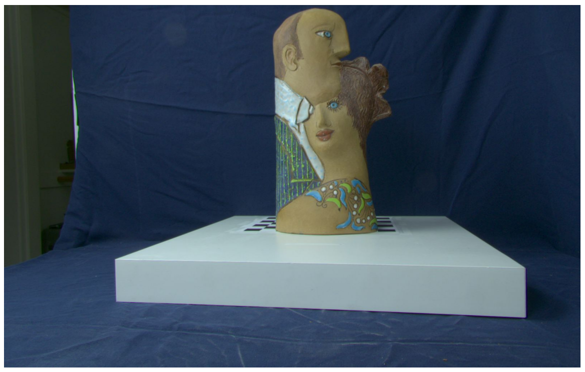
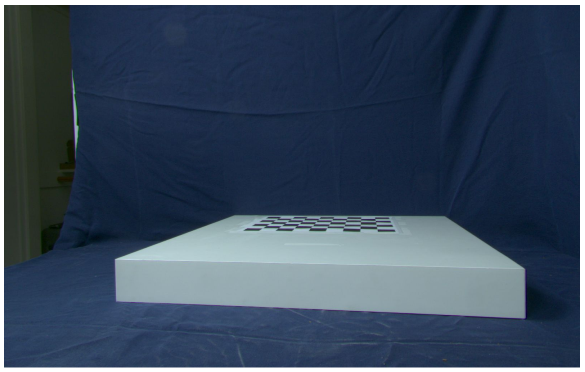



Data:

For our 3D model we had six scans. Each scan had a set of left camera images and right camera images. These were 20 frames of coded light along with a normal image of the object and a background image. The 20 frames are used to decode the object; calculating the surface information of the object by analyzing how light strikes the surface. Here is an example of what the coded frames look like.

The normal and background images are used to mask out the background. The normal image is the object in a normal lighting scenario, and the background is the same setting only with the object removed, this way we could remove the parts in the background.

I used MeshLab to handle mesh alignment and I used Blender to handle the 3D rendering. All of my coding was done in Python. This included camera calibration, mesh reconstruction and mesh pruning.

Algorithms:

For the project, I boiled down the pipeline into three steps: camera calibration, mesh generation, and alignment and rendering.

Camera Calibration was primarily handled using methods from previous assignments. The goal here was to calculate the intrinsic and extrinsic parameters of the cameras that I am using to scan the object. Intrinsic parameters are fixed values that cameras have. In our case we want the focal length and principal point (The perspective center of the camera). These are fixed, meaning that they never change for this camera. Extrinsic parameters are the position and rotation of the camera, these are not fixed values and can change.

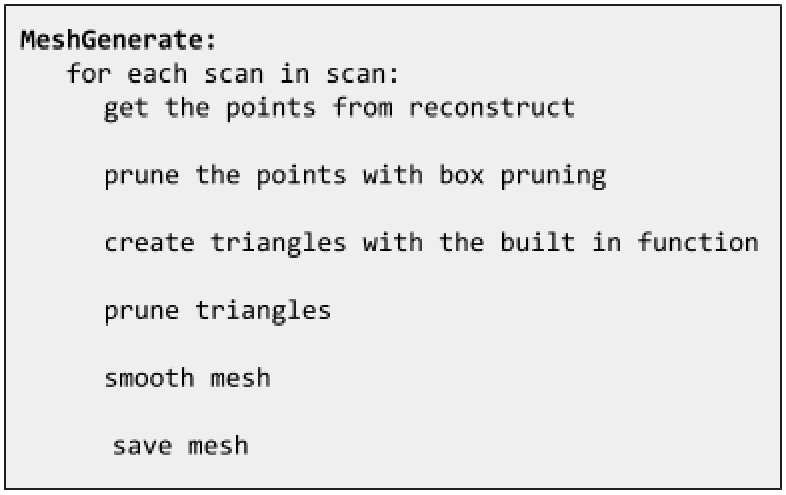
For this project, I already had the intrinsic parameters available to me from a previous assignment, so all I did was import that data. For the extrinsic parameters I used the same process as my previous assignments. Using checkerboard calibration scans given to me for this project, I passed those into the built-in Python library called "cv2". With this, I was able to calculate points for each camera that gave the 2D position of the scans relative to the cameras. Since I know the size of the squares on the checkerboard, I am then able to use these 2D points to get the 3D position of the calibration scan.

With the 2D and 3D points found, I could then calculate the extrinsic parameters. Again, this was the same process I used in Assignment 3, and I used the given calibration function to calculate the extrinsic parameters. This function works by passing into it the camera, the camera's 2D points, the 3D points, and an initial guess for the extrinsic parameters. This doesn't have to be exact, but it should be in the general position of the camera. Then we create an error function that calculates the difference between the 2D points our extrinsic guess gives us when projecting the 2D points, and the actual 2D points that we have. With this error function created, we pass this into the Scipy library's "Least Squares" function to iteratively refine our estimation. With a half-decent estimation, we are able to get an accurate calculation of the camera's extrinsic parameters. Below is a brief pseudocode explanation of the camera calibration.

```
GetCameraParams:
   import intrinsic camera data

   use calibration scans to get 2D and 3D points

   make an initial guess for extrinsic parameters

   iteratively run a Least Squares function with the
   given data to calculate the true extrinsic parameters
```

With our cameras calibrated, next was mesh generation. Mesh generation involved two parts; creating the mesh and smoothing the mesh. To create a mesh I had to pass the object scan files into a reconstruction method. This method took in a file with a set of coded light frames, and I decoded each of these separately to calculate how light bounces deforms on the object. With enough of these coded light frames coming from one angle, I am able to get an accurate depiction of the object (from that angle). Using all these frames and a threshold, I can get an even more accurate description by creating a point mask. This mask states which points in the scan are safe to use, and ignores the points that one or more of the frames couldn't accurately read. Accurate points are determined by calculating the color in grayscale at a point, repeating for the image's inverse, and then calculating the difference to see if it is within a defined threshold. This reconstruction method was given to me, it was something I made in a previous assignment and a refined version was made available to me for this project. I added on to it a function to handle background masking as well. An issue that reconstruction faces is that sometimes the background (for example, a wall or table) of a scan will slip past the masking. This is fixed by taking an image of the object, and then taking another image in the same spot only without the object. When comparing the images, the color at each point should be mostly the same, except for where the object should be. I make a mask listing all the spots where the color is drastically different.

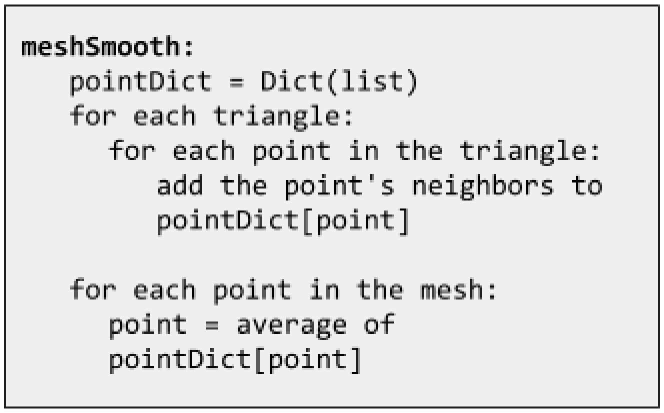
After obtaining the required data from a scan, I can then generate the mesh for it. Here I provided pseudocode explaining my mesh generation process.



```
MeshGenerate:
   for each scan in scan:
      get the points from reconstruct

      prune the points with box pruning

      create triangles with the built in function

      prune triangles

      smooth mesh

       save mesh
```

First I would like to point out the for loop. To create a proper 3D object, I need multiple meshes from different angles, so that means I am doing multiple scans. In this case I did six scans. With that out of the way, there is a process required to create each scan. First I get the points from reconstruction, as explained earlier. Next, I box prune the points. Box pruning is the process of defining the dimensions of a box, and removing points from the reconstruction that don’t fit in that box. The reasoning for this is that sometimes there are random stray points that get past the masking, and we want to remove those. Next I create the triangles that will make up our mesh. This is done using a built in Scipy function called Delaunay. Then comes triangle pruning. Triangle pruning is the process of removing triangles with edges that are longer than a threshold defined by me. This is done by looking at each triangle and getting the absolute value of each edge and checking the length of it. We remove the triangles with long edges. After this, we will have some points that are no longer being used. So we have to remove those too. Once the triangles are pruned and the points removed, we have a mesh for the scan. We could leave it off here, but smoothing the mesh will lead to nicer results.

Smoothing the mesh is the process of cleaning up the jagged or misshapen edges that might come from reconstruction. There are multiple ways to implement smoothing, but I went with the simple method of taking a point, and setting it to the average of all its neighbors. The process works as shown below.

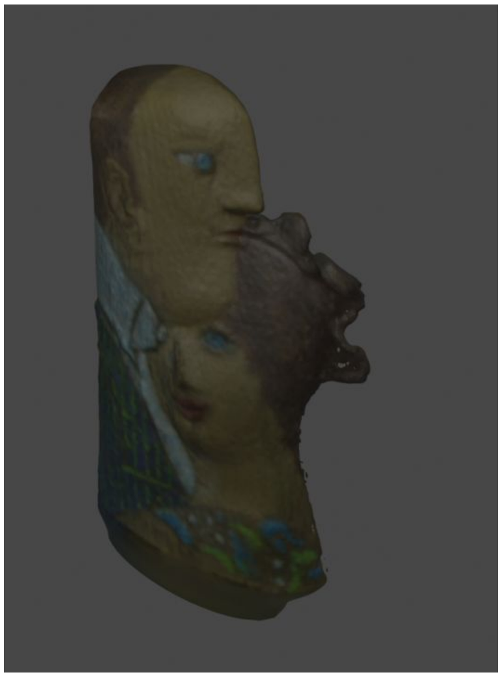
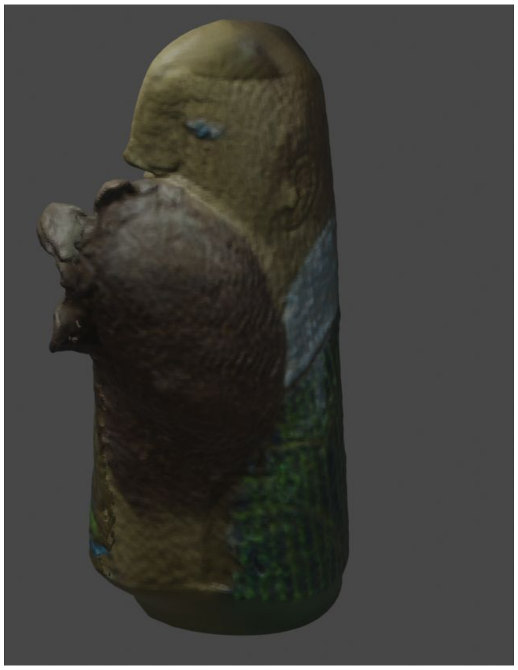
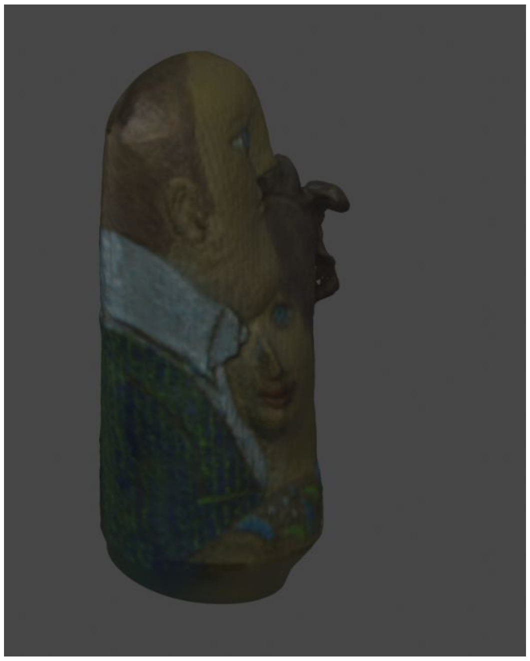
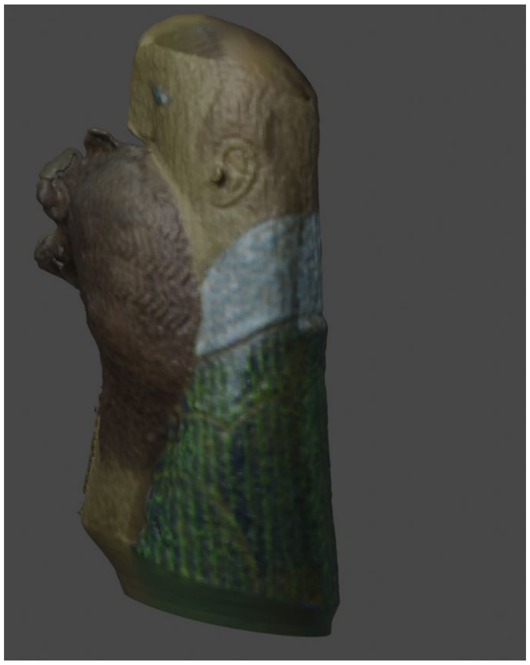


```
meshSmooth:
   pointDict = Dict(list)
   for each triangle:
      for each point in the triangle:
         add the point's neighbors to
         pointDict[point]

   for each point in the mesh:
      point = average of
      pointDict[point]
```

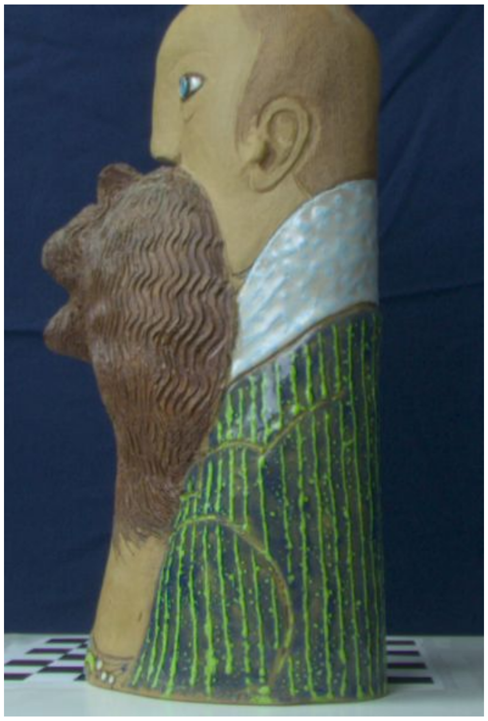
Since I am changing the position of the points based on the position of other points. I don’t want to change the data while working on it. I used a dictionary to store copies of the points, but any method to store lists of points would work. First we look at each triangle and for each point in that triangle, we list the other points in the triangle. Once I list each triangle, there should be a data structure that holds the points, and the list of neighboring points with respect to each point. Next I loop through each point in the mesh, and use the data from the structure to calculate the average of all the neighbors. Once this loop is done, all the points in the mesh have been averaged.

Results:

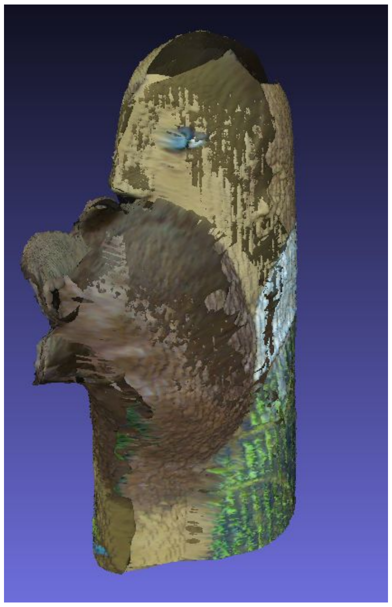
I created my final mesh in MeshLab and rendered it in Blender, here are the renders I created from four different angles

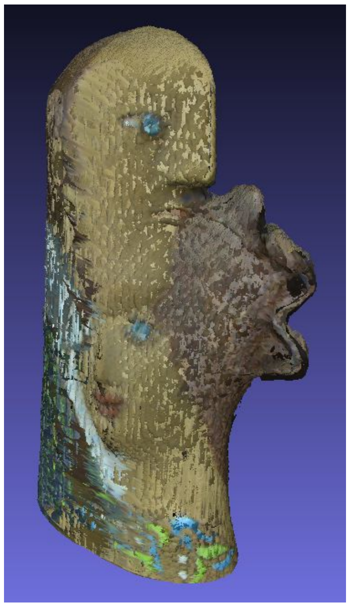





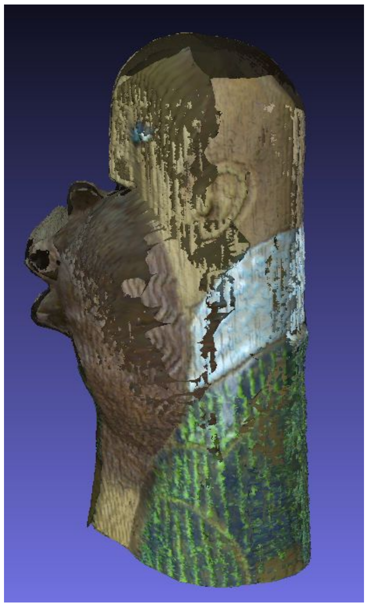
For reference, here are images of the original object.



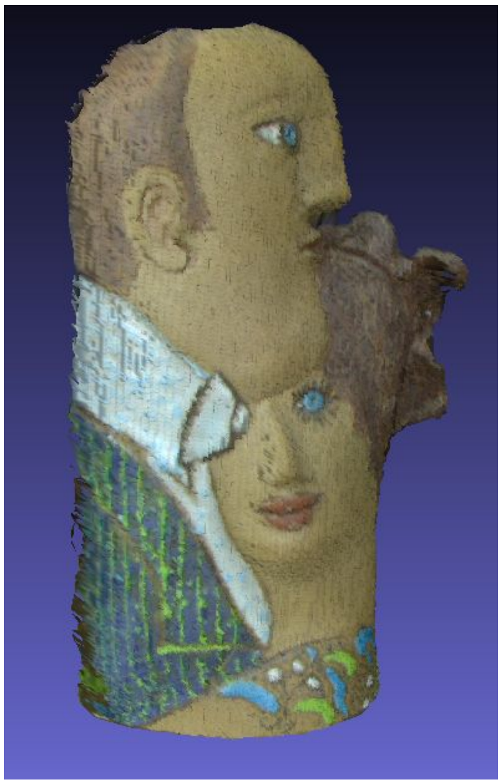
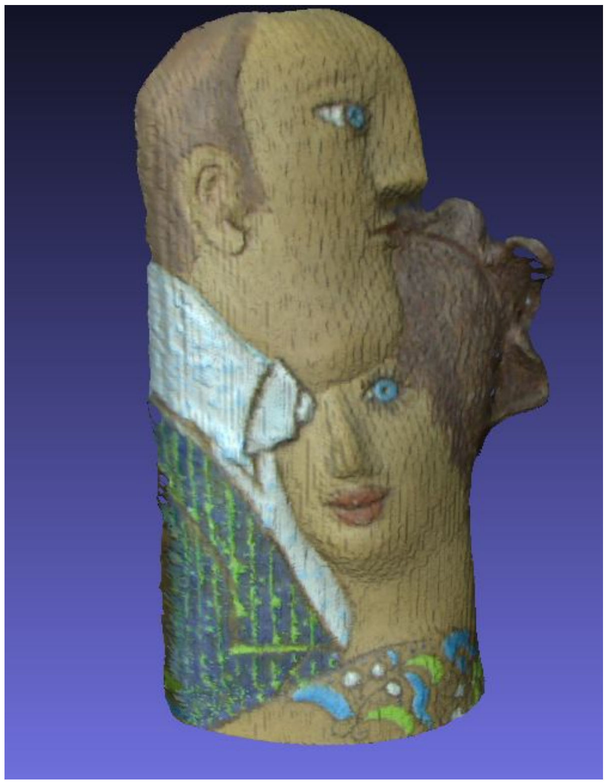
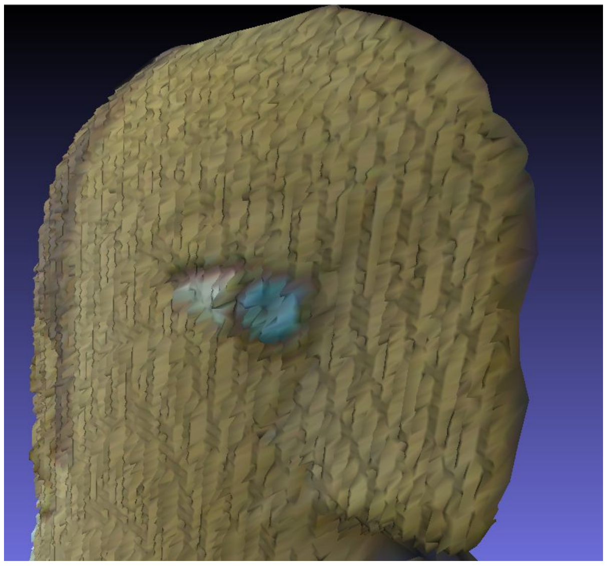
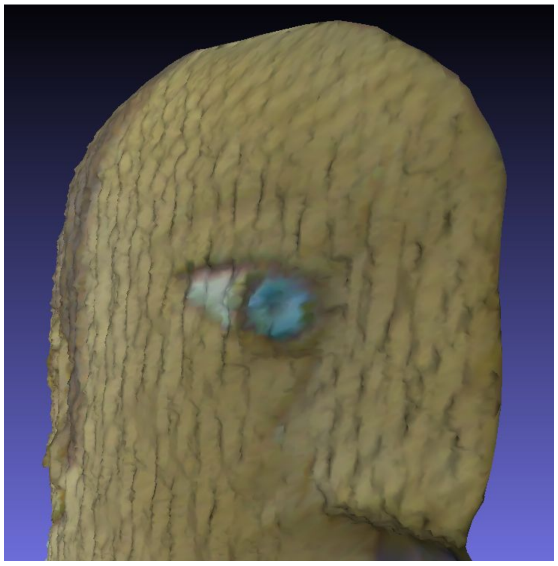
The renders I displayed are after poisson reconstruction and after mesh smoothing. Here is the mesh before poisson reconstruction. Notice the large gap in the head.





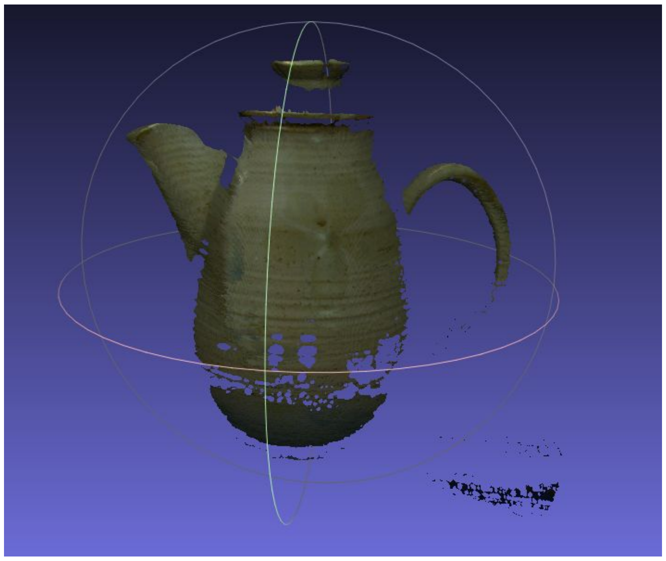


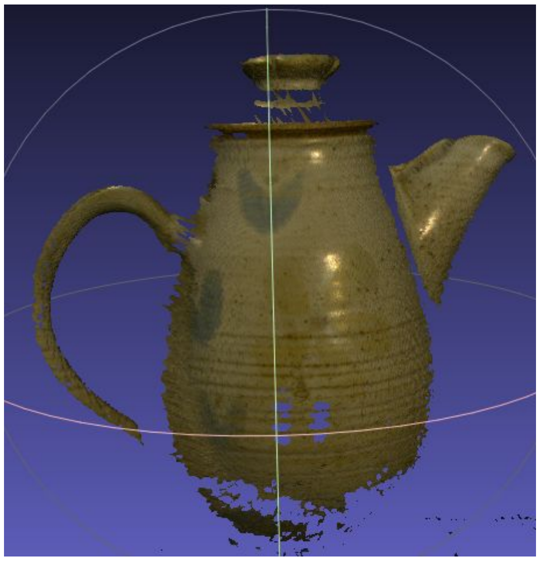
To show the effect that mesh smoothing has, here are before and after examples showing meshes before smoothing (left) and after smoothing (right)

Notice without smoothing the jagged edges in the full image, and the rough texture on the close ups.

As a side note, I initially attempted this project using a different object, a teapot. While unfinished, I do have the meshes for that one too.





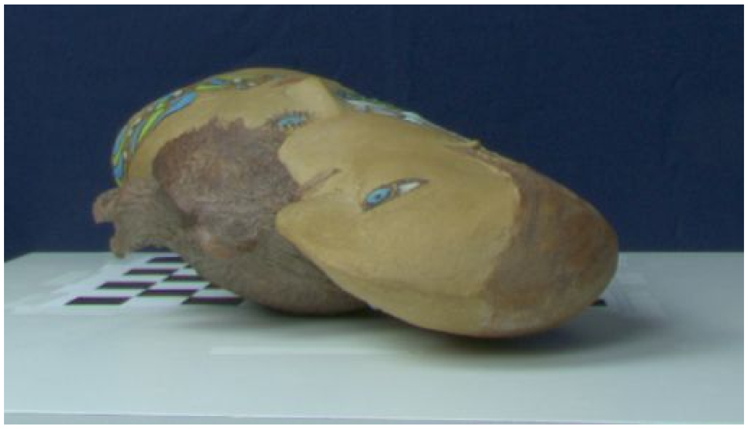
Notice that there is some leftover meshing on the bottom right of both images. This is from the background used in the images and I was unable to remove it. This combined with the number of holds in this mesh made me decide to change my mind and work on a different object.

Assessment and Evaluation:

Overall I believe this is a straightforward task for a computer to handle. While it isn't a quick task to complete, A computer is capable of 2D reconstruction. I can see this taking a much longer amount of time for larger objects, since that would increase the number of points in the mesh, but A computer can still handle the task.

As for my approach to the problem, there are some parts that fall short of my expectations. The mesh smoothing works, but given time I would like to implement a more refined method. I'm unhappy with the fact that I used two for loops in smoothing, I believe there is a way to do it using only array operators, but I have yet to find it. I am also unhappy with my masking and pruning. It works for my final object, but it wasn't up to my expectations for the teapot. I believe this is because of my defined thresholds and dimensions for the masking and pruning, and I would have to take more time to refine those. Another issue I had with my results was the numerous holes in my meshes. I know holes are to be expected, but I feel like my mesh generator was creating more holes then there should have been. There was a large gap in the top of my model. Poisson reconstruction covered it up, but you can see where the reconstruction happened, the colors aren't as detailed as the rest of the object at that spot. Given more time I would refine my parameters to get better meshes, and I would also implement a better smoothing method. I would also try to implement texture mapping if given more time. The mesh coloring is currently dependent on the mesh reconstruction. My theory is to calculate the position of each triangle, and then search in the images for the nearest point to that position and color the triangle the color of the point.

I had some limitation issues but nothing too major. The big issue I had revolved around the head of the object. In a lot of the scans, the head was partially cut off. There was one scan of the head with the object laying on it's side, but part of the head was shaded due to positioning.



I believe these head issues are what lead to my final object having gaps in the top. I know this final image was supposed to fix this, but I was still having issues.

One thing I noticed is that most of my holes were in shaded areas of the object. Even with the teapot, the holes were mostly in the cracks of the object or near the bottom. Because of this, I believe my masking method was running into issues with shading, and it wasn't properly reading the shaded parts. Normally inverse coded light frames fix this issue, but my background masking function seems to be catching the shaded areas as part of the background. The masking was definitely my weakest link.

Despite my issues, I am still happy with my final rendering. There were issues, but nothing was so bad that poisson reconstruction couldn't fix it. There were parts of my code that I felt were too slow or could be improved on, but the end product still bears a strong resemblance to the real thing, so I am satisfied with my results.

Appendix:

GetCams: This function is used to get the intrinsic parameters of the camera. It takes in a picklefile and unpacks the data for the cameras. I made this function for this project, but a lot of the code is reused from my previous assignment

GetExtrinsics: This gets the extrinsic parameters of the cameras. It takes in the calibration images, the cameras, and the initial parameters. It gets the 2D points from the calibration images and passes everything into **CalibratePose** to get the extrinsic parameters. I made this function for this project, but a lot of the code is reused from my previous assignment.

GetColorMask: This function takes in two images; an object and a background, and creates a mask to remove the background. I made this function from scratch.

MeshGenerator: This function handles the generation of the mesh triangles along with the smoothing of these triangles. It also exports these triangles to a .ply file for MeshLab. It takes in a file directory for all the scans and runs **Reconstruction** on each one to calculate points for the mesh. I could've broken this up into multiple functions, but the constant passing around of variables was complicating things. Also I like to have everything condensed together. I made this function from scratch.

The following code was all given to me for this project, but it is all based on previous assignments.

Camera: This is a class object to hold the data for the cameras. This includes both the intrinsic and extrinsic parameters.

CalibratePose: As hinted at earlier, this calibrates the positions of the cameras and returns the extrinsic parameters. It does this by creating an error function out of **Residuals** and using least squares optimization to calculate good parameters. It then updates the camera extrinsics.

Residuals: This takes in 2D points, 3D points, a camera, and a set of camera parameters. It calculates the difference between the 2D points, and the 2D points that the camera would **Project** if given the parameters and 3D points.

Project: This function takes in 3D points and projects them in 2D for a camera.

Triangulate: This does the opposite of **Project**. Given 2D points for two cameras, along with those two cameras, it calculates the 3D points that those two cameras are projecting.

Reconstruction: This function reconstructs the 3D points of an object given a set of scans. It does this by running the **Decode** function on these scans and using the points and masks given to create the 3D points of the mesh. I added in an extra part to handle the mesh coloring as well.

Decode: This is given a directory to a set of scans, and decodes them, along with the mask.

Running the Code:

My code first uses **GetCams** and **GetExtrinsics** to get the camera parameters. With these calculated, we can then get the accurate 3D points by **Triangulating** the cameras.

Then I use **MeshGenerator** to reconstruct meshes from the scans. This does a loop through all the scan files and runs **reconstruction** on all of them to get the points. From there I can create the triangles for the mesh. **MeshGenerator** also exports all the meshes. Once this function is done, I should have six exported ply files, one for each mesh.

From here my code is done and I handle the alignment and poisson reconstruction in MeshLab. Lastly I render my final construction in Blender.