Maud Buffier

Computational Photography Class Project

Structured Light Depth Acquisition

In the folder with the code you have several folders :

- *calibration, which contains Real_Calib and Synthetic_Calib. In each of them we can find the function I used to compute the projected images for each data set, the images, a file with my calibration matrices and the 2 .mat of the toolbox for the projector and the camera.*
- *Reconstruction which contains the code to find the uv coordinates (findPixelsCode.m), the code to generate the 3D reconstruction (reconstruction3D.m) and the code to save the mesh point in a .ply (createPLY.m). You can choose using reconstruction3D.m which calibration do you want to use.*
- *pointCloud in which the point cloud are saved after being computed. However, there were too heavy so I haven't include them in the folder but I put screenshot in my report*
- *uvCode in which there are the UV codes for each 3D point cloud in this report*
- *data_sets in which the data needs to be to be generated from reconstruction3D.m*

Part 1 : Synthetic set of datas

A) 3D reconstruction

Decode the light pattern

In order to decode the light pattern, I used the images provided for each data set (both the illuminations and their inverses). My code to perform that is in 'reconstruction/ findPixelCode' and the results are in the 'uvCode' folder.
First of all, I loaded the images and put them in gray-scale in order to be able to compute the code more easily. Then, for each pixel, I wanted to compute 2 informations (u and v) which encode directly the position of the pixel in the gray-code pattern. To find this u,v coordinates, I computed 2 binary vectors which encode for a particular position in the image. The code is given by the frequency where the pixel is turned on or off across 10 images for u and 10 images for v. Then, I converted those 10*1 binary vectors into a unsigned int position in x or y axes using bi2de(). This position is unique in the gray-pattern.

To be sure if the pixel in on or off in the image, I looked at its intensity with respect to an other image lighted inversely. At the same position for a given gray-code pattern, if the intensity in the first image is smaller than the intensity at the same position for the same gray-code inverted, then the pixel is « off ». And if it's higher, the pixel is « on ». Computing the intensity information relatively to the inverted pattern avoids making errors in places where the albedo of the pixel is low for exemple. The entire method is described in the article provided.

Hence, once I know for each position the good alternance of 1 or 0 (on or off) horizontally and vertically and I can convert those code into a position and I obtained a correspondance of pixels between the camera image plane and the projector image plane.

Eliminate pixels

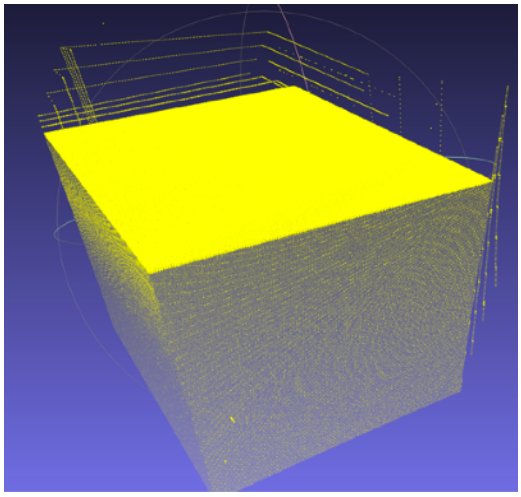
To avoid artifacts, we need to be sure that the code we computed for the pixel is relevant. Otherwise, the pixel could be encoded with a completely wrong position.

To do that, we are going to compute the difference of intensities between the current pattern and the inverted one. By summing this differences across the 10 horizontal and the 10 vertical patterns, we'll obtain a measurement of incertitude about the code of this pixel. Indeed, if the difference is really small, it means that we are not able to tell for sure that a pixel is on and off and we can't be certain about its code. Hence, if the value is below a certain threshold (set to 32 in the article), the pixel is declared 'unsure' and we don't set its u,v position to zeros.

This code is integrated to the 'reconstruction/findPixelCode' function.

Determine the unique depth

Now that we know where a particular pixel of the camera image plane is in the projector image plane, we can use this information to compute an unique depth for each 3D points. However, it's important to notice that the resolutions of the projector and the camera aren't similar. Hence, there is not enough uv code in the gray-code pattern of the projector to describe an unique code value for each pixel. Hence, we can interpolate between the integer codes using a 1D kernel of a window of 7 pixels to obtain a unique value for each pixel. We need to be careful to smooth in the prominent code direction (as u encodes the row in my code, I smoothed u vertically and as v encodes the column information, I smoothed horizontally).
However, smoothing in the case of the synthetic data produces lots of artifacts : as the background was all black, when smoothing, it creates extra boundaries like that :



Hence, I decided not to smooth for the synthetic data in order to obtain better results.
Once I have a position for each pixel I can compute the position of the 3D points w using this equation :

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

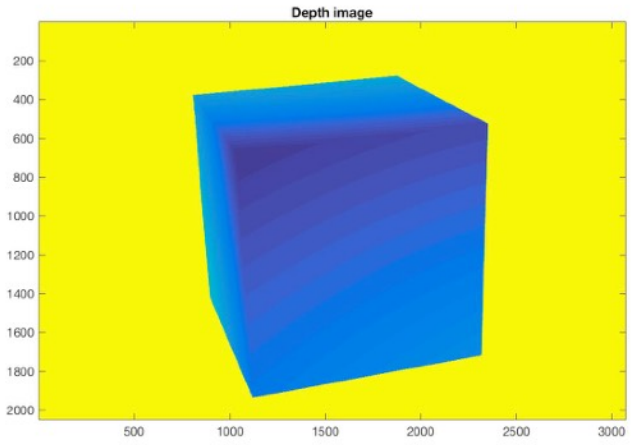
As it doesn't have any close form solution, we can rearrange the equation to find something of the form Ax = b and solve it using least square estimation (see the Simon Prince algorithm).

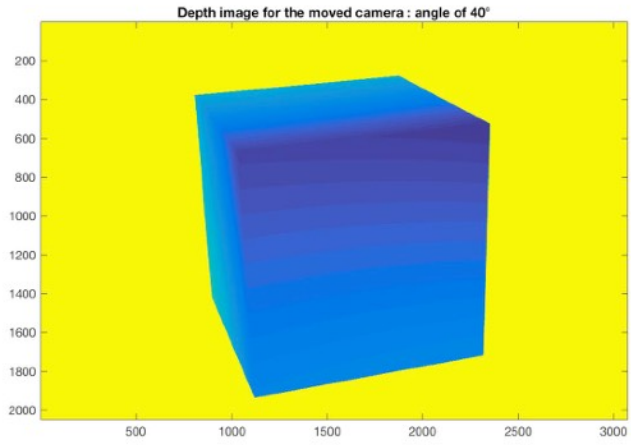
Then, I obtain a 3D point cloud and I can re-project points into camera frame (using the camera extrinsic matrix) in order to obtain the depth of each point with respect to the camera.
If I want to observe the scene with a different point of view, we can re-project in a different frame (the frame for a different point of view of the camera for exemple but multiply the extrinsic matrix by a rotation matrix for example). And then, the depth of the point will be modified and will appear with respect to the new position of the camera.

Results for the data set provided

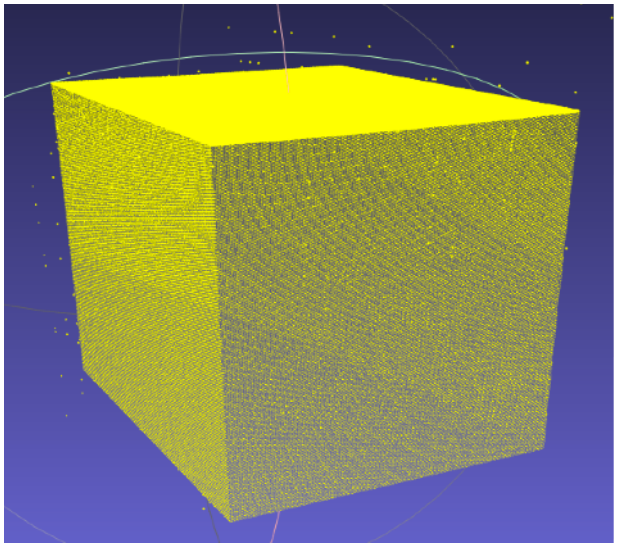
To run the code, use the file 'reconstruction3D(floderName, imageName, first, digit, format,rgb)', the folder name needs to be indicated with the path, the image name, the number of the first images and the typeCalib for this part is 1 (provided calibration)

For the cube :





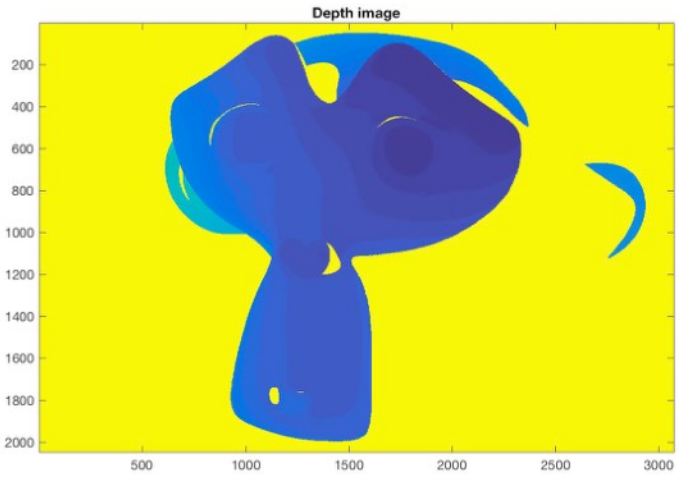
We can see that for the depth image for the moved camera, the points at the right of the cube are closer (darker) as if the camera would have rotate around the vertical axis.
Here is the 3D cloud visualized in meshLab :

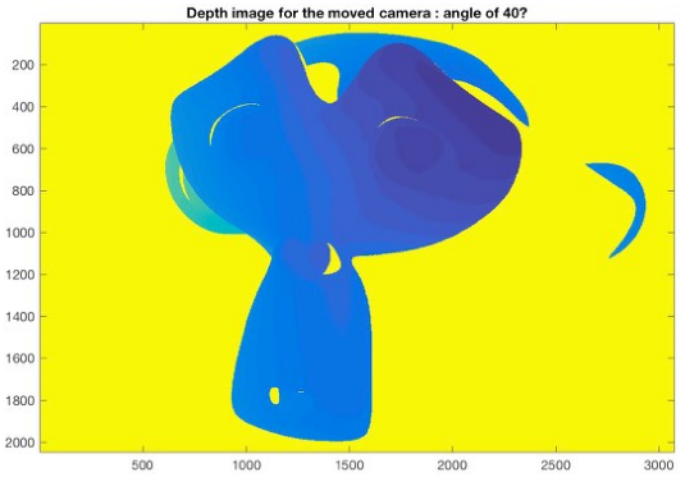


The cube is the easiest object to reconstruct as it's flat and entirely visible. Let's test the other one !

For the monkey :

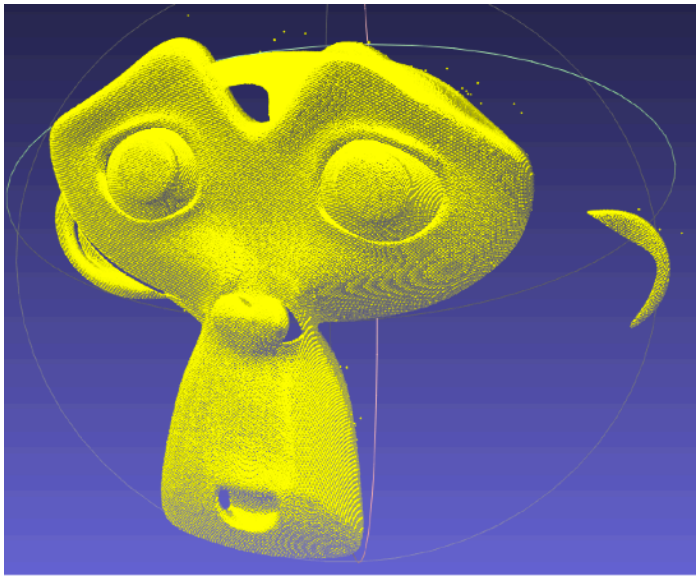
Here are the depth maps for the monkey,





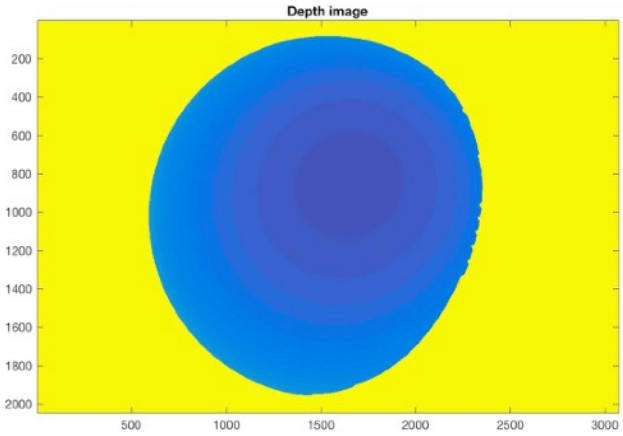
We can't see all the variations of depth really well because some artifacts point have a important depth so the colors are stretch. However, we can still distinguish the eyes of the monkey.

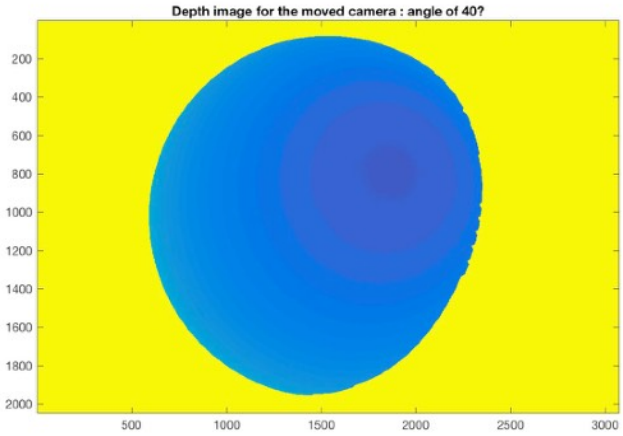
We can visualize it even better using meshlab :



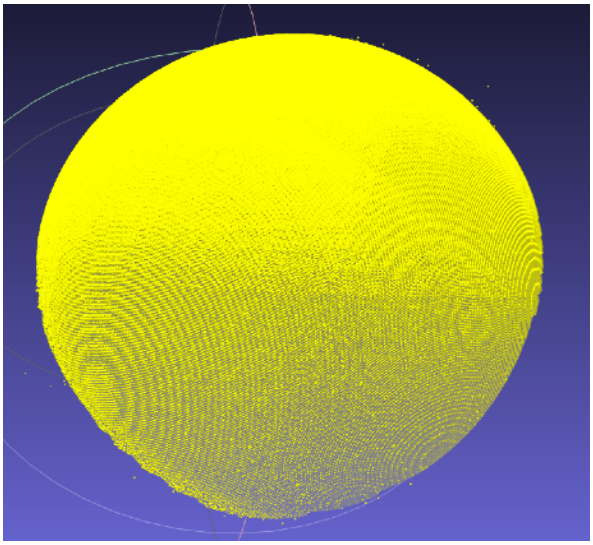
The concavities are well reproduced, even if there are still artefact points at the border due to points with unclear u,v code. As for the cube, I didn't blur the u,v code to avoid seeing artifacts.

For the sphere :



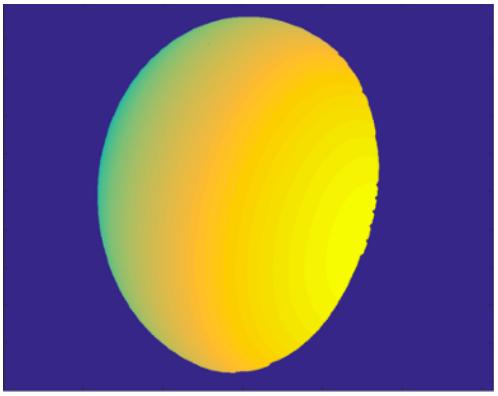


We can see the effect of the moving camera on those depth images very well.
and here is the resulting mesh on meshlab :



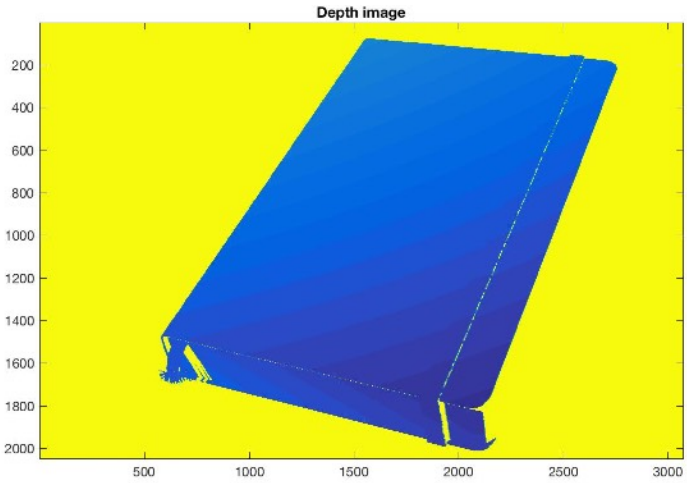
We can also observe the (u,v) code very distinctively on the sphere :

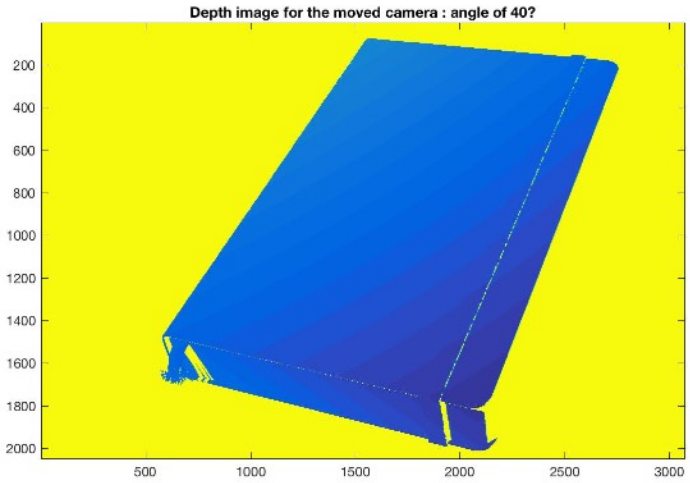
U code (for the rows)



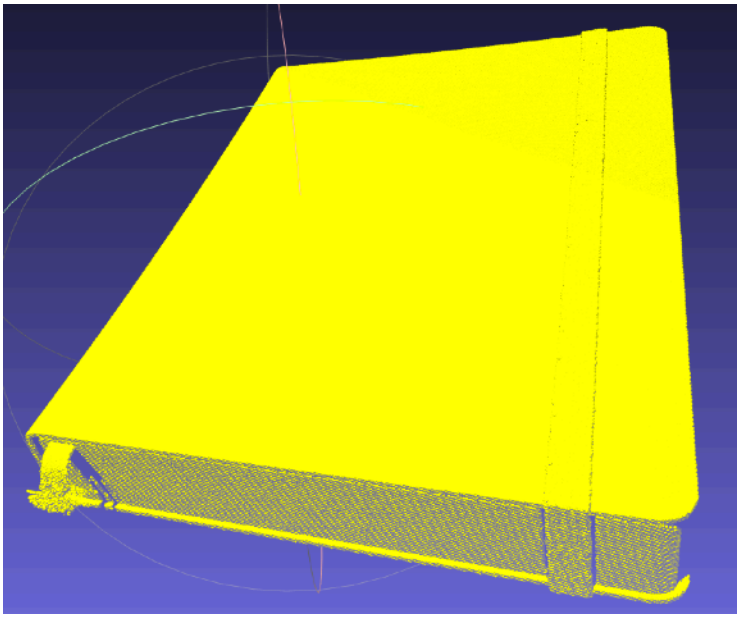
V code (for the colums)

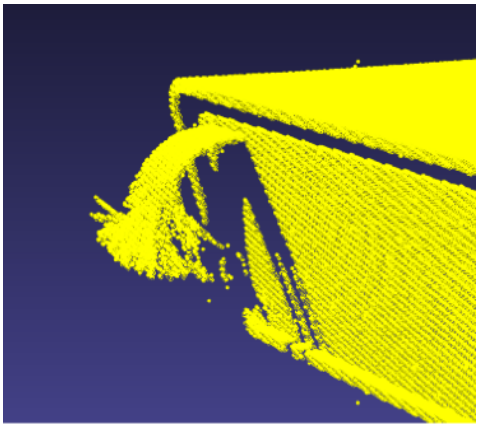
For the notebook :

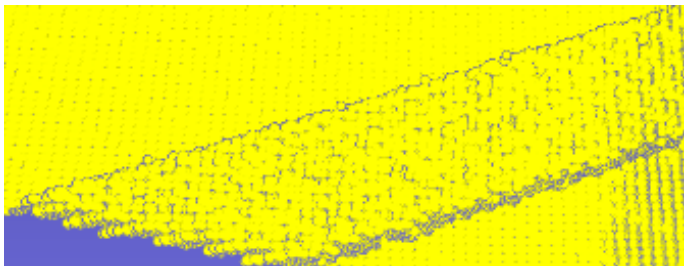




Details at the bottom are visible in the depth maps and in the resulting mesh :



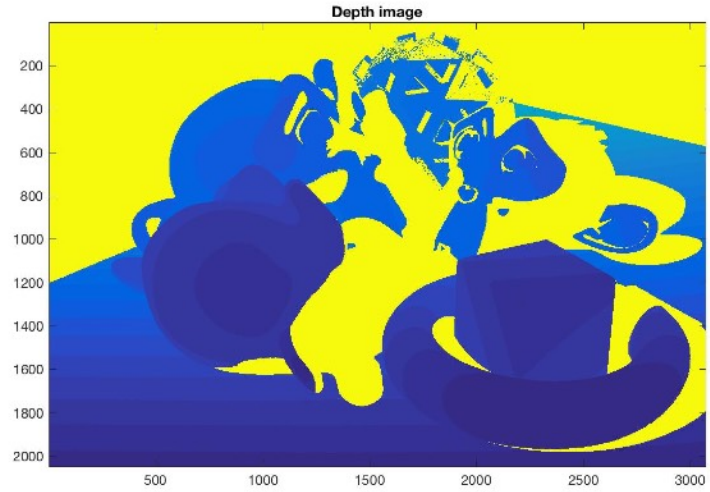


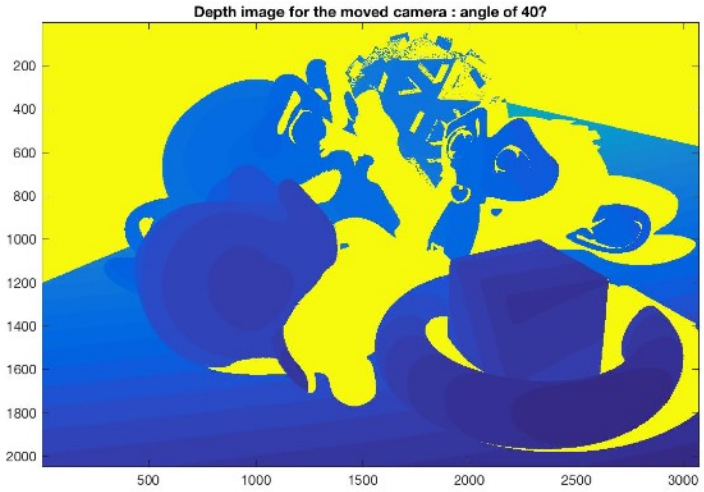


The texture of the band is even reconstructed a little bit (we dont have the nice stripe however). The fine hair gives a good result as well.
However, I think there is a weird artifact : it could look that the book isn't flat in the mesh. It could maybe be explained by perspective effect due to the position of the book.

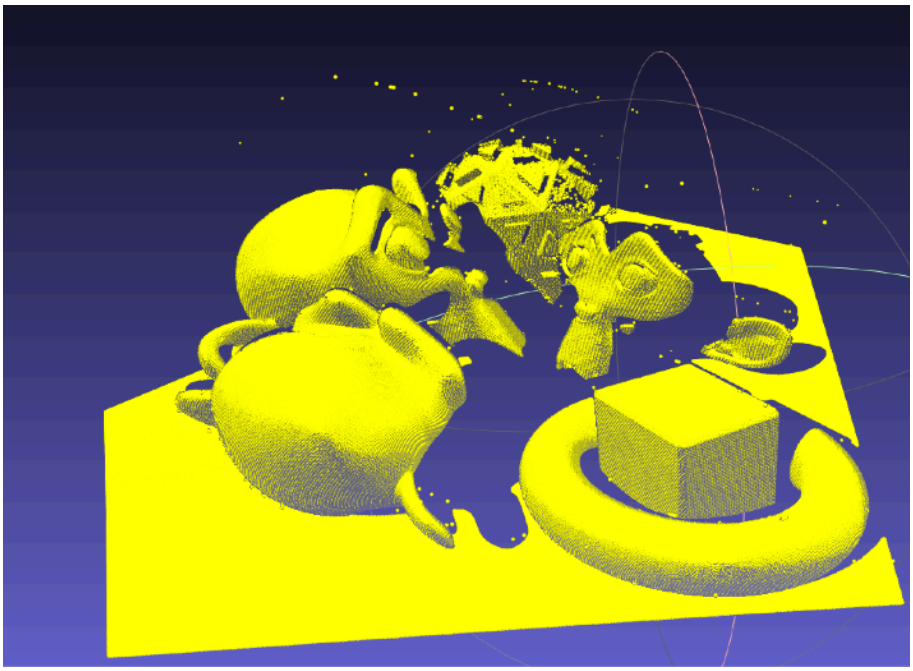
For the red scene :

This scene contains lots of occlusion and even shiny red materials for the monkeys. Moreover, the object in the top is hard to reconstruct because it is partially in the dark.
Hence, lots of pixels are declared « unsure » as you can see in the depth maps :



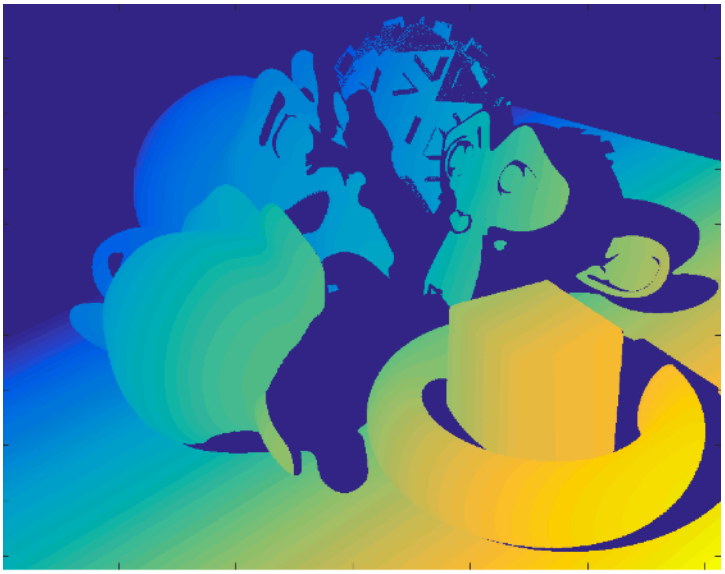


Hence, it leads to some artifact points in the resulting 3D scene :



The top object gives lots of noise. However, the algorithm recovers correctly from the shiny surfaces of the monkeys.
Moreover, we can also visualize the (u,v) code the scene for exemple :

U code

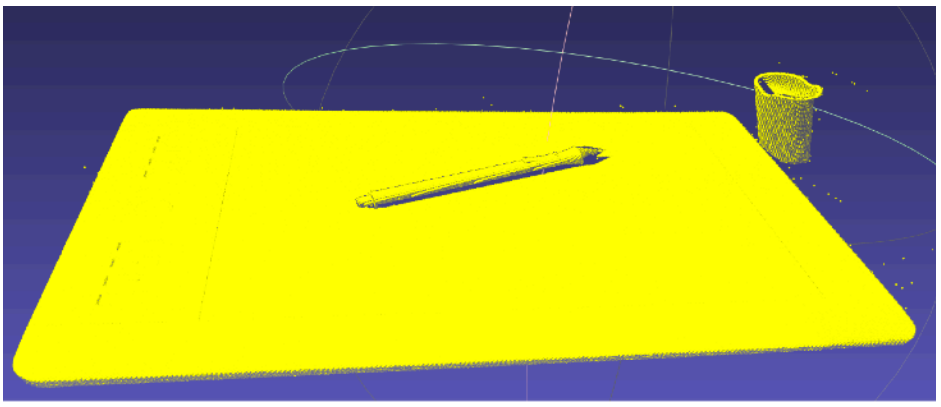


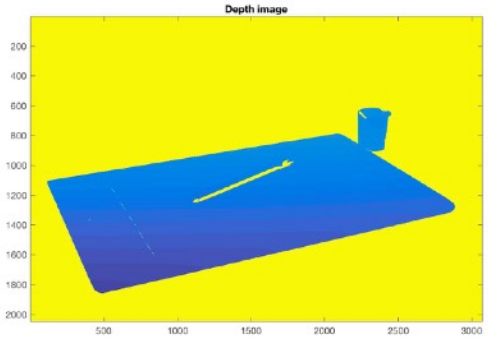
V code

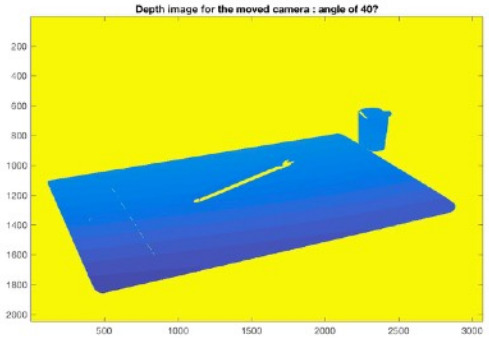
We can see the u code encodes for the rows and the v code for the columns.

For the tablet :

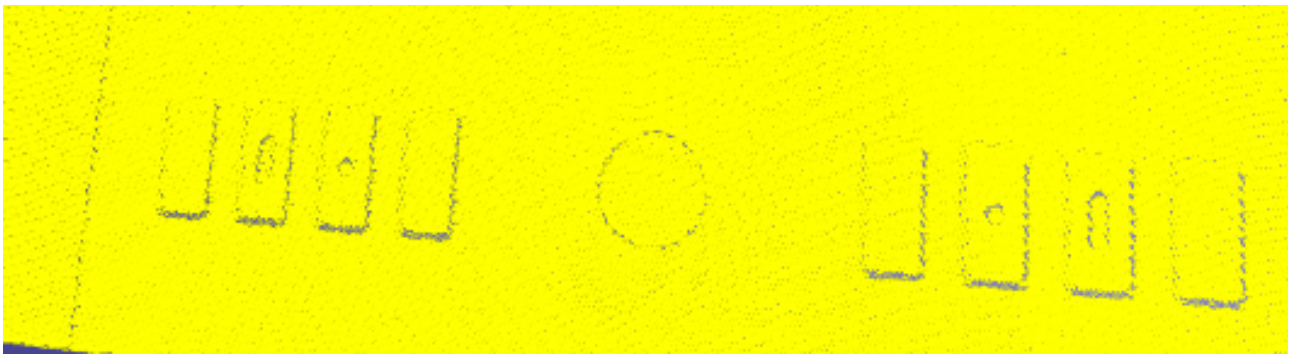
Even if the depth map doesn't show differences in the depth sufficiently, by looking at the mesh :







We can see that the details of the tablet (like the buttons on the left side that we can see here :



Conclusion :

Some artifact appears if the images are complex (some part of the object in the dark, occlusion, fine texture) but the algorithm finds its way generally and produces a convincing 3D mesh for the synthetic data.
Putting the images in gray-scale and using both the pattern and its inverse to compute the code allows to obtain good approximation for the gray code, to reject unsure points and then, being able to reconstruct the scene.

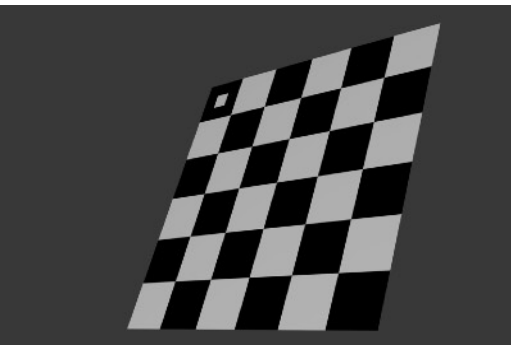
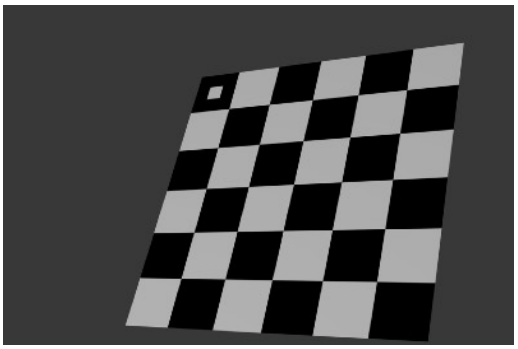
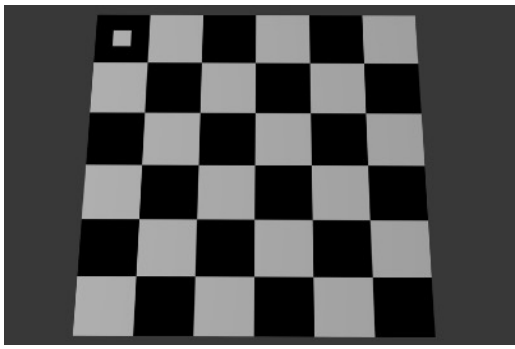
B) Camera calibration

Determination of the projection matrices

To determine the projection matrices we need both to obtain the intrinsic and extrinsic matrices. For both the camera and the projector, to compute the intrinsic matrix, we are going to use a set of images of a known chequerboard in different planes seen by the camera or the projector respectively. This set of images is provided directly for the camera but we need to obtain it by projection for the projector. Indeed, we can't have images of what the projector sees directly as it's a projector and not a camera.
All the code for this part is in the « calibration » folder.

Intrinsic for the camera :

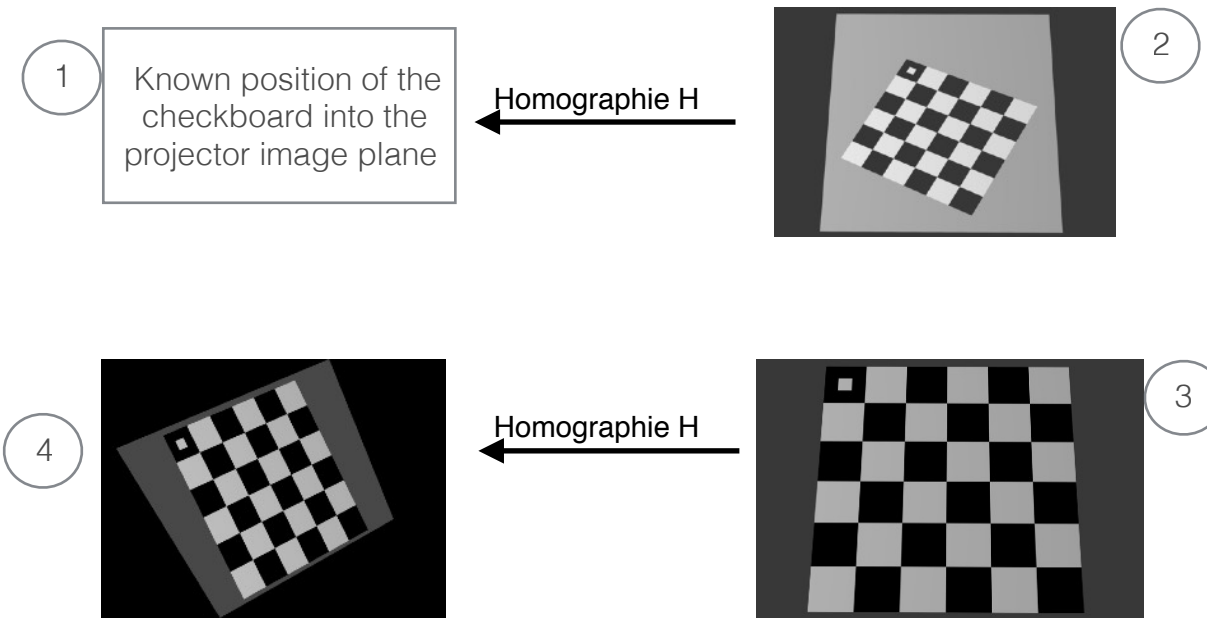
We are going to use this set the images to compute the intrinsic matrix using the toolbox :

The result is in the file 'foundCalib'

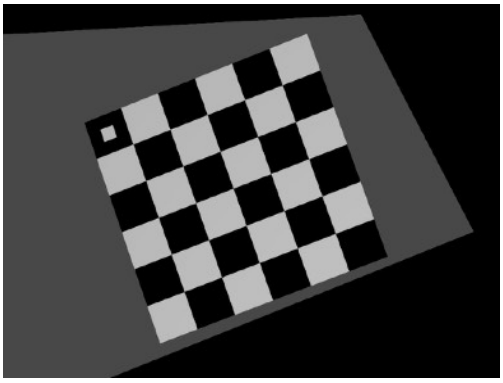
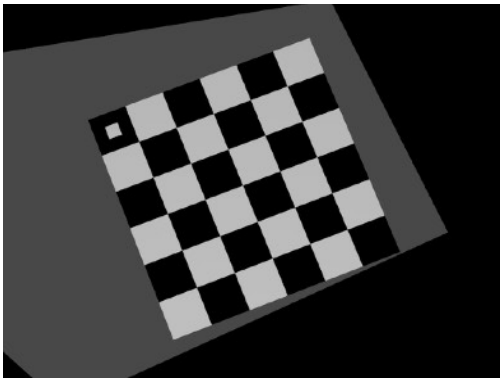
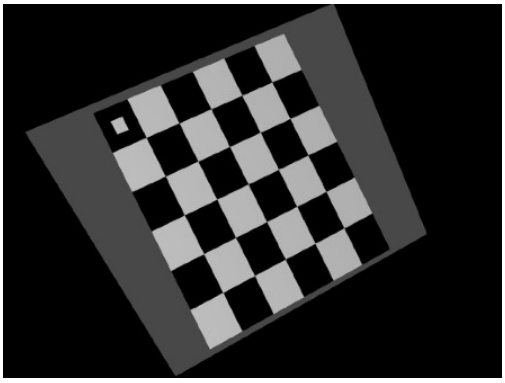
Intrinsic for the projector :

As I said, we need to have images of the chequerboard in different planes seen by the projector :



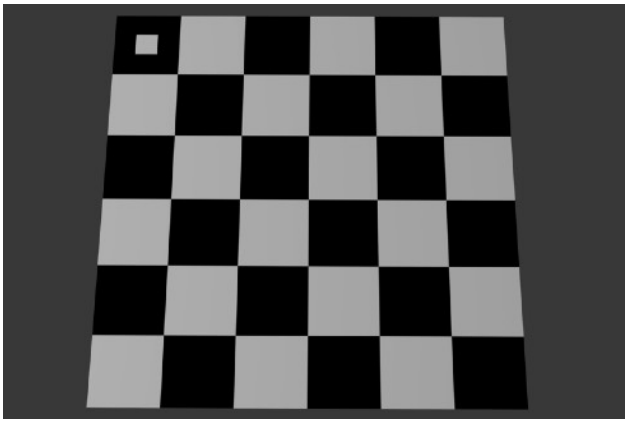
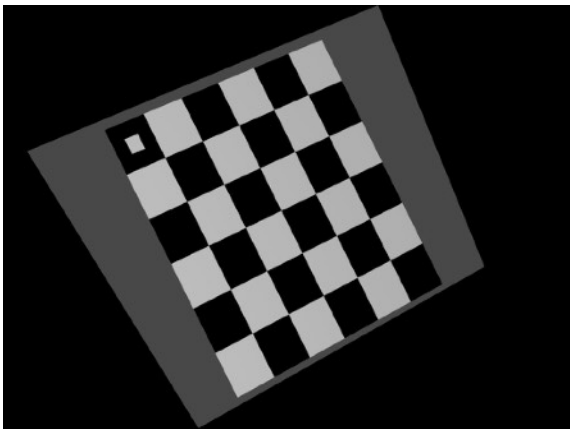
The image 1 is given as we know the position of the chequerboard in the image plane of the projector. Then, we also know what the projection looks like seen by the camera (image 2). Then, we can compute an homography between the projection seen by the camera and the image plane of the projector by taking the 4 corners of the chequerboard for example.
Then, we also have an image of the chequerboard seen by the camera in a plane (3). Hence, we can apply the found homography to find the projection image of this chequerboard into the image plane of the projector.
My code to do this is in the « calibration » folder, « projectorCalibration.m »

Hence, we have a set of 3 images we can use to compute the intrinsic matrix for the projector :

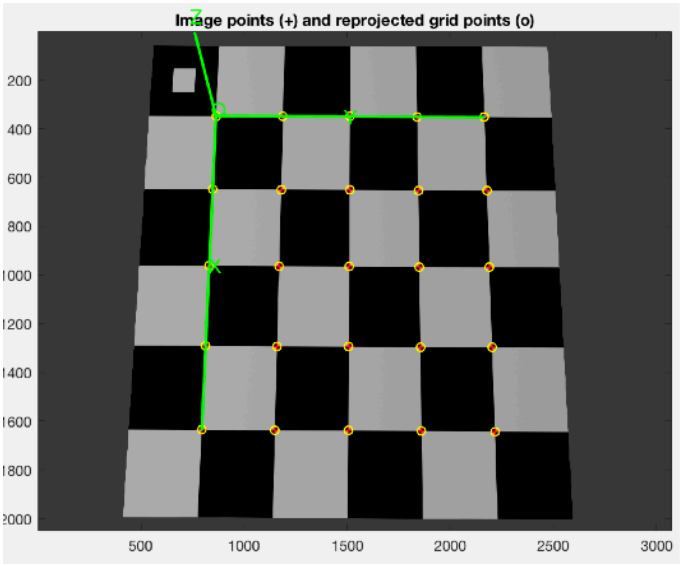
Extrinsic for the camera and the projector :

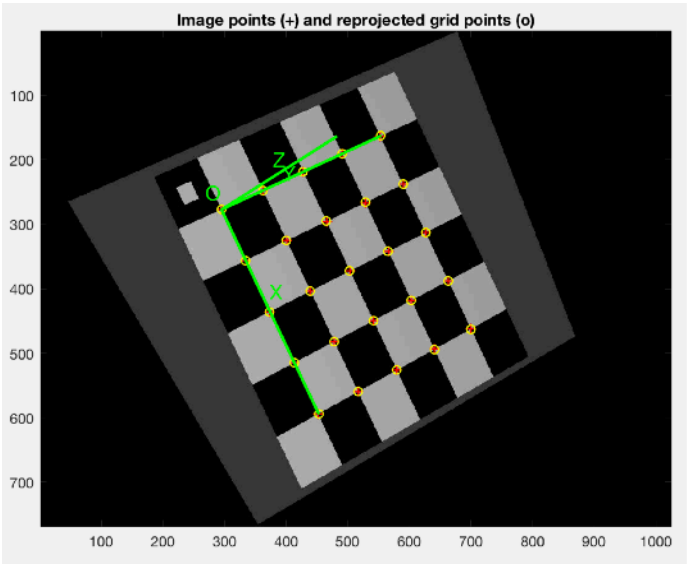
In order to compute the extrinsic matrix for the projector and the camera, we need to have a view of the same chequerboard from the camera point of view and from the projector point of view. As it's what we have compute previously we can for exemple use this pair :

And compute the extrinsic with the toolbox (using calib_gui and then, comp. extrinsic) by being careful to use the good intrinsic matrix for the projector and for the camera.

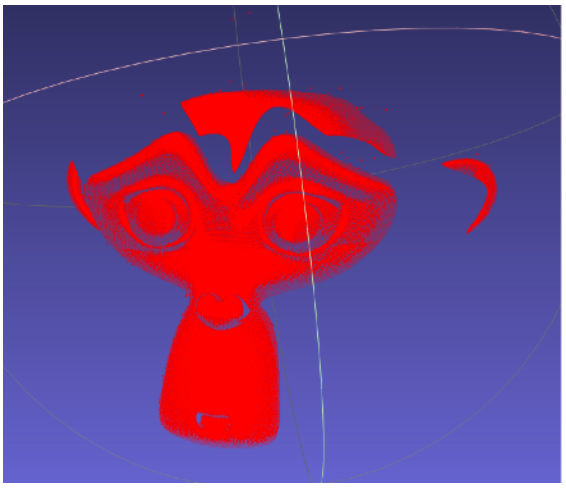
Hence, the frame in the 3D world for the rest of the computation will be this chequerboard. Here are the type of frame I obtained to describe my 3D world



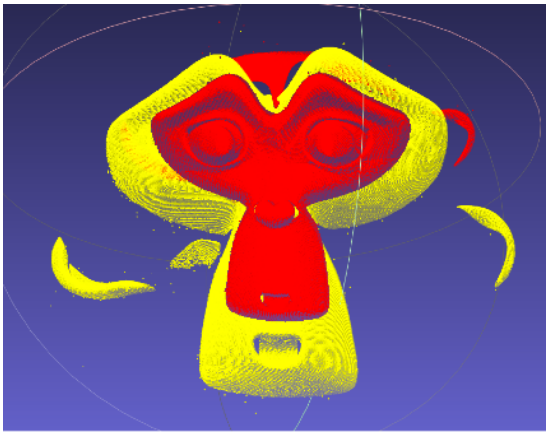


At the end, I obtained extrinsic matrices with very large values for the translation.

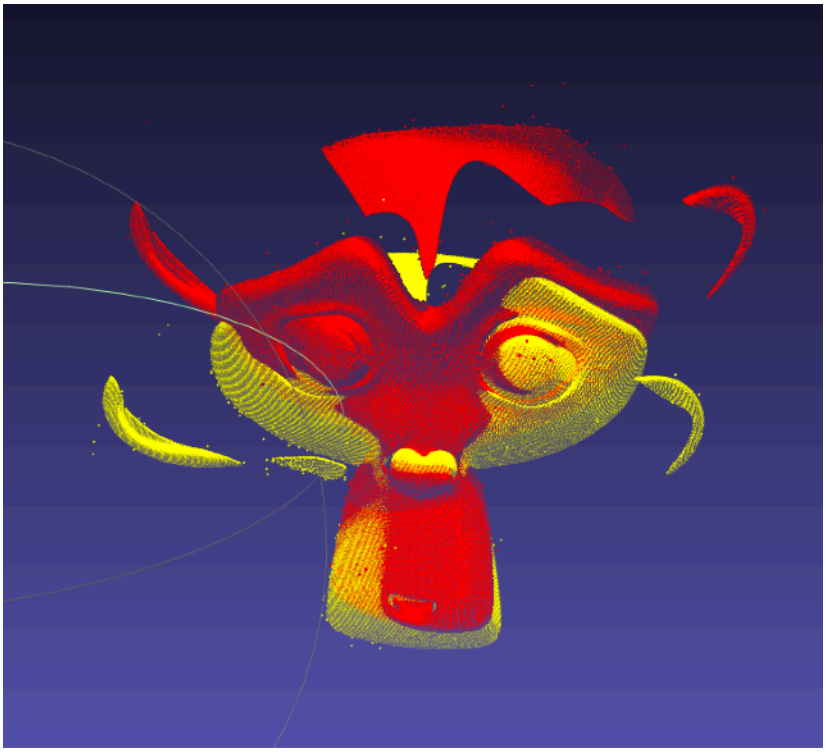
Hence, my mesh cloud seems correct but not at the same scale as the result using the provided matrices.



After rescaling the meshes in a unit box using meshlab, here is my result :



After rescaling again and merging the meshes as much as possible, here is what I obtained :



Hence, we can see that even if the meshes seems similar there aren't exactly. The red one (obtained using my matrices) seems at bit thinner than the yellow one. Moreover it is not exactly in the same plan : as the bottom is approximatively aligned, the tops of the heads diverge.

First of all, the differences in scale are explained by the fact that we haven't taken the same origin for the world : indeed, I took the first pair of images to define the world but I could have taken an other pair of images as well. Hence, my extrinsic matrices wouldn't have been similar as the origine of the 3D world would have differed. However, once rescale, if my matrices were perfect, I thing I should obtain the same mesh.

Those differences come from errors in both the intrinsic and extrinsic matrices. I computed my extrinsic matrices using 3 images by extracting the grid of cornes. However, it's not very precise and, as I only have 3 images, outliers aren't «washed out » enough. Moreover, the same imprecision occurs when I compute the extrinsic matrix using the toolbox.

To finish, as we were working with a projector and a camera, we needed to infer images seen from the projector point of view. Then, I had to compute a homography matrix by clicking on corner points which lead, once again, to imprecisions (even more importante as I used 'getline' which doesn't look for corners as the toolbox).

Then, the resulting mesh is a bit different because the matrices aren't perfect. It demonstrates the need to be precise in the way we compute the calibration matrices.

Part 2 : Real Data Provided

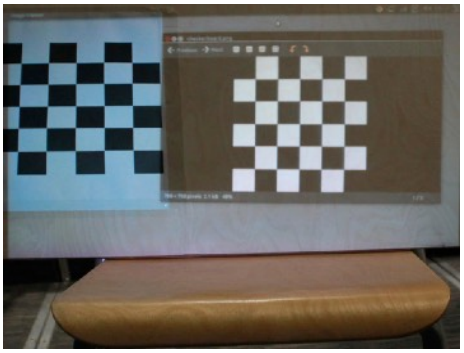
Decode the light patterns

In order to decode the light pattern, I used the same function as for the previous section. However, as loaded the matrix images all in once was too heavy, I computed the u and v code code separately before merging them to obtain an unique matrix with the codes for the scene.
Moreover, I decided to keep the matrices encoding for high frequencies as the file 'projectorcalibrationgridpositions.txt' says that « Both projectors are 1024x768 ». This information seems to be inconsistent with the one provided in the readme.txt : «the projector only supports 800x600 » . Hence, I had to make a choice and I decided to focus on the first affirmation that the projector was 1024*768 resolution on the images (possible with a XGA connection in the documentation of the projector found on line).
In any cases, the last two images with high frequencies encode the last pixels informations so the error will « only be » of a few pixels and not much. If I only use 8 images for exemple to compute the position, I have only 255*255 choices for the gray-code which is very few compared to the resolution of the camera.

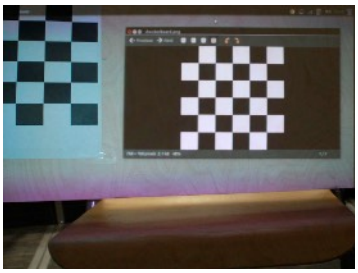
Determination of the projection matrices

To determine the projection matrices, I used the given data calibration images. As for the synthetic part we were provided with images of a projection of the chequerboard and a printed chequerboard in different planes like that :

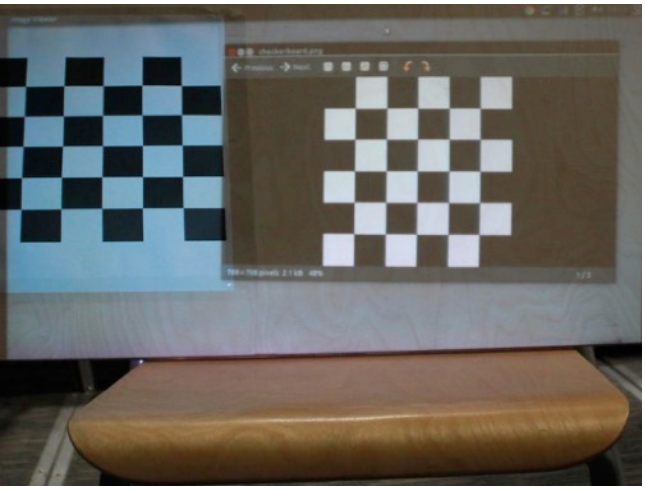
As before, as we know the position of the chequerboard in the image plane of the projector, we can infer an homographie transformation from the image plane of the projector to the world to the plane in the projection plane seen by the camera. Then, by applying this transformation to this image (more precisely to the printed chequerboard to the left) we obtain a printed chequerboard seen by the projector :



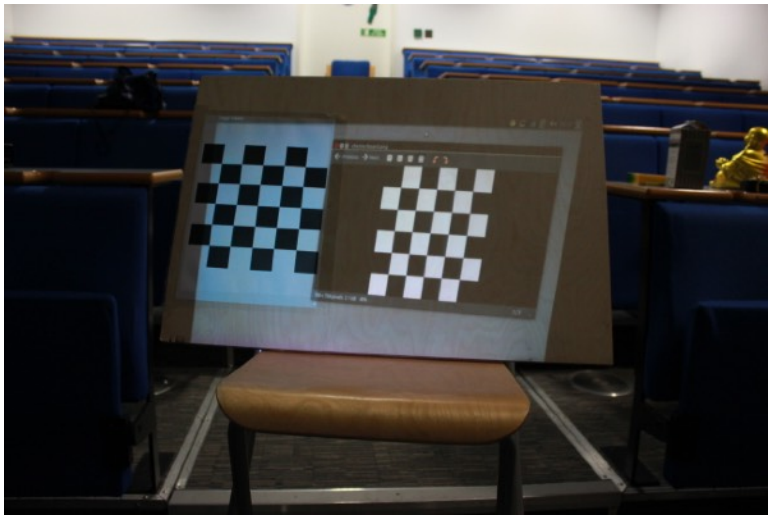
However, sometimes, using this projection, some part of the printed chequerboard are lost. Like here for example :



Hence, to compute the intrinsic matrix for the projector I used only the 3*3 pattern at the bottom right of the 9 images that I computed using the function 'projectorCalibration.m'. Then, to compute the intrinsic matrix for the camera, I used the 9 images provided with the printed chequerboard and I also only used 9 squares. I used arbitrarily 36 for the size of the square (in mm) both in the x and y direction. I think it's a scaling factor so my meshes will be a bit deformed as it's not the exact value (maybe the square are larger in y direction) but I think we can recover from that in meshlab and as I didn't know the exact values..
Then, I needed to find a world frame to compute the extrinsic matrix. I used the first images where almost all the chequerboard was visible,

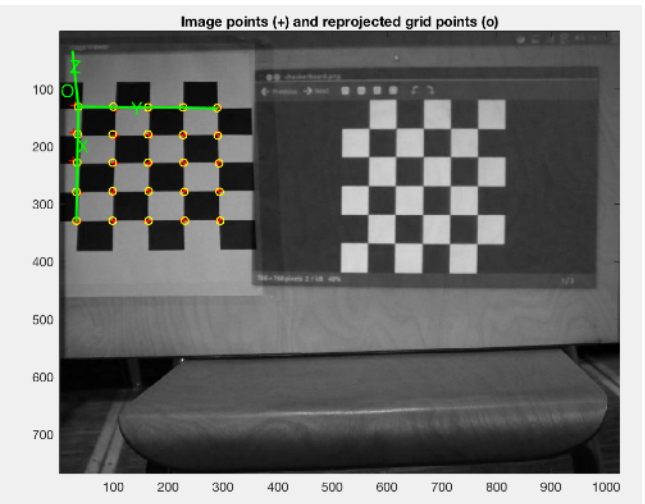


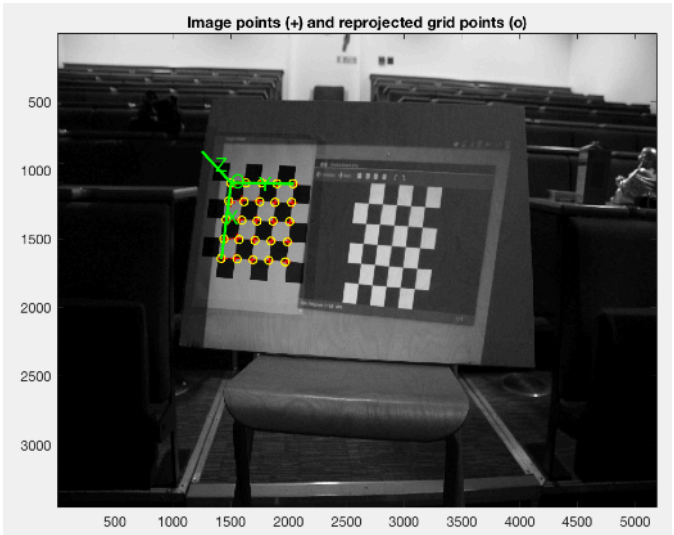
Seen from the projector



Seen from the camera

And by using the toolbox :





World Coordinates system from the projector

World Coordinates system from the camera

The matrices I obtained are in the ‘real_given_calib.m’ file.

Eliminate wrong pixels

In order to eliminate wrong pixel, as in the previous section, I computed for each pixel the sum of the difference between the intensity of this pixel for all the patterns and their inverses. If these sum is above a threshold, it means that the pixel is clear and if it’s below, we can eliminate the pixel because they aren’t enough differences between the lighting of the pixel, then we are not sure about its code. This step is done directly when computing the gray code in « findPixelCode.m ».
However, I needed the threshold to be very high to obtain good result. If it was too low, the depth map was computed correctly but the point cloud in meshlab were impossible to visualize. To find a good threshold, I computed the mean and variance of the difference data and I set the threshold to mean+var. This way, points to noisy are eliminated directly.

Determine the unique depth

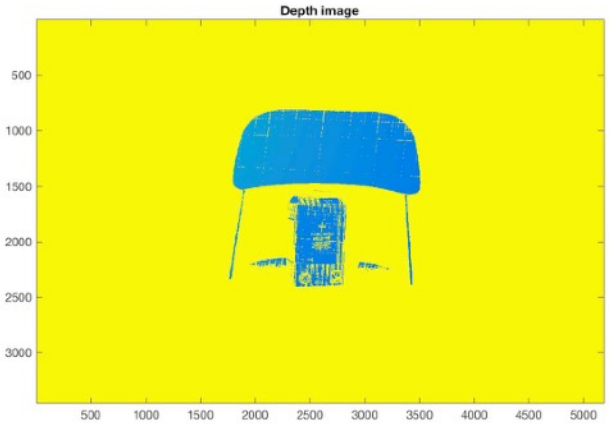
For this part, I used what I explained in the previous section which is a 1D smoothing using the med1filt function of matlab with an order of 5. I smoothed in the main direction of the code : vertically for the part encoding the rows and horizontally for the part encoding the columns.
Then, I can compute the position of the 3D point using the same method than before, by applying the algorithm of Simon Prince to find the point.
I computed the 3D point cloud and visualize them using meshlab.

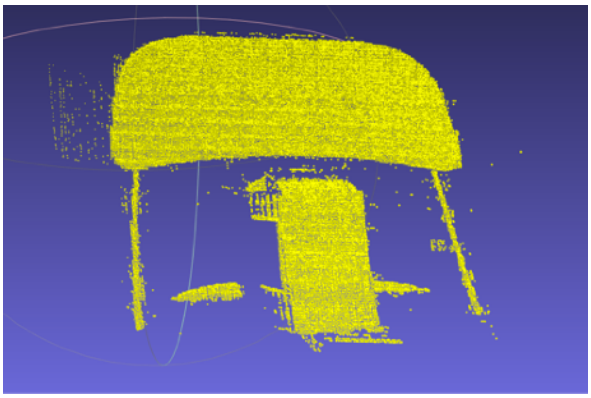
Results for the real data sets provided

I only compute the result for the tea bag and the crayons data sets as it took very long to compute (the images are very high resolution)

Result for the tea

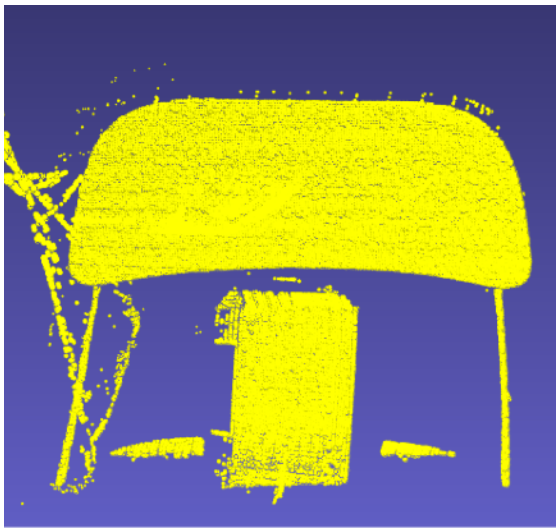
For the tea, here is the depth map and the point cloud I obtained :





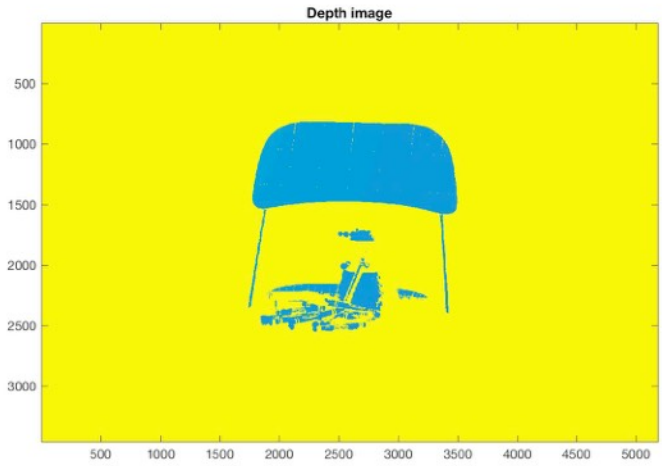
I had to put a very important threshold in order to be able to visualized the point cloud correctly. It explains why the chair isn't reconstruct completely.
However, as the box is quiet simple to reconstruct, putting a high threshold doesn't affect the reconstruction of the box.

Moreover, we can try to blur the uv code obtained (with a median filter of 5 in this case) to obtain a unique depth per point. This is the type of result we obtain, which seems more convincing :

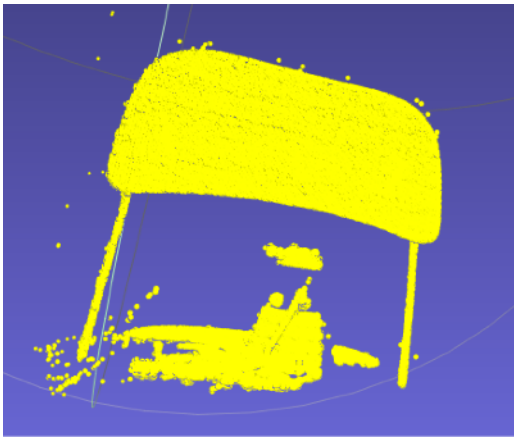


Result for the crayon dalek

Here is the depth image we obtain using a threshold of 4.

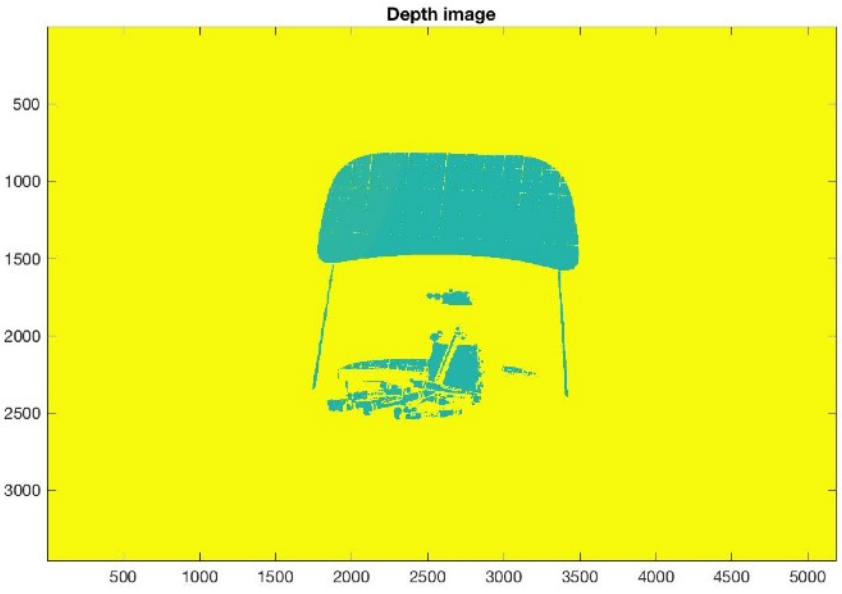


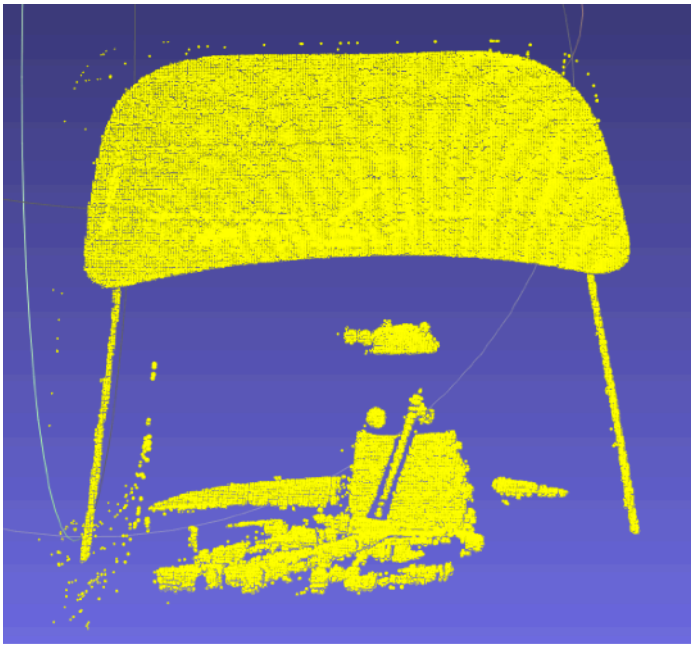
Once again, the main point for this data set was to find a good value for the threshold. Indeed, if the threshold is too low, here is the resulting point cloud :



We can't really distinguish between the crayons and it's very noisy.
However, if the threshold is too high, lots of information will be lost in the process.

Hence, I decided to set the threshold to an median value of 4.5. And here is the result I obtained :

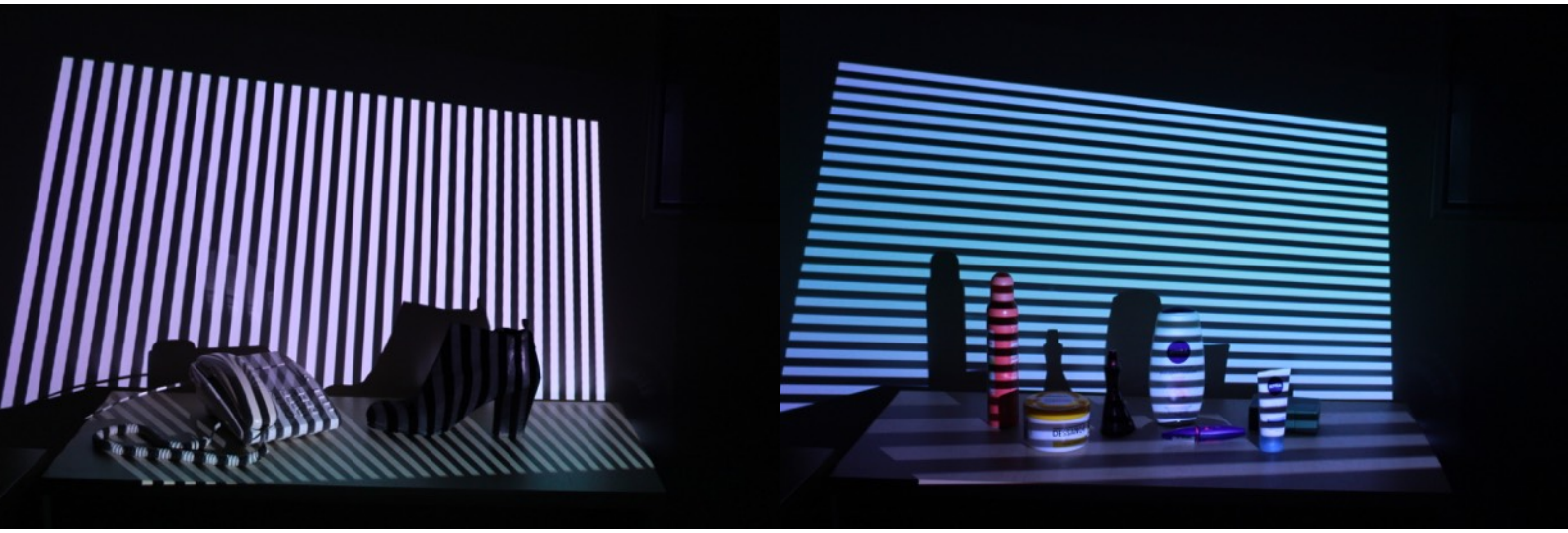




Part 3 : Real Data Captured

For this part, I gathered data with Hugo and Celine

We computed 2 scenes, the first one uses a phone and a shoe. The phone has a cable with high texture which should be hard to reconstruct and as the shoe has a low albedo, it should be hard to find good values in high frequencies too. The second scene is the 'bathroom scene' (sponsored by Nivea). We put some objects with high specular effects and we also add some occlusion between object to see how our algorithm could recover.



We can see some color effects due to the unsynchronization between the camera frequency and the projector frequency. So, I used only one channel of my images in order to deal with that.
We also turn off the auto-focus in the camera to have proper quality images by setting right values for the aperture and shutter speed. We tried not to make the scenes moved at all.
However, we still have aliasing in the high frequencies (see our data in the 'data_set' folder).

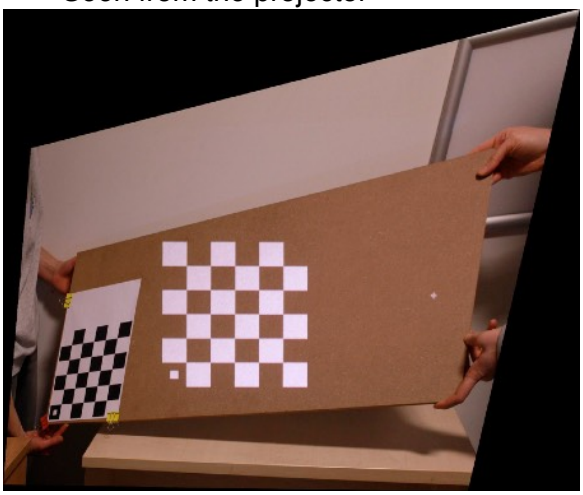
Decode the light patterns, determine the unique depth, eliminate wrong pixel

For all these part, I computed them using the same code than for the real data set explained above. However, I had to put my threshold quite high to be able to get free of sufficient noise to be able to see the objects.. Indeed, the point clouds I obtained were very noisy. The value of the threshold is determined in function of the resulting point-cloud using the mean and the variance as above.
Moreover, I smoothed my data in order to obtain a unique depth for each point (using the function medfilt1 with an order found using test as well. The value depends on the points cloud.

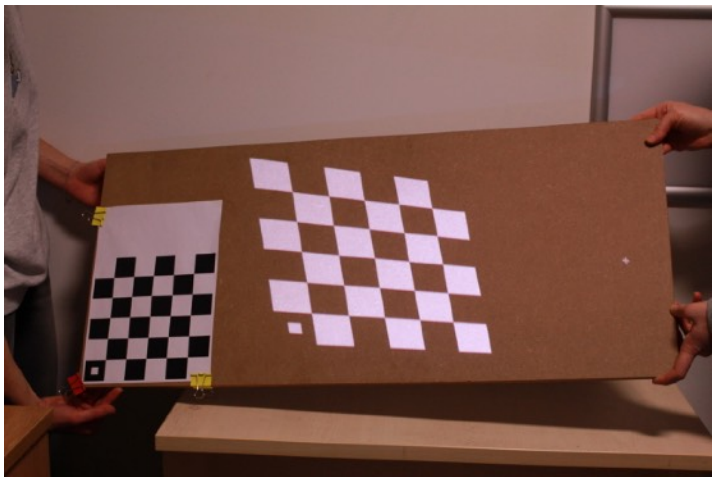
Determination of the projection matrices

To do this part, I mostly did it using the same code as the previous section.
However, we had to be careful while setting our projector to which size we projected the pattern. The resolution of the computer was 1280*800 and we projected the pattern of the chequerboard for the calibration at 400*400 pixel.
Then, using that, I was able to compute the homography matrix for each images (between the projector image plane and the plane of the projection and apply this homography back to our images to obtain the images seen by the projector. My code is is in 'calibration/Real_Calib' and my results in 'calibration/Real_Calib/own_Data_calib'
Here is the type of images I obtained :

Seen from the projector



Seen from the camera

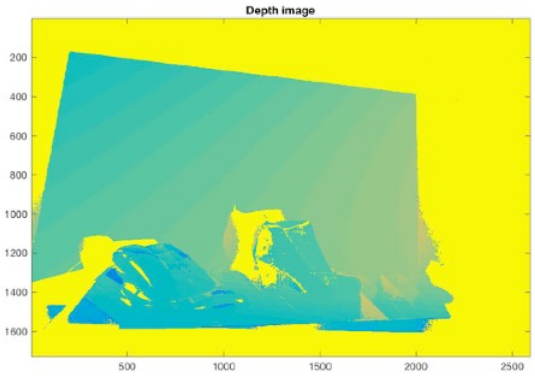


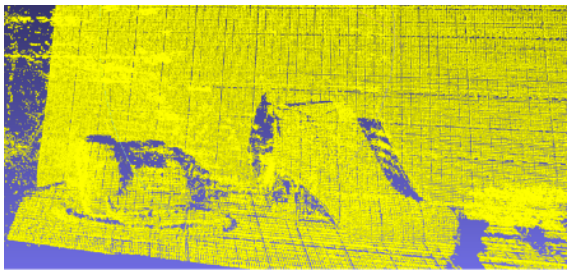
Then, I get back the rotation and translation matrix from the first image.

Result per scene :
Result for the shoe and the phone :

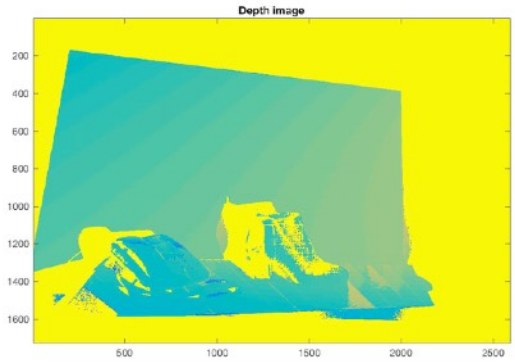
As I said, the result was very noisy so I had to augment the threshold for unsure pixels. On those depth images we can see the result from moving the threshold. Indeed, the point cloud were less noisy but we loose lots of the objects (

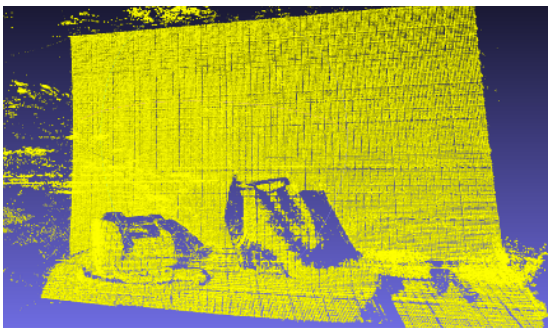
Threshold at 0.27



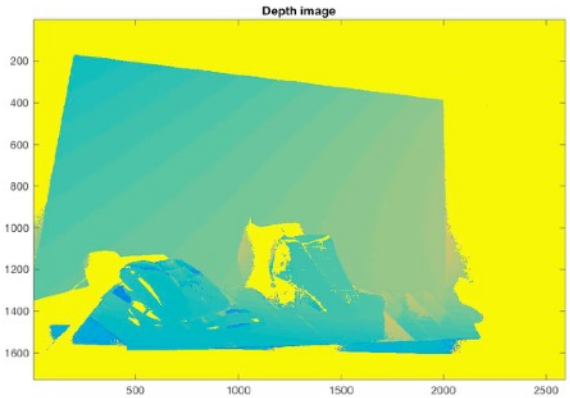


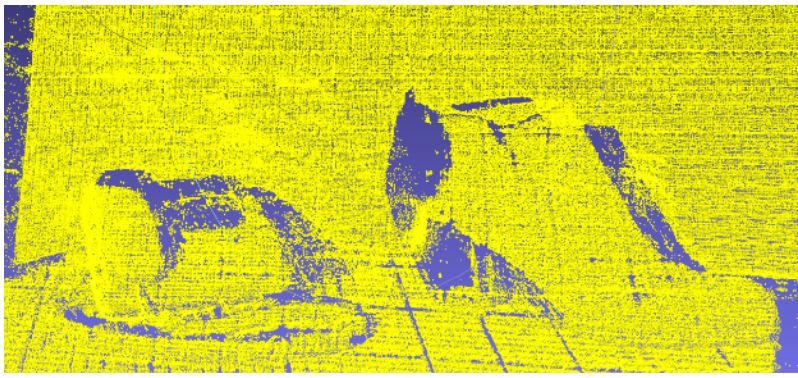
Threshold at 0.50





Seeing that, I put my final threshold to 0.35 and a smoothing level with an order of 10 and here is my final result :

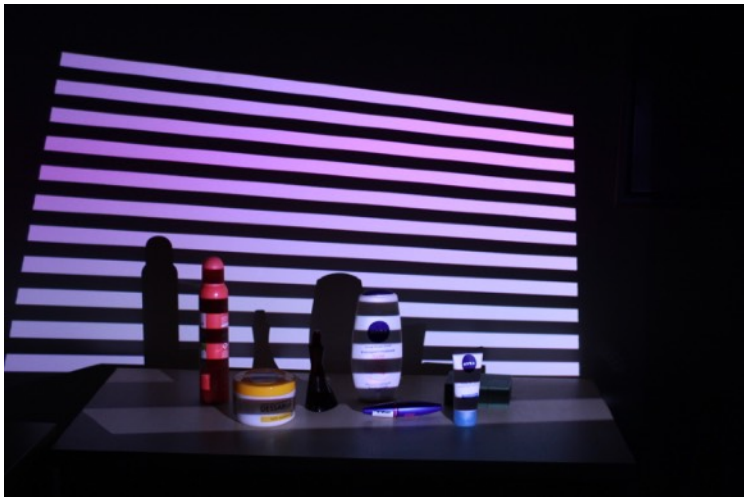




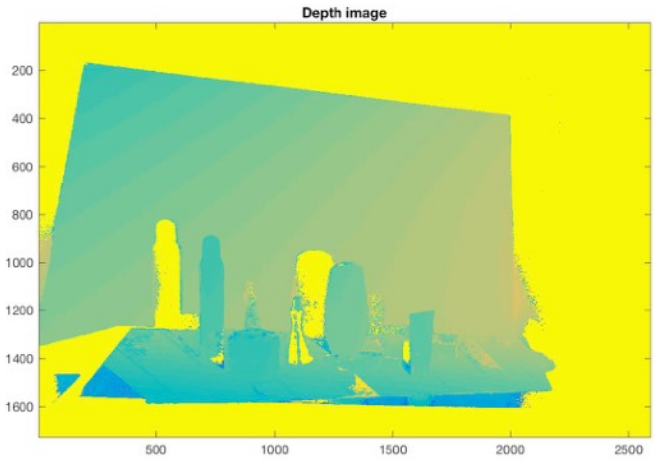
We can that the the scene reconstructed is very noisy even with a high threshold. The specular part of the phone isn't recovered at all (not very surprising) and the shoe with a low albedo not very well easier. Moreover, we can see a rectangular pattern the floor I can't really explained.

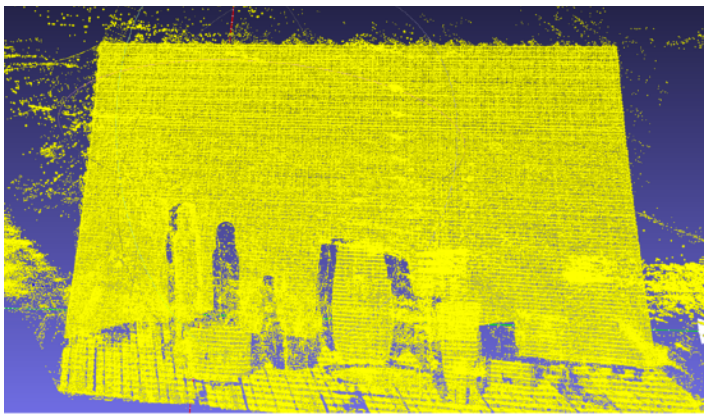
Result for the bathroom scene :

Here is the scene we are trying to reconstruct :



But putting the same threshold as in the previous scene here is the image I obtained :





As for the previous scene, the result is very noisy.

I think it's an artifact in our data which explains why we obtain so much noise. Indeed, we have aliasing for high frequencies. Moreover, I think we took 2 complicated scenes where some objects had low albedo, specular highlight and occlusions. For exemple in the bathroom scene the bottle of parfum is absolutely not reconstructed because of the low albedo plus the specular highlights.

To suppress the noise, we could have try to remove the points outside of certain ellipsoid (knowing how the scene is constructed). We could find this ellipsoid by compute the mean and the variance of the point cloud in every direction. The point outside those values are more likely to be noise. The same could have been done for the point cloud of the real data to avoid having too much noise and loss information by putting a high threshold.