Instant GaussianImage: A Generalizable and Self-Adaptive Image Representation via 2D Gaussian Splatting (Supplementary Material)

Zhaojie Zeng[1], **Yuesong Wang**[1*], **Chao Yang**[2], **Tao Guan**[1], **Lili Ju**[3]
[1] School of Computer Science & Technology, Huazhong University of Science and Technology
[2] NERCGIS, China University of Geoscience (Wuhan)
[3] Department of Mathematics, University of South Carolina
{zhaojiezeng, yuesongwang, qd_gt}@hust.edu.cn, yangchao@cug.edu.cn, ju@math.sc.edu

In this supplementary material, we provide additional explanations for conclusions presented in the main paper, illustrate more technical details, and supplement experimental results.

1. Explanations

1.1. Analysis of CDF Sampling in Sec. 3.3

In Sec. 3.3 of the main paper (*Dithering with* $\mathbb{P}_{\text{pred}}$), we state:

> "An alternative is CDF sampling with a fixed number of Gaussians, as in [5]. While it works when high- and low-entropy regions are balanced, it degrades rendering quality when high-entropy regions dominate, reducing point density where detail is needed. Conversely, in low-entropy images, excessive points are allocated to smooth regions, leading to redundancy."

This implies that when an image is dominated by low-entropy or high-entropy regions, the CDF-based sampling strategy also suffers from overfitting or underfitting issues[3], similar to the threshold-based sampling.

To further analyze the behavior of CDF sampling, we conduct a simple simulation experiment. We generate three types of synthetic image data with different entropy distributions:

- **Balanced entropy** (green) – where pixel counts are evenly distributed across entropy intervals.
- **Low-entropy dominant** (orange) – where low-entropy regions contain more pixels.
- **High-entropy dominant** (blue) – where high-entropy regions contain more pixels.

As shown in Fig. 1 the **top left** plot, these distributions define the initial pixel counts per entropy interval. We then apply CDF sampling to each distribution and obtain the number of sampled points in each entropy interval, illustrated in the **top right** plot. It is evident that in both the balanced and high-entropy dominant cases, more points are allocated to high-entropy regions, while in the low-entropy dominant case, mid-entropy regions receive the highest number of sampled points.

To quantify the sampling efficiency, we normalize the number of sampled points by the corresponding pixel counts in each entropy interval, resulting in the **bottom plot**. Using the balanced case as a reference, we observe that:

- When low-entropy regions dominate, the sampling density is significantly higher than the balanced case, leading to redundancy (*overfitting*).
- When high-entropy regions dominate, the sampling density is notably lower than the balanced case, leading to insufficient representation (*underfitting*).

These results confirm that CDF sampling alone struggles to maintain an optimal balance between high- and low-entropy regions.

1.2. Rationale Behind Using Delaunay Triangulation in Sec. 3.4

In Sec. 3.4 *Ellipse Fitting*, we argue against directly predicting the scaling of each Gaussian using a network. Instead, we first perform Delaunay triangulation and then predict the offsets. The primary reasons for this choice are as follows:

Most deep learning methods handling similar tasks [4] adopt a normalized scale approach, such as learning relative sizes or operating in a specific feature space to enhance adaptability. While some studies directly predict pixel-level absolute values, such methods typically rely on fixed image sizes. If the input dimensions vary, these predictions may become invalid. Given that our task involves images of varying sizes, a normalized scale approach is necessary.

Several normalization strategies exist. One simple approach is to define an absolute scale based on the nearest

*Corresponding author.

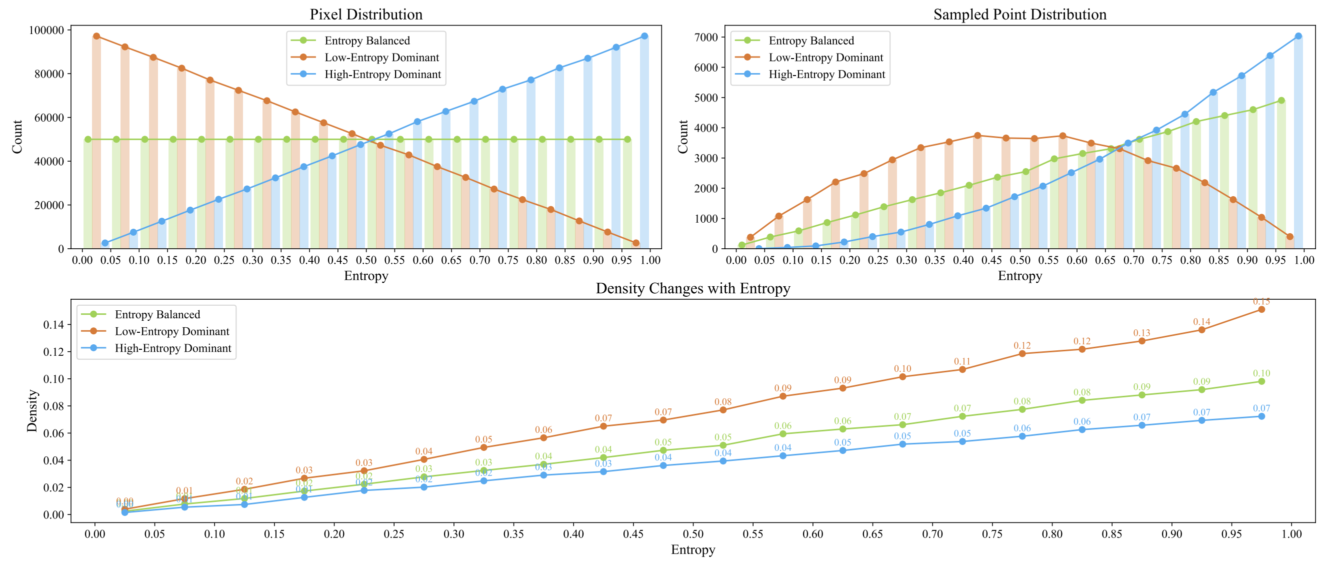

Figure 1. **CDF Sampling Analysis.** A simulated experiment analyzing the effects of CDF sampling under different entropy distributions. **Top left**: Pixel distribution for balanced, low-entropy dominant, and high-entropy dominant cases. **Top right**: Sampled point distribution after applying CDF sampling. **Bottom**: Normalized sampling density, showing redundancy in low-entropy dominant cases (overfitting) and insufficient representation in high-entropy dominant cases (underfitting).

neighbor distance and then learn a relative scale accordingly. While effective for uniformly distributed points, this method struggles with non-uniform distributions—closely spaced points may result in an absolute scale that fails to provide full coverage, increasing training complexity.

Thus, we prioritize an initialization strategy that ensures full image coverage while minimizing overlap between primitives to reduce learning complexity. To achieve this, we adopt **Delaunay Triangulation**.

Specifically, after obtaining the points via Floyd-Steinberg Dithering, we compute the average nearest-neighbor distance δ. To ensure full coverage, we insert additional boundary points at intervals of approximately 3δ along the image edges. After triangulation, we fit ellipses to the generated triangles using OpenCV's `fitEllipse` function [1], which requires at least six discrete points. To meet this requirement, we insert the midpoints of each triangle's three edges, yielding six points per triangle.

The fitted major and minor axes are scaled by 0.5 to serve as the absolute scale of each Gaussian. The fitted ellipse centers and rotation angles are normalized to the range $[-1, 1]$.

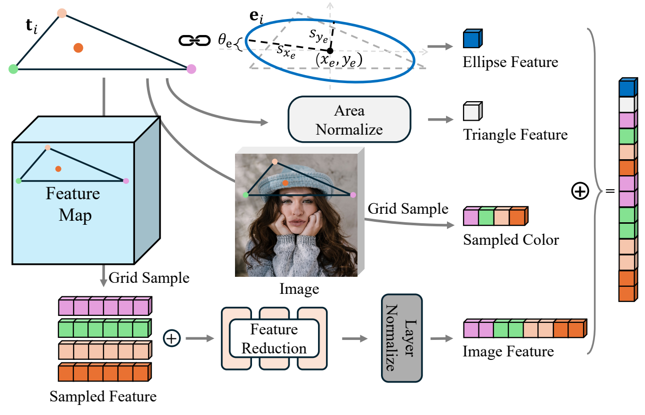

Figure 2. **Feature organization for MLP input.** We construct the MLP input by combining multiple feature components. Geometric features are extracted from the triangle $\mathbf{t}_i$ and its fitted ellipse $\mathbf{e}_i$, while grid sampling provides local deep features and sampled colors from the input image. The extracted features undergo feature reduction and layer normalization before concatenation, forming the final input vector for the MLP.

2. Details

2.1. Feature Extraction Details

Due to space limitations in the main paper, we provide a detailed explanation of how the features in Fig. 2 are obtained. These include the *Triangle Feature*, *Ellipse Feature*, *Sampled Color*, and *Image Feature*. After performing Delaunay triangulation, we obtain a set of triangle representations $\{\mathbf{t}_i\}_{i=0}^{T}$, where each $\mathbf{t}_i$ consists of the three vertex coordinates of the corresponding triangle:

$$\mathbf{t}_i = [\mathbf{v}_a, \mathbf{v}_b, \mathbf{v}_c]^T = \begin{bmatrix} x_a & y_a \\ x_b & y_b \\ x_c & y_c \end{bmatrix}, \tag{1}$$

Triangle Feature. Our goal is to normalize the triangle's area and obtain relative coordinates in a normalized space. Given a set of triangles represented by their three vertex coordinates $\mathbf{t}_i$, we first compute the area of each triangle as:

$$A_i = \frac{1}{2}\left|x_a(y_b - y_c) + x_b(y_c - y_a) + x_c(y_a - y_b)\right|. \quad (2)$$

To normalize the scale, we apply a transformation:

$$s_i = \sqrt{\frac{1}{A_i}}, \quad (3)$$

where s_i is the scaling factor. We then compute the triangle center:

$$\mathbf{c}_i = \frac{1}{3}(\mathbf{v}_a + \mathbf{v}_b + \mathbf{v}_c), \quad (4)$$

where $\mathbf{v}_a, \mathbf{v}_b, \mathbf{v}_c$ are the vertex coordinates. The final normalized triangle representation is given by:

$$\mathbf{t}_i^t = s_i(\mathbf{t}_i - \mathbf{c}_i), \quad (5)$$

which ensures that all triangles have a consistent scale while preserving their relative shape. The obtained normalized coordinates $\mathbf{t}_i^t$ serve directly as our triangle feature (dimension is 6).

Ellipse Feature. We adopt the ellipse fitting method described in [1], which requires at least six points. For each triangle, we construct a vertex set:

$$\mathbf{t}_i^e = \{\mathbf{v}_a, \mathbf{v}_b, \mathbf{v}_c, \frac{\mathbf{v}_a + \mathbf{v}_b}{2}, \frac{\mathbf{v}_b + \mathbf{v}_c}{2}, \frac{\mathbf{v}_a + \mathbf{v}_c}{2}\}. \quad (6)$$

Using these points, we fit an ellipse and obtain a set of ellipses $\{\mathbf{e}_i\}_{i=0}^T$, where each ellipse is characterized by its center position, major and minor axes, and rotation angle:

$$\mathbf{e}_i = \{(x_e, y_e), (s_{x_e}, s_{y_e}), \theta_e\}. \quad (7)$$

The final ellipse feature (dimension is 4) is represented as:

$$\mathbf{f}_i^e = \{x_e, y_e, \frac{s_{x_e}}{s_{y_e} + 10^{-6}}, e_\theta\}. \quad (8)$$

Color Feature To extract the color feature, we sample colors from the image at the three triangle vertices and the triangle center. The sampling positions are defined as:

$$\mathbf{t}_i^c = \{\mathbf{v}_a, \mathbf{v}_b, \mathbf{v}_c, \mathbf{c}_i\}, \quad (9)$$

where the triangle center is computed as:

$$\mathbf{c}_i = \frac{1}{3}(\mathbf{v}_a + \mathbf{v}_b + \mathbf{v}_c). \quad (10)$$

The color values at these positions are obtained using **grid sampling** on image I, which served directly as our color feature (dimension is 12).

$$\mathbf{f}_i^c = I(\mathbf{t}_i^c). \quad (11)$$

Image Feature Similar to the color feature extraction, we sample features from the 64-dimensional feature map produced by the ConvNeXt-based UNet at the three triangle vertices and the triangle center:

$$\mathbf{t}_i^f = \mathbf{t}_i^c. \quad (12)$$

The feature values at these positions are obtained using **grid sampling** on feature map F:

$$\mathbf{f}_i^f = F(\mathbf{t}_i^f). \quad (13)$$

The sampled features are then concatenated and processed through a feature reduction network:

$$\mathbf{f}_i^r = \text{ReLU}(\mathbf{W}_3\text{ReLU}(\mathbf{W}_2\text{ReLU}(\mathbf{W}_1\mathbf{f}_i))), \quad (14)$$

where $\mathbf{W}_1, \mathbf{W}_2, \mathbf{W}_3$ are linear transformation matrices reducing the feature dimension sequentially from $4d \to 4d \to 2d \to d$. Finally, the output feature is normalized using Layer Normalization. The final image feature dimension is $d = 64$.

2.2. Network Design

ConvNeXt-based UNet. For feature extraction, we employ a ConvNeXt-based UNet. The encoder uses the `base` model configuration with:

$$\text{depths} = [3, 3, 27, 3], \quad \text{dims} = [128, 256, 512, 1024]. \quad (15)$$

The decoder follows the implementation of Pytorch-UNet.

Position Field. The Position Field network transforms the feature of each pixel into a probability value to generate the Position Probability Map (PPM). It takes a 64-dimensional feature vector as input and outputs a single probability value:

$$\mathbf{p}_{\text{out}} = \sigma(\mathbf{W}_3\text{ReLU}(\mathbf{W}_2\text{ReLU}(\mathbf{W}_1\mathbf{p}_{\text{in}}))), \quad (16)$$

where $\mathbf{p}_{\text{in}} \in \mathbb{R}^{64}$ is the input feature, $\mathbf{W}_1, \mathbf{W}_2, \mathbf{W}_3$ are linear transformation matrices, and σ denotes the Sigmoid activation function to ensure output values are within $[0, 1]$.

BC Field. The BC Field network transforms the feature of each primitive (dim $= 86$) into barycentric coordinates (dim $= 3$):

$$\mathbf{bc}_{\text{out}} = \text{Softmax}(\mathbf{W}_3\text{ReLU}(\mathbf{W}_2\text{ReLU}(\mathbf{W}_1\mathbf{b}_{\text{in}})), \quad (17)$$

where $\mathbf{bc}_{\text{in}} \in \mathbb{R}^{86}$ is the input feature, $\mathbf{W}_1, \mathbf{W}_2, \mathbf{W}_3$ are linear transformation matrices, and the Softmax activation ensures that the output barycentric coordinates sum to 1.

Σ Field. The Σ Field predicts offsets for scaling and rotation, adjusting the major and minor axes as well as the orientation of the fitted ellipse. It takes a 94-dimensional feature vector as input and outputs a 3-dimensional transformation parameter:

$$\mathbf{\Sigma}_{\text{out}} = \mathbf{W}_3\text{ReLU}(\mathbf{W}_2\text{ReLU}(\mathbf{W}_1\mathbf{\Sigma}_{\text{in}})), \tag{18}$$

where $\mathbf{s}_{\text{in}} \in \mathbb{R}^{86}$ represents the input feature, and $\mathbf{W}_1, \mathbf{W}_2, \mathbf{W}_3$ are linear transformation matrices. The output consists of two scaling factors for the major and minor axes and one rotation offset.

Opacity Field. The Opacity Field predicts the opacity of each primitive based on its feature representation. It takes a 94-dimensional input feature and outputs a single scalar value:

$$o_{\text{out}} = \sigma(\mathbf{W}_3\text{ReLU}(\mathbf{W}_2\text{ReLU}(\mathbf{W}_1\mathbf{o}_{\text{in}}))), \tag{19}$$

where $\mathbf{o}_{\text{in}} \in \mathbb{R}^{86}$ is the input feature, $\mathbf{W}_1, \mathbf{W}_2, \mathbf{W}_3$ are linear transformation matrices, and σ (Sigmoid) ensures the output opacity value is within the range $[0, 1]$.

2.3. Obtain Pseudo PPM.

To train a network capable of predicting the Position Probability Map (PPM) that represents the Gaussian distribution, we first generate high-quality training data via Gaussian decomposition using GaussianImage. We begin with *quadtree-based image partitioning*, where blocks are recursively subdivided based on the mean squared error (MSE) of colors. If the MSE exceeds a predefined threshold, the block is split into four smaller ones, continuing until either the minimum block size of 4×4 is reached or the MSE falls below 0.02. This adaptive partitioning can approximately estimate the required number of Gaussians for good image representation, thus preventing the generation of a low-quality pseudo PPM due to the inadequate number of Gaussians. Next, Gaussians are initialized at the center of each block. Finally, we train GaussianImage for 50,000 iterations.

After Gaussian decomposition, we compute the Gaussian density at each pixel position $\mathbf{x}$ in the image. Specifically, for each $\mathbf{x}$, we estimate the minimal radius of a circle that encompasses K Gaussians, then the local Gaussian density $D_{\mathbf{x}}$ is defined as the unit density of Gaussians in this circle:

$$D_{\mathbf{x}} = \frac{K}{\pi \cdot \max\limits_{i} \left(\|\mathbf{p}_{\mathbf{x}} - \mathbf{p}_i\|\right)^2}, \tag{20}$$

where $\mathbf{p}$ represents position coordinates, and i indexes the top-K nearest neighbors of $\mathbf{x}$. In our experiments, we set $K = 10$. Next, we convert this density map into a PPM, which determines the probability of generating a Gaussian at each pixel location:

$$\mathbb{P}_{\text{pseudo}}(\mathbf{x}) = f(D_{\mathbf{x}}) = \mathcal{N}_{3\sigma}\left(\frac{K}{\max\limits_{i} \left(\|\mathbf{p}_{\mathbf{x}} - \mathbf{p}_i\|\right)^{1/2}}\right), \tag{21}$$

where f is a mapping function for a more smooth distribution and $\mathcal{N}_{3\sigma}$ denotes Three-Sigma Clipped Normalization, formally defined as:

$$\begin{aligned}
\mu &= \text{mean}(\mathbf{x}) \\
\sigma &= \text{std}(\mathbf{x}) \\
x_{\max} &= \min(\mu + 3\sigma, \max(\mathbf{x})) \\
x_{\min} &= \max(\mu - 3\sigma, \min(\mathbf{x})) \\
x' &= \frac{x - x_{\min}}{x_{\max} - x_{\min}} \\
x_{\text{normalized}} &= \text{clamp}(x', 0, 1)
\end{aligned} \tag{22}$$

2.4. Dithering on PPM.

A key aspect of our method is the discretization of the predicted PPM using Floyd-Steinberg Dithering. Direct per-pixel processing would generate an excessive number of sampled points; therefore, we apply *max pooling* to divide the PPM into $k \times k$ patches, where each patch's probability value is determined by the maximum probability within the patch. By adjusting k, we control the number of generated Gaussians to achieve different levels of rendering detail.For example, $k = 3$ for high-detail rendering while $k = 4$ for balanced rendering. To accelerate dithering, we implement a GPU-accelerated version based on [2]. After obtaining the sampled points, we apply *upsampling* to restore their positions to the original resolution.

3. More Experiments

In this section, we provide additional experimental results to validate the settings discussed in the main paper.

3.1. Additional Ablation Study

In the main paper, we noted that while certain settings may not significantly impact final evaluation results, they can introduce instability during training, making the model prone to local optima or even training failure. In Fig. 3, we present PSNR curves for both the training and test sets to illustrate these effects.

The **top row** corresponds to the training curves for "Dither with Img. Grad." and "Color Field," as discussed in Tab. 3 of the main paper. Compared to our final model, predicting Gaussian colors directly instead of opacity leads to slower convergence and a tendency to get trapped in local optima during mid-stage training. While the final model performance remains similar, the opacity-based approach results in a more efficient training process. Furthermore, "Dither with Img. Grad." struggles due to the significant gap between image gradients and the true Gaussian distribution density, leading to both lower performance and a higher likelihood of convergence issues.

The **bottom row** shows an ablation study on feature organization, evaluating the necessity of different features.

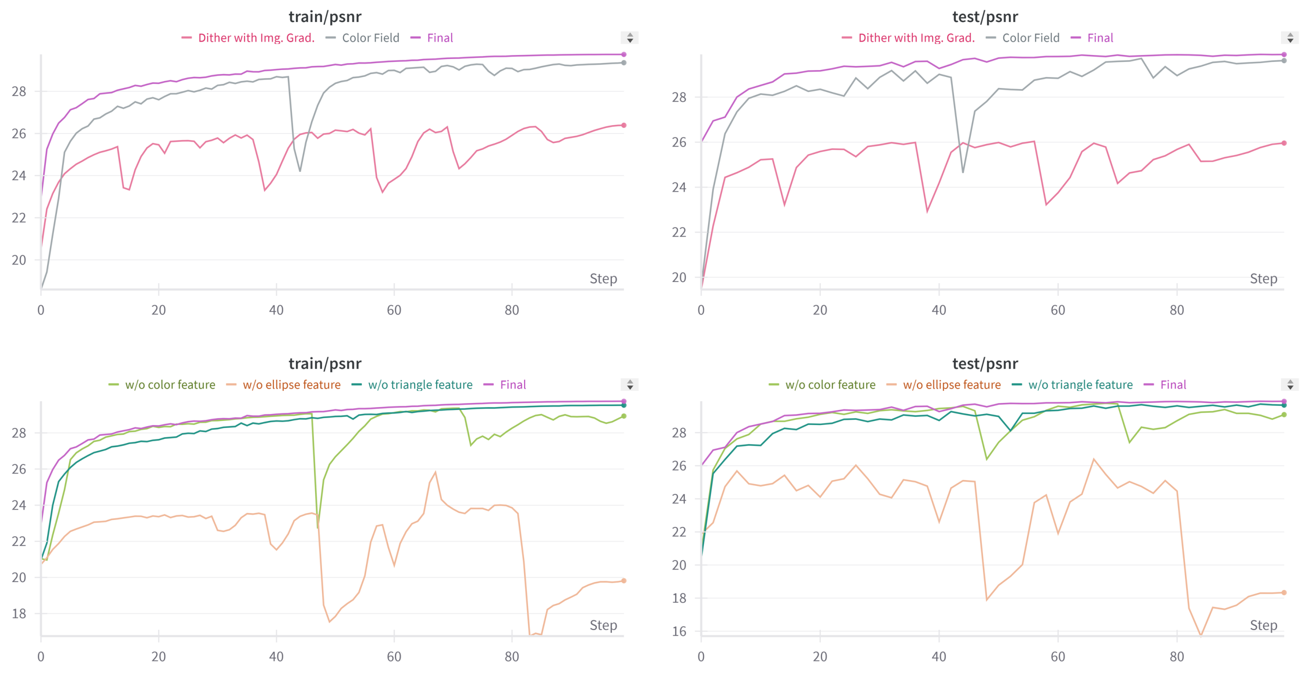

Figure 3. PSNR curves during training process.

We use image features as the default part and analyze the effects of including or excluding *color features*, *ellipse features*, and *triangle features*. The results show that ellipse features are crucial for the final performance, while triangle and color features have a smaller impact but still contribute to faster convergence and improved training stability by preventing local optima.

3.2. Additional Visualization Results

Due to space limitations in the main paper, we provide additional rendering visualizations and comparisons here. We randomly select several images from both the Kodak and Div2k datasets.

In Fig. 4, we compare Gaussian representation initialization using our network (**top row**) and GaussianImage's random initialization strategy (**bottom row**). The visualization includes the initial results, rendering outputs at the 2s mark (including initialization time), and detailed comparisons. From the visual results, our method produces high-quality initialization, leading to superior rendering even before fine-tuning. At the 2s mark, it achieves *near-Ground-Truth* quality, demonstrating finer details compared to GaussianImage.

In Fig. 5, we compare the output of our network-based initialization at the 2s mark with the output of random initialization at 10s. From the detailed comparisons, it is evident that our method achieves results at 2s that nearly achieve or even surpass those obtained after 10s of training with random initialization, further demonstrating the efficiency of our approach.

References

[1] Andrew W Fitzgibbon, Robert B Fisher, et al. *A buyer's guide to conic fitting*. Citeseer, 1996. 2, 3

[2] Giorgia Franchini, Roberto Cavicchioli, and Jia Cheng Hu. Stochastic floyd-steinberg dithering on gpu: image quality and processing time improved. In *2019 Fifth International Conference on Image Information Processing (ICIIP)*, pages 1–6, 2019. 4

[3] Bernhard Kerbl, Georgios Kopanas, Thomas Leimkühler, and George Drettakis. 3d gaussian splatting for real-time radiance field rendering. *ACM Trans. Graph.*, 42(4):139–1, 2023. 1

[4] Tao Lu, Mulin Yu, Linning Xu, Yuanbo Xiangli, Limin Wang, Dahua Lin, and Bo Dai. Scaffold-gs: Structured 3d gaussians for view-adaptive rendering. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 20654–20664, 2024. 1

[5] Yunxiang Zhang, Alexandr Kuznetsov, Akshay Jindal, Kenneth Chen, Anton Sochenov, Anton Kaplanyan, and Qi Sun. Image-gs: Content-adaptive image representation via 2d gaussians. *arXiv preprint arXiv:2407.01866*, 2024. 1

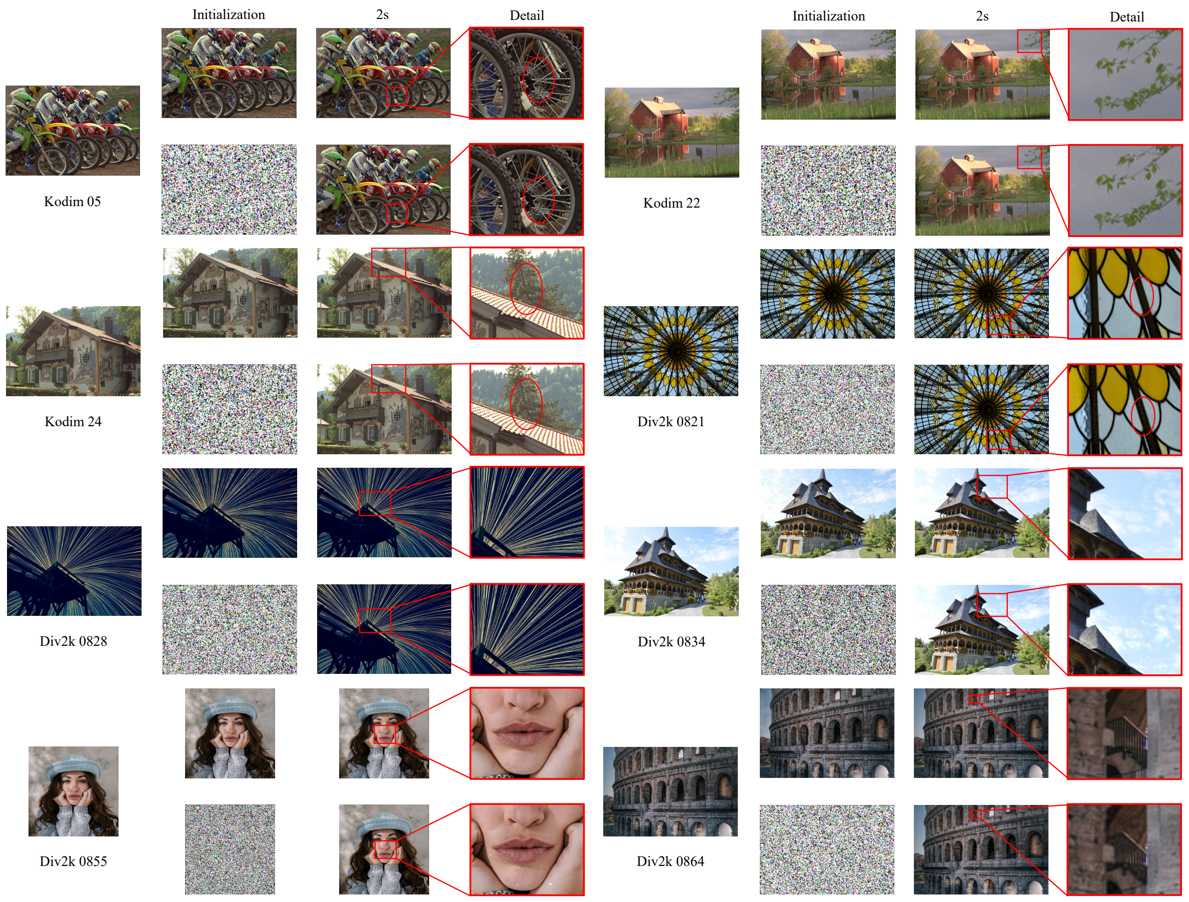

Figure 4. More visualization Results. Please zoom in for more details.

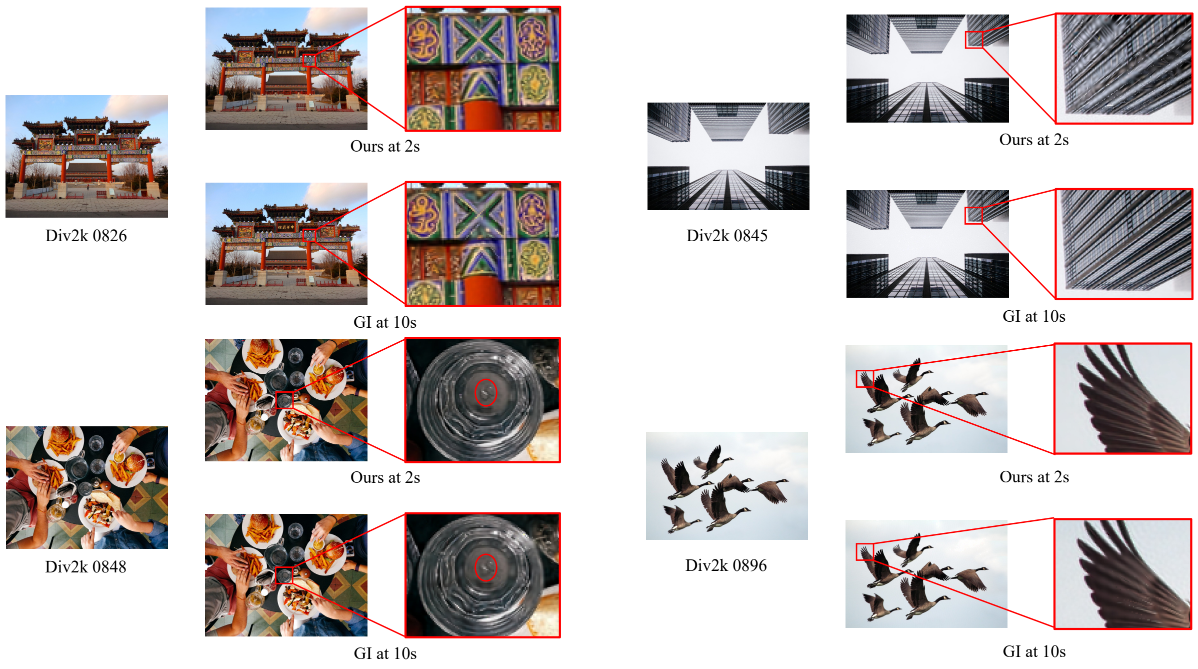

Figure 5. More visualization Results. Please zoom in for more details.