Homework 4 Part 1: NeRF

EECS 498-014 Graphics and Generative Models

2024 Fall

Due date: 11:59 P.M., Nov 13, 2024

Contents

1 Objectives **2**

2 Instructions **2**

2.1 Downloading the Code . . . 2

2.2 Environment Setup . . . 2

2.3 Get access to GreatLakes . . . 3

2.4 Conda Environment Setup . . . 3

3 Overview **4**

4 Helper Functions **5**

4.1 Setup data . . . 5

4.2 Generate Rays . . . 6

4.3 Sampling 3D Points . . . 7

4.4 Positional Encoding . . . 7

4.5 Exclusive Cumulative Product . . . 8

5 NeRF Model **8**

5.1 Volumetric Rendering . . . 8

5.2 Testing your rendering . . . 9

5.3 Training Your NeRF! . . . 10

5.4 Rendering from different viewpoints . . . 11

6 Optional: Extract Mesh from NeRF **11**

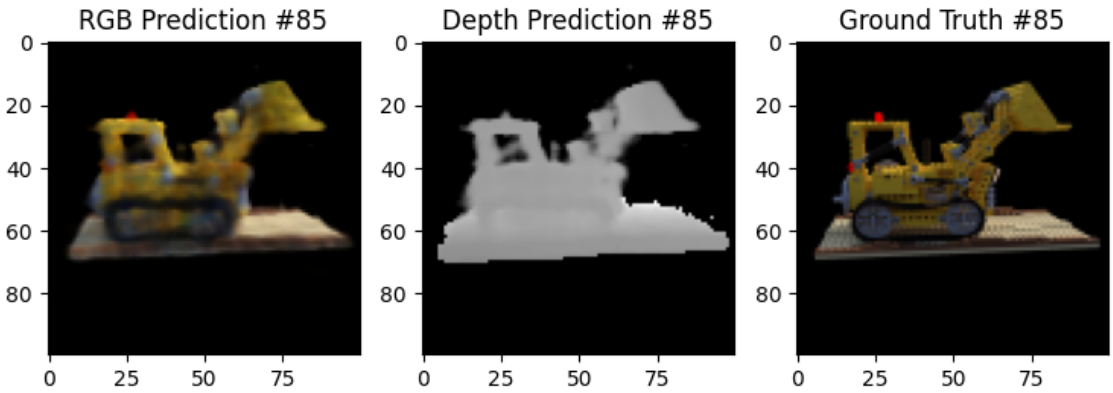
7 Submission and Grading **11**

1 Objectives

This assignment is a 2 weeks project on implementing Neural Radiance Fields (NeRFs). The learning objectives of this assignment are to:

- You will gain familiarity with 3D Reconstruction
- You will obtain hands-on experience in implementing NeRFs, one of the most popular approaches in 3D reconstruction.

After completing the tasks, you will be able to reconstruct 3D objects like the following figure.



2 Instructions

2.1 Downloading the Code

You can download the starter code here

2.2 Environment Setup

In this homework, we will use Python as the programming language and Pytorch as our deep learning framework. To support GPU usage, we provide GreatLakes (GL) server.

For people who haven't used the GL before, please first sign up at this website.

Importantly, you shouldn't use the GPU for any other purposes outside of the class. You will get zero grades for the homework if you violate this rule.

2.3 Get access to GreatLakes

GreatLakes (GL) is a high-performance computing cluster in UMich, which provides powerful computation resources. Please follow these steps to connect to GreatLakes.

1. Before connecting to the GreatLakes Server, make sure you are using the campus network or have connected to the network through UMVPN.

2. From your local terminal, type `'ssh ${UNIQUE_NAME}@greatlakes.arc-ts.umich.edu'` to connect to login node of GreatLakes, where `'${UNIQUE_NAME}'` is your UMICH unique name. It requires your UMich account password and Duo Certification via Duo Push or passcodes. More details can be found here. When you log in to VSCode by ssh, you can open the folder to: `/scratch/eecs498s014f24_class_root/eecs498s014f24_class/Your_UniqName`.

3. GreatLakes uses SLURM to manage the computation resource.

   To query compute nodes with GPUs, in the login node, you can execute the following command:

   ```
   salloc --cpus-per-task=4 --gpus=1 --mem-per-gpu=44GB --partition=spgpu --time=0-02:00:00
   --account=eecs498s014f24_class
   ```

   We know that it is hard to copy from the PDF, so we **provide a copy of the execution codes in the README.md**. This command will allocate a **A40 GPU with 42 GB memory for 2 hours**. All the program execution inside this terminal window will be accessible to the GPU (not for other terminal windows). If you close the window of the terminal, the GPU will be released and needs to be requested again. If there are multiple students in this course request at the same time, other students may need to wait. The **maximum number of GPUs that can be allocated by all the students in this course is 10**. Thus, please **start the homework earlier**. The account is shared among all students and the quota for each individual is around **300 hours until the end of the semester**. The storage allocated for each student is around **80GB** (including the conda environment space).

   Note that the terminal window with GPU allocated may be slow in executing the git and conda env commands. We recommend opening another terminal window for these CPU executions.

   Please note that GreatLakes is a shared computation cluster and there are many others using it. Please be considerate to others, including but not limited to:

   - Release the GPU after you are done with your work.
   - Don't run computationally heavy jobs in the login node, which will lead to the login node being slow.

2.4 Conda Environment Setup

We use conda to manage the Python package.

1. Load Conda module on GreatLakes:

```
module load python3.9-anaconda
```

2. Create a Conda environment

```
conda create -n nerf python=3.9
```

3. Activate conda environment

```
conda activate nerf
```

4. Install the required packages:

   Please go to `Readme.md` and follow the instructions there.

For Pytorch, we use version 2.4.1 with cuda 12.1.

3 Overview

In this problem set, you will implement Neural Radiance Fields (NeRF), a groundbreaking technique in computer vision and graphics for 3D scene reconstruction and novel view synthesis.

As shown in Figure 3, NeRF represents a 3D scene as a continuous function using a Multi-Layer Perceptron (MLP). This function maps from a 5D input vector (3D spatial location x,y,z and 2D viewing direction θ, ϕ) to a 4D output vector (RGB color and opacity σ). The key aspects of NeRF are:

1. Input: 5D coordinate (x, y, z, θ, ϕ)

- (x, y, z): 3D spatial location in the scene
- (θ, ϕ): 2D viewing direction

2. Output: 4D vector (r, g, b, σ)

- (r, g, b): RGB color at the given location and viewing direction
- σ: opacity or density at the given location

3. Training Data: A set of 2D images with known camera poses

4. Novel View Synthesis: After training, NeRF can generate new views of the scene from arbitrary camera positions

In this assignment, you will implement NeRF from scratch and build the complete training pipeline. The main steps include:

1. **Ray Generation**: Create rays for each pixel based on camera poses and intrinsic matrix.

2. **Sampling and MLP Query**: Sample points along these rays in 3D space and query the MLP for color and opacity on these points.
3. **Volume Rendering**: Given the opacities and colors of points along a ray, calculate the ray's RGB value through a weighted integration of the colors.
4. **Loss Calculation and Training**: Compute the loss between the rendered image and the corresponding ground truth image, and update the MLP parameters through backpropagation.

By completing this assignment, you'll gain a deep understanding of how NeRF works, including its core concepts like volumetric rendering, positional encoding, and the integration of classical computer graphics techniques with deep learning.

Functions to implement:

1. Function `get_rays(H, W, K, c2w)`
2. Function `positional_encoder(x, L_embed=6)`
3. Calculation of the 3D points sampled along the rays
4. Functions for cumulative product `cumprod_exclusive`
5. Calculation of `rgb_map`, `depth_map`.

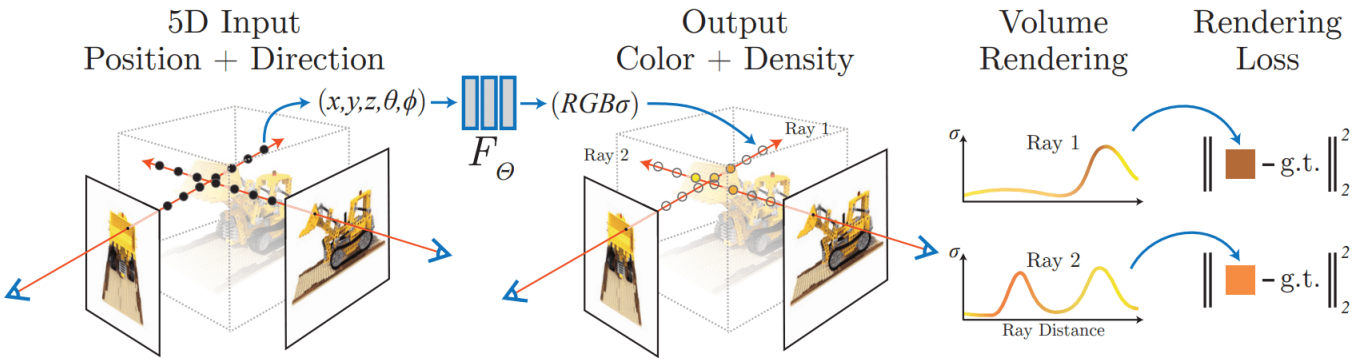


Figure 1: **Whole pipeline of NeRF.** The image is credited to the original NeRF paper.

4 Helper Functions

4.1 Setup data

NeRF takes a set of posed RGB images as input and reconstructs the corresponding 3D model. To minimize training time, we will use a tiny dataset that contains only 106 images and their associated camera poses. We already provide

the dataset `tiny_nerf_data.npz` in the github repo.

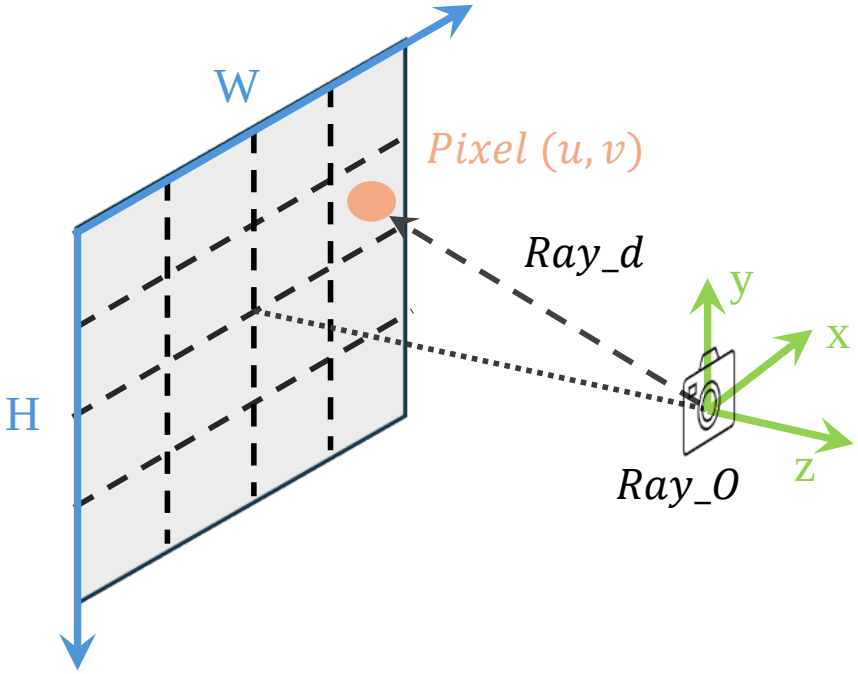


Figure 2: **Illustration of ray generation of NeRF.**

4.2 Generate Rays

Ray generation is a fundamental component of NeRF. For each pixel in an image, a ray is cast from the camera center through the corresponding pixel. Each ray is defined by two key components:

1. **Origin** (`rays_o`): The starting point of the ray, typically the camera center.

2. **Direction** (`rays_d`): The vector indicating the ray's direction.

As shown in the Figure, we utilize the function `get_rays` to generate these rays, which form the basis for subsequent sampling and rendering processes.

Task 1: Implement the `get_rays` function in `nerf_model.py`, which could be finished by following the steps below:

Step 1: Obtaining the pixel coordinates `(u, v)` for all pixels on the image plane using `torch.meshgrid`. Here, `u` ranges from `0` to `W-1` and `v` ranges from `0` to `H-1`.

Step 2: Transforming the pixel coordinates `(u, v)` to the camera coordinate system using the camera intrinsic matrix `K`. The camera origins `rays_o` should be `(0, 0, 0)` in the camera coordinate system. The ray directions `rays_d` are calculated as the difference between the pixel's 3D coordinates in the camera coordinate system and the camera origins.

Note (Camera Coordinate System). We are using the following conventions for the camera coordinate system (the green axes in the figure):

- **x-axis**: right
- **y-axis**: up
- **z-axis**: away from the screen. The camera is looking at the -z direction.

Please note that pixel coordinate `(u, v)` is in the image coordinate system, where the origin is at the top-left corner of the image. The x-axis points to the right, and the y-axis points down.

The image plane is placed at $z = -1 * \text{focal length}$, which allows you to obtain the 3D coordinates of the pixel in the camera coordinate system.

Step 3: Convert the ray directions from camera coordinates to world coordinates, by simply applying the camera transformation matrix. Note the `pose` is the camera-to-world transformation matrix.

Step 4: Get the ray origins `rays_o`, which are the `translation` part of the `pose` matrix. We provide the code for this step.

4.3 Sampling 3D Points

After generating rays, we need to sample 3D points along the rays, and further query the MLP for color and opacity. Given the ray origins `rays_o` and ray directions `rays_d`, we can obtain the 3D points `pts` by:

$$(x, y, z) = \textbf{rays\_o} + t \times \textbf{rays\_d} \tag{1}$$

where t is a scalar, sampled from `near` and `far`.

4.4 Positional Encoding

Given the sampled 3D points, we query the MLP for color and opacity based on the 3D points and view directions. It is shown that simply using the 3D points and view directions as input to the MLP is not sufficient for representing high-frequency details, which limits the expressiveness of the MLP. To address this, we employ Fourier positional encoding to model the high frequencies.

Given an input p (p is a scalar), we will encode it into:

$$\gamma(p) = (p, \sin(2^0 p), \cos(2^0 p), \sin(2^1 p), \cos(2^1 p) \cdots, \sin(2^{L\_embed-1} p), \cos(2^{L\_embed-1} p)) \tag{2}$$

Task 2: finish the missing code in function `positional_encoder`.

4.5 Exclusive Cumulative Product

Here we also need to implement a helper function called `cumprod_exclusive`, which computes the exclusive cumulative product of a tensor along its last dimension.

Given a tensor with value $[x_0, x_1, \ldots, x_{N-1}]$, the function should return a new tensor with value $[1, x_0, x_0 \cdot x_1, x_0 \cdot x_1 \cdot x_2, \ldots, x_0 \cdot x_1 \cdot x_2 \cdot \cdots \cdot x_{N-1}]$ along the last dimension.. **Exclusive** means that the cumulative product at each element does not include the element itself. We will use this function in the later volume rendering process.

Task 3: Implement the `cumprod_exclusive` function, which would be used later.

5 NeRF Model

Neural Radiance Fields (NeRF) utilize a multi-layer perceptron (MLP) as a continuous function to represent the 3D scene. Specifically, given a 3D position $\boldsymbol{x} = (x, y, z)$ and view directions $\boldsymbol{v}$ as inputs, the MLP $\boldsymbol{F}$ outputs the RGB color and opacity σ at the corresponding 3D position.

$$(\mathbf{RGB}, \sigma) = \boldsymbol{F}(\boldsymbol{x}, \boldsymbol{v}) \tag{3}$$

The module `VeryTinyNerfModel` implements this MLP for the NeRF model. To optimize training time, we will employ a compact MLP architecture as our network. This MLP takes the output from the positional encoder as input and predicts both the opacity and RGB values.

5.1 Volumetric Rendering

To render an image from the NeRF model, we first need to sample 3D points along the rays. The rays can be calculated based on the camera poses and intrinsic matrix, as implemented in `get_rays`.

Along each ray, we sample N points from *near* t_n to *far* t_f. To enhance the representation resolution, we employ a stratified sampling approach, where the interval $[t_n, t_f]$ is divided into N evenly spaced bins, and 3D points are uniformly sampled at random from within each bin:

$$t_i \sim U[t_n + \frac{i-1}{N}(t_f - t_n), t_n + \frac{i}{N}(t_f - t_n)] \tag{4}$$

With the sampled t_i, we can compute the corresponding 3D position in world space using the ray origins $rays\_o$

and ray directions $rays\_d$. Subsequently, the MLP processes these positions and view directions (both with positional encoding) to output the colors $\boldsymbol{c}_i$ and opacity σ_i.

The final pixel color $\hat{C}(\boldsymbol{r})$ for ray $\boldsymbol{r}$ is calculated by the volumetric rendering equation:

$$\hat{C}(\boldsymbol{r}) = \sum_{i=1}^{N} T_i(1 - \exp(-\sigma_i \delta_i))\boldsymbol{c}_i \tag{5}$$

where

$$T_i = \exp(-\sum_{j=1}^{i-1} \sigma_j \delta_j) \tag{6}$$

and $\delta_i = t_{i+1} - t_i$ is the distance between adjacent samples.

Task 4: Now you are expected to finish the function `render` in `nerf_model.py` and implement the volumetric rendering equation. It includes two parts.

Part 1: Sample points along the rays given the ray origins `rays_o`, ray directions `rays_d` and value t. In the code, we use z to represent the t_i in the above equation, where z is a tensor with shape `[ray_batch_size, num_samples]`.

Part 2: Given the σ_i and $\boldsymbol{rgb}_i$ of points along the rays, which are outputs from the MLP, calculate the rays' RGB value through a weighted integration of the colors.

Hint1: Feel free to use the implemented `cumprod_exclusive` function to calculate the transmittance T_i.

Hint2: $\exp(a + b) = \exp(a) \cdot \exp(b)$. Think about how to use it with `cumprod_exclusive`.

Hint3: You should be comfortable to set δ_N to be a very large value, e.g. 1e10, which is the distance between the last sample to a point at infinitely far away.

5.2 Testing your rendering

For the sake of easy debugging, we provide you with a pretrained weight and the inference code. By running `python train.py`, it will automatically download the pretrained weight and first test the rendering result, which results are saved as `rendering_test_results_provided_model`. Before running the code, you are expected to finish the `TODO` code block inside the `main` function in `train.py`, which requires one line of code calling the `render` function you have implemented. You should expect something like the following: The results look blurry as we only train with limited data and time.

> **Note.** `train.py` File Please take a look at `main` function in `train.py`. It contains the whole pipeline, including visualization of dataset, testing the rendering function with the provided model, and training the model by yourself. Feel free to commit unnecessary blocks when you are debugging. For instance, you should feel comfortable committing the code block after testing your rendering when you are working on this part.

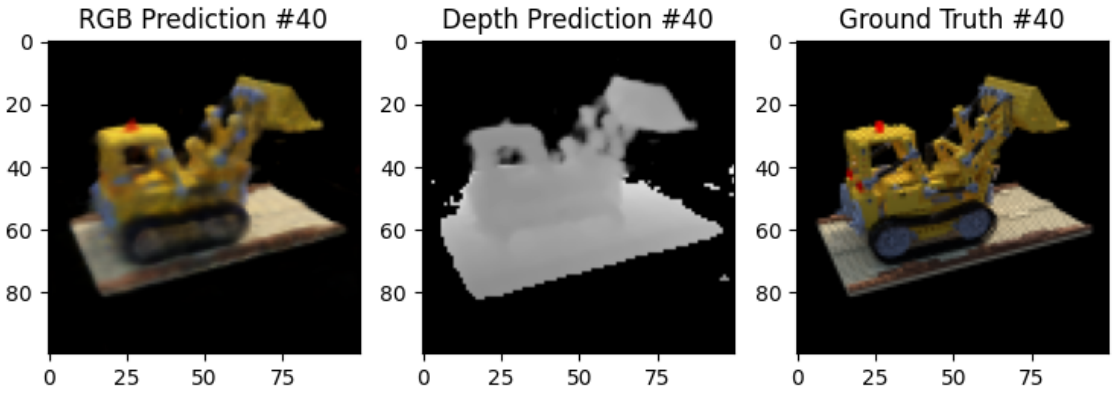


Figure 3: **Illustration of testing renderer results.**

5.3 Training Your NeRF!

Now we are ready to train your NeRF model! To train the model, you need to first finish the `train` function in `train.py`. **Task 5** Finish the `train` function.

It involves the following steps:

1. Step 1: Get rays for the image
2. Step 2: Render the image using the model with the provided rays. You can set the near and far planes to be 2.0 and 6.0 respectively.
3. Step 3: Compute the loss between the rendered image and the corresponding ground truth image using the simple MSE loss.
4. Step 4: Update the model parameters using the computed loss, including `optimizer.zero_grad()`, `loss.backward()` and `optimizer.step()`.

> **Note** (Pytorch Optimizer). Pytorch optimizer is a package that contains the optimizer you need. You can find the optimizer you need in here. Remember to call `optimizer.zero_grad()` before computing the loss.

After finishing the `train` function, you can train your model by:

```
python train.py
```

It will save the intermediate results in `training_results_step_number` and you should expect something like the following after 2500 steps:

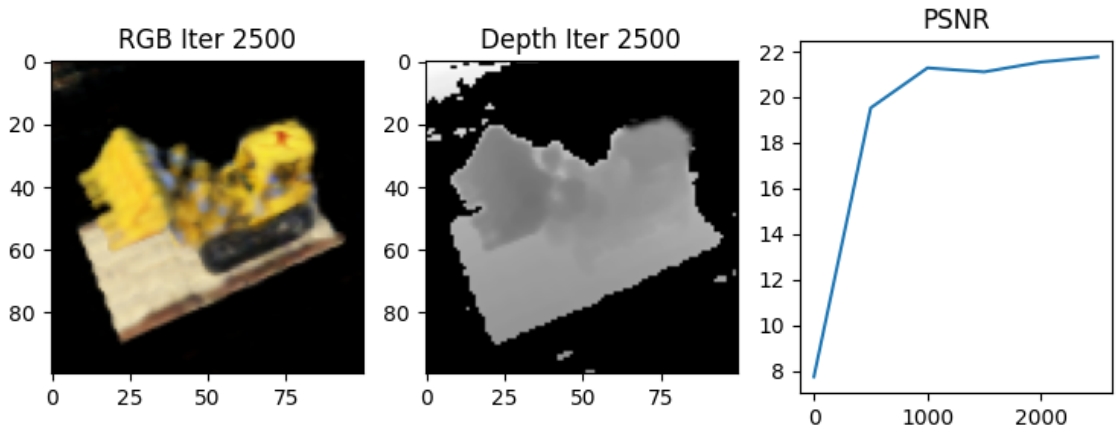


Figure 4: **Intermediate results of training.**

5.4 Rendering from different viewpoints

Given the trained model, we can query it with different camera poses. In this example, we simply pick three camera poses from the dataset and render the prediction images from them.

Please check the results in `rendering_test_results_trained_model.png`.

6 Optional: Extract Mesh from NeRF

Please note this part is optional and there would be no credit for this part.

It would be interesting to extract the mesh from the trained NeRF model. To do it, you can sample the color and opacity from the trained NeRF model on a dense grid of points, and meshlize the points with high opacity.

You can use the algorithm called marching cubes, which is a widely used algorithm for extracting isosurface from a 3D scalar field. PyMcube is a Python package that implements the marching cubes algorithm.

For more detailed instructions, you can check this notebook from the original NeRF paper and see if you can extract the mesh from your trained NeRF model.

7 Submission and Grading

HW4 has two parts: NeRF and GS. Each part takes 50% of grades and is graded individually. For submission, please combine the submission materials from two parts into one single .zip file.

The total score of this assignment is 100 points. As we are building the whole pipeline of NeRF, all five tasks are linear and dependent. So we assign 20 points for each task.

To submit your assignment, please zip your code as well as the generated images and upload them to Canvas. Please don't include any dataset images, or trained checkpoints in your uploaded zip file.

Homework 4 Part 2: Gaussian Splatting

EECS 498-014 Graphics and Generative Models

2024 Fall

Due date: 11:59 P.M., Nov 13, 2024

Contents

1 Objectives **2**

2 Instructions **2**

3 Overview **2**

4 Data Setup **3**

5 Representing 3D Scenes with Gaussians Splatting **3**

5.1 Gaussian Representation . . . 3

5.2 Optimizable parameters . . . 5

5.3 Quaternions to Rotation Matrix . . . 6

5.4 Spherical Harmonics . . . 6

6 Gaussian Splatting Rendering **7**

6.1 2D covariance matrix . . . 7

6.2 Tile-based point rasterization . . . 8

7 Gaussian Splatting Training **9**

7.1 Overfitting on a single image . . . 10

7.2 Training on all images . . . 10

8 Play with the Gaussian Splatting package **11**

9 Submission and Grading **13**

10 Optional: Capture your own 3D Gaussians **14**

1 Objectives

This assignment is a 2 weeks project on Gaussian Splatting. In this assignment, you will implement a simplified Gaussian Splatting algorithm using pure Python, which allows you to explore the core idea of Gaussian Splatting.

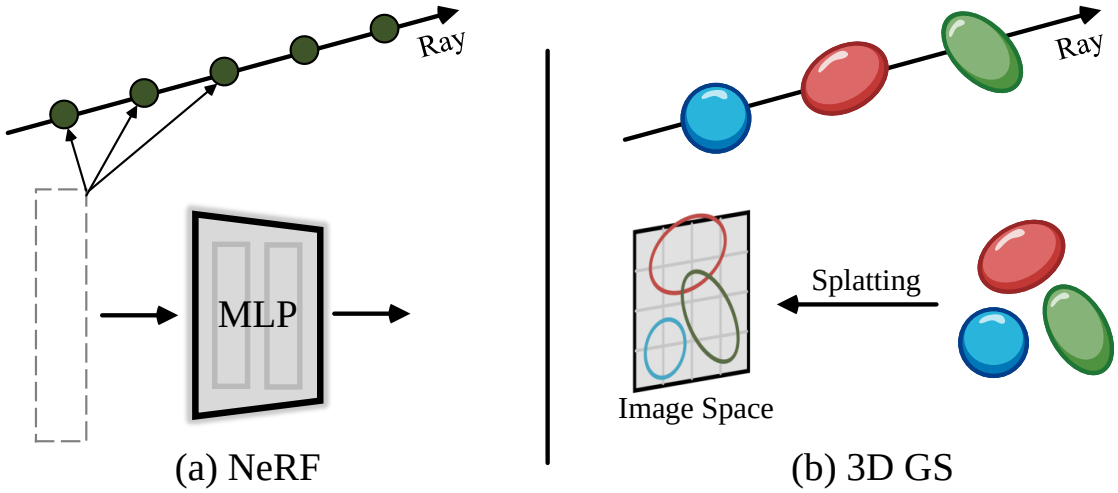


Figure 1: **Comparison between NeRF and Gaussian Splatting**. Unlike NeRF representing the 3D scene as an implicit function, Gaussian Splatting represents 3D scenes using a set of gaussian points and renders them via *splatting* 3D gaussian into 2D images. Image credit: A Survey on 3D Gaussian Splatting .

2 Instructions

You can download the starter code here. Please refer to the instruction section in HW Part 1 for how to install the Python environment and how to use GreatLakes. **Again, all commands in this part could be found in the `Readme.md`.**

3 Overview

In this problem set, you will implement a simplified version of Gaussian splatting.

Gaussian splatting is a popular and powerful representation of 3D scenes. Instead of utilizing an implicit function (MLP) to represent attributes of all 3D positions in 3D space, it consists of a set of explicit Gaussian points in 3D space and uses point rasterization for differentiable rendering. Consequently, it is much more efficient. In this homework, we will focus on the core idea of Gaussian splatting and implement a much-simplified version of it. Since we are not implementing it using CUDA for parallelization and are not using any heuristics for optimization, the entire process is slow and the results may not be very good. However, it will provide you with a better understanding of how Gaussian splatting works.

For better usage of Gaussian splatting, you are encouraged to check the original Gaussian splatting implementation at Inria and this open-source package gsplat.

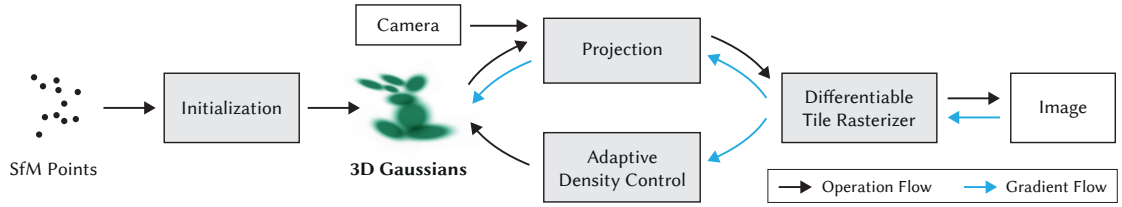


Figure 2: **Gaussian Splatting Pipeline Overview.** 3D Gaussian Splatting optimizes a set of 3D Gaussians to fit the 3D scene, and renders them into 2D images using a differentiable rasterization process. Image Credit.

4 Data Setup

Similar to NeRF, Gaussian splatting also takes a set of posed RGB images and reconstructs the corresponding 3D model. In general, Gaussian splatting requires point clouds for initialization to achieve optimal reconstruction results, which can be obtained using COLMAP or other methods. In this homework, we do not use point clouds for initialization and instead work with a simple dataset called `chair`. We load its images and corresponding camera intrinsics/extrinsic using the `load_data` function in the `data_loader.py` file. Feel free to test other datasets after completing this assignment!

Figure 3: **Training Scene Visualization.**

5 Representing 3D Scenes with Gaussians Splatting

5.1 Gaussian Representation

Gaussian Splatting represents 3D scenes using a set of 3D Gaussians. In Figure 4, we show an example of using 2D Gaussians to represent a 2D shape, and you can imagine how it works in 3D. A set of 2D Gaussians (ellipses) are placed in 2D space, and their combined curve is close to the target 2D shape (shown in red).

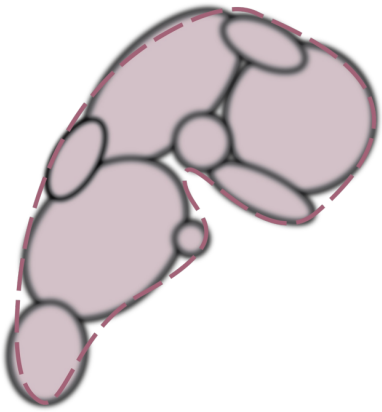


Figure 4: **Example of using Gaussian to represent shapes.** Here we show a toy example demonstrating how to use a set of 2D gaussians to fit a 2D shape. The red curve is the target shape, and the pink bubbles are 2D gaussians. By optimizing the parameters of 2D Gaussians (location, scales, rotations), we can approximately fit the target curve. Image credit.

As you can observe in the figure, the 2D Gaussians have different sizes, orientations, and center positions, which is the same as 3D Gaussians in 3D space. Now we will introduce how to parameterize a 3D Gaussian.

In general, an anisotropic Gaussian (ellipse) in 3D space centered at μ can be expressed as:

$$G(\mathbf{x}) = \exp\left(-\frac{1}{2}(\mathbf{x}-\mu)^T\Sigma^{-1}(\mathbf{x}-\mu)\right) \tag{1}$$

where Σ is a full 3D covariance matrix defined in world space. As shown in Figure 5, the full 3D covariance matrix Σ is related to the scale and rotation of the Gaussian. In the figure, the x, y, and z directions in black refer to the world space coordinates, while the yellow arrow indicates the x, y, and z axes of the Gaussian. S_x, S_y, and S_z are the lengths along the x, y, and z axes, respectively.

Thus, the full 3D covariance matrix Σ can be expressed as:

$$\Sigma = RSS^TR^T \tag{2}$$

where R is a rotation matrix (3x3) representing the transformation between the world coordinates and the Gaussian's XYZ axes. S is a diagonal matrix (3x3) representing the scaling of the Gaussian along the XYZ axes.

$$S = \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & S_z \end{bmatrix} \tag{3}$$

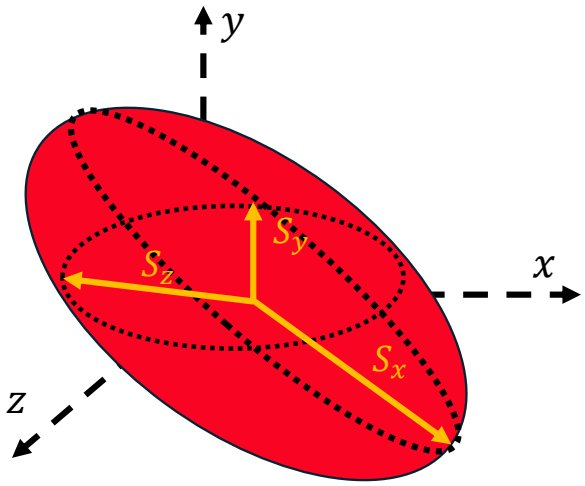


Figure 5: **3D Gaussian in 3D space.** Here we show a 3D gaussian in 3D space (anisotropic ellipse). The x, y, z are the 3D world coordinate, and S_x, S_y, S_z are the x, y, z axes length of the ellipse, respectively.

5.2 Optimizable parameters

Gaussian splatting optimizes the parameters of a set of Gaussians to fit the 3D scene. In this section, we discuss how to parameterize a Gaussian and how to optimize these parameters.

A 3D scene is represented by a set of (N) Gaussian, with the following parameters to be optimized:

1. **_xyz**: [N, 3], the XYZ positions of the Gaussian center in world space.
2. **_feature**: [N, D], the features of Gaussians, where D is the dimension of features. In general, Spherical Harmonics (SH) are often utilized as Gaussian features, which include **_feature_dc** ([N, 3], RGB color bases) and **_feature_rest** ([N, 3, D'], color basis vectors that will be produced with the view direction. This allows for different colors depending on the view direction). We also use SH in this homework, which we will discuss in the next section.
3. **_rotation**: [N, 4], rotations of Gaussians represented by quaternions. We will discuss this later.
4. **_scaling**: [N, 3], scaling of Gaussians along the XYZ directions, $[S_x, S_y, S_z]$.
5. **_opacity**: [N, 1], opacity of Gaussians.

As homework, we initialize the above parameters using a very simple manner instead of from point clouds, which may give degraded performance. We randomly sample N points in 3D space as **_xyz**, and calculate **_feature_dc** and **_feature_rest** from sampled RGB colors. All **_rotatoin** are initialized as $[0, 0, 0, 1]$, **_scaling** are 0.02, **_opacity** are 0.1

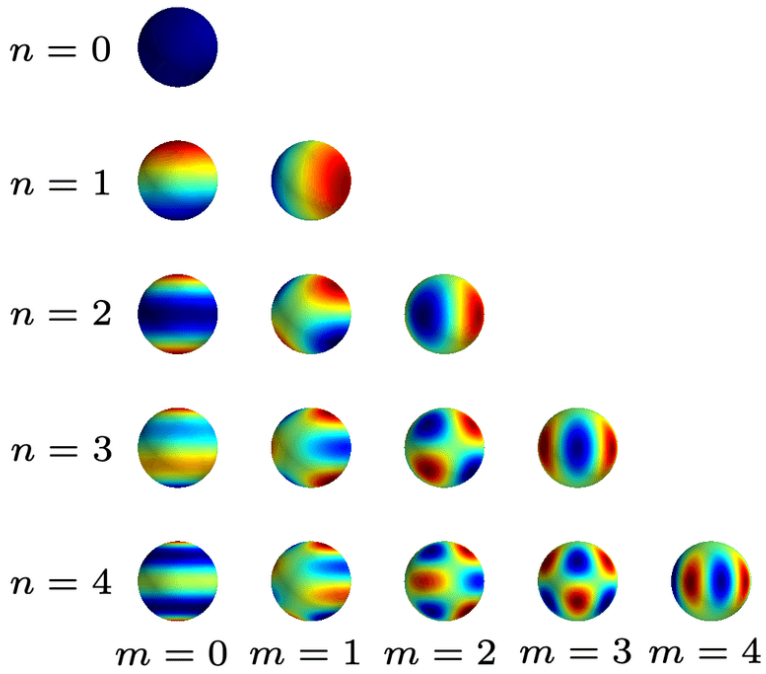


Figure 6: **Spherical Harmonics (SH) Visualization.** We visualize the basis function of SH with different degrees for a scaler on the sphere. n is the degree of SH, and m is the basis index in the degree. Image credit.

5.3 Quaternions to Rotation Matrix

As mentioned above, we need to optimize the rotation of Gaussians. The most straightforward way to represent rotations is using the 3x3 rotation matrix, while it is not easy to optimize due to the orthogonality constraint. Instead, we use quaternions to represent rotations, which is more efficient and easier to optimize. Quaternions use 4 scaler values to represent rotations, and the rotation matrix can be easily calculated from quaternions. In this homework, we are going to implement the conversion between quaternions and rotation matrices. Quaternions to rotation matrix could be found here: `https://www.songho.ca/opengl/gl_quaternion.html`

Task 1. Finish function `build_rotation` and `build_scaling_rotation` in `gs_render.py` file. Here we only need the 3×3 rotation matrix, which could be obtained by removing the last row and column of the 4x4 rotation matrix.

5.4 Spherical Harmonics

Spherical Harmonics (SH) are a set of orthogonal basis functions defined on the surface of a sphere. By combining these bases with different coefficients, we can theoretically represent any function on the sphere. Here, we present the visualization of SH basis functions in Figure 6, which represents a scalar function (value for a single channel) on the sphere. To represent the RGB values, we use three sets of independent SH basis functions.

You may be curious why we want to have different values for different positions on the same sphere. This is because object colors typically vary when viewed from different directions due to lighting and material effects. By using SH to represent the features of Gaussians, we can naturally obtain different colors for different view directions, which is

referred to as view-dependent effects.

Spherical Harmonics (SH) can have different degrees, as shown in Figure 6. Larger degrees can represent more complex functions but also require more parameters. In this homework, we set the SH degree to 3 ($n = 3$), which means we have 1 'dc' feature and 9 'rest' features for each color channel. Please check the `eval_sh` function in `utils.py` to see how to evaluate SH with different view directions.

6 Gaussian Splatting Rendering

After initialization, we need to render the 3D Gaussians to 2D images in a differentiable manner, which allows us to optimize the parameters of Gaussians based using the photo-metric loss.

In this section, we will implement how to render 3D Gaussians to 2D images using a simplified version of Gaussian splatting.

6.1 2D covariance matrix

After obtaining Gaussian's 3D covariance matrix, we need to project it to image space for rendering. Thanks to the property of Gaussian, this projection has a closed-form solution, as the projection of a 3D Gaussian is a 2D Gaussian.

EWA volume splatting achieves this closed-form solution by constructing a 2D covariance matrix for the projected 2D gaussian.

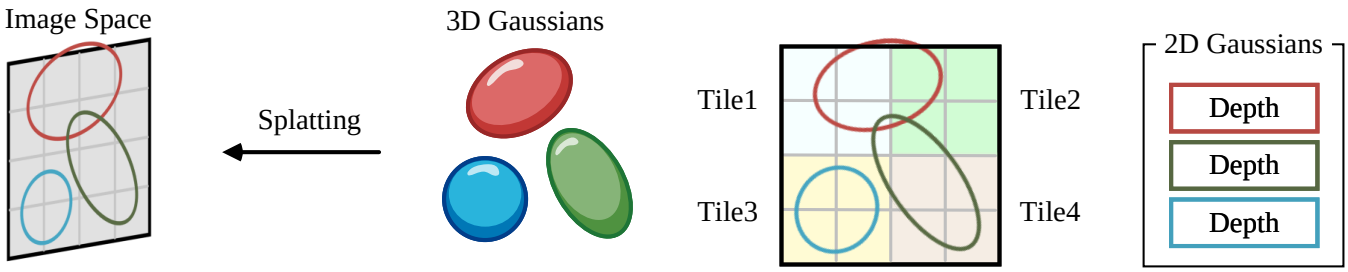
The 2D covariance matrix Σ' could be calculated as:

$$\Sigma' = JW\Sigma W^T J^T \tag{4}$$

where J is the Jacobian of the affine approximation of the projective transformation (first-order Taylor expansion), and W is the camera transformation matrix (purely rotation). Note here we directly project 3D Gaussian to image space, so that W is the world-to-camera transformation.

Task 2. Finish the missing parts in the `build_covariance_2d` function in `gs_render.py` file. You are expected to calculate the 2D covariance matrix of the projected Gaussian given J, 3D covariance matrix Σ, and world-to-image transformation W.

> **Note** (Affine Approximation of Projective Transformation). You are not required to derive the Jacobian matrix J in this homework but it is recommended to check how it is derived in detail. The *locally affine approximation* is calculated by Eq.29 in EWA Splatting.



(a) Projecting 3D Gaussians to 2D Image Space **(b) Tile-based Rasterization**

Table 1: Gaussian Splatting Rendering.

6.2 Tile-based point rasterization

Now we obtain the projected 2D Gaussians in the image space by projecting 3D Gaussians, as shown in Figure 1 (a). The next step is to rasterize these 2D Gaussians into 2D images (for each pixel). As shown in Figure 1 (b), we divide the image into a set of tiles, with each tile being a 2D block of pixels (e.g., 16x16). We calculate the intersection between the pixels in a tile and the 2D Gaussian and accumulate the Gaussian's RGB values to the pixels. Importantly, although we project the 3D Gaussian into 2D space, we still store their depths in 3D space, allowing us to sort the projected 2D Gaussians according to their depths. This information is critical for calculating the accumulated RGB values for pixels.

Task 3. Complete the code in `GaussRenderer.render` (in `gs_render.py`) to obtain `in_mask` based on the projected Gaussian's bounding box (`rect`) and the tile's bounding box. `in_mask` should be a boolean tensor with shape [B,] indicating whether the Gaussian overlaps with the tile.

To get the `in_mask`, we need to know the bounding box of the 2D Gaussian, as well as the tile's range. The bounding box of the 2D Gaussian is represented by the location of the top-left corner and the bottom-right corner of the rectangle. In the code, we already get the 2D Gaussian's top-left corner `rect[0]` and bottom-right corner `rect[1]`, where `rect[0]`, `rect[1]` are tensors with shape [B, 2] (x,y position). The tile ranges could be simply inferred from the loop of the image: the top-left corner is (h, w) and the bottle-right corner is $(h + \text{tile\_size}, w + \text{tile\_size})$. For simplicity, both of them are in the pixel coordinate, and you don't need to do any transformation.

With the `in_mask`, we can then calculate the pixel-wise Gaussian values in the tile and accumulate them into the image. **Task 4**. Follow the instructions and complete the code for Task 4 in `GaussRenderer.render` to obtain `acc_alpha`, `tile_depth`, and `tile_color` for each pixel within the tile.

It should be the following steps:

1. Step 1: calculate the gaussian weights given `dx` and `sorted_inverse_conv`. `dx` is the difference between the pixel position and the Gaussian center, and each element is 2-dimensional about x,y position. `sorted_inverse_conv` is the inverse of the covariance matrix, which is a 2x2 matrix for each Gaussian. Following the normal Gaussian distribution formula, the weight should be $\exp(-0.5 * (dx)^T \Sigma^{-1} (dx))$.

2. Step 2: calculate alpha. alpha = (gauss_weight[..., None] * sorted_opacity[None]).clip(max=0.99)

3. Step 3: calculate the accumulated alpha, color and depth.

3DGS utilizes the point-based α-blending to calculate the accumulated alpha, color and depth. To calculate the color:

$$C = \sum_{i=1}^{N} T_i \alpha_i c_i + (1 - \text{acc\_alpha}) \cdot C_{\text{bg}} \tag{5}$$

where: C_{bg} is the background color.

$$T_i = \prod_{j=1}^{i-1} (1 - \alpha_j) \tag{6}$$

To calculate the accumulated depth:

$$\text{depth} = \sum_{i=1}^{N} T_i \alpha_i z_i \tag{7}$$

where z_i is the depth of the i-th Gaussian.

For accumulated alpha:

$$\text{acc\_alpha} = \sum_{i=1}^{N} \alpha_i T_i \tag{8}$$

Note (Acceleration of Tile-based Rasterization). As you may have noticed, the rendered pixel values inside one tile are independent of the pixel values in other tiles. Therefore, we can perform parallel computation for each tile by spawning a CUDA kernel for rasterization, which could significantly speed up the rendering process. In the original paper on 3D Gaussian Splatting, a highly optimized CUDA kernel is deployed to enhance efficiency. It first calculates the indices of intersecting Gaussians (within the tile frustum) and sorts them according to their depths. Then, it loads the Gaussians into shared memory for efficient memory access. Finally, it performs parallel computation for each Gaussian and accumulates their RGB values to the pixels.

In this homework, we do not use CUDA for parallelization for the sake of simplicity. Instead, we perform the tile-based rasterization in a for-loop, which is slower but relatively easy to implement.

7 Gaussian Splatting Training

Now we have already implemented all the necessary components for Gaussian Splatting, and we are ready to train our own models.

Figure 7: 100 steps

Figure 8: 500 steps

Figure 9: **Overfitting results on a single image**. We show the results at 100 and 500 steps.

Similar to NeRF, the Gaussian splatting model is trained by optimizing the loss on 2D images: between rendered images and the provided target images.

Here we use a similar loss as the paper:

$$\mathcal{L} = (1 - \lambda)\mathcal{L}_1 + \lambda(1 - \text{SSIM}) \tag{9}$$

where $\mathcal{L}_1$ is the L1 loss, and SSIM is the Structural SIMilarity index.

7.1 Overfitting on a single image

For ease of debugging, we highly recommend you to first overfit your model on a single image, which is much faster than training on all images.

```
python train.py --single_image_fitting
```

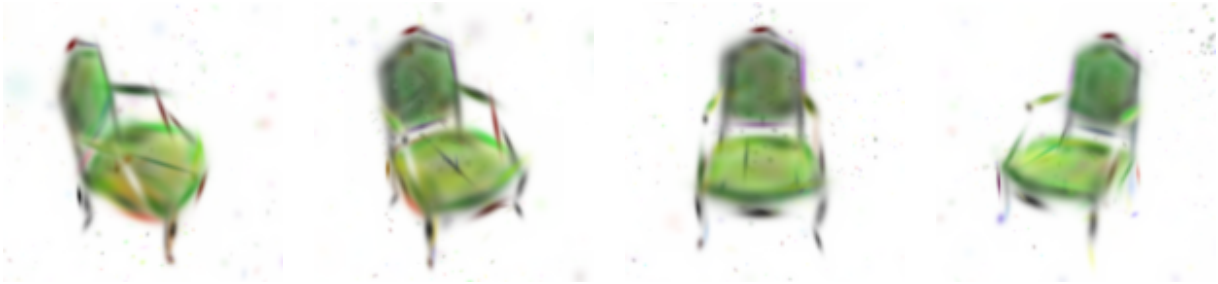
The whole overfitting takes 500 steps, requiring around 3 minutes on GreatLakes.

Results are saved in `result/single_image_fitting`. You may expect results shown in Figure 9

7.2 Training on all images

After overfitting on a single image, we now train our model on all images. It will train for 2000 steps, taking around 12 minutes on GreatLakes. Intermediate results are saved in `result/multi_image_fitting`, and the test view results of the final model are saved in `result/multi_image_fitting/test_views`. You may expect similar quality as shown in Figure 7.2 on the test views.

```
python train.py
```



view 3 view 4 view 5 view 6

Our results would be blurry and not very good, which is expected, as we use a much-simplified version of Gaussian splatting (we don't include hierarchical pruning and splitting, which are very important for performance). Also, our pure-python implementation is slower than the optimized CUDA version, and we only optimize very limited times.

8 Play with the Gaussian Splatting package

Now we will play with the real 3D Gaussian package, implemented in CUDA with tons of optimizations.

Install the package

Firstly, we need to install the package.

```
git clone https://github.com/nerfstudio-project/gsplat.git --recursive
cd gsplat

module load cuda/12.1.1
module load gcc/10.3.0
python -m pip install -e .

python -m pip install -r examples/requirements.txt
python examples/datasets/download_dataset.py
```

The installation process will take a while to finish.

Note the above command is for GreatLakes. In particular, we load the pre-installed CUDA and GCC modules by calling `module load`. You can try to install the package on your own machine while not using the module loading.

Setup the port forwarding

We typically use a visualizer to examine and verify the 3D Gaussian results. To view the results running on the server, we need to forward the port from the server to our local machine.

NOTE: You should be able to find the following command in `Readme.md`

On your local machine, run the following command to set up the port forwarding:

```
ssh -L {local_port}:localhost:{server_port} {your unique name}@greatlakes.arc-ts.umich.edu
```

Here, {local_port} refers to the port on your local machine, and {server_port} refers to the port on the server. {your unique name} is your unique identifier on GreatLakes. You can set both {local_port} and {server_port} to 8090, for example.

On the login node of GreatLakes, you will first need to request a compute node (**allocated with a GPU**). Then, on the compute node, run the following command to obtain the node address:

```
hostname -f
```

You will receive the node address, e.g., gl1710.arc-ts.umich.edu.

Next, we need to set up the port forwarding from the login node to the compute node. On the **login node**, run the following command:

```
ssh -L {server_port}:localhost:{server_port_compute} {node_address}
```

Here, {server_port_compute} is the port on the compute node. Setting it to 8090 is also feasible.

Now we have successfully set up the two port forwarding connections: from the local machine to the login node, and from the login node to the compute node. On the **compute node** and go to the directory of the Gaussiance homework (gsplat), we can start training the Gaussian splatting model:

```
python examples/simple_trainer.py default --data_dir data/360_v2/garden/ --data_factor 4 --
    result_dir ./results/garden --port ${server_port_compute} --max_steps 20000
```

You can access the web UI on your local machine by entering the following address in your web browser:

```
http://localhost:{local_port}
```

You should see something similar to Figure 10.

Task 5. Complete the above steps to explore the real Gaussian splatting. You are expected to submit a 3-second video clip of the reconstruction results, showcasing changing camera viewpoints. This task should be relatively straightforward, as you do not need to write any code.

For submission, please ensure that the video is compressed to be less than 3MB.

Figure 10: **GSplat UI.** You can view the Gaussian splatting results in real time.

9 Submission and Grading

HW4 has two parts: NeRF and GS. Each part takes 50% of grades and is graded individually. For submission, please combine the submission materials from two parts into one single .zip file.

This homework consists of five tasks, with the first four tasks each worth 22.5 points, while the last task is optional and worth 10 points. The final score will be the sum of the scores from each task.

For submission, please provide a zip file containing the following files:

1. `gs_render.py`
2. `gs_model.py`
3. `result/multi_image_fitting/test_views` folder
4. `result/single_image_fitting/test_views` folder
5. A 3-second video clip of your results using the gsplat package.

Please refrain from submitting any other files, particularly large files, downloaded data, or checkpoints.

10 Optional: Capture your own 3D Gaussians

It is now time to capture your own 3D Gaussians! Explore some fascinating and creative environments, and capture them using your iPhone. Discover how these scenes can be reconstructed with your own Gaussians.