Introduction to Multimodal Learning
Problem Set 3

Due: June 11, 2024

May 27, 2024

In this homework, you will implement the volume rendering method in neural radiance field (NeRF) model. You will then utilize this method to render multi-view images from a pretrained NeRF model.

1 Requirements

Please submit a zip file containing the following files or folders:

- A implemented version of 'Problem3_NeRF' that contains your implementation of nerf model.
- A result folder that save your visualization results in 'images' folder.
- A report.pdf file briefly summarizing your results and observation. It should include the results figures. You may also include any other findings you think are relevant.

Setup. Install Anaconda. Then you can set up the environment with the following commands:

```
conda env create -f environment.yml
conda activate nerf
```

Code Structure There are five main components of our codebase:

- The camera: pytorch3d.CameraBase
- The ray data structure: RayBundle in ray_utils.py
- The scene: SDFVolume and NeuralRadianceField in inplicit.py
- The sampling schedule: StratifiedSampler in sampler.py
- The renderer: VolumeRenderer in renderer.py

To perform volume rendering, you need to implement the following procedures:

- **Ray sampling from cameras.** You will implement some functions in ray_utils.py to generate rays in the world coordinate system from a particular camera.
- **Point sampling along rays.** You will implement the StratifiedSampler class to generate sample points along each ray.
- **Rendering.** You will implement the VolumeRendering class to evaluate a volume function at each sample point along a ray, and then aggregate the evaluations to form rendering results.

Problem 1: Ray Sampling

In this problem, you are required to implement the ray sampling function. Please look at the **render_images** function in **main.py**. The function enumerate each predefined camera view and render an RGB image from the camera. The first step for the rendering is to acquire all the pixels from an image and generate corresponding camera rays in the world coordinate system:

```
xy_grid = get_pixels_from_image(image_size, camera)
ray_bundle = get_rays_from_pixels(xy_grid, image_size, camera)
```

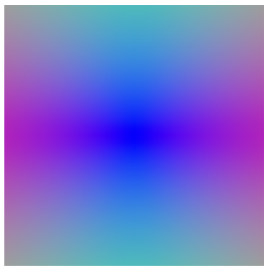
Requirements. You need to implement the **get_pixels_from_image** and **get_rays_from_pixels** in **ray_utils.py**. The **get_pixels_from_image** function generates pixel coordinates in the range $[-1, 1]$. The **get_rays_from_pixels** function generates rays from pixels and transforms the rays from the camera space to the world space.

You can run the code for this problem by the following command:

```
python main.py --config-name=box
```

After you have implemented these functions, please verify that your output (in **images** folder) matches the reference output results in **reference_images** folder. The visualization of grid and ray should look like the following picture in **reference_images/grid_vis.png** and **reference_images/ray_vis.png**

(a) Grid



(b) Ray

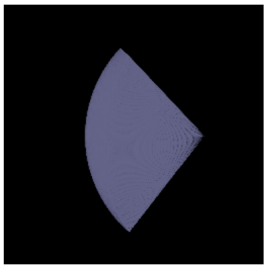
Problem 2: Point Sampling

The second step for the rendering is to sample multiple 3D points along a ray with a uniform sampling strategy.

Requirements. You need to implement the forward method of **StratifiedSampler** in **sampler.py** with the following procedure:

- Uniformly generate a set of distances between $[near, far]$.
- Use these distances to compute point offsets from ray origins **RayBundle.origins** along ray directions **RayBundle.directions**.
- Store the sampled distances and points in **RayBundle.sample** points and **RayBundle.sample** lengths.

After you have finished this method, you can visualize the **point_vis.png** result in **images** folder and check the reference output results in **reference_images** folder. The results should look like:



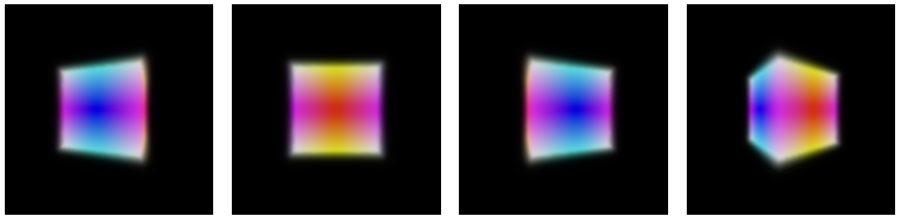
Problem 3: Rendering.

In this part, you will need to implement the volume rendering. In specific, you need to implement the forward method of **VolumeRenderer** in **renderer.py**. Two functions **_compute_weights** and **_aggregate** are used in the forward method. The **_compute_weights** function computes the weight $w_i = T(x_0, x_i)(1 - e^{-\sigma_i \Delta t_i})$ for each sample point, where x_0 is the ray origin, $x_{i \geq 1}$ is the sample point, σ is the density, Δt is the length of current ray segment and $T(x_0, x_i) = T(x_0, x_{i-1}) e^{-\sigma_{i-1} \Delta t_{i-1}}$ is the transmittance. Note that $T(x_0, x_1) = 1$. The **_aggregate** function aggregates emissions by a weighted sum:

$$L(x, \omega) = \sum_{i=1}^{n} w_i L_e(x_i, \omega) \tag{1}$$

where ω is the ray direction, L_e is the emission, and L is the final rendered color. You will also render a depth map in addition to color from a volume.

Requirements-1: Implement the **VolumeRenderer** in **renderer.py**.
After you have implemented the method, please run the command in Problem1. Your rendering results will be saved to **images/render_cube.gif**. The results should look similar to the following figure.



Requirements-2: Run the following command and check your rendering results on 3D lego model. The results should look similar in the following figure. The nerf model is loaded from a pretrained checkpoint, we only do the rendering part.

```
python main.py --config-name=nerf_lego_render
```

Optional: (Feel free to try it or not.) We have also provide a nerf training code in **train_nerf** in **main.py**. You can train a simple NeRF network from scratch by the following command:

```
python main.py --config-name=nerf_lego
```