Introduction to Computer Vision (Spring 2025)
Assignment 3

Release date: May 2, due date: May 17, 2025 11:59 PM

This assignment includes 4 tasks: camera calibration, transforming a depth image to a point cloud, sampling a point cloud from a mesh and implementing Marching Cube, which sums up to 100 points and will be counted as 10 points towards your final score of this course. This assignment is fully covered by the course material from Lecture 8, 9 &10.

The objective of this assignment is to get you familiar with processing 3D data. We offer starting code for all the tasks and you are expected to implement the key functions using Python and Numpy.

Policy on using for loop/while: for some questions that don't allow for loop/while, using them will be penalized (2 points for 1 use). Some useful Numpy functions are included in Appendix for your information.

Submission: To submit your homework, please compress your code **and your results** using our provided script *pack.py* following the original path structure, and submit to course.pku.edu.cn. Feel free to post in the discussion panel for any questions and we encourage everyone to report the potential improvements of this assignment with a bonus of up to 5 points.

1. **Camera Calibration (20 points):**

   Given a set of corresponding pairs of 3D coordinates in world coordinate space and 2D coordinates in image coordinate space, we can solve the intrinsic $\mathbf{K}$ and extrinsic $[\mathbf{R}, \mathbf{T}]$ of a *perspective* camera. In this question, you are required to implement this process of camera calibration.

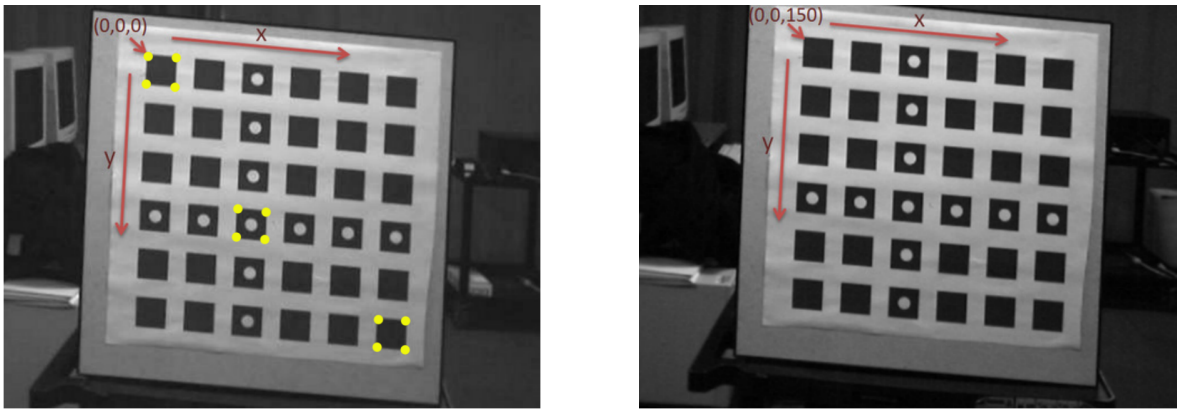

Figure 1: Left: Image of calibration grid at Z=0. Right: Image of calibration grid at Z=150.

In practice, we often utilize a calibration grid to get correspondences. The left image of Figure 1 shows a calibration grid in the rest/zero position. This calibration grid has 6×6 squares and each square is $50mm \times 50mm$. The separation between adjacent squares is $30mm$, so the entire grid is

$450mm \times 450mm$. For the right image in Figure 1, the calibration grid was moved back along its normal from the rest position by $150mm$. The process of calibrating a camera is shown below:

First, to get the 3D coordinate of each corner of the squares, we define the top left corner of the calibration grid in the rest/zero position as the origin of world coordinate space. The X-axis runs left to right parallel to the rows of squares and the Y-axis runs top to bottom parallel to the columns of squares, as shown in Figure 1. The Z-axis is then perpendicular to the calibration grid and points backward to the plane. Now, we can obtain the 3D coordinates of the corners on both the two images, *e.g.* the bottom right corner of the right image is at (450, 450, 150).

Second, we need the 2D coordinates of each corner of the squares, which could be obtained by the corner detector *e.g.* Harris Corner detector in Assignment 1. Here, you don't have to bother implementing all the tedious details since we have already computed some of the 2D coordinates with corresponding 3D coordinates of corners.

Finally, with the corresponding 3D coordinates and 2D coordinates pairs, we can solve the intrinsic and extrinsic parameters of the camera by the techniques introduced in class. For-loop is **allowed**.

a)[5 points] Compute the Corresponding 3D coordinates

In this question, you have 12 pre-computed 2D coordinates of corresponding pairs for each image, which are indicated by the yellow points in the left image in Figure 1. Another 12 pre-computed 2D coordinates of the right image correspond to the same grid corners as in the left image. And you are required to calculate all 24 3D coordinates of them.

b)[10 points] Construct the Equation $\mathbf{Pm} = \mathbf{0}$ and Solve $\mathbf{m}$

For the 2D coordinates $p_i = [u_i, v_i]^\top$ $(0 \leq i \leq 24)$ and their corresponding 3D coordinates $P_i = [x_i, y_i, z_i]^\top$, we have:

$$\lambda \begin{bmatrix} p_i \\ 1 \end{bmatrix} = \mathbf{K}[\mathbf{R}\ \mathbf{T}]P_i. \tag{1}$$

And we can further define the detailed transform Equation 1 as

$$\begin{bmatrix} u_i \times w_i \\ v_i \times w_i \\ w_i \end{bmatrix} = \begin{bmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \end{bmatrix} \begin{bmatrix} x_i \\ y_i \\ z_i \\ 1 \end{bmatrix} \tag{2}$$

You are required to follow Equation 2 to construct a homogeneous linear system $\mathbf{Pm} = \mathbf{0}$ and solve $\mathbf{m}$, in which $\mathbf{P}$ is a 48x12 matrix and $\mathbf{m}$ is a 12x1 matrix. Please refer to the slides of Lecture 8 for the details.

c)[5 points] Solve K and [R T] from m

Please follow the instruction on the slides to solve $\mathbf{K}$ and $[\mathbf{R}\ \mathbf{T}]$ from $\mathbf{m}$. The results will be saved in *calibr.npy* for submission.

2. **Backprojection: Transform a Depth Image to a Point Cloud (20 points):**

   In this question, you are required to transform a depth image to a point cloud. The depth image is synthesised by a standard perspective camera.

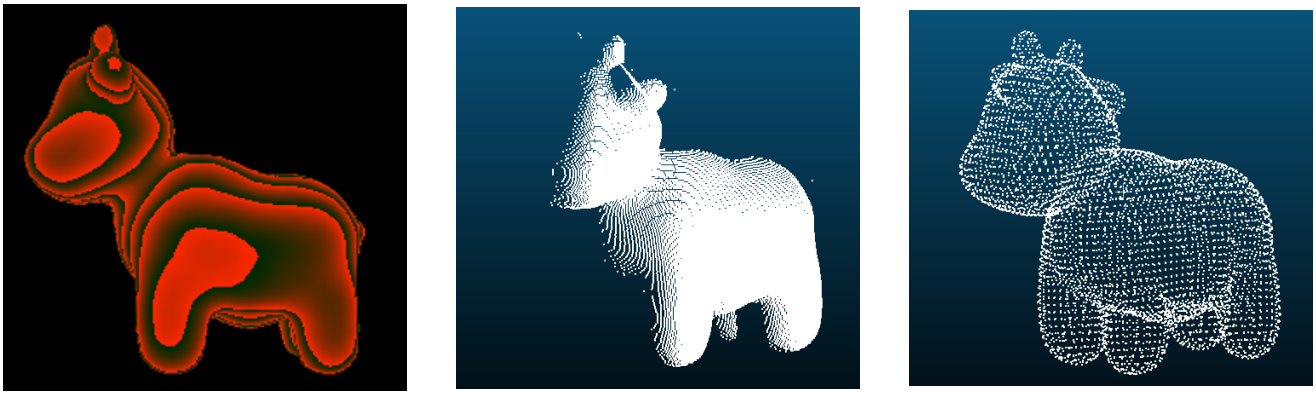

Figure 2: Left: raw depth image. Middle: transformed partial point cloud with another viewpoint. Right: ground truth complete point cloud.

   After backprojection, compute the one-way Chamfer distance from your generated partial point cloud to the ground truth complete point cloud, namely

$$\text{one way chamfer} = \frac{1}{|S_1|} \sum_{x \in S_1} \min_{y \in S_2} \|x - y\|_2$$

   where S_1 is the partial point cloud and S_2 is the complete point cloud. We only care about one-way chamfer distance here because another-way chamfer distance is meaningless due to occlusion.

   You are required to only use Numpy to finish this task. For-loop is **not allowed** in this question.

3. **Sample a Point Cloud from a Mesh (35 points):**

   a)[15 points] Uniform Sampling

   An easy and fast way to sample a point cloud from a mesh is uniform sampling. First, you need to compute the area of each individual face and use it to compute the probability of each face to be sampled. Then, independent identically distributed (i.i.d.) sample faces according to the probabilities. Finally, for each sampled face, uniformly sample one point inside the triangle.

   In this question, you are required to implement this algorithm and you should expect a result as Figure 3. Note that for-loop is **not allowed** in this question.

   b)[15 points] Farthest Point Sampling

   Though uniform sampling is easy and fast, this method often results in irregularly spaced sampling as shown in Figure 3. Farthest point sampling (FPS) is another sampling strategy, which can ensure that the sampled points are far away from the others.

   Under the greedy approximation, this algorithm will first uniformly sample a large number of points U, and randomly choose one point p_0 as the initialization of a point set S. Then, for each iteration, pick up the point $p_i \in U$, which is the farthest to the current point set S, and add p_i to S. This pick-and-add process is repeated until the point set S have enough points as we want. The result is shown in Figure 3.

Note that for-loop is **allowed** in this question, since the sampled point of each iteration relies on the results of previous iterations.

c)[5 points] Metrics

Finally you are required to compute the Chamfer distance (CD) and earth move distance (EMD) of those two point clouds sampled by different methods. To give you a feeling about which metric is more sensitive to sampling, you are required to repeat sampling and computing metrics for 5 times (and for-loop is **allowed** here), and save the mean and variance of CD and EMD for submission. For distance calculation, you will calculate the **mean** value of distance as the result (which is different from definition in slides). For EMD, you can directly use this repository. Or, you can write your own EMD method: if you have the distance (cost) matrix, you can solve the minimum matching problem using scipy.optimize.linear_sum_assignment to get the point matching.

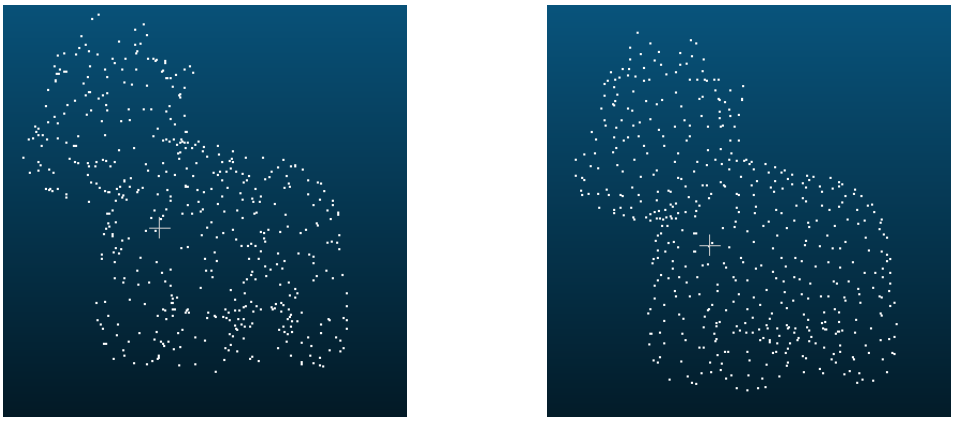

Figure 3: Left: result of uniform sampling. Right: result of farthest point sampling.

4. **Marching Cube (25 points):**

   Marching Cube is one of the most classic and famous algorithm to transform a signed distance field (SDF) to a mesh. In this question, you are expected to implement this algorithm. To make it easier to start, we provide the lookup table and some useful function, as well as a detailed demonstration of how to use them. Please see the hints in *lookup_table.py* for more details.

   You are required to only use Numpy to finish this task. To help you debug and get a feeling of this algorithm, we also provide two SDFs for you. The meshes generated from Marching Cube algorithm should look like Figure 4. Note that for-loop is **allowed** in this question to reduce the difficulty.

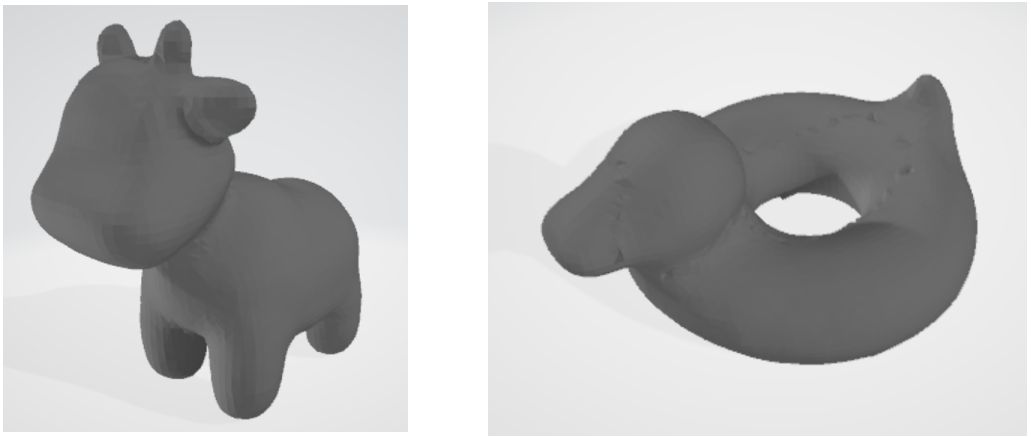

Figure 4: Generated meshes from Marching Cube algorithm. Left: Spot. Right: Bob.

Appendix

1. We recommend some handy Numpy functions which may help your tensor-style coding.

   - meshgrid, https://numpy.org/doc/stable/reference/generated/numpy.meshgrid.html
   - concatenate, https://numpy.org/doc/stable/reference/generated/numpy.concatenate.html
   - where, https://numpy.org/doc/stable/reference/generated/numpy.where.html
   - argmax, https://numpy.org/doc/stable/reference/generated/numpy.argmax.html
   - linalg.svd, https://numpy.org/doc/stable/reference/generated/numpy.linalg.svd.html

2. We recommend some useful software for visualize point clouds and meshes.

   - CloudCompare, https://www.danielgm.net/cc/
   - MeshLab, https://www.meshlab.net/