Problem Set 5

Due May 20, 2024 at 11:59 PM Pacific

Turn in your problem set by Gradescope. Include any code you write in your solutions as well. Mark down how much time you spent on the homework set.

1. **Camera Projections with Depth Images:** Let's familiarize ourselves with the pinhole camera as well as the very confusing fact that there are two primary conventions for camera poses. In class, we saw that we can project 3D points into 2D using the camera projection equation

$$w \begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = K \left[R_w^c \mid t^c \right] \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix},$$

that transforms 3D points in the world frame into a local camera frame using R_w^c and t^c (i.e., the world-to-camera transformation), before using the camera intrinsics matrix K to project these transformed 3D points onto the image plane. w tells us how far in the $z-$direction (in terms of focal lengths) the point (now expressed in pixel coordinates) is. Therefore, we need to divide the vector by w to find the (u, v) on the image plane at one focal length.

We can also go the opposite way, inverse projecting 2D pixels into 3D, if we know how far the pixels terminate. Note that the following equation

$$\underbrace{w\, K^{-1} \begin{pmatrix} u \\ v \\ 1 \end{pmatrix}}_{d} = \left[R_w^c \mid t^c \right] \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix},$$

on both sides represent the vector that starts at the camera origin and ends at the 3D point expressed in the local camera frame. However, given just an image, we only have access to (u, v) of a pixel, and not w. Nevertheless, having access to pixel-aligned depth t along the ray direction d allows us to find the left-hand side, that is

$$wK^{-1} \begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = t \cdot d.$$

Confusingly, there are two camera pose conventions (Figure 1): OpenGL and OpenCV. OpenCV convention is typically used for software that works with images and sensor data (e.g. OpenCV, Intel Realsense) because the (x, y) coordinates are axis-aligned with the way images are stored as a matrix. z is oriented towards the front.

OpenGL convention is used in 3D software, such as CAD and Blender. OpenGL is a more natural interpretation in the sense that (x, y) are aligned how people typically draw (x, y) in plots. z is oriented towards the back. For more information, check out `https://github.com/nex-mpi/nex-code/wiki/Pose-convention`.

Your job in this problem is to project depth images into 3D space from two viewpoints and ensure that the joint point cloud is sensible. The equations above are only suitable for OpenCV convention (and can be made OpenGL compliant through slight modifications), while the given poses are poses that transform points in the camera frame to the world frame using the OpenGL camera convention.

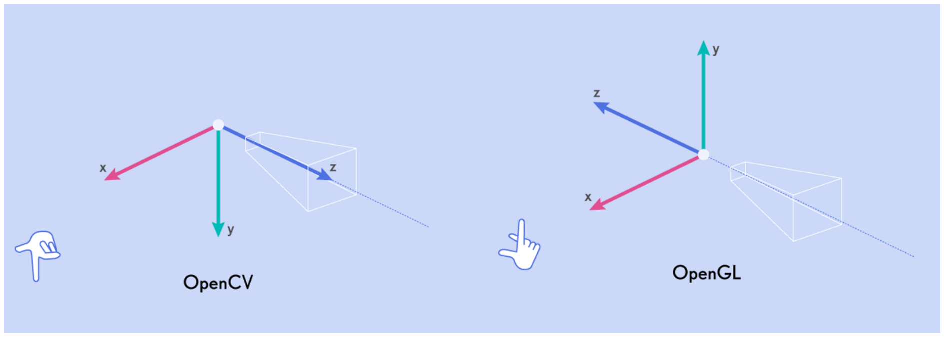

Figure 1: OpenCV vs OpenGL.

(1) Given the two RGB-D images (as .png and .npy files) and their poses (in `poses.json` in OpenGL format), first plot the RGB images and their depths.

(2) Then, project these colored images as **colored** 3D point clouds. Show visualizations of the joint colored point cloud using any 3D plotting software or library of your choice (e.g., Open3D).

(3) Report the bidirectional Chamfer distance $CD(P_1, P_2) + CD(P_2, P_1)$.

(4) Report the OpenCV equivalent of the ground-truth poses.

2. **Deriving Volumetric Rendering:** You are tasked with implementing several components of the Neural Radiance Field (NeRF) pipeline, specifically the ray-marching and the rendering equation. The NeRF model architecture and network weights are provided to you.

Before implementing anything, let's familiarize ourselves with the rendering equation, which can be viewed from a probabilistic lens. A form of Beer's law for the volumetric attenuation of light along a ray $r(t) = o + t \cdot d$ is given by

$$T(t) = \exp\left[-\int_{t_n}^{t} \sigma(\tau) d\tau\right],$$

where T is the transmittance of light (i.e., the fraction of total light that can pass from t to t_n), o is the camera origin, d is the viewing direction, t the distance along the ray, and σ the NeRF density. We can equivalently interpret T as the probability that a ray of light is *not occluded* between t_n and t, with the occlusion event being a Bernoulli random variable. To paint this equivalence, if the transmittance $T = 0.6$, then under the traditional interpretation, light radiated at t will have its intensity reduced by 40% by the time it reaches t_n. Under the probabilistic interpretation, if we shoot N light rays from t to t_n and 40% of the rays never reach t_n while 60% reach t_n, then the total light transmitted will also be 60%.

Using the probabilistic interpretation of transmittance, the probability that a ray of light is occluded between t_n and t is

$$F_{\mathcal{T}}(t) = Pr(\mathcal{T} \leq t) = 1 - T(t) = 1 - \exp\left[-\int_{t_n}^{t} \sigma(\tau) d\tau\right]$$

where $\mathcal{T}$ is a random variable that is the distance along the ray at which a ray stops. It should be noted that $Pr(\mathcal{T} \leq t)$ is the *Cumulative Distribution Function* of $\mathcal{T}$.

(1) Express the **expected attribute** (e.g., color, depth, embeddings) of a ray $\mathbb{E}_t\left[c(r(t))\right]$ as a function of $c(r(t))$, $\sigma(r(t))$, t, t_n, and t_f. Your expression should match the *intergral* form of the NeRF rendering equation.

Hint: The derivative of a CDF is the PDF, that is $p(x) = \dfrac{dF(x)}{dx}$ *for a PDF labeled as* $p(x)$ *and CDF labeled as* $F(x)$.

(2) Using the integral form of the NeRF rendering equation, derive the *quadrature* form of the rendering equation

$$C = \sum_{i=0}^{N} c_i \exp\left[-\sum_{j=0}^{i} \sigma_j \delta_j\right] (1 - \exp[-\sigma_i \delta_i]).$$

Hint: Assume that σ and c is piecewise-constant between sample points i and $i+1$. The distance between those points is δ_i, where δ_i need not be the same throughout the samples, thereby requiring the index i.

(3) Complete the `get-rays` function (in `render-helpers.py`) that takes the camera calibration matrix K and camera pose (R, t) to output the camera origins and the ray directions coming out of each pixel. You may reuse your code from Problem 1.

(4) Complete the `raw2outputs` function (in the same file) that takes the outputs of the neural network and returns the RGB and depth map.

(5) Plot renders of the NeRF at the given poses in the provided code. For each pose, there should be an RGB render and a depth render. The plotting is already implemented for you. To run the code, simply run `run.py`.