Homework 2

1 Volume Rendering[7pt]

Problem 1. In this exercise, you will recap what you have learned in Lecture 6, and implement a simple volume rendering method. You will then utilize this method to render multi-view images from a pre-trained NeRF.

We provide a codebase in `Problem1` folder and what you need to do is to complement some core codes of volume rendering.

Setup. First install Miniconda or Anaconda. Then you can set up the codebase with the following commands:

```
conda env create -f environment.yaml
conda activate hw2_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 routine: `StratifiedSampler` in `sampler.py`
- The renderer: `VolumeRenderer` in `renderer.py`

`StratifiedSampler` provides a simple ray marching method that samples multiple points along a ray traveling through the scene. The sampler and the renderer jointly make up a rendering pipeline. Like traditional graphics pipelines, this rendering procedure is independent of the scene and camera.

The scene, sampler and renderer are all packaged together in the `Model` class in `main.py`, where the forward method executes a rendering process with a scene and a sampling strategy as input.

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

- **Ray sampling from cameras.** You will complement some functions in `ray_utils.py` to generate rays in the world coordinate system from a particular camera.

- **Point sampling along rays.** You will complement the `StratifiedSampler` class to generate sample points along each ray.
- **Rendering.** You will complement the `VolumeRendering` class to evaluate a volume function at each sample point along a ray, and then aggregate the evaluations to form rendering result.

1. [Programming Assignment] Ray sampling. 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:

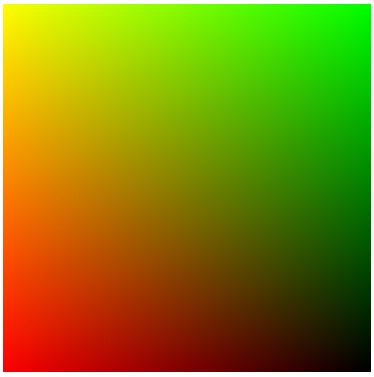
```
xy_grid = get_pixels_from_image(image_size, camera) # TODO (1):
    implement in ray_utils.py
ray_bundle = get_rays_from_pixels(xy_grid, image_size, camera) #
    TODO (1): implement in ray_utils.py
```

   You need to implement `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.

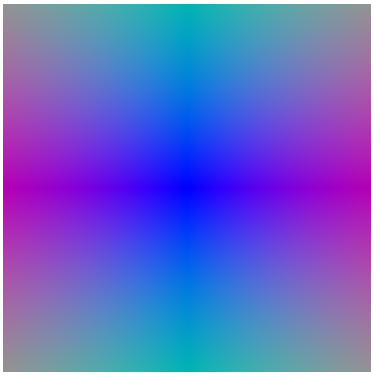
   You can run the code for this problem by the following command:

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

   After you have implemented these functions, please verify that your output matches the TA's output by visualizing `xy_grid` and `ray_bundle` in `render_images`. The visualizations of grid and ray should look like `ta_images/grid_vis.png` and `ta_images/ray_vis.png`:



(a) Grid



(b) Ray

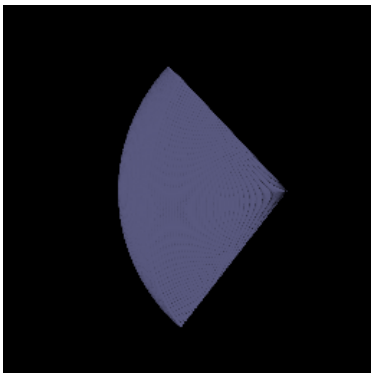
2. [Programming Assignment] Point sampling. The second step for the rendering is to sample multiple 3D points along a ray with a uniform sampling strategy. You need to implement the forward method of `StratifiedSampler` in `sampler.py` with the following routine:

   (a) Uniformly generate a set of distances between [near, far].

   (b) Use these distances to compute point offsets from ray origins `RayBundle.origins` along ray directions `RayBundle.directions`.

   (c) Store the sampled distances and points in `RayBundle.sample_points` and `RayBundle.sample_lengths`.

   After you have finished this method, you can visualize the result by first filling out the relevant codes in `render_images` and then running the command in Problem 1-1. The visualization of sample points should look like `ta_images/point_vis.png`:



3. Theory of transmittance calculation. Here we come to the transmittance calculation, which is the core part of volume rendering. Considering a ray going through a non-homogeneous medium shown in Figure 1, please compute the transmittance from x_4 to x_1 and write down your result in your report.

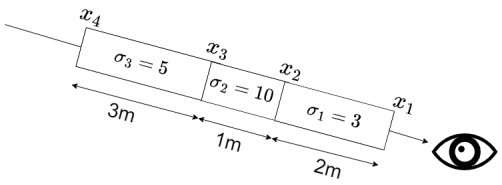


Figure 1: An example of a non-homogeneous medium.

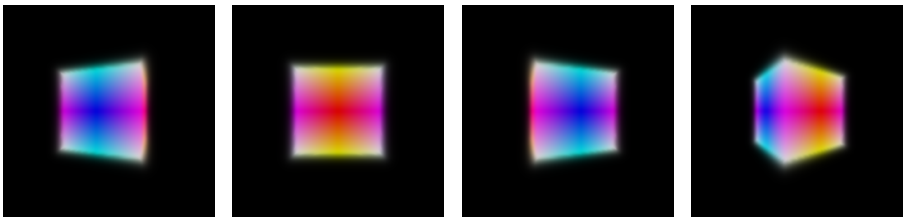
4. [Programming Assignment] Rendering. Let us come back to the implementation of volume rendering. The final step for the rendering is to integrate emissions along a ray to form a color observation at a pixel. You need to complement the forward method of `VolumeRenderer` in `renderer.py`. Two functions `_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 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.

After you have implemented the method, please finish the `render_images` function and run the command in Problem 1-1. Your rendering result will be written to `images/render_cube.gif`:



Up to now, you have finished this problem. In addition to the previously rendered cube that is represented by a very simple signed distance field, TA also provides a pre-trained NeRF of a Lego model. Run the following command:

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

Your rendering result will be written to `images/render_nerf.gif`, it should look like:

Please attach all of your rendering results to your submission file.

Note: We provide the NeRF training code under `train_nerf` in `main.py`. Based on your implementation of volume rendering, you can train a simple NeRF network by the following command:

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

Feel free to try it!

2 Single Image to 3D[7+2pt]

Problem 2. In this problem, you will design a neural network for the task of single image to 3D with two different 3D losses and analyze the effect of these losses. Different from Problem 1, you need to implement the network design and the training and evaluation routines on your own (fyi: We provide a codebase under `Problem2_framework` folder, you can either utilize this codebase or implement from scratch).

The problem here is to estimate the object's point cloud from its single image. You can use what you learn in the class to design your neural network backbone that first encodes the image to a high-level feature vector and then decodes the feature vector to a 3D point set as the final estimation.

The loss design for the network could be various. In addition to the Chamfer Distance(CD) loss we mentioned in the class, we will introduce the Hausdorff Distance(HD) loss in Problem 2-1. You will implement these two losses in your network training and compare their performances.

We focus on point cloud estimation for specific shape primitives with simple geometries to reduce the shape uncertainty and reduce the difficulty. We provide a synthetic dataset in `Problem2` folder and you need to select data from it for both training and inference procedures in Problem 2-2. The dataset contains 100 cubes with diverse scales and colors. For each cube, 16 rendered images from various views and corresponding ground-truth point clouds are included.

1. (3D losses) Let P and Q be two finite sets of points in 3D space, namely $P = \{p_i\}_{i=0}^{n-1}$ and $Q = \{q_j\}_{j=0}^{m-1}$ where $p_i, q_j \in \mathbb{R}^3$. We use $d(p_i, q_j) = ||p_i - q_j||_2$ to represent the Euclidean distance between p_i and q_j. The Hausdorff Distance between P and Q is defined as

$$h(P,Q) = \max\{d(P,Q), d(Q,P)\}, \tag{2}$$

where $d(P,Q) = \max_{p_i \in P} [\min_{q_j \in Q} d(p_i, q_j)]$ and $d(Q,P) = \max_{q_j \in Q} [\min_{p_i \in P} d(q_j, p_i)]$.

- (a) [Extra credit] Prove that the Hausdorff Distance is a metric, i.e. it has properties of positive, reflexive and triangular inequation:
  - Positive: $h(X,Y) \geq 0$.
  - Reflexive: $h(X,Y) = 0 \rightarrow X = Y$.
  - Triangular inequation: $h(A,C) \leq h(A,B) + h(B,C)$.
- (b) [Programming Assignment] Implement Hausdorff Distance(HD) loss and Chamfer Distance(CD) loss in python. The implementation of these two losses should be differentiable for network training.

2. [Programming Assignment] (Network design) Given an RGB image, you will design a neural network to predict the object point cloud. You need to select a loss function implemented in Problem 2-1 to measure the difference between the predicted and the ground-truth point clouds and thus train the network. For the network design, a simple encoder-decoder network that mentioned in the class could be competent. The encoder leverages CNNs for encoding the image input to a feature vector representing the whole image. The decoder predicts the point cloud from the feature vector. Various effective decoder designs can be found in https://arxiv.org/pdf/1612.00603.pdf.

   Although our dataset is generated by rendering from the known object shape in the synthetic scenario, sometimes we may not have the CAD model and will need to capture the object geometry in the real world to train the network. The multi-view RGB or RGB-D reconstruction algorithms could be used to capture the object geometry, but the reconstruction result is noisy due to the sensor noise and the error of the algorithms. Therefore, we provide a dataset with clean point clouds that corresponds to the most ideal synthetic scenario, and a dataset with noisy point clouds that simulates the real-world noise to some extent.

   - (a) Implement your neural network in pytorch. Besides the network architectures mentioned in the class, you are encouraged to design different networks that you consider is effective.
   - (b) Select at least 10 cubes for training and at least 10 cubes for evaluation from the dataset in `Problem2` folder. Use HD loss and CD loss respectively to train the network in the `clean` dataset. Evaluate the networks trained by HD loss and CD loss with the same loss in the training. Report the loss values during network training and evaluation and visualize the predicted point clouds of each network.

- (c) Use HD loss and CD loss respectively to train the network in the `noisy` dataset and compare the results with those from the `clean` dataset.
- (d) From what you have discovered, analyze why HD loss is less used in practice.

Note: The file structure of the dataset in `Problem2` folder is shown below:

- `cube_dataset` – The dataset folder.
  - `clean` – The dataset with clean point clouds.
    - `0` – The data of cube 0. The 16 data pairs in the same folder is rendered from the same cube.
      - `0.png` – The image from view 0 of cube 0.
      - `0.ply` – The point cloud in the camera coordinate system from view 0 of cube 0.
      - ...
      - `15.png` – The image from view 15 of cube 0.
      - `15.ply` – The point cloud in the camera coordinate system from view 15 of cube 0.
    - ...
    - `99` – The data of cube 99.
  - `noisy` – The dataset with noisy point clouds. The structure is the same as the `clean` dataset.

3 Surface Reconstruction[6pt]

Problem 3. In this problem, you will design a routine to reconstruct the surface of an object from its point cloud, using what you have learned in the class.

When we need to transfer a point cloud to its mesh surface, the implicit function is always treated as an intermediate representation. In this problem, the position and normal of each point are given, thus we are able to easily discriminate the inner and outer space of the surface and design a Signed Distance Function (SDF) as the intermediate representation to build up the bridge between the point cloud and the mesh surface.

Finding a resonable SDF from the point cloud could be solved by interpolation methods, here we use the Moving Least Squares (MLS) algorithm mentioned in the class. We first construct a constraint at each point of the object point cloud in Problem 3-1. We then voxelize the object in 3D space and estimate the object SDF at each voxel vertex with the MLS interpolation algorithm in Problem 3-2. The SDF of all the vertices will finally form the implicit function of the object's surface.

Estimating the mesh surface from the the object's implicit function can be solved by the Marching Cube algorithm with manually designed surface shape for each cube. We leverage PyMCubes, an off-the-shelf Marching Cube implementation, to obtain the reconstructed object mesh in Problem 3-3.

1. [Programming Assignment] MLS constraints. Given an object point cloud bunny.ply containing normal vectors, we define an implicit polynomial function $f(p), p = (x, y, z)$ in the whole 3D space, where the input point cloud fits the zero-level set of f (i.e. for every point p_i in the point cloud, we have $f(p_i) = 0$). The normal direction of f should be consistent with the normal vectors of the point cloud.

   Given the value f_i on some specific constraint points, f can be estimated by solving an optimization problem:

   $$f = \arg\min_{\phi} \sum_{i} \|\phi(p_i) - f_i\|^2, \tag{3}$$

   where $\phi(p)$ is a polynomial function $b_k(p_i)^T a$ for a degree k polynomial. The variable is the polynomial coefficients a.

   The first step is to design constraints p_i. Intuitively, the point cloud should be part of the constraints with $f = 0$. However, if there are no other constraints, most interpolation methods will produce a zero-valued function as the result. To address this issue, since the normal vector of a point in the point cloud is known, we can approximate the value of f along the normal and thus add more constraints.

   We can use the following process to generate all the constraints:

   - (a) For each point p_i in the point cloud, add a constraint $f(p_i) = 0$.
   - (b) Select an ϵ. For each point p_i, compute $p_{i+N} = p_i + \epsilon n_i$, where n_i is the normal vector of p_i. Check whether p_i is the closest point to p_{i+N}, if not, decrease ϵ and recompute p_{i+N} until the closest point is p_i. Add a constraint $f(p_{i+N}) = \epsilon$.
   - (c) Repeat the same routine in (b) with $-\epsilon$.

   Finally, we have designed $3N$ constraints for the implicit function f. Visualize all the points in these constraints and use different colors for different function values.

2. [Programming Assignment] MLS interpolation. Given the constraints computed in Problem 3-1, we then leverage MLS interpolation to construct the implicit function. At each point p in the 3D space, $f(p)$ can be estimated by a polynomial function defined on the nearby constraint points. The information of various constraint points is aggregated by a weighted sum, where the weight for each point is determined by the point's position.

(a) Voxelize the 3D space with fixed voxel size and fixed number of voxels. Report your voxel size and voxel number.

(b) Compute the implicit function value at each voxel vertex using MLS interpolation and build up the implicit function. You can design your own weight function θ in MLS. You need to implement the $0, 1, 2$ polynomial basis functions and use them respectively to define f and use them to build up the implicit function with MLS. You may properly use a specific number of nearby constraint points for each grid point in MLS to speed up the computing process.

3. [Programming Assignment] Marching Cube. Once we have estimated the implicit function of the object, the last thing is to transfer the implicit function to the final mesh surface. Here we use an off-the-shelf implementation of the Marching Cube algorithm to solve this.

   Install PyMCubes with the following commands:

```
pip install pycollada
pip install --upgrade PyMCubes
```

   More information can be found in pycollada and PyMCubes.

   (a) Use the implicit function defined by 0 polynomial basis function f_0 and 1 polynomial basis function f_1 respectively, visualize their reconstructed mesh surfaces and compare them.

   (b) Use the implicit function defined by 2 polynomial basis function f_2, and find out whether it can reconstruct a better mesh surface, why or why not?