3D RECONSTRUCTION BASED ON STRUCTURED LIGHT

Mutian Xu 16083274   Instructor: Charless Fowlkes

ABSTRACT

This is a final course project report of UCI CS117-Project in Computer Vision, instructed by Prof. Charless Fowlkes. In this project, we reconstruct a 3D model based on structured light method using the scan images of a 'teapot'. This report includes an introduction of the main method, a basic description of the algorithms, a final 3D model and an evaluation of the result.

1. OVERVIEW

The goal of this project is to produce a high-quality 3D reconstruction of a teapot from a collection of structured light scans.

A structured-light 3D scanner is a 3D scanning device for measuring the three-dimensional shape of an object using projected light patterns and a camera system. Projecting a narrow band of light onto a three-dimensionally shaped surface produces a line of illumination that appears distorted from other perspectives than that of the projector, and can be used for geometric reconstruction of the surface shape (light section).

A faster and more versatile method is the projection of patterns consisting of many stripes at once, or of arbitrary fringes, as this allows for the acquisition of a multitude of samples simultaneously. Seen from different viewpoints, the pattern appears geometrically distorted due to the surface shape of the object.

Although many other variants of structured light projection are possible, patterns of parallel stripes are widely used. The picture shows the geometrical deformation of a single stripe projected onto a simple 3D surface. The displacement of the stripes allows for an exact retrieval of the 3D coordinates of any details on the object's surface.

2. DATASET

In this project, we used the scanner to get several scans from different views. Every scan is a 1920*1200-pixel image, stored in a file called teapot, consisted of 7 files named 'grab_0_u' ~ 'grab_6_u'. Every file is split into 4 images whose prefixes are 'color' to provide the color information, 40 images whose prefixes are 'frame_C0' indicating the frame from one view and 40 images whose prefixes are 'frame_C1' from another view. In every 40 images, the first half of them ended up with 0~19 are scanned vertically and others ended up with 20~39 are scanned horizontally. Figure 1 shows a horizontal-scanned image from one view.

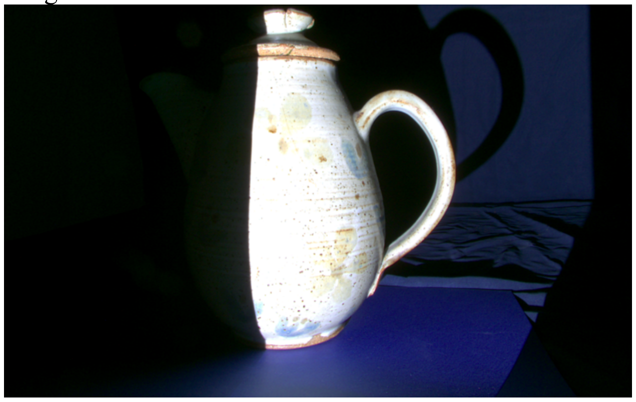
Figure 1: A horizontal-scanned image named 'frame_C0_00'

In this project, we will align 7 outputs corresponds to 7 files into one model using MeshLab.

In addition, we use 22 pairs of chessboard pictures in the directory 'calib_jpg_u' to get accurate parameters for the scanner cameras. Figure 2 shows one of chessboard images for calibration.

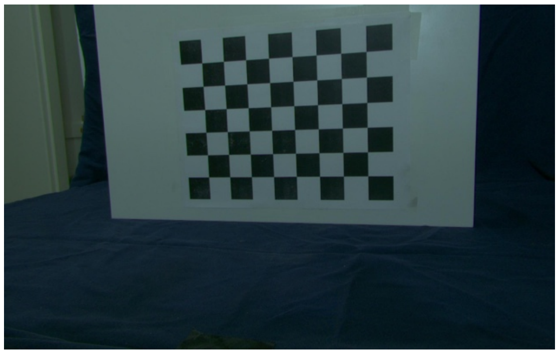
Figure 2: A chessboard image for calibration

3. ALGORITHMS

3.1 Camera Calibration

In this section, we firstly use the function from OpenCV library to get the intrinsic parameters of the scanner cameras. Then generate the known 3D point coordinates of points defined as *pts3* on the checkerboard in cm using meshgrid function in numpy library. We specify an initial

guess *parameters_intial* of the camera and the reprojection error *residuals*, next use *scipy.optimize.leastsq* to get the optimal extrinsic parameters *params_update* in order to minimize the reprojection error, which can be represented as:
params_update=scipy.optimize.leastsq(lambda params: residuals(pts3,pts2,cam, parameters_intial)

3.2 3D reconstruction based on Structured Light method

a. Decode

Project both an image and its inverse and compare the two frames to see which is brighter. A pixel in each camera image is "decoded" by looking at the sequence of light and dark across all the frames.

For each pair of images, recover the bit by checking to see that the first image is greater or less than the second. We also maintain a seperate binary array (mask) the same size as the images in which our mark "undecodable" pixels for which the absolute difference between the first and second image in the pair is smaller than some user defined threshold. This will allow us to ignore pixels for which the decoding is likely to fail (e.g., pixels that weren't illuminated by the projector). We mark a pixel as bad if any of the 10 bits was undecodable.
Also we maintain a *color_mask* to avoid triangulating pixels that are part of the background.

After thresholding the pairs we should have 10 binary images. We first converting the 10 bit code from the gray code to standard binary (binary coded decimal). Once we have converted to BCD, we then produce the final decimal value using the standard binary-to-decimal conversion (i.e., $\Sigma^{9}_{n=0}$ B[9-n]*2^{n}).

Gray-to-Binary Conversion: XOR each binary code bit generated to the Gray code bit in the next adjacent position. ex: convert the Gray code word 11011 to binary.

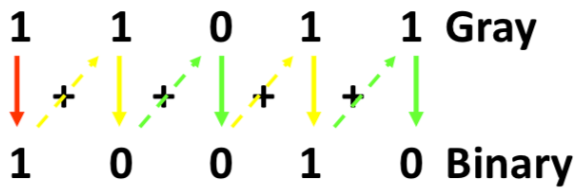

Figure 3: Convert the Gray code word 11011 to binary

Because our projector is 1024=2^10 pixels wide, we use a binary code with 10 bits so we project 10 different patterns in a single view.

b. Reconstruction

Call the decode function four times to decode the horizontal and vertical images for both the left and right cameras. We combine the horizontal and vertical codes to get a single (20-bit) integer for the left and right cameras using ***Code = Horizontal + 1024*Vertical***. We also combine the corresponding binary masks so only pixels with both good horizontal and vertical codes are marked as valid.

For each pixel in the left image which was succesfully decoded, find the pixel in the right image with the corresponding code. One way to do this effeciently is using the ***numpy.intersect1d*** function with return_indices=True in order to get indices of matching pixels in the two images.

Now that we have corresponding pixel coordinates in the two images and the camera parameters, use triangulate to get the 3D coordinates for this set of pixels.

Moreover, we need to **record the color** of the corresponding pixel in the color image. We store the RGB values for the points in a 3*N array where N is the number of points, **using the pts as the index** matched to the RGB 3D array, which can be represented as the following code: *color = color_image[pts2[x]][pts2[y]]*

3.3 Meshing

To display the reconstructed scan as a surface, we need to generate triangular faces of the mesh which connect up the points. Figure 4 shows an example of triangulated meshes.

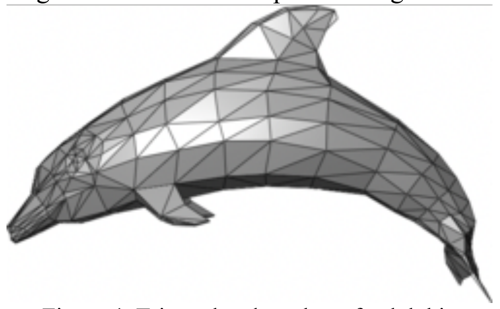

Figure 4: Triangulated meshes of a dolphin

a. Delaunay Triangulation:

Create a "nice" triangulation by choosing triangulation whose smallest angle is as large as possible... avoid skinny triangles. A triangulation fulfills the **circle criterion** if and only if the circumcircle of each triangle of the triangulation does not contain any other point in its interior. Figure 5 shows how Circumcircle Criterion works.

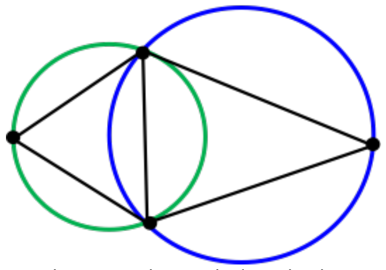

Figure 5: Circumcircle Criterion

Figure 6 shows an example of building a triangulation:
-Successively add in one point at a time;
-Add in edges connecting ***p*** to neighbor vertices, then flip edges as necessary to assure circumcircle/smallest angle property holds true.

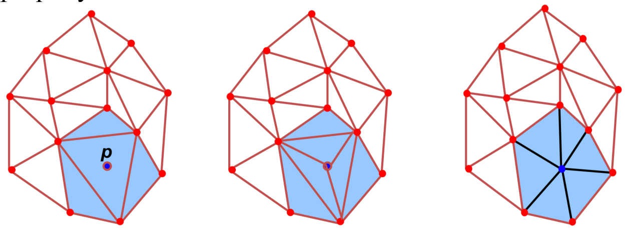

Figure 6: An example of building a triangulation

b. Mesh Clean:
Some limitations of connecting up image neighbors will cause noise of our model.
-There will be some points where we don't have data due to noise, occlusion of projected light;
-Connections don't respect discontinuities in the surface.
Solution: Since we have 3D from triangulate function, only connect points if they are not too distant in space.

Idea 1-Tiangle Pruning: Remove triangles from the surface mesh that include edges that are longer than a user-defined threshold. Typically when we have points on the surface, they will be relatively close by so this gives a way to get rid of noisy points off the surface. Figure 7 shows this method.

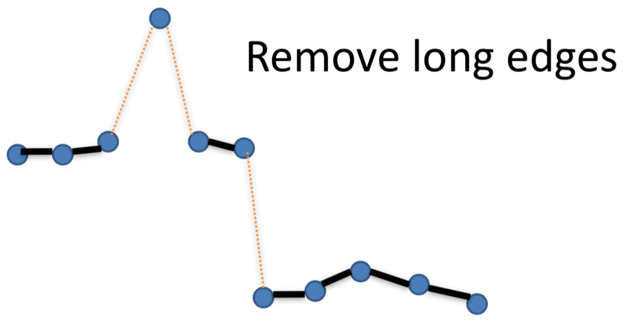

Figure 7: Remove long edges

Idea 2- Smooth surface: Figure 8 shows how to smooth surface: Move each point towards the average of its neighbors in the mesh.
We need to repeat this process multiple times, each time the mesh will get smoother. I repeat 3 times in my code.

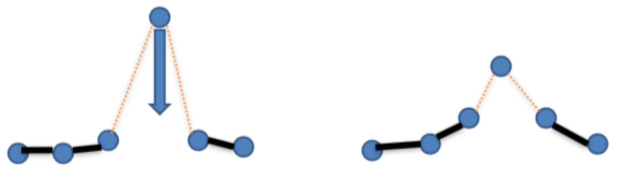

Figure 8: Smooth surface

We need to **update the color array and pts array simultaneously** while pruning. After mesh cleaning, we will find points that are no-longer connected to any neighbor in the mesh (e.g. they are no longer in any mesh triangle). Find these points and remove them as well so that the final mesh we produce doesn't include any unreferenced vertices.

3.4 3D alignment
As mentioned before, we need to align 7 outputs corresponds to 7 files of scanned pictures into one model using a third-party tool, **MeshLab**.

We need to save out each individual mesh data as **.ply** file and load them all into the MeshLab. Figure 9 shows how to combine meshes into an aligned scan.

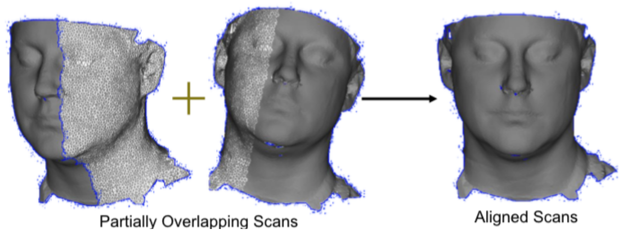

Figure 9: An example of combining meshes

3.5 Poisson Reconstruction
Main idea: The normal vectors on a surface are the same as the gradient of an implicit function F defining the surface.
-Figure 10 relationship between the normal field and gradient of indicator function.

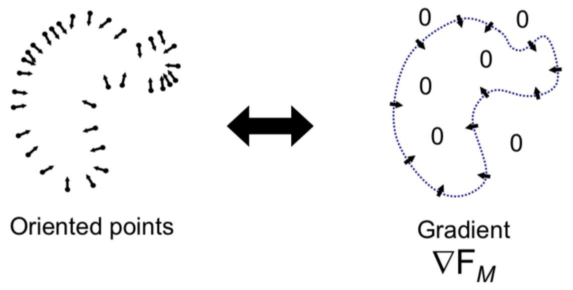

Figure 10: Relationship between the normal field and gradient of indicator function

-Represent the points on the surface by a vector field $\vec{V}$;
-Find the function F whose gradient best approximates $\vec{V}$:

$$\min_F Q(F) = \min_F \| \nabla F - \vec{V} \|$$

-Can transform this into a "Poisson problem" which is a well-studied mathematical equation (describes heat flow, gravity, electrostatic interactions):

$$\nabla \cdot (\nabla F) = \nabla \cdot \vec{V} \Leftrightarrow \Delta F = \nabla \cdot \vec{V}$$

In our project, we will directly use the **MeshLab** to perform Poisson surface reconstruction.

4. RESULTS

4.1 Camera Calibration

We called our function to re-project the used the chessboard corner. Figure 11 shows that the reprojection points all mapped with the corner, which means we successfully get accurate parameters.

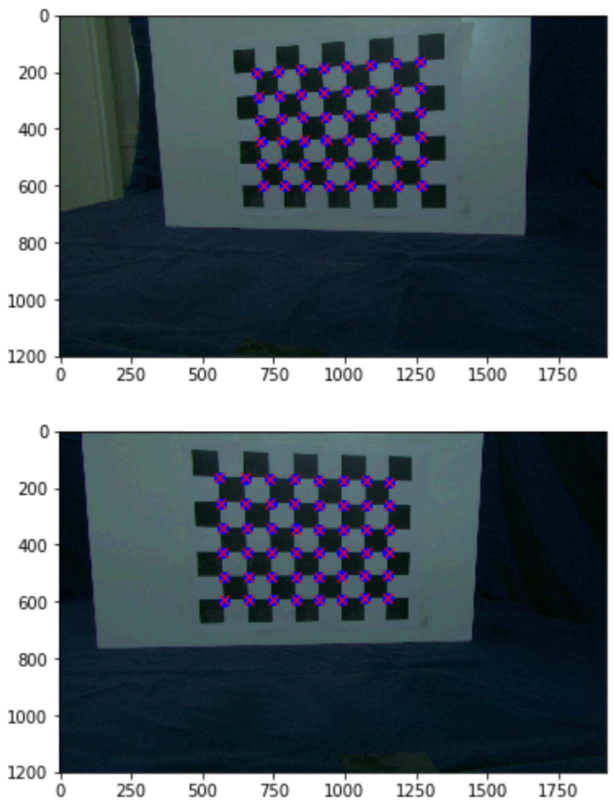

Figure 11: Reprojection result

4.2 Code, Mask and Color_mask

We successfully decoded the scanned images and got the mask and color_mask, which are shown in Figure12 respectively.

4.3 Initial reconstruction

Figure 13 shows the reconstruction visualization from XY-view. We successfully reconstruct although with some noisy points, which will be cleaned in the next step.

Reconstruction provides us with the set of verticies. To find the faces of our mesh, we use the 2D coordinates of the points as they were visible in the left image (pts2L). Use the function *scipy.spatial.Delaunay* on these 2D coordinates to get the list of triangles. These faces along with the 3D coordinates of the points provide a description of the surface in 3D.

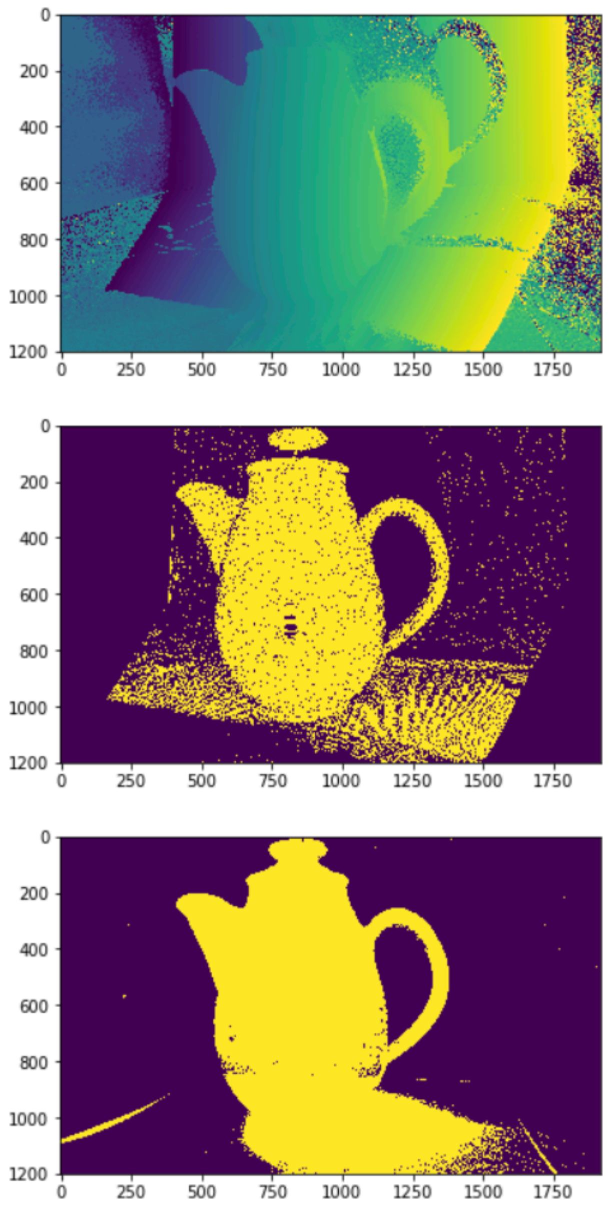

Figure 12: Reprojection result

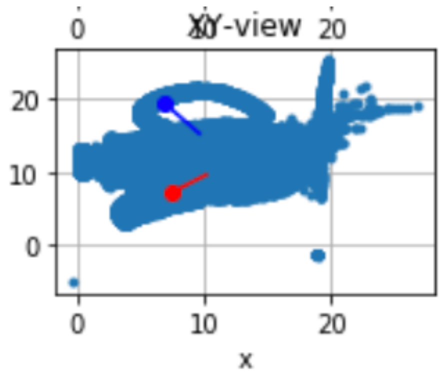

Figure 13: The initial reconstruction visualization from XY-view

4.4 Reconstruction after Mesh Clean

I tested every grab with different threshold and define the best threshold in the python notebook. Figure 14 shows the reconstruction visualization from XY-view after Mesh Clean. Compare with figure 13, obviously the surface becomes **more smoothed** and many noise points are **pruned**.

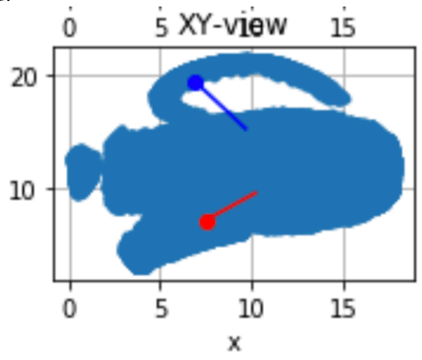

Figure 14: The reconstruction visualization after Mesh Clean from XY-view

4.5 3D Alignment Result

Figure 15 shows 7 grabs before 3D alignment.

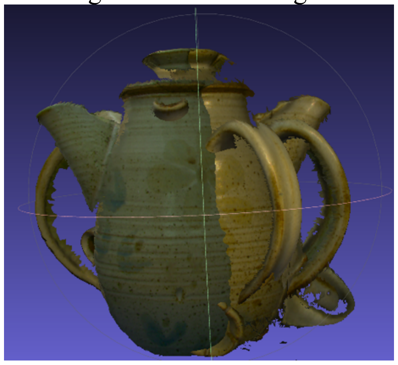

Figure 15: 7 grabs before 3D alignment

Figure 16 shows the result after 3D alignment using MeshLab. It is clear that there still exist some **missing parts**, which **need to be resolved in Poisson Reconstruction.**

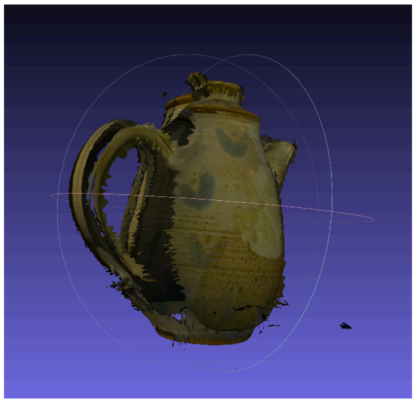

Figure 16: The result after 3D alignment using MeshLab

4.6 Poisson Reconstruction Result

Figure 17 shows the result after Poisson Surface Reconstruction using MeshLab, saved in **poisson.ply**. **The missing part problem is resolved.**

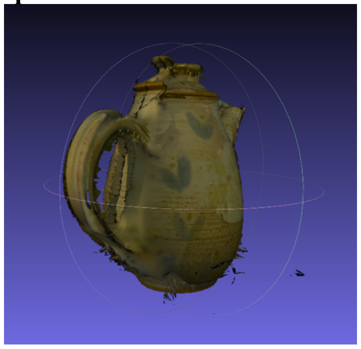

Figure 17: The result after Poisson Surface Reconstruction

4.7 Blender Visualization

Figure 18 shows the final result shown in **Blender.**

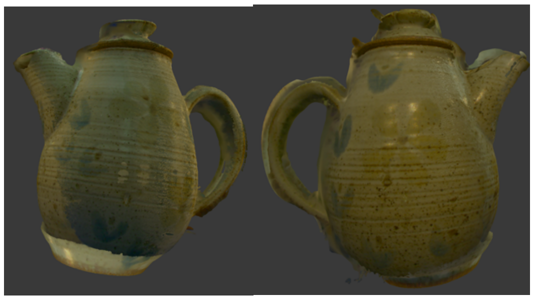

Figure 18: Final result shown in Blender

5. ASSESSMENT AND EVALUATION

5.1 Assessment of Difficulty

In this project, we need to use computer programming to solve most of the work, trying to combine all the algorithms and code from every assignment. It's not an easy job for me because we need to **understand the principle** of every step and use the programming language to implement it. It is involved in different kinds of problems about **matrix analysis, geometry mathematics, image processing and 3D vision**. In the code level, we need to be familiar with numpy and other useful libraries in computer vision field.

5.2 Success

In the code level, I successfully realized most of the work using my own code. As mentioned in RESULT part, the **decode**, **reconstruction** and **mesh-clean** function works well. For the optimization of the previous assignment, I wrote code for add color_mask, color map and mesh smoothing, which all get good results. I also encapsulate the code and make my code's structure easy to follow and refine. It **can work on different scans of image** with little revision of the file path and threshold.

Moreover, for the result of every grab, my code works very well because **most of the noise are filtered**, keeping the **complete shape, smooth surface** and **true color** of the object, which also means the threshold and other **user-assisted value are set correctly**.

Although use the MeshLab, the result is good because **the right operation of the software**. For example, the 3D alignment **nearly combine all the grabs** and the Poisson Reconstruction fill the missing part and possibly **create a closed object.**

5.3 Limitation

For my approach, the biggest limitation is that I haven't write code by myself to realize the **3D alignment and Poisson Reconstruction** but directly **get the help from MeshLab**. However, these two parts directly decide the final performance of the project. For example, in 3D alignment, we need to **select corresponding points by ourselves**. This will definitely cause some errors, which can be seen from figure 16 with a big missing part.

In addition, my mesh generation part **needs the help of users** to decide the threshold such as box-limit and trithresh, and **needs several human experiments** to decide these parameters which are actually **not the super-perfect** values.

5.4 Data Defect

In our project, we can find that the scan image are not perfect because there exists some **light difference** between different views. For example, the brightness of grab 0 is obviously less than grab4. Figure 19 shows this problem. **This will directly cause the color difference of our final result.** As you can see in the figure 18, some parts of the object are green while others are yellow.

Plus, **the number the scanned images is not adequate**, which cause our initial result of every grab is not complete because we need to reconstruct based on more views. This also **makes hard to align them** because **the missing part of the object is too much.**

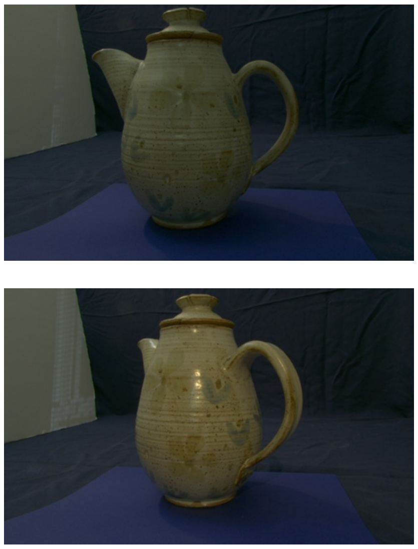

Figure 19 The brightness difference between grab0(above) and grab4(below).

5.5 Difficult Part

In this project, I got stuck in several parts:

-**Record the color of the corresponding pixel:**

I think this part is a little tricky because it's hard to figure out the relationship between **the pixel and the pts.** And writing code for this part also took me some time.

-The map tri part:

Removing unconnected points is a bit tricky because tri contains indicies into the pts arrays so if we remove elements from pts, the indicies in tri will no longer be valid and **need to be updated as well.** We need to create a map list and figure out how to implement in the mesh generation part.

-Mesh smoothing:

It's a little hard to find the neighbor of the pts and the tutorial of the Delauny is not very clear of this method. We need to print out several times to **understand how this function works and how to add this into our previous code.**

-3D Alignment:

It's a little hard to select corresponding points because of the unfamiliarity of the MeshLab. Also the big **missing part of every grabs** makes this task difficult. We need to select grabs as close as possible so that we can gradually build a complete object.

5.6 Future Work

If I have more time on this project in the future, I will firstly scan **more images** in **better light condition**.

Moreover, I will also try to use some **machine learning** method to let machine to **learn the optimal threshold parameters** instead of defining them by users.

Last, it's necessary to **implement the ICP algorithm by myself** in 3D alignment and try to write code for Poisson Reconstruction referenced from related works such as JUH's open code.

5.7 Conclusion

In this project, we create a 3D model based on scanned images using structured light method. We successfully implement different 3D vision related function such as triangulation, calibration and mesh generation. This project requires us to fully understand the basic knowledge about light geometry, matrix analysis and image processing in 3D reconstruction field.

After three-months learning and this project, I built a basic system of Computer Vision problem. Fortunately, the final result of this project is **nice** and basically **completed all the initial millstones** and **prove the validity and good performance of our work.**

(See the last page for Appendix and Acknowledgement)

APPENDIX

In this project, for getting the best performance and good implementation of object-oriented programming, I wrote most of functions by myself and also import some functions from instructor's code.

Code from instructor:
-Use Camera, triangulate, calibratePose, makerotation, decode in camutils.py;
-Use vis_scene in visutils.py to visualize 3D view;
-Use calibrate.py to get intrinsic parameters of the camera;
-Use the code from HW3 and calibratePose in camutils.py to get the extrinsic parameters of the camera.
-Use meshutils.py to get the .ply file.

Code wrote by myself:
-Add **color_mask** in the **decode function** in camutils.py to get rid of the background of the scanned object;
-Add **color record** to the **reconstruction function** in camutils.py**;**
-Referenced Instructor's code to **modify bounding-box pruning and triangle pruning** in **mesh generation** in camutils.py;
-Referenced map tri in the lecture slide to modify the **mesh generation** in camutils.py;
-Add the code for **mesh smoothing** in **mesh generation**;
-**Encapsulate** the mesh generation function in camutils.py to fit several directories of scan images and different threshold.
-Create a Jupyter Notebook **execution.ipynb** to **show every result** respectively mentioned in 4.RESULT and **get 7 .ply files** corresponding to 7 grabs.

ACKNOWLEDGEMENT

Thanks for the patient instruction during this course, every assignment and the final project from Prof. Charless Fowkless and Miss. Samia Shafique, who also ignite my interest of this area.