Introduction to Computer Vision (Spring 2026)
Assignment 3

Release date: May 8, due date: May 23, 2026 11:59 PM

This assignment includes 4 tasks: Mask R-CNN for simple-shape instance segmentation, transforming a depth image to a point cloud, sampling a point cloud from a mesh, and implementing the rendering core of NeRF. The assignment sums up to 100 points and will be counted as 10 points towards your final score of this course.

The objective of this assignment is to help you practice several classic computer vision components across both 2D and 3D settings. We provide starter code for all tasks, and you are expected to implement the key functions using Python and Numpy. The *MaskRCNN* part also requires PyTorch.

Policy on using for loop/while: for some questions that do not allow for loop/while, using them will be penalized (2 points for 1 use). Some useful Numpy functions are included in the 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.

1. **Mask RCNN (20 points):**

   Mask RCNN is a classic instance segmentation baseline. In this question, you will work with a small Mask RCNN that predicts simple shapes such as rectangles, circles, and triangles. The content of this question is reused from Assignment 4 so that you can directly use the existing starter code in the folder *MaskRCNN*. The provided model uses *torchvision* and a lightweight *mobilenet_v2* backbone. The training setup has been simplified so that it can run on CPU with a very small dataset.

   a)[10 points] Prepare the dataset

   Complete the *MultiShapeDataset* class in *MaskRCNN/dataset.py*. You should randomize the background color, randomize the type, position, and size of the shapes, use NMS to remove shapes with large overlap, and update the masks to handle occlusions. After that, run `python dataset.py` to visualize the dataset.

   b)[5 points] Train the Mask RCNN

   Run `python train.py` to train the model. The release script starts from scratch when there is no checkpoint in *MaskRCNN/results/*, and resumes from the latest checkpoint if you rerun it. The training setting is intentionally small: the dataset size is 10 samples, the number of epochs is 3, and the device is CPU.

   c)[5 points] Visualize the Mask RCNN results

   Run `python visualize.py` to visualize the model predictions on the single-shape dataset. The results do not need to be perfect; reasonable outputs will be accepted.

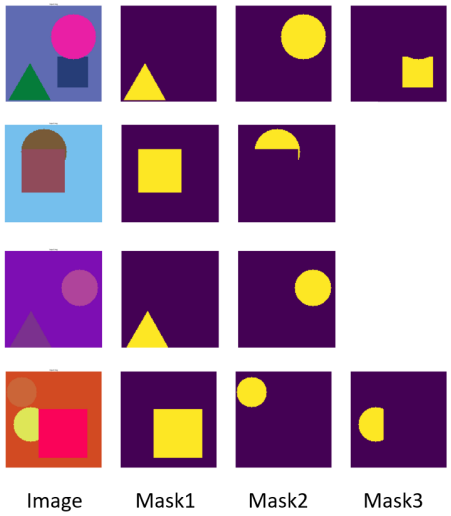

Figure 1: Input data for Mask RCNN

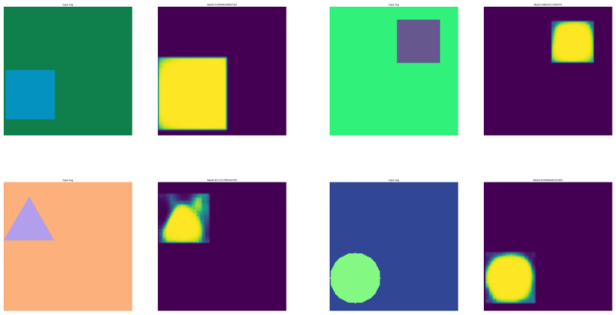

Figure 2: Results of the Mask RCNN

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.

   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

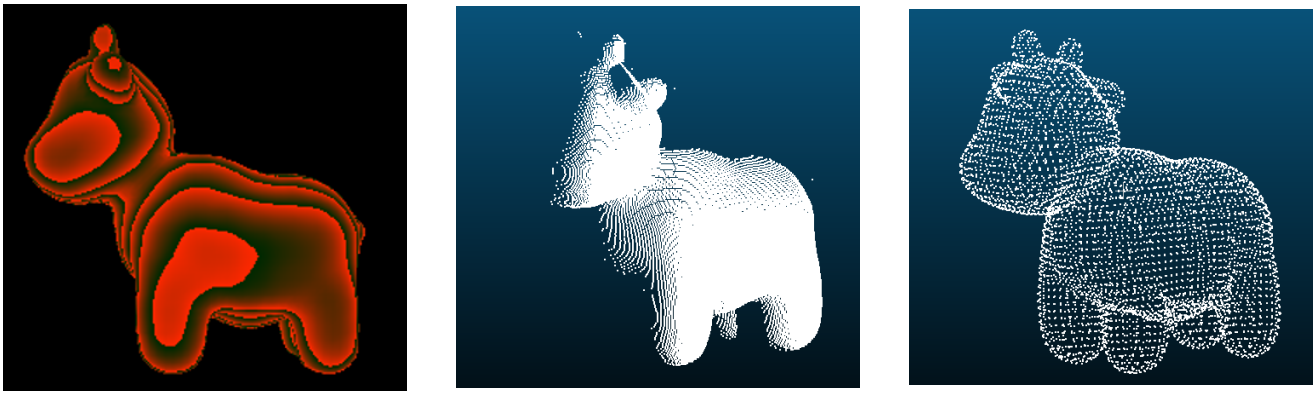

Figure 3: Left: raw depth image. Middle: transformed partial point cloud with another viewpoint. Right: ground truth complete point cloud.

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, independently 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 4. 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 4. Farthest point sampling (FPS) is another sampling strategy, which can ensure that the sampled points are far away from one another.

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 the point $p_i \in U$ that is farthest from the current point set S, and add p_i to S. This process is repeated until the point set S has enough points as we want.

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, compute the Chamfer distance (CD) and Earth Mover's Distance (EMD) of the two point clouds sampled by different methods. To give you a feeling about which metric is more sensitive to sampling, repeat the sampling and metric computation 5 times, and save the mean and variance of CD and EMD for submission. For distance calculation, use the **mean** value of distance as the result (which is different from the definition in the slides). For EMD, you may use this repository or write your own implementation.

4. **Mini NeRF Rendering (25 points):**

NeRF represents a scene with a radiance field and renders a pixel by integrating color and density values along a camera ray. In this assignment, we only focus on the **rendering core**. We provide an analytic radiance field in the starter code, so you do **not** need to train an MLP.

a)[5 points] Generate camera rays

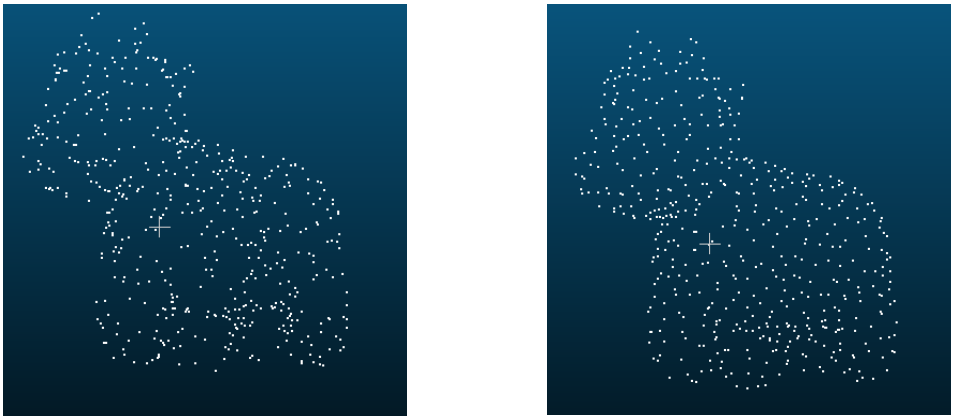

Figure 4: Left: result of uniform sampling. Right: result of farthest point sampling.

Given the image height H, image width W, focal length, and camera-to-world matrix **c2w**, generate one ray origin and one ray direction for each pixel.

b)[5 points] Sample points along rays

Uniformly sample N points between the near plane and the far plane for every ray, and obtain their 3D coordinates.

c)[10 points] Volume rendering

Given the density σ_i, color $\mathbf{c}_i$, and interval length δ_i at each sample point, implement alpha compositing:

$$\alpha_i = 1 - \exp(-\sigma_i \delta_i), \tag{1}$$

$$T_i = \prod_{j<i} (1 - \alpha_j), \tag{2}$$

$$w_i = T_i \alpha_i, \tag{3}$$

$$\mathbf{C}(r) = \sum_i w_i \mathbf{c}_i. \tag{4}$$

You should also compute a rendered depth map using the same weights.

d)[5 points] Save your render outputs

For this release, we do not distribute reference RGB/depth arrays in the student package. Save the rendered RGB image, the rendered depth map, and a few simple self-check metrics from your local run. The hidden grader will compare your saved outputs against the instructor reference.

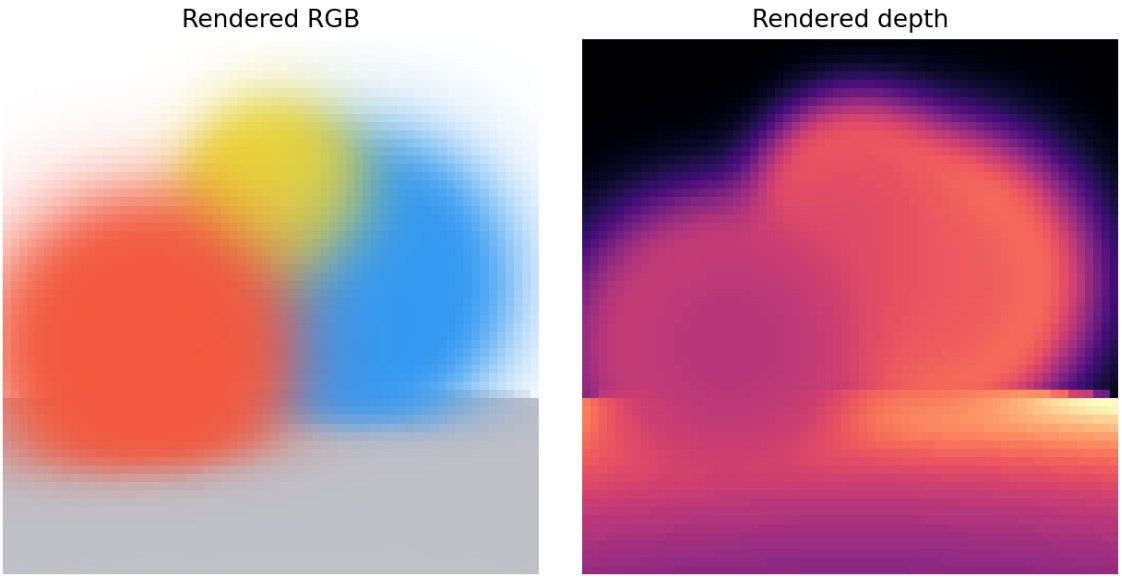

Figure 5: Reference render from the provided analytic radiance field. Left: RGB image. Right: depth map.

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
   - cumprod, https://numpy.org/doc/stable/reference/generated/numpy.cumprod.html

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

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