# CS 117 FINAL REPORT

**name**: Leyang Cheng

## Project Overview:

I am trying to build a pipeline that utilize a set of 2D structure light scan pictures from 2 cameras in different perspective to reconstruct a color, high-quality 3D model. I am choosing koi, a fish sculpture for this project. The project starts from calibrating the cameras and then decoding the images of objects that have sequence coded pattern on it to get the correspondences between two cameras. Then using triangulation to get a 3D point cloud. Later merged different perspective into one continuous surfaces. During reconstruction, there will be some strategies for mesh clean up. After getting 5 scans from 5 sets of different angle images, I use rigid alignment to align all 5 mesh and merging via Poisson by Meshlab.

## Data:

All raw data comes from the dataset provided with the course. The subject is a koi fish sculpture, scanned 5 times on a turnable chessboard. Each grab is captured by two cameras, C0 and C1, and each camera records 39 Gray-code pattern frames, so a single grab is 2 × 39 = 78 frames and the full set is 390 frames. All images are at 1920 × 1200 resolution. Each grab also includes the object color images and a background image used for masking. Another calib directory holds the checkerboard calibration images for initial calibration.



The provided data have one issue that the background color image beside grab 0 are not actually empty background image but same as the object images, which means, according to Ed discussion, we are only using grab 0's background image for foreground filtering. There will be some mismatch on the turnable chessboard.

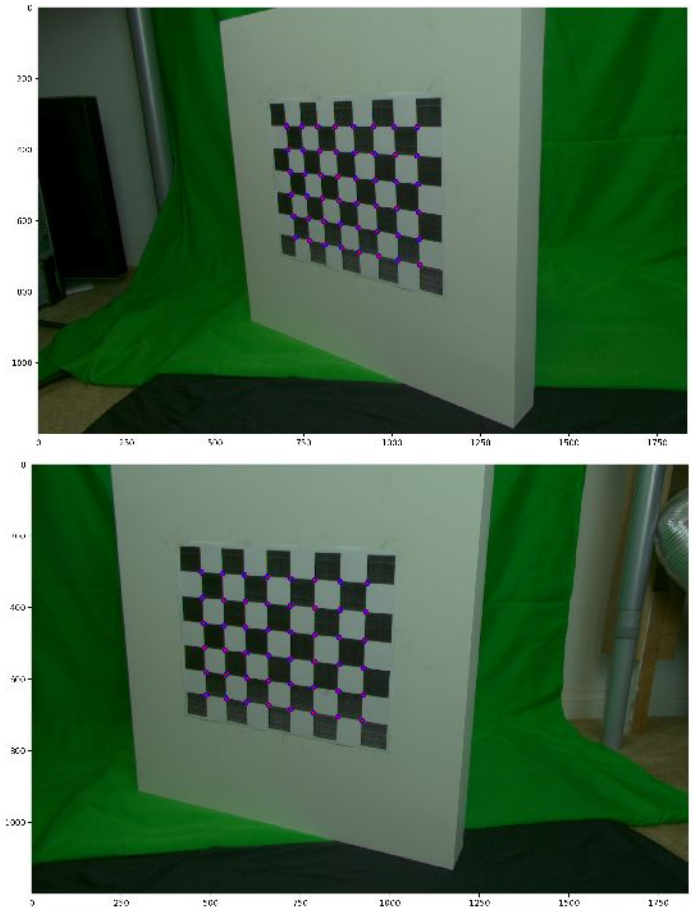
## Algorithms:

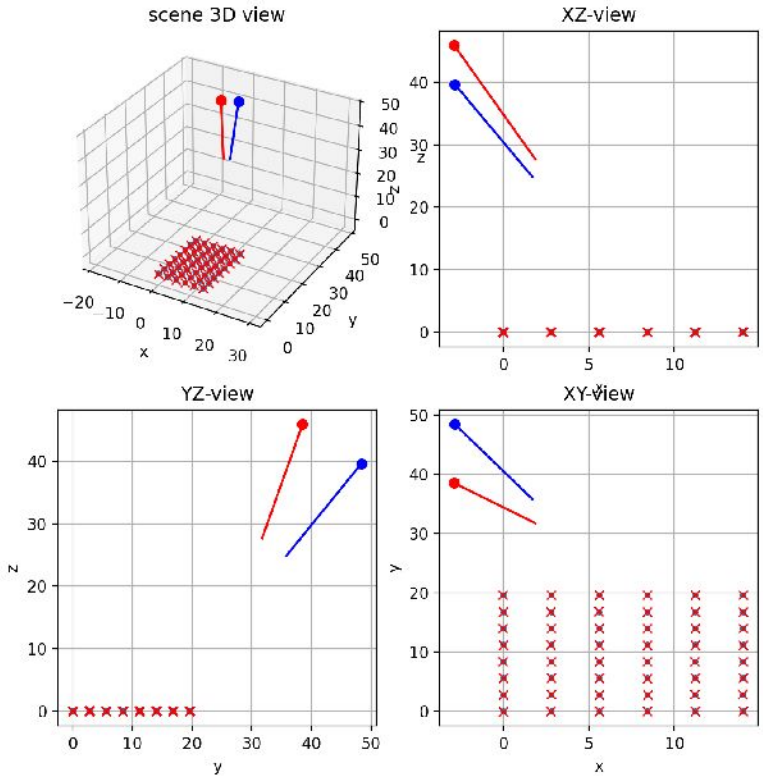
The pipeline is organized into utility modules — `camutils.py`, `calibrateUtils.py`, `reconstructUtils.py`, `meshutils.py`, `selectpoints.py`, `visutils.py` — driven by a notebook (`finalProject.ipynb`) that executes each stage and visualizes intermediate results.

1. Camera calibration, intrinsics (`calibrate.py`): camera calibration reuses code from earlier assignments with some modification. Intrinsic parameters were estimated with the provided `calibrate.py`, which detects 8×6 checkerboard corners in the `calib` images with OpenCV and runs `cv2.calibrateCamera`, yielding focal lengths, principal point,

and radial distortion for each camera (saved as `calib_C0.pickle` / `calib_C1.pickle`).

2. Camera calibration, extrinsics (`calibrateUtils.py`): extrinsic parameters are calibrated by `calibrateUtils.calibrate_extrinsics`, adapted from my Assignment 3 code with some modification. It calls `camutils.calibratePose` (from `camutils.py`, provided in Assignments 3 and 4), which solves for the six extrinsic parameters (three rotation angles, three translations) by nonlinear least squares (`scipy.optimize.leastsq`). After calibration, I use `visutils.vis_calibrate_extrinsic` to visually verify the result:

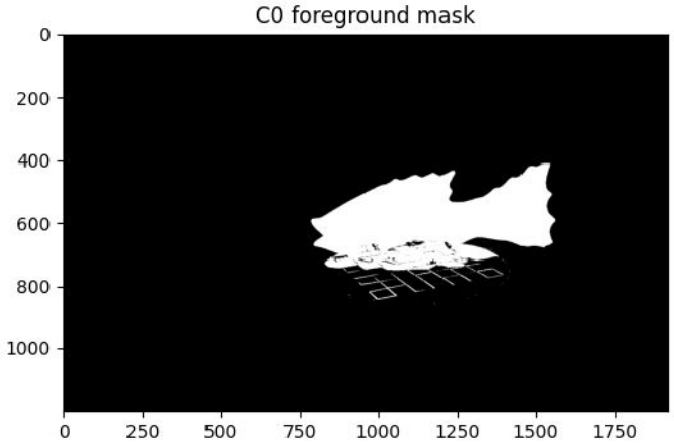





3. Foreground masking ( `reconstructUtils.compute_foreground_masks` ): classifies each pixel by comparing the object color image against the grab_0 background image. Dissimilarity is the per-pixel sum of squared differences over the three color channels; a pixel is foreground when this exceeds `color_thresh` (final value 0.065). The method follows the background-subtraction masking described in lecture 13.

```
color_mask[x,y] = |rgb[x,y,:] - rgb_bg[x,y,:]|^2 > thresh
Rmask = color_mask & R_h_good & R_v_good
```

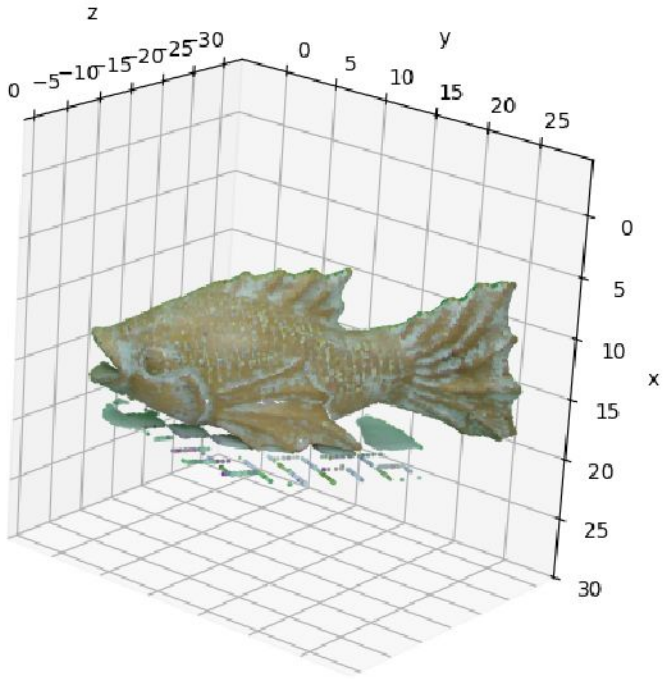
The resulting mask is combined with the decoding mask inside `reconstruct` , so only pixels that both differ from the background and decode reliably are triangulated.




4. Decoding and triangulation are adapted from Assignment 4. `reconstructUtils.decode` decodes the 10-bit Gray-code sequences with `decode_thresh` (0.015). `reconstructUtils.reconstruct` combines the horizontal and vertical codes into a 20-bit identifier per pixel, matches codes across the two cameras (restricted by the decode

and foreground masks), and triangulates each match with the provided `camutils.triangulate`. The modification made for this project is color: for every triangulated point, the RGB value at its left-image pixel is recorded into a 3×N `colors` array returned alongside `pts3`, `pts2L`, `pts2R`.

5. With `reconstructUtils.mesh_cleanup`, I prune and build the surface mesh. first with the mesh clean up is the user input method, bounding-box cropping: A per-grab axis-aligned box `boxlimits = [xmin, xmax, ymin, ymax, zmin, zmax]` keeps only points inside the object's spatial extent. Each grab gets its own box since the fish faces a different direction in every grab. The main target of the crop is the checkerboard under the fish: it sits on the turntable and rotates with it, but the background reference is always grab_0's image, so in grabs 1–4 the rotated black-and-white squares no longer line up with the background so `color_thresh` can't remove them. Thus, I remove it by user input the bounding box cropping. To set the limits precisely, I render each trial reconstruction as a 3D scatter plot colored by the per-vertex RGB values.




6. The second part is from Assignment 4: the surviving 2D points are triangulated with scipy.spatial.Delaunay, and any triangle with a 3D edge longer than trithresh (final value 0.6) is discarded to remove the stretched "webbing" faces that span depth discontinuities. Vertices no longer referenced by any triangle are then dropped, and the triangle array is remapped.
7. The last part of reconstruction is smoothing the mesh: `meshutils.smooth_mesh`. Inside this function, I compute the average location of each 3D point's neighbors in the mesh and replace the point coordinate with this average, and iterate this process to increase smoothness. `meshutils.smooth_mesh` first builds a neighbor set per vertex in one pass over the triangle list: for every triangle (a, b, c), each vertex adds the other two to its set. Each smoothing iteration then replaces every vertex position with the mean of its neighbors' positions. The smoothing is repeated 3 times in my case. The logic follows the lecture slides and the project guideline.

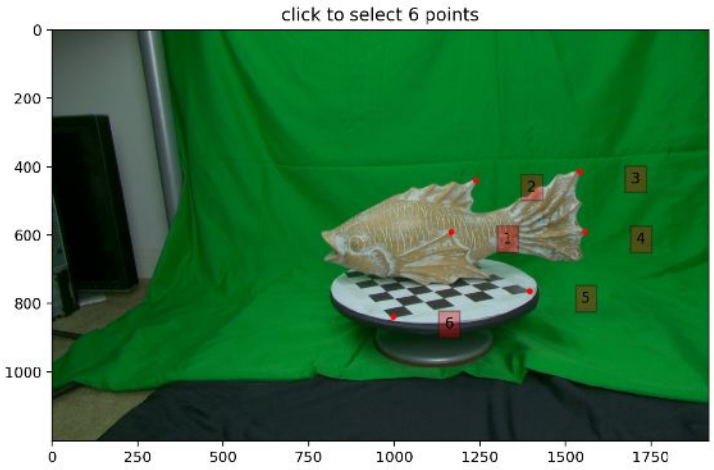


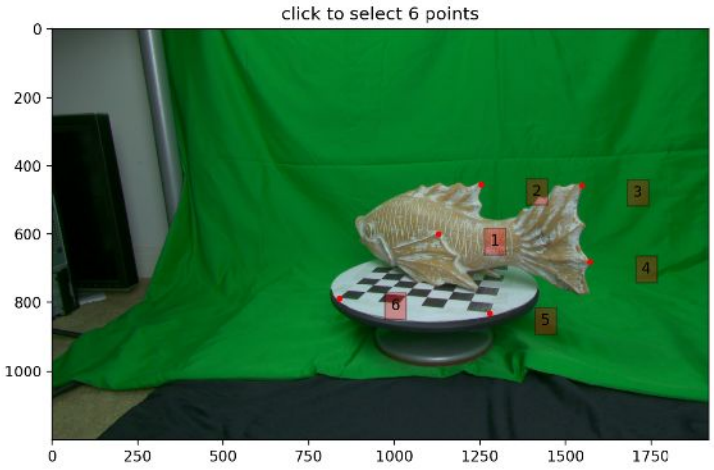
Pseudocode to help understanding:

```
  def smooth_mesh(pts3, tri, iterations):
      # build 1-ring neighbor sets from the triangle list
      neighbors[v] <- {} for every vertex v
      for each triangle (a, b, c) in tri:
          neighbors[a] += {b, c}
          neighbors[b] += {a, c}
          neighbors[c] += {a, b}

      repeat iterations times:
          newpts3 <- empty array same shape as pts3
          for each vertex v:
              newpts3[v] <- mean of pts3[u] for u in neighbors[v]   #
smooth the mesh
          pts3 <- newpts3
      return pts3
```

8. Multi-scan alignment ( `selectpoints.py` + notebook): each grab is reconstructed in the camera's own coordinate frame, so the five scans must be brought into one common frame before merging. I use grab_0 as the reference and rigidly align grabs 1–4 to it directly.
   For every pair, I click `npoints = 6` corresponding features (eye, fin tips, pattern corners) in the color images of the two grabs using `selectpoints.select_k_points`, which draws and numbers each click so the ordering stays consistent between the two images. `selectpoints.clicks_to_pts3` then converts clicks to 3D: for each click it finds the nearest reconstructed point in `pts2L` by Euclidean distance and looks up the same column in `pts3`.



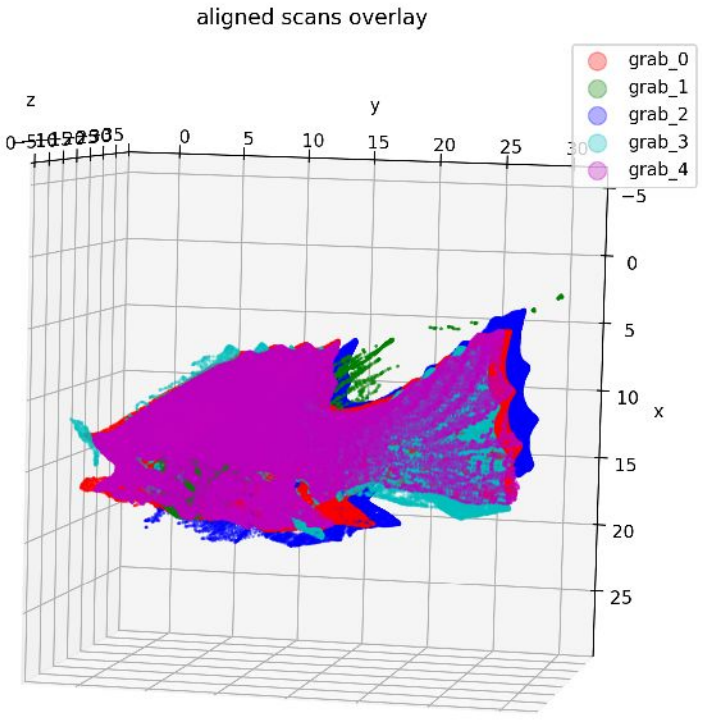

The rigid transform (R, t) is the closed-form SVD solution from lectures 12–13, implemented in the notebook:

```
# svd rigid alignment of source onto reference (lec 12)
m1 = np.mean(X1, axis=1, keepdims=True)
m2 = np.mean(X2, axis=1, keepdims=True)
X1c = X1 - m1
X2c = X2 - m2
H = X2c @ X1c.T
U, S, Vt = np.linalg.svd(H)
R = Vt.T @ U.T
t = m1 - R @ m2
X2aligned = R @ X2 + t
```

The transform is estimated from only the six clicked pairs but applied to the entire source point cloud.

I was considering doing alignment through neighboring grabs (grab_2 → grab_1 → grab_0), which would give larger overlap per pair and easier clicking. But I eventually choose to make grab_0 as reference and other to align to grab_0. The reason is every link would have slightly error of the previous one, and at the dimension of 1920×1200, there would be more error come from the accuracy of my click, so align with neighbors would amplify human error across each grabs.

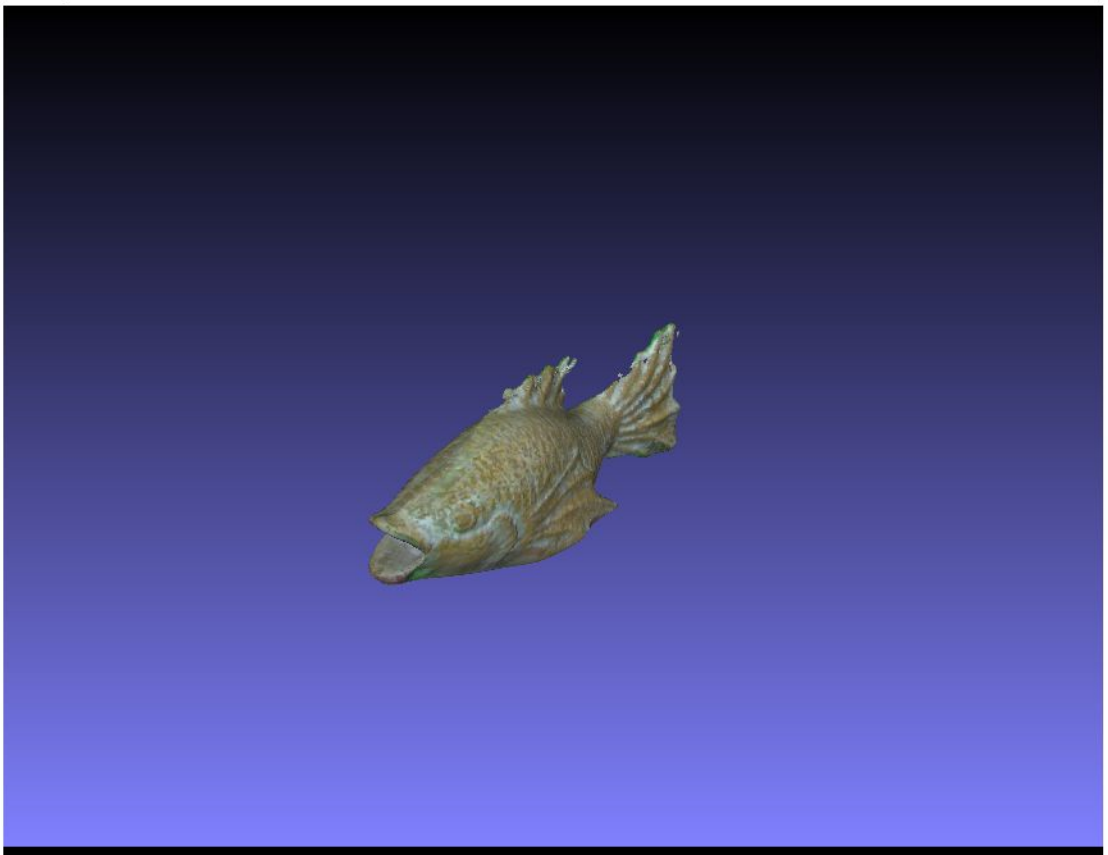
Alignment quality is verified visually by overlaying all five aligned clouds in one 3D scatter plot, one color per grab at low opacity; correct alignment shows the colored clouds coinciding on the fish surface:

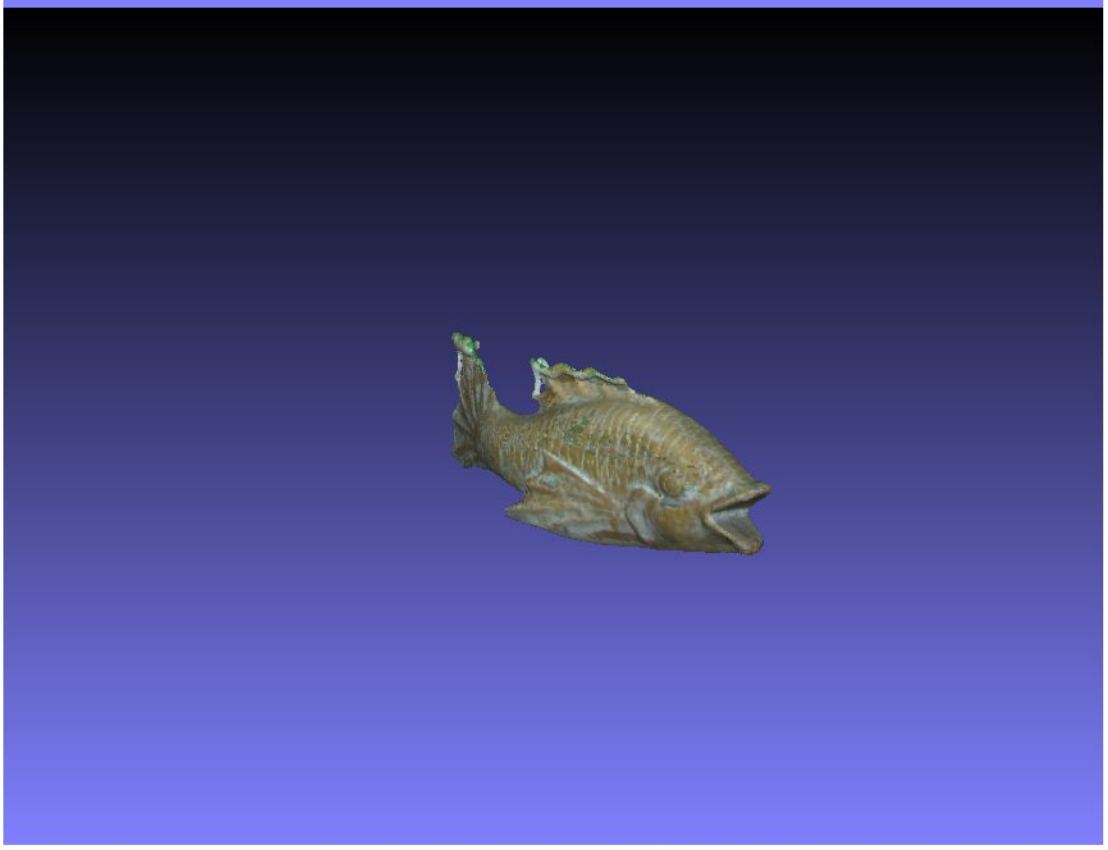


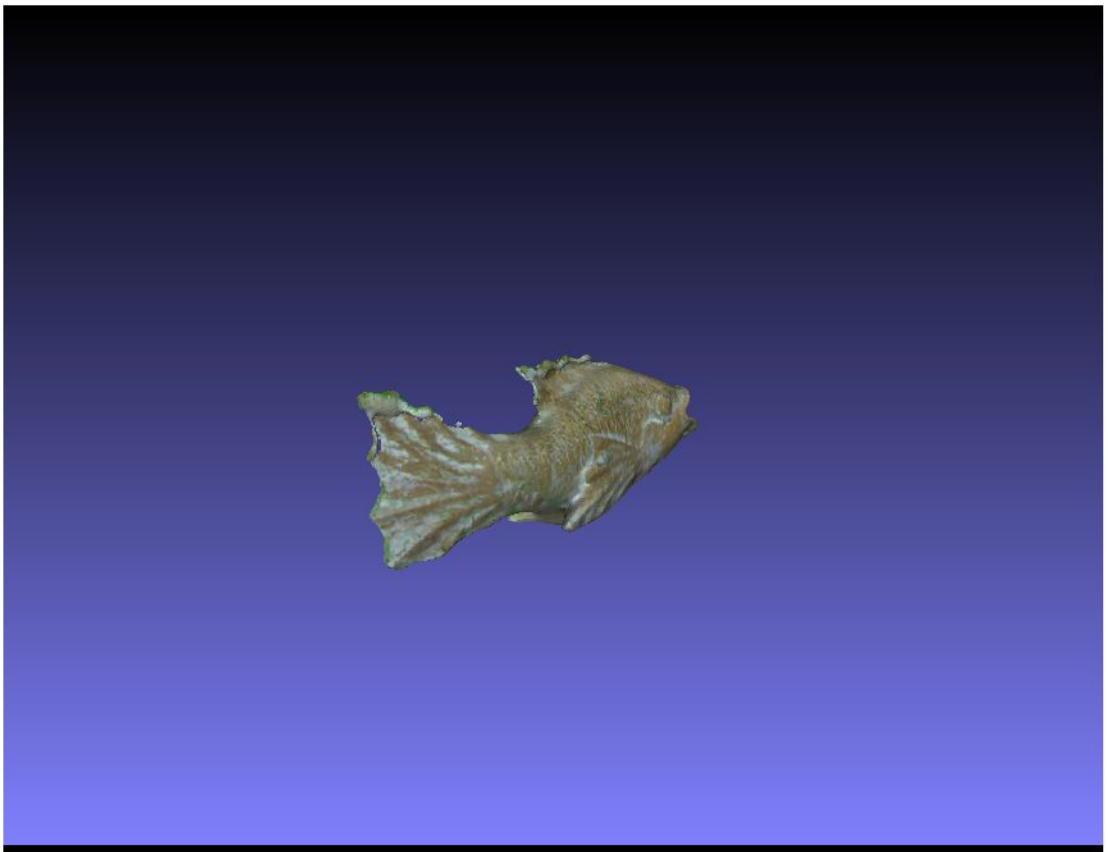

9. Merging and Poisson surface reconstruction (`meshutils.writeply` + MeshLab): after alignment, each grab is written out as a colored `.ply` file using the provided `meshutils.writeply`, applying the per-grab vertex offset so the triangle indices stay valid. The five files are then merged in MeshLab: Then use the meshlab tool of Filter -> Poisson to extrapolates a closed surface. Since the Poisson output has no color, the per-vertex colors are transferred from the merged five-grab point cloud onto the mesh with MeshLab's Vertex Attribute Transfer. The final mesh is exported as `model.ply` (53,883 vertices, 107,762 faces). Figure show below on Result sections.

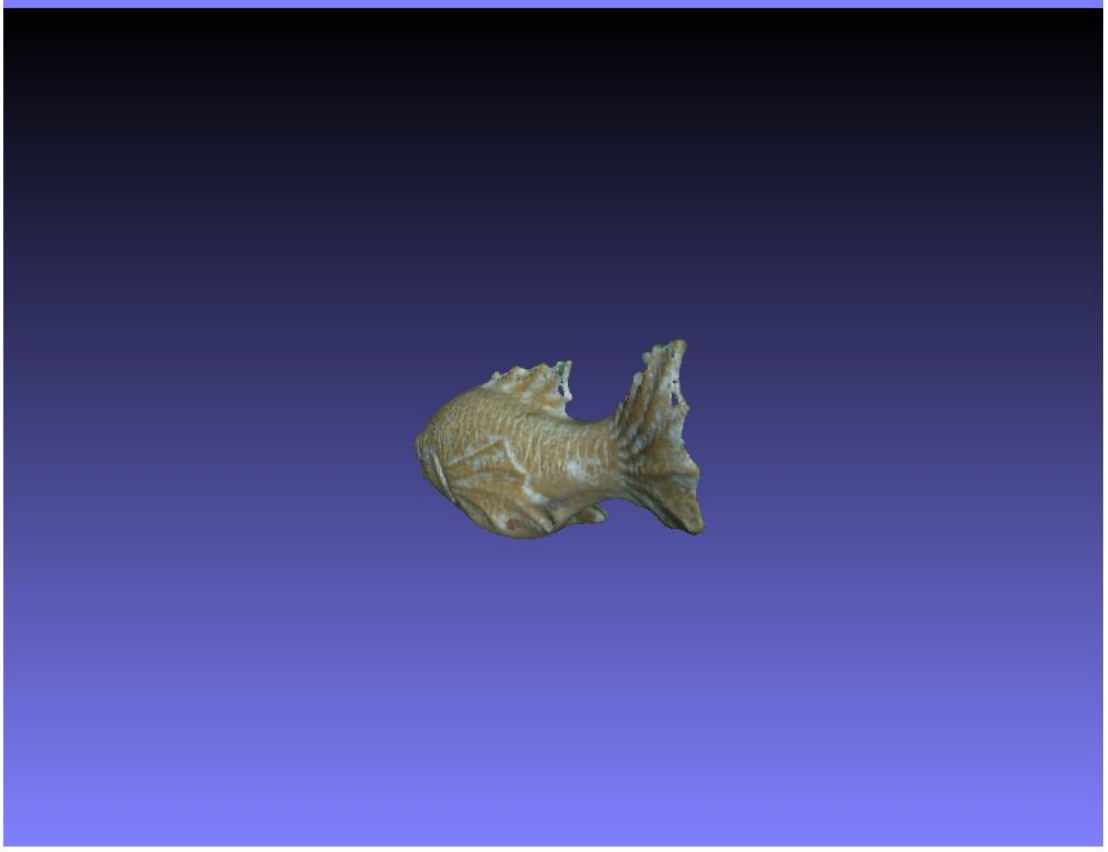
# Results:

Snapshots:





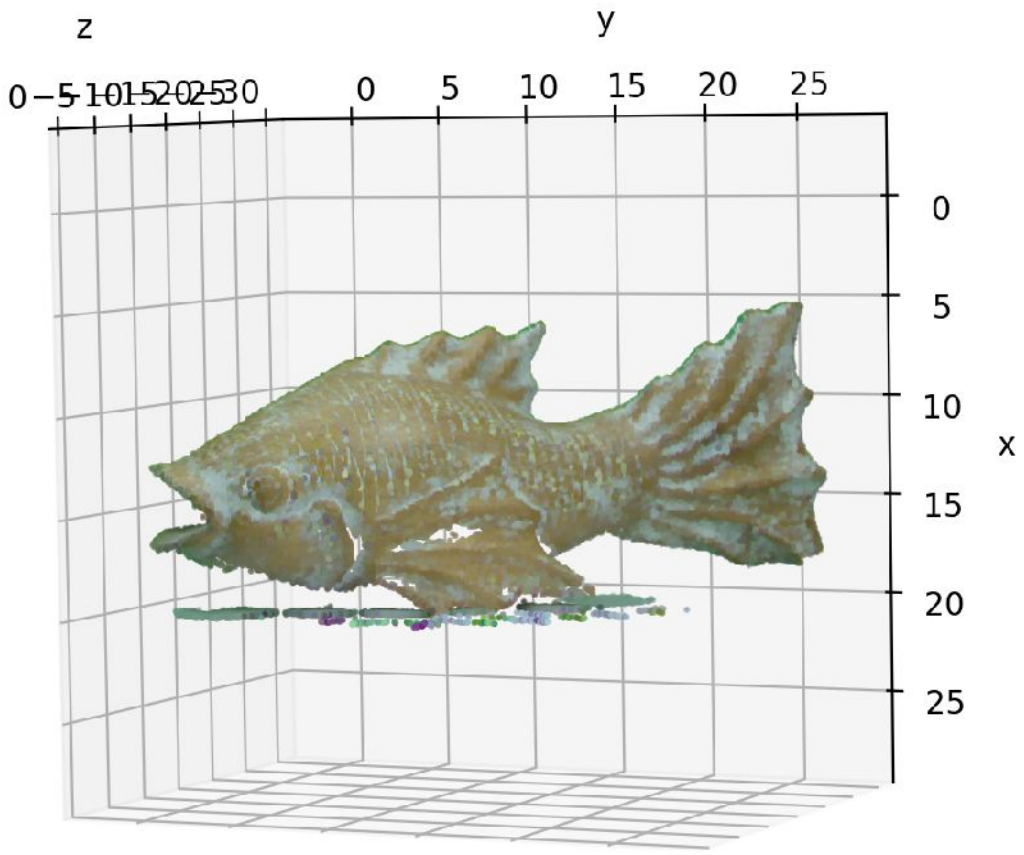


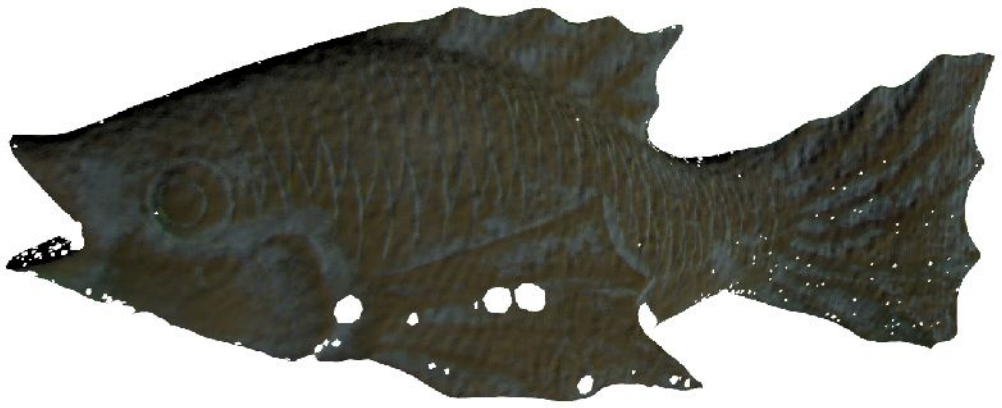


Real Figure

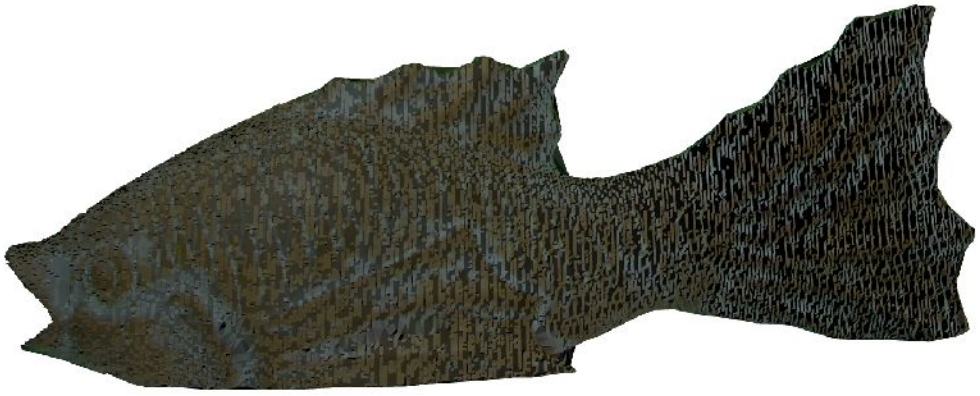


Effect of mesh clean: Before vs After



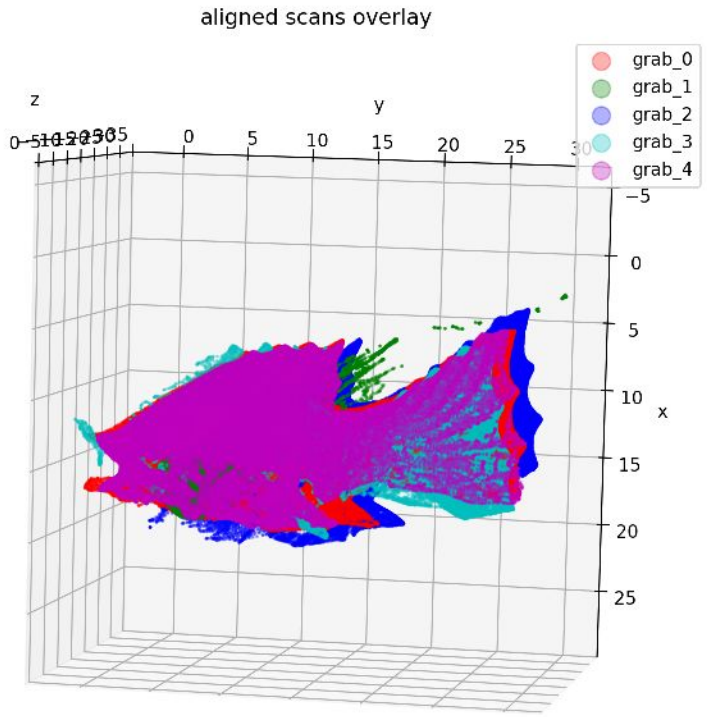


Effect of smoothing: Before vs After:





Alignment result in scatter plot




Parameters:
All five grabs share the same thresholds; only the bounding box is per-grab.
`decode_thresh` =0.015
`color_thresh` =0.065
`trithresh` =0.6

`boxlimits` [xmin, xmax, ymin, ymax, zmin, zmax]

`boxlimitlist` = [
np.array([5, 19.5, -3, 28, -29, -4]),
np.array([5, 19.5, -3, 28, -30, 0]),
np.array([5, 19.5, -4, 33, -30, 1]),
np.array([5, 19.5, -4, 33, -30, 8]),
np.array([5, 19.5, -10, 33, -25, 1]),
]

# Assessment and Evaluation

I believe the task is modern difficulty, the major time costing is adjusting thresholds and clicking points for alignments.

For success:
I finish the whole pipeline that can runs end to end from raw scan directories to a single colored model. I get a pretty nice boxlimits for each grab. I do try to use this pipeline on different object, such as david. However, I need to adjust the boxlimit again and I found difficulty in alignment for david so here I do not have a nice model for david. The color and

detail pattern of koi is still preserved in my model, especially the fish scale, is nicely visible. That is also what I am satisfied for.

For limitation:
mostly grab1 of the koi that only limited fish body is being scan, difficult in alignment:



The fins and tails are too thin for structured light so there would be more likely to have holes on mesh.
The background for foreground mask is always from grab_0. This limit the foreground masking which was able to easily remove the checkerboard at the first time.
There are slightly color reflection of green on my object.
One practical obstacle: MeshLab's Poisson filter repeatedly crashed on my macbook until I found that limiting it to a single thread avoids the bug.
Difficulty in alignment: the image dimension is only 1920 x 1200, a slightly click error would make a visible 3D offset, Furthermore, there amount of grab I believe is not enough. In fact, to gain a better result, I did use align tool from MeshLab after my align version to slightly adjust the detail of the mesh for better effect.

If have more time:
I would choose to improve the camera and background first: I would increase the camera's resolution and also get each grab's clear background image to help foreground mask.
I would also try to get more grabs from different angle. (what if I have lets say 36 cameras around the object?)
I would also add ICP refinement after SVD to enhance the effect of alignment.
I would use more technique from meshlab to make my mesh look better.
I would also try to replace manual alignment and study for some automatic way to do so since manual way is too time consuming and annoying.

# Appendix

Self-written code:

`reconstructUtils.compute_foreground_masks`
`reconstructUtils.reconstruct_scan`
`meshutils.smooth_mesh`
`selectpoints.clicks_to_pts3`
`finalProject.ipynb` (alignment code inside, including SVD rigid alignment)

Modified from Previous Assignments :

`calibrateUtils.calibrate_extrinsics` (A3)
`reconstructUtils.decode` (A4)
`reconstructUtils.reconstruct` (A4)
`reconstructUtils.mesh_cleanup` (A4)
`visutils.vis_calibrate_extrinsic` (A3)

Instructor provided (also include the provided file from previous assignment):

`camutils.py`
`meshutils.writeply()`
`calibrate.py`
`selectpoints.SelectPoints` (from cs116 assignment 3 panorama)
`selectpoints.select_k_points` (from cs116 assignment 3 panorama)
`visutils.set_axes_equal_3d`
`visutils.label_axes`
`visutils.vis_scene`

Outsides sources:

MeshLab