Project 1: Structured Light Depth Acquisition

Student ID:13028702 Student Name: Yifei Gao

Synthetic Data Set Report

3D Reconstruction:

Decode gray pattern

I decided to convert all color images into grayscale before decoding their pattern, as compute the uv_code at each color channel does not provides extra information and compute one channel only is more efficient. My strategy of decoding the light pattern is to compare the value of intensity in different projection pattern pairs, the pattern itself and its inverse, then uses this value to decide whether current pixel is 1 or 0 (on or off) in the binary sequence. I decide to go with higher intensity in the pattern than its inverse as 1, and lower as 0. As we have 20 pattern each in both direction (horizontal u direction and vertical v direction), the length of the binary sequence in each direction is 10 bits . These binary sequences are converted into decimal number for further processing. This uv_code we obtained then can be used with the image coordinate to estimate the depth map. Example of the uv_code I've obtained using 'decode_uv.m' is shown as follow:

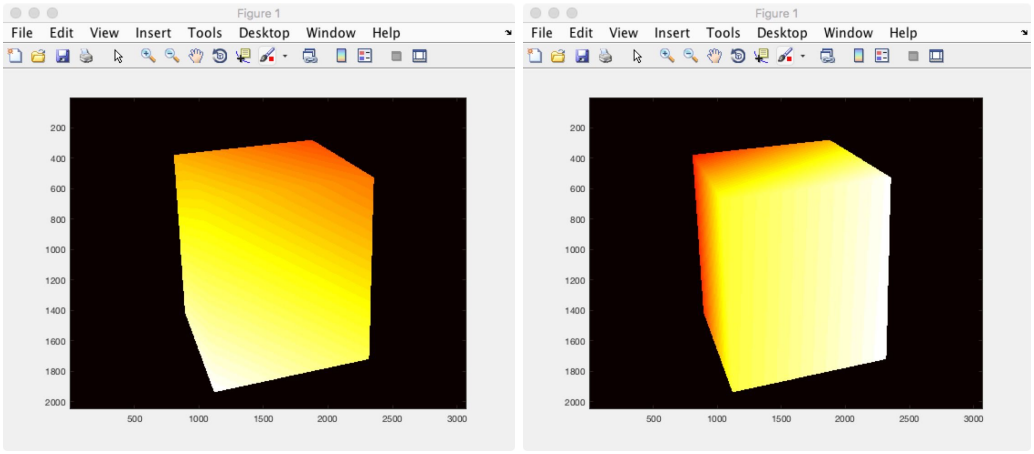


Figure 1. UV_Code result.
(Left: u_code of 'Cube_T1'; Right: v_code of 'Cube_T_1')

Eliminate pixels

In order to filter out the uncertain gray code in the data I also computes the total unsigned distance of each pixel while decoding the gray pattern. In the paper 1 they measure this by signed difference overall colorbands and using a very high threshold, but as the synthetic data we're given is 'perfect' so I've set a very low threshold for synthetic data. I labeled these uncertain pixel with a -1 flag to avoid further computation of its depth.

Determinate the depth map

As we already have two set of data at this stage, the camera image plane coordinate of each pixel and the decoded uv_code of the projector image plane, and a given set of calibration data, we can estimate the depth using the routine we learned in ‘Lab 2’. We are solving for

$$\hat{\mathbf{w}} = \underset{\mathbf{w}}{\operatorname{argmin}} \left[\sum_{j=1}^{J} \left(\mathbf{x}_j - \mathbf{pinhole}[\mathbf{w}, \mathbf{\Lambda}_j, \mathbf{\Omega}_j, \boldsymbol{\tau}_j]\right)^T \left(\mathbf{x}_j - \mathbf{pinhole}[\mathbf{w}, \mathbf{\Lambda}_j, \mathbf{\Omega}_j, \boldsymbol{\tau}_j]\right) \right]$$

in this part, so we can stack the linear constraint using both data set and solve for Ax=b least square problem. As we do not have the problem of projector having lower resolution than we expected, so I didn’t do any smoothing to the uv_code to cancel artefacts in this part. (We do not estimate the depth for pixel that we have labeled as uncertain in the uv_code.) Example of resulting depth map is shown as follow:

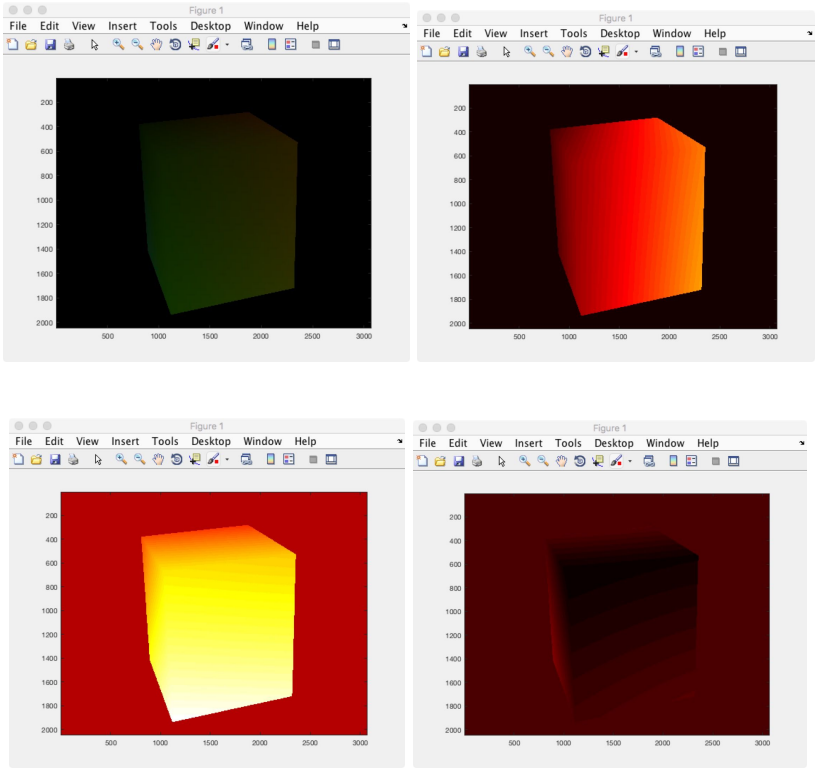
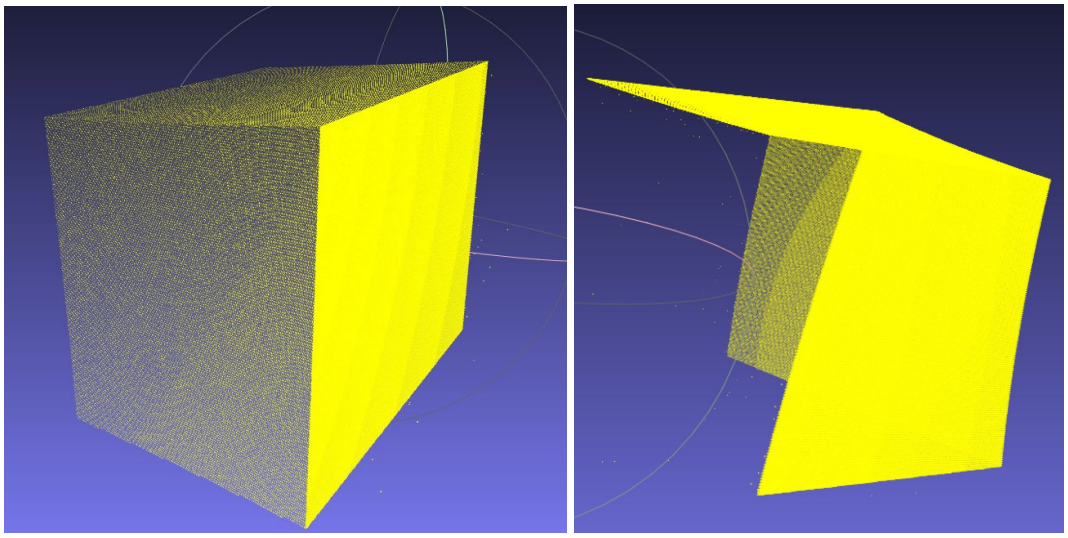


Figure 2. Depth map result.
(From top-left to bottom-right: combine(u,v,w); u_value; v_value; w_value)

Visualize result

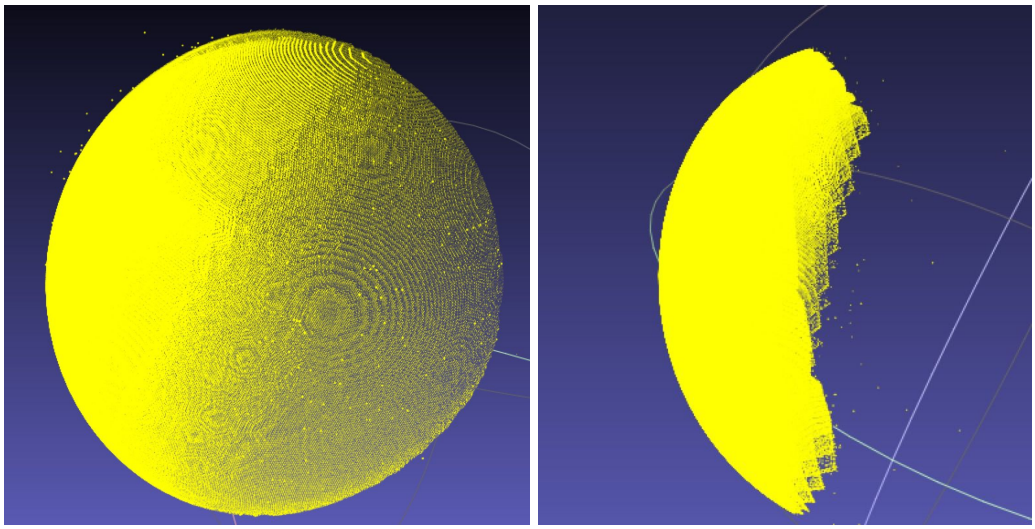
I’ve used meshlab to visualize the result of the point cloud generated for each dataset.

Cube_T1:



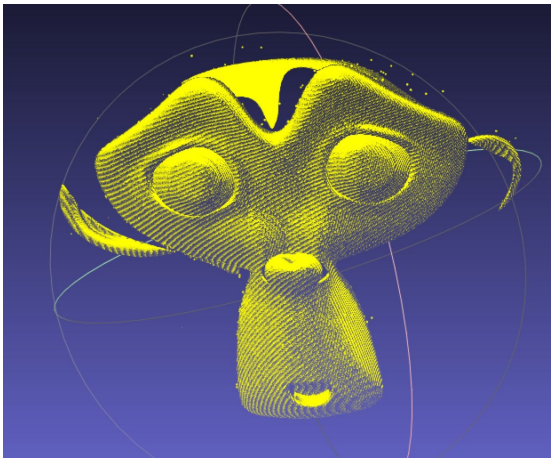
Camera Perspective | Side Perspective

Sphere_T1:

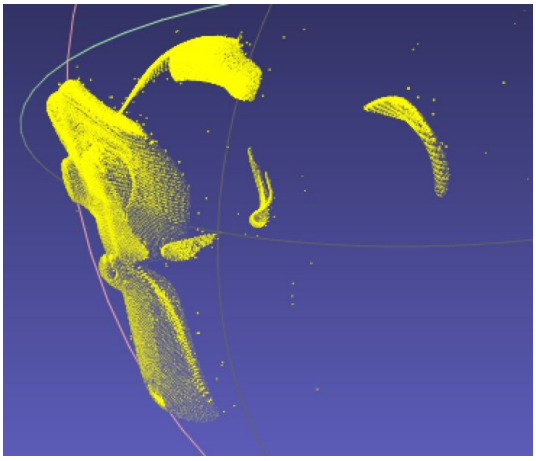


Camera Perspective | Side Perspective

Monkey_T1:

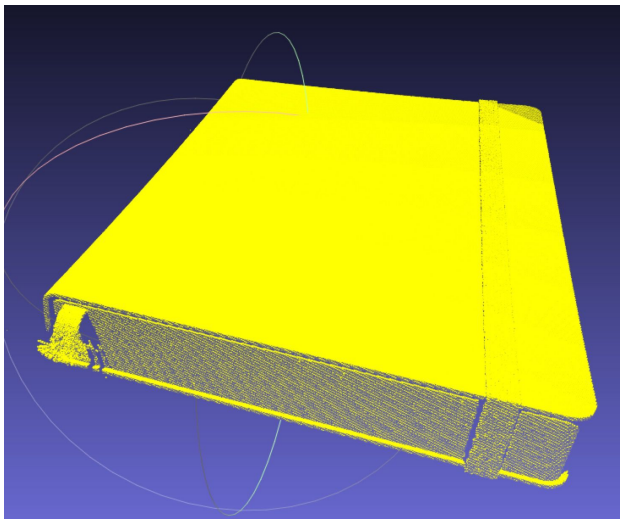


Camera Perspective

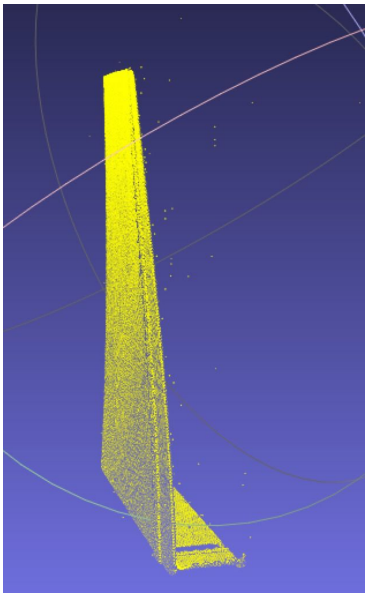


Side Perspective

Notebook_T1:

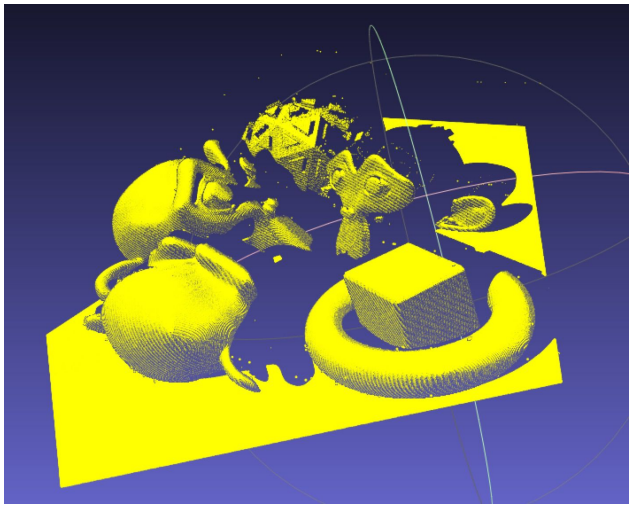


Camera Perspective

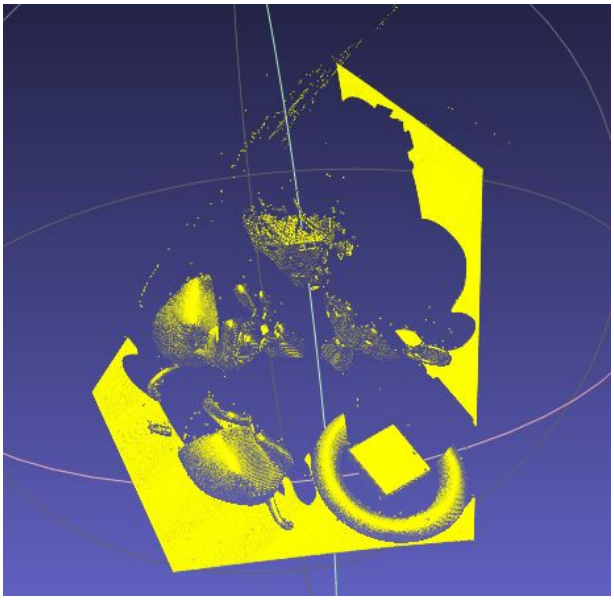


Side Perspective

Red_T1:

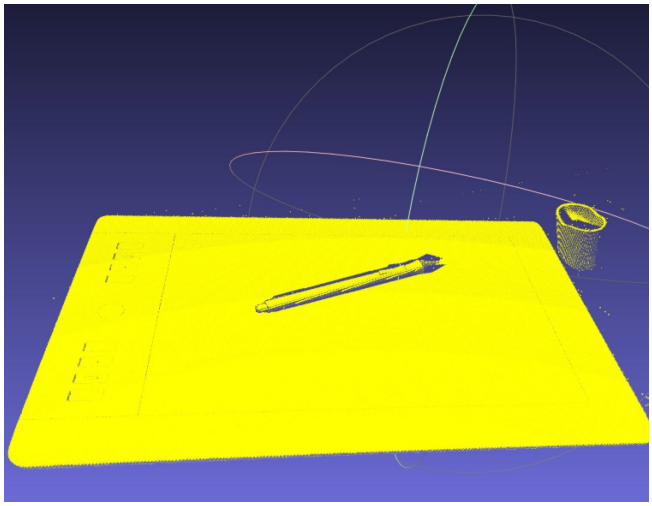


Camera Perspective

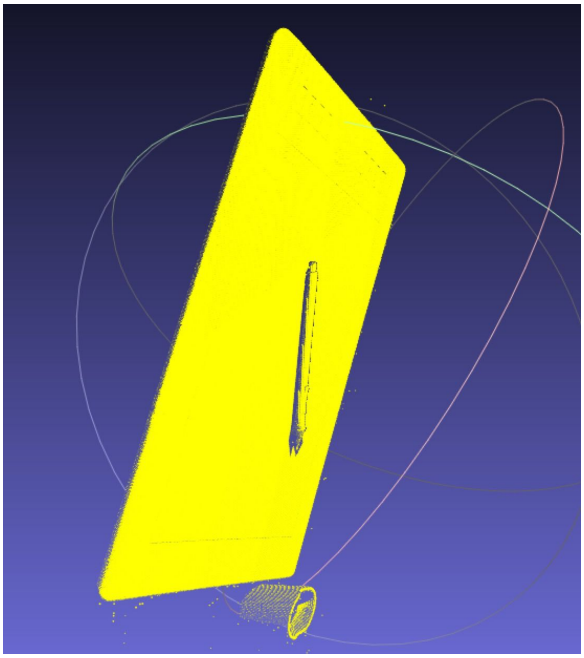


Above Perspective

Tablet_T1:



Camera Perspective



Side Perspective

The quality of the point cloud in general is very good, for some complicated objects like in ‘Red_T1’ it can still obtain the depth of most objects, although part of objects that is covered by the shadow other object in front of it are eliminated in the result. It’s able to recover some very fine depth change in the image, like the buttons in the ‘Tablet_T1’ and the bookmark in ‘Notebook_T’. We can still see some noise remaining in the reconstruction, and as the scenes

gets more complex the noise trends to increases as well. This might be caused by many reason like shadow casting, transparency and the material of the object. But in general the result are very convincing and the quality of the mesh we obtained is very good.

Camera Calibration:

In order to estimate the projection matrix of the camera and the projector, we need to firstly figure out the intrinsic and extrinsic matrix of the camera using the toolbox. As the image we're provided are in the camera's image plane, so this estimation can be done by tracking the deformation of printed pattern in the image. But the extrinsic and intrinsic matrix of the projector can not be estimate on the raw image as they are not in the projector's image plane. So we need a way to transform our raw image in camera image plane to projector's image plane. My strategy is to use the conners coordinates of projected pattern with the given coordinate of the conners to compute a homography matrix. Then by applying this transformation to the raw image we're able to get a new image that represents the view of the projector.

I've reused the code for calculating best homography from Machine Vision lab to do this task, this part of the code can be found in 'reproject_pattern.m'. The result of reprojecting the pattern is shown below:

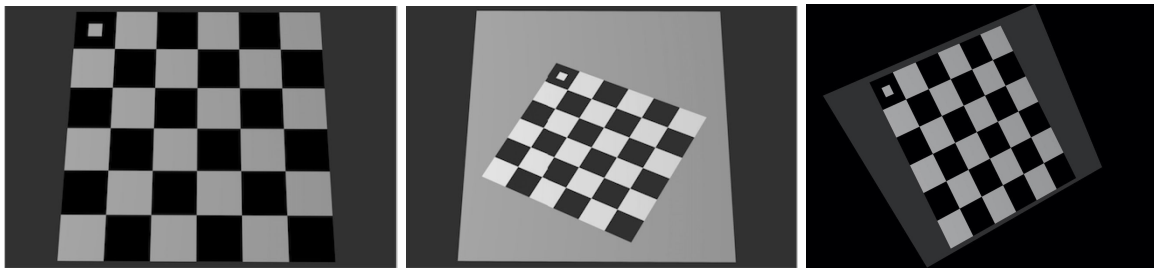


Figure.3 Calibration Steps
(Left to right: Printed pattern in Camera image plane;
Projected pattern in camera image plane;
Reprojected result that shows the printed pattern in projector's image plane)

By repeating this step we are able to get a set of images in the projector's image plane, we can then uses the tool box to estimate the intrinsic and extrinsic matrix of the projector.

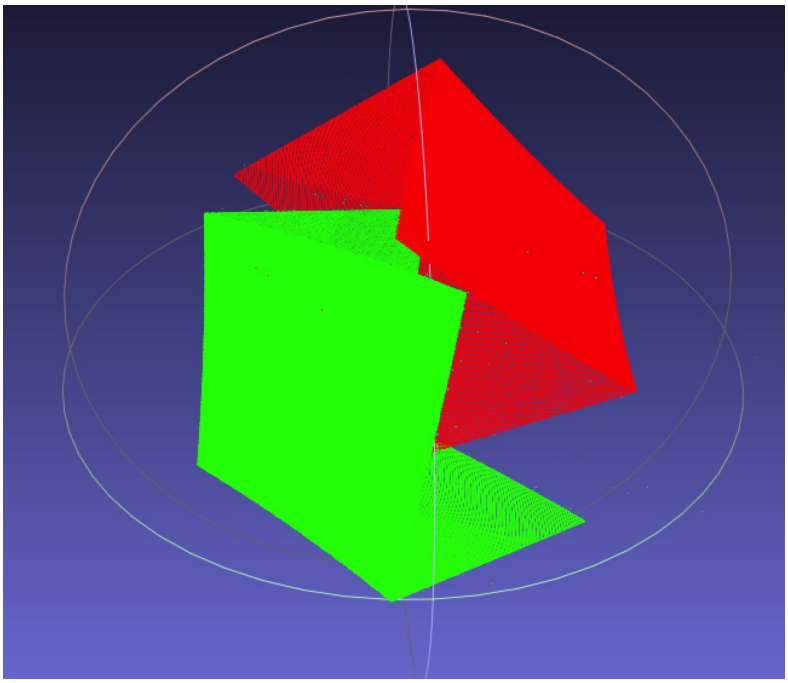
estimate_synthetic_cube.ply

```
ply
format ascii 1.0
element vertex 2146842
property float x
property float y
property float z
end_header
344.492617 930.382482 -71.095126
344.290196 931.078246 -71.775148
344.087747 931.774232 -72.455287
343.885271 932.470438 -73.135541
343.682767 933.166866 -73.815911
345.394281 933.847894 -70.458687
345.192118 934.543959 -71.138185
344.507048 922.832347 -72.564204
344.304511 923.527196 -73.244738
346.012635 924.217485 -69.896111
345.810439 924.911976 -70.575774
345.608215 925.606688 -71.255554
```

given_synthetic_cube.ply

```
ply
format ascii 1.0
element vertex 2146842
property float x
property float y
property float z
end_header
0.119381 0.047161 0.007665
0.119470 0.047140 0.007750
0.119558 0.047119 0.007835
0.119646 0.047099 0.007920
0.119735 0.047078 0.008005
0.119821 0.047283 0.007550
0.119909 0.047262 0.007635
0.118409 0.047165 0.007887
0.118498 0.047144 0.007972
0.118585 0.047349 0.007517
0.118673 0.047328 0.007602
0.118761 0.047307 0.007687
0.118850 0.047286 0.007772
0.118938 0.047266 0.007857
0.119026 0.047245 0.007942
0.119240 0.047225 0.007606
```

The images above shows that the point cloud I've obtained using my own calibration have larger scale than the given calibration. So when I put them both in the meshlab it's hard to visualize them together. I've rescale them to the same size and the result is below:

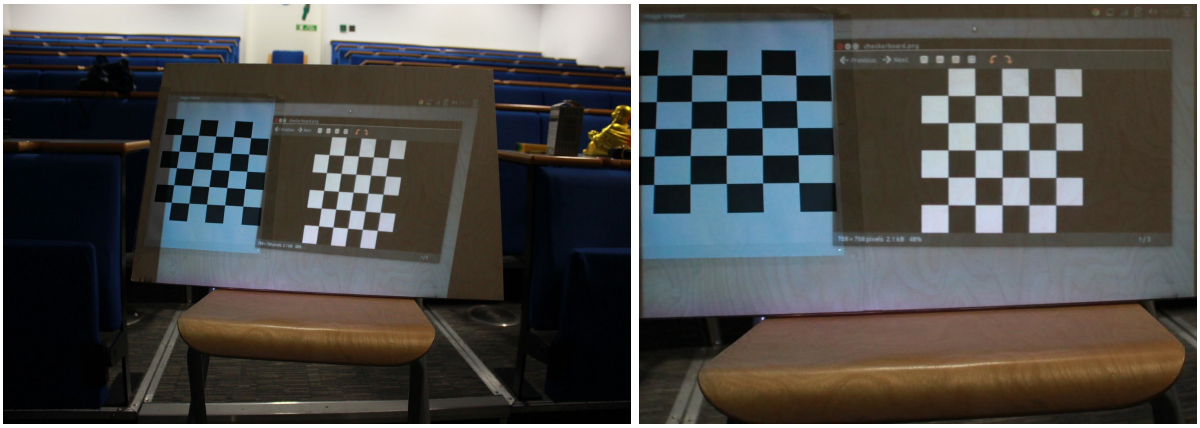


The green point cloud is the estimate calibration result and the red point cloud is the given calibration result. You can see the global rotation are different but the depth are similar.

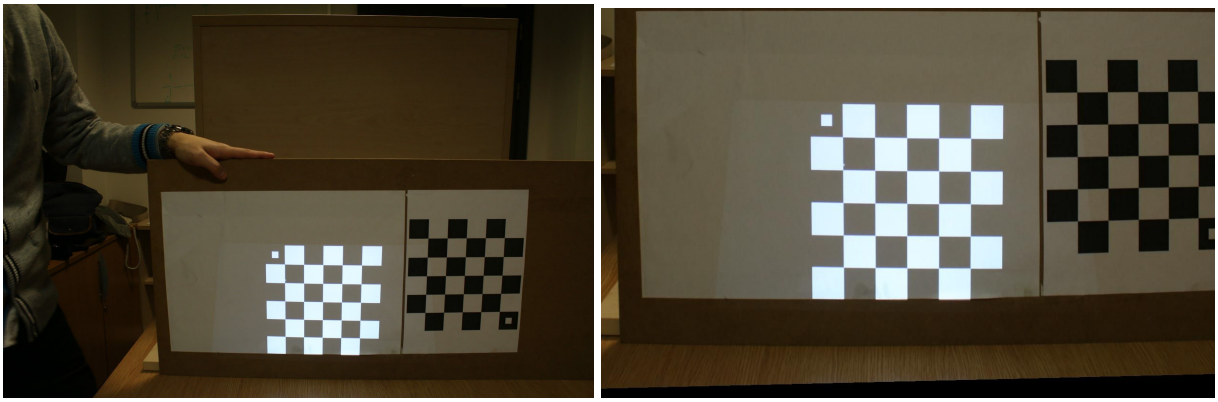
As there are lots of human effort of plotting connors in finding the homography, extrinsic and intrinsic matrix. So the resulting calibration we have will contains lots of error compare to the given calibration. This causes the global rotationation and scaling of the point cloud different to the given calibration.

Real Data and my own data

I tried to apply the same routine on the real and my own dataset images but the outcomes are not as expected. I tried to eliminates the high frequency by only using the low frequency pattern and smooth the uv code using a 1d median filter as the projector resolution can not present these high frequency pattern very well. I reproject the pattern using the same technique as I've used aboved and here are some example of that:

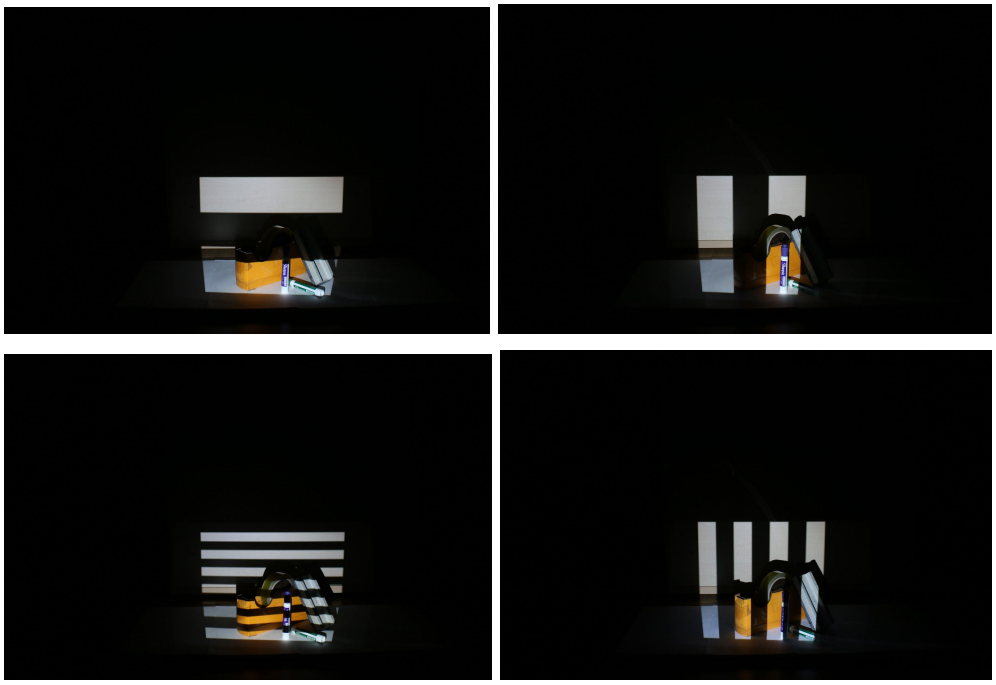


Reprojecting the **real calibration**.

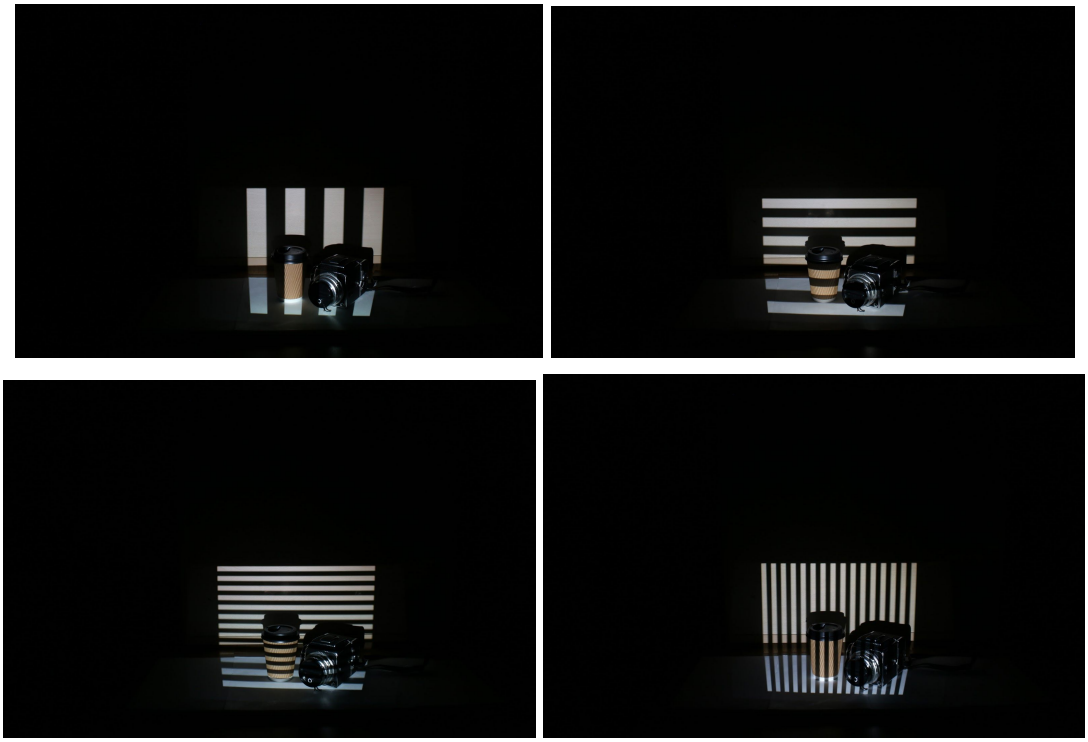


Reprojecting our **own calibration**.

I have capture two set of data with George parperis and here are some sample of it:



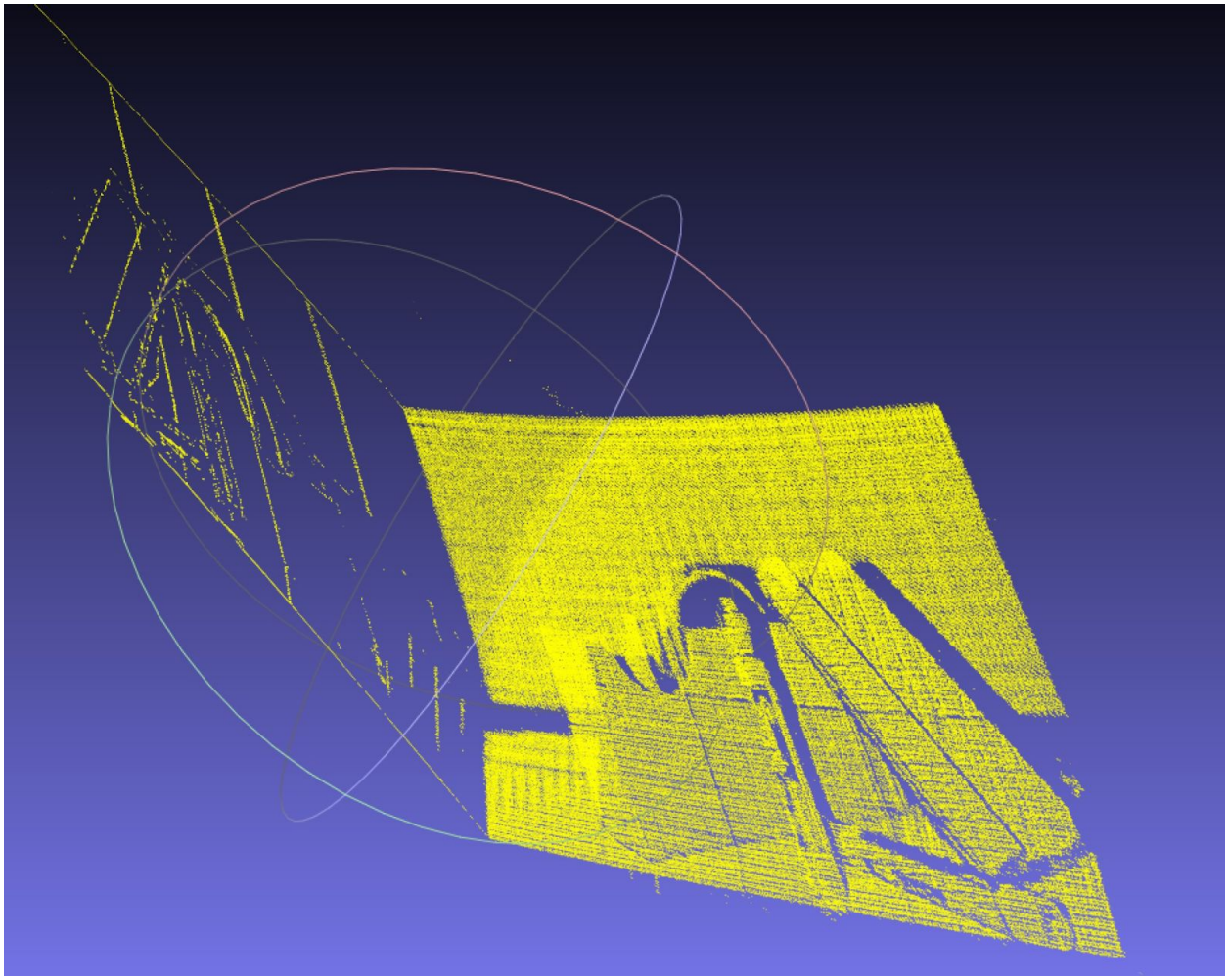
Set 1: CTape, eraser and marker



Set 2: Coffee and camera

I have try the algorithm on my own data and the best point cloud I've get is shown below:

You can see lots of noise in the background and the depth in general is deformed, I think the problem is in the calibration step or reprojecting. As the extrinsic matrix I obtained from the reproject image has very large translation.



As the data is too large, I have put it the point cloud I Generated, together with my own data on dropbox.
Dropbox link is:
https://www.dropbox.com/sh/dh8h14gkwfs9hcy/AACGkJn7aQtHNTsHLkF5h_qKa?dl=0