Embodied Artificial Intelligence

HW1: 3D Gaussian Splatting

Department: Department of Automation

Class: Class 31

Name: Zuo Gou

ID: 2023012274

1 Experiment objectives

- Understand and implement the 3D Gaussian Splatting workflow for novel view synthesis.
- Implement first-degree spherical harmonics for improved rendering quality.
- Explore post-hoc pruning strategies to optimize rendering performance while maintaining visual fidelity.

0. Setting up the environment

We followed the repository README. Key steps we performed and verified:

1. Install dependencies: `pip install -r requirements.txt`.
2. Ensure CUDA toolkit and `nvcc` are installed.
3. Build and install the gaussian rasterization kernel in the submodule:

```
cd submodules/diff-gaussian-rasterization
python setup.py build
python setup.py install
```

Quick functional test we ran (generates example renders in `renders/`):

```
python render.py
```

After the above runs, the pipeline and the extension built successfully, and we obtained **(incorrectly) rendered images** in the renders/ folder. Below is one we obtained:

fig1: Example of incorrectly render output after building the rasterizer (task0).

1. Preparation

This section documents dataset handling and COLMAP stages, and shows how to visualize the provided point cloud.

1.1. COLMAP reconstruction stages

By using COLMAP's automatic reconstruction pipeline on the provided dataset via the following CLI:

```
DATASET_PATH="$(pwd)/datasets/fruit" \
colmap automatic_reconstructor \
--workspace_path "$DATASET_PATH" \
--image_path "$DATASET_PATH/images" 2>&1 | tee "$DATASET_PATH/colmap_autorecon.
    log"
```

The attached COLMAP automatic reconstruction log (`colmap_autorecon.log`) confirms the standard SfM stages and provides concrete timing and scale information for this dataset. The observed stages are (task1.1):

1. **Feature extraction:** COLMAP processed 220 input images. Per-image SIFT feature counts vary (examples in the log range from 1.7k up to 4.0k features per image). The feature extraction pass finished quickly (elapsed time reported 0.42 minutes).
2. **Exhaustive feature matching:** pairwise matching is the most expensive early stage for this dataset; the log reports exhaustive matching over multiple blocks with a total elapsed time of about 84.83 minutes (matching blocks printed with per-block times in the log).
3. **Database loading and graph construction:** COLMAP loads cameras (220), matches (21169) and images (220) and builds the correspondence graph (log shows "Building correspondence graph..." completed).
4. **Initialization:** the automatic reconstructor selected an initial image pair (the log shows "Initializing with image pair #87 and #112") which seeded the incremental reconstruction.
5. **Incremental registration & triangulation:** images were registered incrementally; the log contains many "Registering image #..." and "Retriangulation" entries where newly registered images produced additional 3D points.
6. **Bundle adjustment (BA):** repeated local/global BA steps were executed throughout the incremental pipeline. The log contains multiple "Global bundle adjustment" sections; a representative BA run shows a CPU single-precision solver and reports detailed iteration statistics. For example, one BA decreased MSE from 0.992939 to 0.989297 (the BA reports list iterations, cost, gradient norms and timings). Bundle adjustment was the major time consumer and should be monitored in logs (iterations, final reprojection error, and timing).

7. **Optional dense reconstruction / meshing:** not required but available as a subsequent stage.

Summary of observations: the log confirms the standard SfM pipeline (feature extraction, matching, initialization, incremental registration, triangulation and iterative BA). For this dataset, matching and bundle adjustment dominated runtime; we used the provided precomputed `points3D.ply` to skip heavy reconstruction steps when appropriate.

1.2. Point cloud initialization

We used `assets/points3D.ply` to initialize Gaussians. To visualize we used MeshLab: File -> Import Mesh -> `points3D.ply` -> Render -> Show Vertex Color. Bellow is a screenshot of the loaded point cloud in Meshlab:

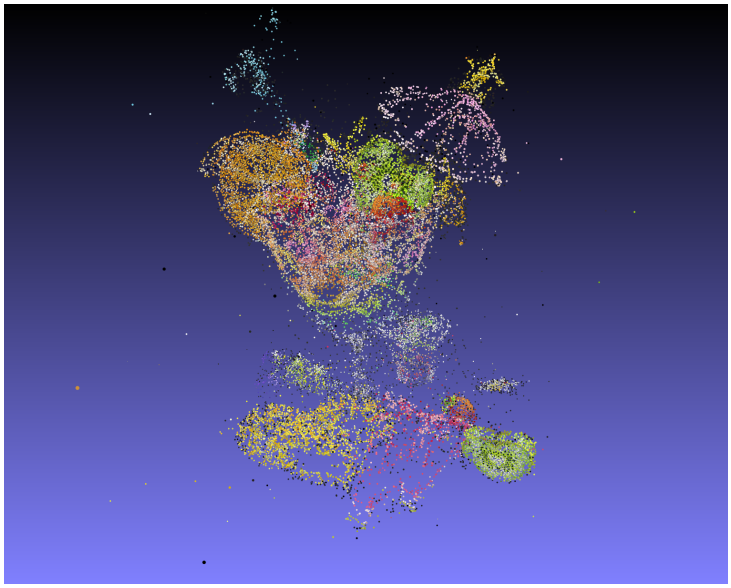

fig2: Visualization of the provided point cloud in Meshlab (task1.2).

2. Building Gaussian primitives

This section documents our derivations and implementations for Gaussian covariance construction and projection, namely how to construct elliptic Gaussian primitives and splat them into screen space.

2.1. Constructing the covariance matrix

Derivation of covariance from scale and rotation (task2.1 writing)

Let a Gaussian in local coordinates have zero mean and diagonal covariance given by the squared local standard deviations: $S^2 = \text{diag}(s_x^2, s_y^2, s_z^2)$ with $s_i > 0$. A rotation matrix $R \in SO(3)$ maps local to world coordinates. The covariance in world coordinates is

$$\Sigma = RS^2R^\top. \tag{1}$$

This follows from the rule $\text{Cov}(AX) = A\text{Cov}(X)A^\top$ for a linear transform A.

Implementation of `get_covariance` (task2.1 coding)

We implemented `get_covariance` in `models/gaussian_point.py` as follows:

```
def get_covariance(self, scaling_modifier = 1):
    s= self.get_scaling # scaling vector
    q= self.get_rotation # rotation quaternion
    ## Begin code 2.1 ##
    # todo: build 3dgs covariance matrix. Equation (6) of 3dgs paper.
    R = build_rotmat_from_quat(q)
    S = torch.diag_embed(scaling_modifier * s)
    L = R @ S
    ## End code 2.1 ##

    actual_covariance = L @ L.transpose(1, 2)
    symm = strip_symmetric(actual_covariance)
    return symm
```

Here, `build_rotmat_from_quat` constructs rotation matrices from quaternions, and we apply an optional `scaling_modifier` to uniformly scale the standard deviations.

2.2. Viewing transform (task2.2)

Writing the viewing transformation matrix in block form as

$$\mathbf{T}_v = \begin{pmatrix} \mathbf{R}_v & \mathbf{t}_v \\ \mathbf{o}^\top & 1 \end{pmatrix}$$

mapping world coordinates to camera coordinates.

Writing the point in homogeneous form, the mean transforms as

$$\mu_c \;=\; \mathbf{T}_v \begin{pmatrix} \mu \\ 1 \end{pmatrix} \;=\; \mathbf{R}_v\,\mu + \mathbf{t}_v.$$

The covariance transforms according to the linear part (rotation) only:

$$\Sigma_c \;=\; \mathbf{R}_v \, \Sigma \, \mathbf{R}_v^\top,$$

In implementation, extract the 3×3 rotation block $\mathbf{R}_v$ from the view matrix and use the above formulas to obtain the Gaussian parameters in camera space before applying the projection/Jacobian step.

2.3. Projection Jacobian and projected covariance (task2.3)

Suppose a point in camera coordinates is $x_c = (x, y, z)^\top$ and the pinhole projection (to pixel units) is

$$p : \mathbb{R}^3 \to \mathbb{R}^2, \qquad p(x, y, z) = \begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} f_x \dfrac{x}{z} \\ f_y \dfrac{y}{z} \end{pmatrix},$$

where f_x, f_y are the focal (pixel) scalings. The mapping is nonlinear because of the division by z, but we approximate it locally at the mean by its Jacobian.

The Jacobian of p evaluated at a mean $\mu_c = (\bar{x}, \bar{y}, \bar{z})^\top$ is

$$J_p(\mu_c) = \begin{pmatrix} \frac{\partial u}{\partial x} & \frac{\partial u}{\partial y} & \frac{\partial u}{\partial z} \\ \frac{\partial v}{\partial x} & \frac{\partial v}{\partial y} & \frac{\partial v}{\partial z} \end{pmatrix} = \begin{pmatrix} \dfrac{f_x}{\bar{z}} & 0 & -\dfrac{f_x \, \bar{x}}{\bar{z}^2} \\ 0 & \dfrac{f_y}{\bar{z}} & -\dfrac{f_y \, \bar{y}}{\bar{z}^2} \end{pmatrix}.$$

If the original Gaussian in world coordinates has mean μ and covariance Σ and the rigid viewing transform (world -> camera) has linear part R (so $\mu_c = R\mu + t$ and $\Sigma_c = R\Sigma R^\top$), then the approximate 2D covariance in image (pixel) space is obtained by the linearization

$$\Sigma_{2D} \approx J_p(\mu_c) \, \Sigma_c \, J_p(\mu_c)^\top.$$

Equivalently, combining the camera rotation with the projection Jacobian,

$$J_{\text{img}} = J_p(\mu_c) \, R, \qquad \Sigma_{2D} \approx J_{\text{img}} \, \Sigma \, J_{\text{img}}^\top.$$

Practical notes used in the implementation: - Evaluate J_p at μ_c using $\bar{z} = \max(\mu_{c,z}, \varepsilon)$ to avoid division by zero. - The focal factors f_x, f_y are included in J_p so the result is already in pixel units. - To ensure a minimum visible size, add a small constant to the diagonal of Σ_{2D} (one-pixel floor), then return the symmetric 2D covariance components.

2.4. Implementing `computeCov2D` in the rasterizer (task2.4)

We implemented `computeCov2D` in `submodules/diff-gaussian-rasterization/diff_gaussian_raster` as follows:

```
// Begin Code 2.4
float z = max(t.z, 1e-6f);  // avoid division by zero
glm::mat3 J = glm::mat3(
        focal_x / z, 0.0f, -focal_x * t.x / (z * z),
        0.0f, focal_y / z, -focal_y * t.y / (z * z),
        0.0f, 0.0f, 0.0f);  // projection Jacobian
glm::mat3 T = W * J;  // combined transform: camera rotation + projection
    Jacobian, T is equivalent to J_img
glm::mat3 cov = glm::transpose(T) * sigma * T;  // propagated covariance
// End Code 2.4
```

Here, `t` is the point in camera coordinates, `W` is the camera rotation matrix, and `sigma` is the 3D covariance in world coordinates.

2.5. Computing 99% bounding box (task2.5)

To compute the 99% bounding box of the projected 2D Gaussian, we can use the 2D covariance `Sigma_2D` obtained from the previous step. The 99% confidence region of a 2D Gaussian with covariance Σ_{2D} is the ellipse

$$\{x : (x-\mu)^\top \Sigma_{2D}^{-1}(x-\mu) \le \chi_2^2(0.99)\},$$

where for 2 degrees of freedom

$$\chi_2^2(0.99) = -2\ln(1-0.99) \approx 9.21034.$$

If the eigenvalues of Σ_{2D} are $\lambda_1 \ge \lambda_2 > 0$, the ellipse semi-axes lengths are

$$a_i = \sqrt{\chi_2^2(0.99)\,\lambda_i}, \qquad i = 1, 2.$$

A centered axis-aligned square that fully contains this ellipse must have half-side at least the largest semi-axis length, hence the required square side length is

$$s \;=\; 2\sqrt{\chi_2^2(0.99)\,\lambda_{\max}}, \qquad \lambda_{\max} = \lambda_1.$$

For a screen-space (2×2) Gaussian covariance

$$\Sigma_{2D} = \begin{pmatrix} \sigma_{xx} & \sigma_{xy} \\ \sigma_{xy} & \sigma_{yy} \end{pmatrix}$$

the eigenvalues can be computed in closed form:

$$\lambda_{1,2} = \frac{\sigma_{xx}+\sigma_{yy}}{2} \pm \sqrt{\left(\frac{\sigma_{xx}-\sigma_{yy}}{2}\right)^2 + \sigma_{xy}^2},$$

so take $\lambda_{\max} = \max(\lambda_1, \lambda_2)$ and compute

$$s = 2\sqrt{9.21034\,\lambda_{\max}}.$$

2.6. generating rendered images

After implementing the above steps, we generated correctly rendered images by repeating commands in section 0:

```
cd submodules/diff-gaussian-rasterization
python setup.py build
python setup.py install
python render.py
```

Below is one of the correctly rendered images we obtained:

fig3: Example of correctly rendered output after implementing Gaussian construction and projection.

3. Better rendering

This section documents our implementations for first-degree spherical harmonics and post-hoc pruning strategies.

3.1. First-degree spherical harmonics (task3.1)

Implementation of `eval_sh`

We implemented `eval_sh` in `utils/sh_utils.py` as follows:

```
if deg ==1:
    # Begin code 3.1 ##
    result += (
        SH_C1_0 * sh[..., 1] * dirs[..., 1:2]
```

```
        + SH_C1_1 * sh[..., 2] * dirs[..., 2:3]
        + SH_C1_2 * sh[..., 3] * dirs[..., 0:1]
    )
    # End code 3.1 ##
```

Here, `dirs` are the normalized view directions in camera space, and `sh` are the spherical harmonics coefficients per point.

Evaluating rendering performances of different SH degrees

To evaluate rendering performance for 0th and 1st spherical-harmonics (SH) degrees, we used two complementary comparisons: a composite visual comparison using an error heatmap and a quantitative comparison (paired PSNR across the five test views).

- **Method A: composite visual comparison**

  For each test view we create a heatmap showing per-pixel absolute error between the 0th and 1st degree SH renders. The heatmap uses a perceptual colormap to highlight error magnitudes. We then create a composite figure per view with 4 panels: (1)Ground Truth, (2) SH0 render, (3) SH1 render, (4) error heatmap for SH0 and SH1 side-by-side for visual comparison. This allows inspection of where and how the two SH degrees differ in fidelity.

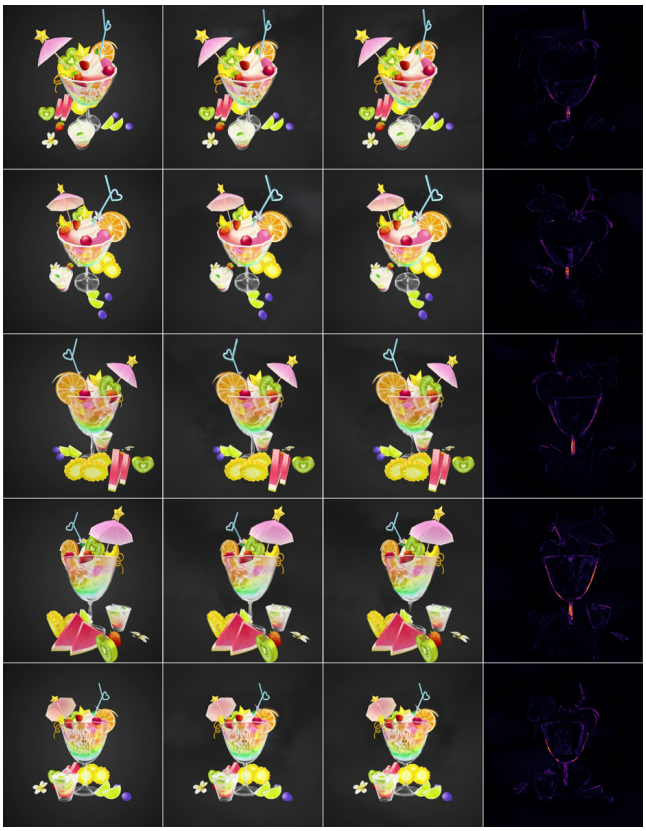

fig4: Example composite figure for 5 test views comparing GT, SH0, SH1, and error heatmaps.

From the comparison image, we can see that the two SH degrees produce relatively significant difference at the edge of the objects, especially at the neck of the cup in the rendered images.

- **Method B: paired PSNR comparison and summary plot**

  The PSNR value between two images I_1, I_2 is computed as

$$\mathrm{PSNR}(I_1, I_2) = 10 \log_{10} \left(\frac{L^2}{\mathrm{MSE}(I_1, I_2)} \right),$$

  where L is the maximum pixel value (255 for 8-bit images) and MSE is the mean squared error. We computed PSNR between SH0 and GT, and SH1 and GT for each of the 5 test views, resulting in paired PSNR values. We then summarized the results in a plot showing paired PSNR values across views and their means for SH0 and SH1:

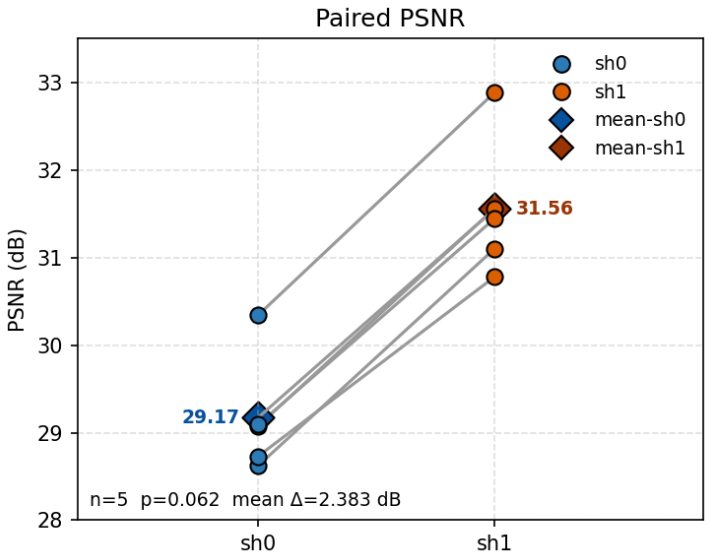

fig5: Paired PSNR comparison between SH0 and SH1 across 5 test views.

From the PSNR plot, we can see that the first-degree SH consistently outperforms the zero-degree SH across all test views, with an average PSNR improvement of approximately 2.4 dB.

3.2. Post-hoc pruning (task3.2)

Implementation of 2 pruning strategies

Apart from opacity-based pruning which removes Gaussians with opacity below a threshold (task3.2.1), we also implemented another pruning method - importance-based pruning (task3.3, bonus). Here importance is computed by rendering each Gaussian from multiple camera views

and accumulating a score based on its opacity, screen-space area, and visibility, namely:

$$\text{importance} = \sum_{\text{views}} \text{opacity} \times \text{area} \times \text{visibility},$$

where area is computed as πr^2 with r being the Gaussian's projected radius in pixels, and visibility is a binary indicator (1 if the Gaussian contributes to the view, 0 otherwise). Gaussians with importance below a certain quantile threshold are pruned.

We implemented these two pruning strategies by modifying `prune_point` in `gaussian_model.py` and `render.py`:

```
# In gaussian_model.py
def prune_points(self, th=0.0, mode=1, cameras=None):
    ## Begin code 3.2 ##
    opacity = self.get_opacity()
    valid_points_mask = torch.ones((self.get_xyz.shape[0],), dtype=torch.bool,
        device=self._xyz.device)

    if mode == 1 or cameras is None:
        with torch.no_grad():
            valid_points_mask = (opacity.view(-1) > th)
            if valid_points_mask.sum().item() == 0:
                return

    elif mode == 2:
        with torch.no_grad():
            importance = torch.zeros((self.get_xyz.shape[0],), dtype=torch.
                float32, device=self._xyz.device)
            means3D = self.get_xyz
            means2D = torch.zeros((self.get_xyz.shape[0], 3), dtype=self._xyz.
                dtype, device=self._xyz.device)
            cov3D_precomp = self.get_covariance(1)

            for cam in cameras:
                tanfovx = math.tan(cam.FoVx * 0.5)
                tanfovy = math.tan(cam.FoVy * 0.5)
                raster_settings = GaussianRasterizationSettings(
                    image_height=int(cam.image_height),
                    image_width=int(cam.image_width),
                    tanfovx=tanfovx,
                    tanfovy=tanfovy,
                    bg=torch.tensor([0, 0, 0], dtype=torch.float32, device=self.
                        _xyz.device),
                    scale_modifier=1.0,
                    viewmatrix=cam.world_view_transform,
                    projmatrix=cam.full_proj_transform,
                    sh_degree=self.active_sh_degree,
                    campos=cam.camera_center,
```

```
                    prefiltered=False,
                    debug=False
                )

                # Precompute colors_per_point in this view (same as render path)
                shs_view = self.get_features.transpose(1, 2).view(-1, 3, (self.
                    max_sh_degree + 1) ** 2)
                dir_pp = (means3D - cam.camera_center.repeat(means3D.shape[0],
                    1))
                dir_pp_normalized = dir_pp / (dir_pp.norm(dim=1, keepdim=True) +
                     1e-8)
                sh2rgb = eval_sh(self.active_sh_degree, shs_view,
                    dir_pp_normalized)
                colors_precomp = torch.clamp_min(sh2rgb + 0.5, 0.0)

                rasterizer = GaussianRasterizer(raster_settings=raster_settings)
                _, radii = rasterizer(
                    means3D=means3D,
                    means2D=means2D,
                    shs=None,
                    colors_precomp=colors_precomp,
                    opacities=opacity,
                    scales=None,
                    rotations=None,
                    cov3D_precomp=cov3D_precomp
                )

                radii = radii.view(-1)
                visible = (radii > 0).float()
                area = (torch.pi * (radii ** 2)).float()
                importance += (opacity.view(-1) * area * visible)

            cutoff = torch.quantile(importance, th)
            valid_points_mask = (importance > cutoff)
            if valid_points_mask.sum().item() == 0:
                return

    ## End code 3.2 ##
    self._xyz = self._xyz[valid_points_mask]
    self._features_dc = self._features_dc[valid_points_mask]
    self._features_rest = self._features_rest[valid_points_mask]
    self._opacity = self._opacity[valid_points_mask]
    self._scaling = self._scaling[valid_points_mask]
    self._rotation = self._rotation[valid_points_mask]
```

```
# In render.py
### Begin Code 3.2 ###
```

```
print("Number of gaussians before pruning: ", gaussians.get_xyz.shape[0])
mode = int(getattr(args, 'prune_mode', 0))
th = float(getattr(args, 'prune_threshold', 0.0))
if mode == 0:
    print('No pruning applied')
elif mode == 1:
    print('Pruned gaussians by opacity')
    gaussians.prune_points(th=th, mode=1)
elif mode == 2:
    print('Pruned gaussians by importance')
    gaussians.prune_points(th=th, mode=2, cameras=cameras)
print("Number of gaussians after pruning: ", gaussians.get_xyz.shape[0])
### End Code 3.2 ###
```

Pruning mode can be chosen via command-line arguments: `--prune_mode 0` for no pruning, `--prune_mode 1` for opacity-based pruning and `--prune_mode 2` for importance-based pruning, with threshold specified by `--prune_threshold`. For example, you can run:

```
python render.py --sh_degree 1 --prune_mode 1 --prune_threshold 0.05
```

to prune Gaussians with opacity below 0.05.

Selecting maximum pruning ratio (task 3.2.2)

We tested different pruning thresholds for opacity-based pruning (task3.2.2) to find the maximum threshold that maintains acceptable visual quality compared to the baseline (no pruning). Below is a comparison of rendered images at 5 different opacity thresholds (0.02, 0.04, 0.06, 0.08, 0.10):

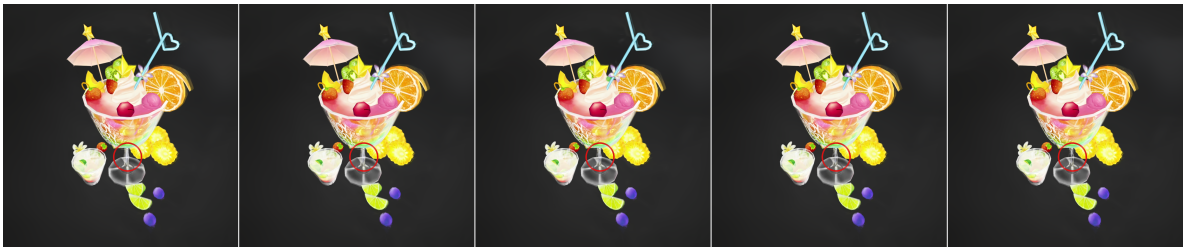

fig6: Comparison of rendered images at different opacity pruning thresholds. From left to right: threshold=0.02, 0.04, 0.06, 0.08, 0.10.

From the comparison, we observe that thresholds up to 0.06 maintain good visual quality with minimal artifacts. At threshold 0.08, some degradation becomes noticeable, especially missing pixels at the neck of the cup. Therefore, we select **0.06** as the maximum opacity pruning threshold that preserves acceptable visual quality, at which the maximum pruning ratio equals $69737/257882 \approx 0.270$.

Evaluation and comparison of 2 pruning strategies

We compare the opacity-based pruning and importance-based pruning by calculating PSNR and SSIM between the pruned renders and ground truth images at different pruning threshold, then plotting PSNR/SSIM vs amnount of dropped points for both methods as follows:

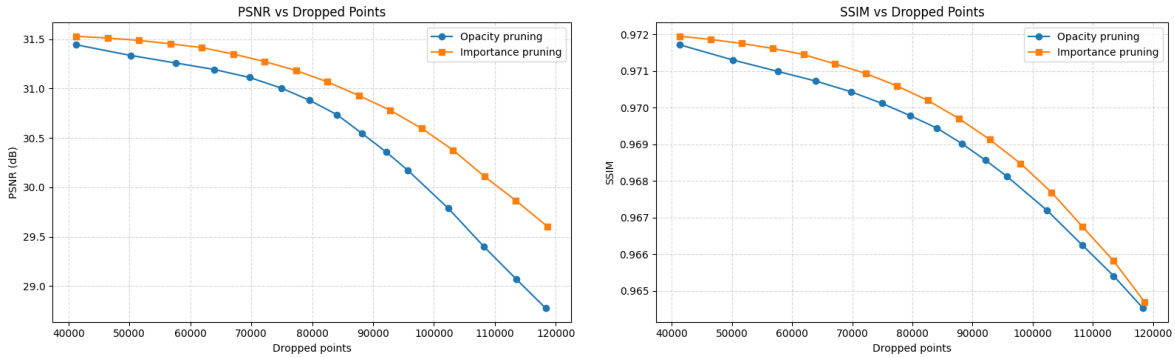

fig7: Comparison of opacity-based and importance-based pruning: (left) PSNR vs dropped points, (right) SSIM vs dropped points.

From the plots, we observe that importance-based pruning consistently outperforms opacity-based pruning in both PSNR and SSIM metrics at various levels of dropped points. This indicates that importance-based pruning is more effective at retaining visual quality while reducing the number of Gaussians.

Conclusion

- We successfully implemented the 3D Gaussian Splatting pipeline, including Gaussian construction, projection, and rendering.
- We implemented first-degree spherical harmonics, which significantly improved rendering quality as evidenced by visual comparisons and PSNR metrics.
- We explored opacity and importance-based pruning. Importance-based pruning demonstrated superior performance in maintaining visual fidelity while reducing the number of Gaussians.

References

- 3DGS: `https://arxiv.org/abs/2308.04079`
- COLMAP: `https://colmap.github.io/`
- Spherical harmonics table: `https://en.wikipedia.org/wiki/Table_of_spherical_harmonics`