ReSplat: Learning Recurrent Gaussian Splatting

Haofei Xu[1,2], Daniel Barath[1], Andreas Geiger[2,3], and Marc Pollefeys[1,4]

[1]ETH Zurich [2]University of Tübingen, Tübingen AI Center [3]KE:SAI [4]Microsoft

Fig. 1: Learning recurrent Gaussian splatting in a feed-forward manner. ReSplat iteratively refines 3D Gaussian Splatting (3DGS) [24] for sparse view synthesis (2–32 views), a challenging regime where optimization-based 3DGS typically struggles. For initialization (iteration 0), we introduce a compact reconstruction model that predicts Gaussians in a 16× subsampled space. This yields 16× fewer Gaussians and 4× faster rendering compared to per-pixel baselines MVSplat [6] and DepthSplat [55]. The reduced Gaussian count enables highly efficient subsequent refinement. Compared to the optimization-based 3DGS [24], ReSplat is 100× faster thanks to its feed-forward design, while maintaining the benefits of iterative updates. Here we show the results for 8 input views (512 × 960 resolution) on the DL3DV [27] dataset (metrics in Tab. 1).

Abstract. While existing feed-forward Gaussian splatting models offer computational efficiency and can generalize to sparse view settings, their performance is fundamentally constrained by relying on a single forward pass for inference. We propose ReSplat, a feed-forward recurrent Gaussian splatting model that iteratively refines 3D Gaussians without explicitly computing gradients. Our key insight is that the Gaussian splatting rendering error serves as a rich feedback signal, guiding the recurrent network to learn effective Gaussian updates. This feedback signal naturally adapts to unseen data distributions at test time, enabling robust generalization across datasets, view counts, and image resolutions. To initialize the recurrent process, we introduce a compact reconstruction model that operates in a 16× subsampled space, producing 16× fewer Gaussians than previous per-pixel Gaussian models. This substantially reduces computational overhead and allows for efficient Gaussian updates. Extensive experiments across varying number of input views (2, 8, 16, 32), resolutions (256×256 to 540×960), and datasets (DL3DV, RealEstate10K, and ACID) demonstrate that our method achieves state-of-the-art performance while significantly reducing the number of Gaussians and improving the rendering speed. Our project page is at haofeixu.github.io/resplat.

Keywords: Feed-Forward Gaussian Splatting · Learning to Optimize · Iterative Refinement

1 Introduction

Feed-forward Gaussian splatting [4, 44] aims to directly predict 3D Gaussian parameters from input images, eliminating the need for expensive per-scene optimization [24] and enabling high-quality sparse-view reconstruction and view synthesis [6, 30, 50, 62]. Very recently, significant progress has been made in this line of research: feed-forward models [7, 10, 55, 58, 61] can now produce promising reconstruction and view synthesis results from sparse input views.

Despite these advances, performance remains largely concentrated on standard in-domain benchmarks [27, 64], and existing feed-forward models often struggle to generalize to unseen scenarios. Most current methods [4, 6, 10, 55, 61] learn a single-step mapping from images to 3D Gaussians, an approach inherently limited by network capacity in complex scenes. In contrast, per-scene optimization [24] achieves high-quality results via iterative updates but is computationally expensive, requiring thousands of gradient-based updates. This motivates our approach: using learned recurrent steps to progressively improve reconstruction, balancing feed-forward efficiency with adaptability of iterative optimization.

We identify that the rendering error provides a valuable feedback signal, informing the model about the quality of its prediction. This allows the network to adapt to the test data, reducing the dependence on the training distribution and leading to robust generalization. Furthermore, by iterating this process, the model incrementally refines its prediction. This recurrent mechanism reduces learning difficulty by decomposing the task and increases model expressiveness with each update, effectively simulating a deeper, unrolled network [3, 16].

Driven by this observation, we begin with a single-step feed-forward Gaussian reconstruction model to initialize the recurrent process and then perform recurrent updates to improve the initial Gaussians. Since the recurrent updates occur in 3D space, where a large number of Gaussians would impose a significant computational burden, we design our initial model to predict Gaussians in a $16\times$ subsampled space. This contrasts with most existing feed-forward models [4, 43, 44, 61] that predict one Gaussian per pixel, which scales poorly with increasing number of views and image resolutions. Our method achieves a $16\times$ reduction in the number of Gaussians while maintaining performance.

Based on this compact initial reconstruction, we train a weight-sharing recurrent network that iteratively improves the initial prediction. Crucially, the network leverages the rendering error of the input views to determine how to update the Gaussians. Specifically, we render the input views (available at test time) using the current prediction, compute the rendering error, and propagate it to the 3D Gaussians. The recurrent network then predicts the parameter updates from this error and the current Gaussians, without explicit gradient computation.

We validate our method through extensive experiments across diverse scenarios. On the challenging DL3DV [27] dataset, using 8 input views at 512×960 resolution, our learned recurrent model improves PSNR by 3.5dB, while using only 1/16 of the Gaussians and achieving $4\times$ faster rendering speed. We also demonstrate that our recurrent model leads to robust generalization to unseen datasets, view counts, and image resolutions, where previous single-step feed-forward models

usually struggle. With 16 input views at 540×960 resolution, we outperform Long-LRM [10] by 0.8dB PSNR while using $4\times$ fewer Gaussians. On the commonly used two-view RealEstate10K [64] and ACID [29] benchmarks, ReSplat also achieves state-of-the-art results, demonstrating its strong performance.

2 Related Work

Feed-Forward Gaussian Splatting. Significant progress has recently been made in feed-forward Gaussian models [10, 21, 31, 55, 57, 61]. However, two major limitations persist: First, most existing feed-forward models predict one or multiple Gaussians for each pixel [4, 6, 10, 43, 61], which produces millions of Gaussians when handling many input views and/or high-resolution images and thus limits scalability. Second, most existing methods are developed with single-step feed-forward inference. While conceptually simple, the achievable quality is bounded by network capacity for challenging and complex scenes. In this paper, we overcome these two limitations by first reconstructing Gaussians in a $16\times$ subsampled space, and then performing recurrent Gaussian updates based on the rendering error, which significantly improves efficiency and quality. Unlike SplatFormer [9], which introduces a *single-step non-recurrent* refinement network for *optimized* 3DGS parameters, we propose a *weight-sharing recurrent* network to iteratively improve the results from a *feed-forward initialization*. In addition, SplatFormer is evaluated only on object-centric datasets and it is non-trivial to make it work for complex scenes (it is 2dB PSNR worse than our method). In contrast, our ReSplat targets scene-level benchmarks, and we demonstrate the effectiveness of the rendering error as an informative feedback signal, which we find crucial but is missing in SplatFormer.

Learning to Optimize. Many tasks in machine learning and computer vision can be formulated as minimization problems with an optimization objective, where the solutions are found by iterative gradient descent [1, 35, 42]. Modern approaches [19, 36, 37, 45] try to simulate the optimization process by iteratively updating an initial prediction with a weight-sharing network, which usually achieves superior results compared to single-step regression methods, especially for out-of-distribution generalization. In vision, such a framework has been successfully applied to optical flow [45], stereo matching [28, 51], scene flow [47], SLAM [46], Structure-from-Motion [26], and Multi-View Stereo [49]. Unlike prior work that often relies on feature correlations [45] for the recurrent process, we investigate this paradigm for the feed-forward Gaussian splatting task and identify the Gaussian rendering error as an informative feedback signal.

Learning to Optimize for View Synthesis. In the context of view synthesis, DeepView [15] predicts multi-plane images with learned gradient descent, where explicit gradient computation is necessary. In addition, G3R [8] learns to iteratively refine 3D Gaussians with the guidance of the explicitly computed gradients. However, our method is gradient free. Moreover, G3R requires well-covered 3D points for initialization and struggles with sparse points, while we directly predict initial Gaussians from posed images, without requiring any initial 3D points. Like

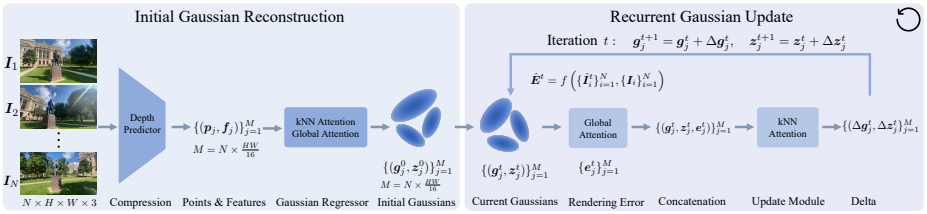


Fig. 2: ReSplat. Given N posed input images, we first predict per-view depth maps at $1/4$ resolution and then unproject and transform them into a point cloud with image features $\{(\boldsymbol{p}_j, \boldsymbol{f}_j)\}_{j=1}^{M}$, where $M = N \times \frac{HW}{16}$ is the number of points. We then reconstruct an initial set of 3D Gaussians $\{(\boldsymbol{g}_j^0, \boldsymbol{z}_j^0)\}_{j=1}^{M}$ in a $16\times$ subsampled 3D space using a kNN and global attention-based Gaussian regressor. Next, we learn to refine the initial Gaussians recurrently. At each recurrent step t, we use the current Gaussian prediction to render input views and then compute the rendering errors $\hat{\boldsymbol{E}}^t$ between rendered and ground-truth input views. Global attention is then applied on the rendering error to propagate these errors to the 3D Gaussians. A kNN attention-based update module then takes as input the concatenation of the current Gaussian parameters $\boldsymbol{g}_j^t$, the hidden state $\boldsymbol{z}_j^t$, and the rendering error $\boldsymbol{e}_j^t$, and predicts the incremental updates $\Delta\boldsymbol{g}_j^t$ and $\Delta\boldsymbol{z}_j^t$. We iterate this process for a total of T steps.

G3R, QuickSplat [32] also relies on gradient computation but focuses on surface reconstruction. Another related work LIFe-GOM [52] iteratively updates the 3D reconstruction, but it focuses on human avatars with a hybrid Gaussian-mesh 3D representation. In contrast, our method aims to improve the quality and generalization of feed-forward Gaussian splatting models for general scenes.

3 Approach

Given N input images $\{\boldsymbol{I}^i\}_{i=1}^{N} (\boldsymbol{I}^i \in \mathbb{R}^{H\times W\times 3})$ with their intrinsic $\{\boldsymbol{K}^i\}_{i=1}^{N}$ $(\boldsymbol{K}^i \in \mathbb{R}^{3\times 3})$ and extrinsic $\{(\boldsymbol{R}_i, \boldsymbol{t}_i)\}_{i=1}^{N}$ $(\boldsymbol{R}_i \in \mathrm{SO}(3), \boldsymbol{t}_i \in \mathbb{R}^3)$ matrices, our goal is to predict a set of 3D Gaussian primitives [24] $\mathcal{G} = \{(\boldsymbol{\mu}_j, \alpha_j, \boldsymbol{\Sigma}_j, \mathbf{sh}_j)\}_{j=1}^{M}$ to model the scene, where M is the total number of Gaussian primitives and $\boldsymbol{\mu}_j$, α_j, $\boldsymbol{\Sigma}_j$, and $\mathbf{sh}_j$ are the 3D Gaussian's position, opacity, covariance, and spherical harmonics, respectively. The reconstructed 3D Gaussians can be efficiently rasterized, enabling fast and high-quality novel view synthesis.

Unlike previous feed-forward models [4, 10, 61] that perform a single-step feed-forward prediction, we learn to estimate the Gaussian parameters recurrently. This not only reduces learning difficulty by decomposing the task into smaller, incremental steps but also enables higher reconstruction quality. In particular, we first predict an initial set of 3D Gaussians and then iteratively refine them in a gradient-free, feed-forward manner. Given that the Gaussian update occurs in 3D space, a large number of 3D Gaussians will introduce significant computational overhead during the update process. Thus, in our initial reconstruction stage, we predict a compact set of 3D Gaussians in a $16\times$ subsampled space. More specifically, we perform $4\times$ spatial compression when predicting per-view depth maps, which leads to $16\times$ fewer Gaussians compared to previous per-pixel representations [4, 44]. Consequently, the number of Gaussians M in our model is

$N \times \frac{HW}{16}$, which scales efficiently to many input views and high-resolution images. Fig. 2 provides an overview of our pipeline.

3.1 Initial Gaussian Reconstruction

Subsampled 3D Space. Our initial Gaussian reconstruction model is based on the DepthSplat [55] architecture. However, unlike DepthSplat, we predict Gaussians in a spatially 16× subsampled 3D space ($N \times \frac{HW}{16}$), and thus we produce 16× fewer Gaussians than DepthSplat. To achieve 16× subsampling, we resize the full-resolution depth predictions from the depth model in DepthSplat to 1/4 resolution ($N \times \frac{H}{4} \times \frac{W}{4}$), and then unproject and transform them into 3D via camera parameters to obtain a point cloud with $M = N \times \frac{HW}{16}$ points. Each 3D point $\boldsymbol{p}_j \in \mathbb{R}^3$ is also associated with a feature vector $\boldsymbol{f}_j \in \mathbb{R}^{C_0}$ ($C_0 = 256$ for our small model and 512 for our base model) extracted from the input images:

$$\{\boldsymbol{I}_i, \boldsymbol{K}_i, \boldsymbol{R}_i, \boldsymbol{t}_i\}_{i=1}^N \rightarrow \{(\boldsymbol{p}_j, \boldsymbol{f}_j)\}_{j=1}^M. \tag{1}$$

Since we now have 16× fewer 3D points, naïvely predicting Gaussian parameters from the point features $\boldsymbol{f}_j$ will lead to considerable performance loss. However, we find that using additional kNN attention [63] and global attention [48] layers to encode the 3D context [5, 54] information can compensate for this loss.

Aggregating the 3D Context. We use six alternating blocks of kNN attention and global attention to model both local and global 3D contexts, which enables communication between different 3D points and produces 3D context-aggregated features $\boldsymbol{f}_j^* \in \mathbb{R}^{C_0}$ with increased expressiveness:

$$\{(\boldsymbol{p}_j, \boldsymbol{f}_j)\}_{j=1}^M \rightarrow \{(\boldsymbol{p}_j, \boldsymbol{f}_j^*)\}_{j=1}^M. \tag{2}$$

Decoding to Gaussians. We use the point cloud $\{\boldsymbol{p}_j\}_{j=1}^M$ as the Gaussian centers, and other Gaussian parameters are decoded using a lightweight Gaussian head (two-layer MLP) from the 3D context-aggregated features $\{\boldsymbol{f}_j^*\}_{j=1}^M$. Accordingly, we obtain an initial set of 3D Gaussians with parameters $\{(\boldsymbol{\mu}_j, \alpha_j, \boldsymbol{\Sigma}_j, \mathbf{sh}_j)\}_{j=1}^M$ and feature vectors $\{\boldsymbol{f}_j^*\}_{j=1}^M$. We use $\boldsymbol{g}_j^0 \in \mathbb{R}^{C_1} (C_1 = 59)$ to denote the concatenation of all the Gaussian parameters $(\boldsymbol{\mu}_j, \alpha_j, \boldsymbol{\Sigma}_j, \mathbf{sh}_j)$ for the j-th Gaussian at initialization, where C_1 is the total number of parameters for each Gaussian. We use $\boldsymbol{z}_j^0$ to denote the initial hidden state of the j-th Gaussian for the subsequent recurrent process, and initialize it with the feature $\boldsymbol{f}_j^*$: $\boldsymbol{z}_j^0 = \boldsymbol{f}_j^* \in \mathbb{R}^{C_0}$. Thus, the initial Gaussians can be represented as

$$\mathcal{G}^0 = \{(\boldsymbol{g}_j^0, \boldsymbol{z}_j^0)\}_{j=1}^M. \tag{3}$$

3.2 Recurrent Gaussian Update

Based on the initial Gaussian prediction in Sec. 3.1 (Eq. (3)), we train a recurrent network that iteratively refines the initial prediction. In particular, at iteration

t ($t = 0, 1, \cdots, T-1$, where T is the total number of iterations), the recurrent network predicts incremental updates to all Gaussian parameters $\Delta \boldsymbol{g}_j^t \in \mathbb{R}^{C_1}$ and their hidden state $\Delta \boldsymbol{z}_j^t \in \mathbb{R}^{C_0}$ as:

$$\boldsymbol{g}_j^{t+1} = \boldsymbol{g}_j^t + \Delta \boldsymbol{g}_j^t, \quad \boldsymbol{z}_j^{t+1} = \boldsymbol{z}_j^t + \Delta \boldsymbol{z}_j^t. \tag{4}$$

To predict the incremental updates $\Delta \boldsymbol{g}_j^t$ and $\Delta \boldsymbol{z}_j^t$, we propose to learn the update in a gradient-free, feed-forward manner from the rendering error of input views.

Computing the Rendering Error. Given that we have access to the input views at test time, we are able to create a feedback loop to guide the recurrent network to learn the incremental updates. Specifically, we first render the input views $\{\hat{\boldsymbol{I}}_i^t\}_{i=1}^N$ based on the current Gaussian parameters at iteration t and then measure the difference between the rendered and ground-truth input views. We evaluate several different methods to compute the rendering error and observe that a combination of pixel-space and feature-space errors performs best.

In particular, we first use $\{\hat{\boldsymbol{I}}_i^t - \boldsymbol{I}_i\}_{i=1}^N$ to measure the rendering error in the pixel space, and then perform $4\times$ spatial downsampling with pixel unshuffle to align with the number of 3D Gaussians. For the feature-space rendering error, we extract the first three stage features (at 1/2, 1/4 and 1/8 resolutions) of the ImageNet [13] pre-trained ResNet-18 [20] for the rendered input views and ground-truth input views, and bilinearly resize the three scale features to the same 1/4 resolution, followed by concatenation. We denote the extracted features as $\{\hat{\boldsymbol{F}}_i^t\}_{i=1}^N$ and $\{\boldsymbol{F}_i\}_{i=1}^N$ ($\hat{\boldsymbol{F}}_i^t, \boldsymbol{F}_i \in \mathbb{R}^{\frac{H}{4} \times \frac{W}{4} \times C_2}$, where $C_2 = 256$) for the rendered and ground-truth input views, respectively. We then compute the difference between the features with subtraction $\{\hat{\boldsymbol{F}}_i^t - \boldsymbol{F}_i\}_{i=1}^N$. We combine pixel-space and feature-space rendering errors via element-wise addition. To match channel dimensions, the pixel-space error is first projected to the feature space using a linear layer followed by Layer Normalization [2]. This can be expressed as

$$\hat{\boldsymbol{E}}^t = f\left(\{\hat{\boldsymbol{I}}_i^t\}_{i=1}^N, \{\boldsymbol{I}_i\}_{i=1}^N\right) = \{\hat{\boldsymbol{F}}_i^t - \boldsymbol{F}_i\}_{i=1}^N + \text{proj}(\{\hat{\boldsymbol{I}}_i^t - \boldsymbol{I}_i\}_{i=1}^N), \tag{5}$$

where "proj" is the operation mentioned before to match dimensions. We denote all rendering errors as $\hat{\boldsymbol{E}}^t = \{\hat{\boldsymbol{e}}_j^t\}_{j=1}^{N \times \frac{H}{4} \times \frac{W}{4}}$, where $\hat{\boldsymbol{e}}_j^t \in \mathbb{R}^{C_2}$ is the j-th feature difference of dimension C_2 at iteration t.

Propagating the Rendering Error to Gaussians. To propagate the rendering error to 3D Gaussians such that they can guide the network to update the Gaussians, a straightforward approach is to concatenate the rendering error $\hat{\boldsymbol{e}}_j^t$ with the Gaussians $(\boldsymbol{g}_j^t, \boldsymbol{z}_j^t)$ in a spatially aligned manner, since they have the same number of points ($N \times \frac{HW}{16}$). However, with this approach, the j-th Gaussian can only receive local information around the j-th rendered pixel, even though it can also contribute to other rendered pixels during the rendering process. To propagate the rendering error more effectively, we propose to apply global attention across all the $N \times \frac{H}{4} \times \frac{W}{4}$ rendering errors $\hat{\boldsymbol{E}}^t$, which enables each Gaussian to receive information from all rendering errors. This process can be formulated as follows:

$$\boldsymbol{E}^t = \text{global\_attention}(\hat{\boldsymbol{E}}^t) = \{\boldsymbol{e}_j^t\}_{j=1}^{N \times \frac{HW}{16}}, \tag{6}$$

where $\boldsymbol{e}_j^t$ is the j-th rendering error, which has aggregated the original point-wise rendering error $\hat{\boldsymbol{e}}_j^t$ globally. We then concatenate the Gaussians with the globally aggregated rendering errors as $\{(\boldsymbol{g}_j^t, \boldsymbol{z}_j^t, \boldsymbol{e}_j^t)\}_{j=1}^M$, which are then used to predict the incremental update (illustrated in Fig. 2).

Recurrent Gaussian Update. Letting the Gaussians at iteration t be $\mathcal{G}^t = \{(\boldsymbol{g}_j^t, \boldsymbol{z}_j^t)\}_{j=1}^M$, our update module predicts the incremental updates of Gaussian parameters and hidden state as:

$$\{(\boldsymbol{g}_j^t, \boldsymbol{z}_j^t, \boldsymbol{e}_j^t)\}_{j=1}^M \rightarrow \{(\Delta\boldsymbol{g}_j^t, \Delta\boldsymbol{z}_j^t)\}_{j=1}^M. \tag{7}$$

These updates are then added to the current prediction (Eq. (4)). This process is iterated T times. We observe that our model converges after 4 iterations. During training, we randomly sample the number of iterations T between 1 and 4, and our model supports a different number of iterations at inference time, allowing a flexible speed-accuracy trade-off with a single model. Since the recurrent process occurs in 3D space, we choose to use four kNN attention [63] blocks as the recurrent architecture to model the local structural details. The Gaussian updates $\boldsymbol{g}_j^t$ are decoded with a lightweight head (four-layer MLP).

3.3 Training Loss

Our model is trained in two stages. In the first stage, we train an initial Gaussian reconstruction model to provide a compact initialization to our subsequent updates. The training loss is a combination of a rendering loss ℓ_{render} and a depth smoothness loss [18] ℓ_{smooth} on the predicted depth maps of the input views:

$$L_{\text{1st}} = \sum_{v=1}^{V} \ell_{\text{render}}(\hat{\boldsymbol{I}}_v, \boldsymbol{I}_v) + \alpha \cdot \sum_{i=1}^{N} \ell_{\text{smooth}}(\boldsymbol{I}_i, \hat{\boldsymbol{D}}_i), \tag{8}$$

$$\ell_{\text{render}}(\hat{\boldsymbol{I}}, \boldsymbol{I}) = \ell_1(\hat{\boldsymbol{I}}, \boldsymbol{I}) + \lambda \cdot \ell_{\text{perceptual}}(\hat{\boldsymbol{I}}, \boldsymbol{I}), \tag{9}$$

$$\ell_{\text{smooth}}(\boldsymbol{I}, \hat{\boldsymbol{D}}) = |\partial_x \hat{\boldsymbol{D}}| e^{-|\partial_x \boldsymbol{I}|} + |\partial_y \hat{\boldsymbol{D}}| e^{-|\partial_y \boldsymbol{I}|}, \tag{10}$$

where V is the number of target views to render in each training step, and N is the number of input views. The perceptual loss $\ell_{\text{perceptual}}$ [23] measures the distance in VGG [41] feature space, which is also used in previous methods [22, 61]. The depth smoothness loss ℓ_{smooth} does not require ground-truth depth and serves as a regularization term on the estimated depth maps of the input views to encourage the depth gradient to be similar to the image gradient [17, 18]. We use $\alpha = 0.01$ and $\lambda = 0.5$ for all the experiments.

In the second stage, we freeze our initial reconstruction model and train only the recurrent model end-to-end. We use the rendering loss ℓ_{render} of rendered and ground-truth target views to supervise the network. All Gaussian predictions during the recurrent process are supervised using the rendering loss, applying exponentially ($\gamma = 0.9$) increasing weights:

$$L_{\text{2nd}} = \sum_{t=0}^{T-1} \gamma^{T-1-t} \sum_{v=1}^{V} \ell_{\text{render}}(\hat{\boldsymbol{I}}_v^t, \boldsymbol{I}_v). \tag{11}$$

4 Experiments

Implementation Details. We implement our method in PyTorch [39]. We choose $k = 16$ for kNN attention in the initialization model following Point Transformer [63], and we use $k = 8$ for the recurrent model to focus more on local details. Our Gaussian splatting renderer is based on the Mip-Splatting [60] implementation in gsplat [59]. We optimize our model with the AdamW [33] optimizer. More training details are presented in the appendix.

Efficient Global Attention Implementation. Our model contains several global attention layers. Considering that performing global attention on $N \times \frac{H}{4} \times \frac{W}{4}$ features would be expensive for high-resolution images, we first perform $4\times$ spatial downsampling with pixel unshuffle (reshaping from the spatial dimension to the channel dimension) and then compute global attention on the $N \times \frac{H}{16} \times \frac{W}{16}$ features. Finally, we upsample the features back to $1/4$ resolution using pixel shuffle (reshaping from the channel dimension to the spatial dimension). This implementation enables our model to scale efficiently to high-resolution images.

Efficient kNN Implementation. For typical point counts (e.g., < 300K), we default to a customized CUDA implementation [12] of global kNN. However, because this GPU implementation has $O(N^2)$ complexity, it becomes a computational bottleneck at higher resolutions or view counts. To ensure scalability in these regimes, we introduce an efficient local kNN alternative that reduces complexity to $O(N \cdot C)$ by restricting the search to a small candidate set of size C. This set is constructed using the known camera parameters to aggregate spatial neighbors from the same view and cross-view neighbors projected from nearby cameras. Computing exact 3D distances strictly within this constant-sized set guarantees efficient top-k selection even for massive point clouds. We provide more computational analysis in the appendix.

Model Sizes. Our default model (ReSplat-Base) uses a ViT-B [14, 38, 56] backbone as part of our depth prediction model, which has 223M parameters in total (209M for the initialization model and 14M for the recurrent model). For ablation experiments, we use a ViT-S backbone (ReSplat-Small) to save compute, which has 77M parameters in total (62M for the initialization model and 15M for the recurrent model). In Tab. 2, we additionally train a large initialization model (ReSplat-Large, 559M) using a ViT-L backbone to evaluate scalability.

To facilitate reproducibility, we release our code and pre-trained models publicly at `https://github.com/cvg/resplat`.

Coordinate System. Since our recurrent network operates within a global 3D space, the selection of a coordinate system is critical, as it directly determines the spatial distribution of the Gaussian's centers. For our datasets, camera poses are estimated from COLMAP [40]. We evaluated aligning the global reference frame to various views within the sparse input set. Empirically, we observed that using the spatially central (*e.g.*, the middle frame in a sequential trajectory) input view as the reference coordinate system yields the best performance (see Tab. 6b). We posit that this centers the coordinate system, reducing the maximum transformation distance to the most distant input views and effectively balancing the spatial positions of the 3D Gaussians.

Table 1: Evaluation of 8 input views (512×960) on DL3DV. The standard optimization-based approach 3DGS [24] requires several thousand iterations to reach convergence, while our feed-forward ReSplat is 100× faster and benefits from additional iterations. In contrast to per-pixel feed-forward models such as MVSplat [6] and DepthSplat [55], which produce millions of Gaussians, our ReSplat compresses the Gaussian count by 16×, resulting in a 4× faster rendering speed.

Method	Category	#Iterations	PSNR ↑	SSIM ↑	LPIPS ↓	#Gaussians	Recon. Time (s)	Render Time (s)
3DGS [24]	Optimization	1000	20.36	0.667	0.448	9K	15	0.0001
		2000	23.18	0.763	0.269	137K	31	0.0005
		3000	23.42	0.770	0.232	283K	50	0.0008
		4000	23.46	0.770	0.224	359K	70	0.0009
MVSplat [6]	Feed-Forward	0	22.49	0.764	0.261	3932K	0.129	0.0030
DepthSplat [55]	Feed-Forward	0	24.17	0.815	0.208	3932K	0.190	0.0030
ReSplat	Feed-Forward	0	26.21	0.842	0.185	246K	0.311	0.0007
		1	27.15	0.859	0.169	246K	0.437	0.0007
		2	27.51	0.865	0.163	246K	0.563	0.0007
		3	27.65	0.867	0.161	246K	0.689	0.0007
		4	**27.70**	**0.868**	**0.160**	246K	0.816	0.0007

Evaluation Settings. We mainly consider three evaluation settings. First, we evaluate view synthesis from 8 input views at 512×960 resolution on the DL3DV [27] dataset, where we retrain 3DGS [24], MVSplat [6], and DepthSplat [55] with their public code for fair comparisons. Second, we consider view synthesis from 16 input views at 540×960 resolution on DL3DV following Long-LRM [10]. Third, we evaluate on the commonly used 2-view (256×256) setting on RealEstate10K [64] and ACID [29], where we compare with 2-view methods like GS-LRM [61] and LVSM [22].

4.1 Main Results

8 Views at 512×960 Resolution. We report the results on the DL3DV [27] benchmark split (140 scenes) in Tab. 1. Regarding 3DGS [24], we perform per-scene optimization on the 8 input views for all 140 scenes, while for feed-forward models, we perform zero-shot inference. We observe that 3DGS optimization typically converges with 4K optimization steps, and optimizing longer can lead to overfitting due to the sparse input views; thus, we report the best results (at 4K iterations). As shown in Tab. 1, 3DGS optimization is computationally expensive due to the large number of iterations required, while our feed-forward ReSplat is 100× faster and is able to benefit from recurrent iterations. Previous per-pixel feed-forward models MVSplat [6] and DepthSplat [55] produce millions of Gaussians, while our ReSplat compresses the number of Gaussians by 16×, resulting in a 4× faster rendering speed. Overall, our ReSplat outperforms 3DGS by 4.2dB PSNR and DepthSplat by 3.5dB PSNR with superior efficiency on the number of Gaussians and the rendering speed. Visual comparisons provided in Fig. 3 and the appendix demonstrate the higher rendering quality of our method.

Optimization-Based *vs*. Feed-Forward Refinement. To further demonstrate the efficiency of our feed-forward refinement, we compare it against per-scene

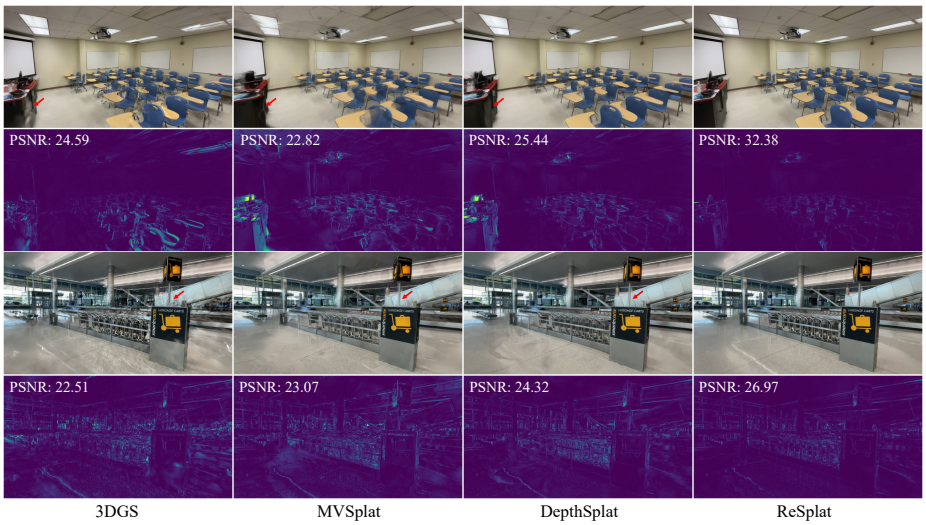
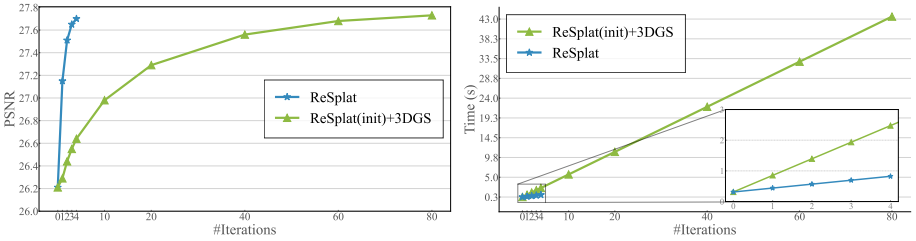


Fig. 3: View synthesis on DL3DV. Our ReSplat outperforms both optimization-based and feed-forward methods, demonstrating significantly smaller rendering errors.

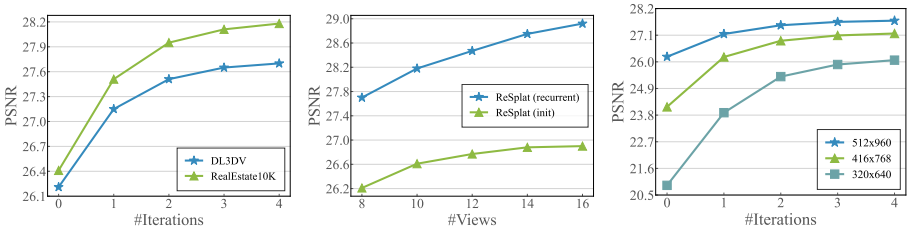


(a) PSNR *vs*. number of iterations. **(b) Time *vs*. number of iterations.**

Fig. 4: Optimization-based *vs*. feed-forward refinement. Starting from the *same ReSplat initialization*, we compare our feed-forward refinement against per-scene optimization using 3DGS [24]. Our ReSplat improves the rendering quality significantly faster (4 *vs*. 80 iterations) and provides a 53× speedup in reconstruction time. Furthermore, as highlighted in the zoomed-in region of (b), our per-iteration speed is faster than standard optimization since our approach eliminates the need for gradient computation.

optimization using the *same ReSplat initialization*. As illustrated in Fig. 4, our ReSplat is significantly faster than 3DGS optimization-based refinement thanks to our gradient-free, feed-forward architecture.

Generalization Across Datasets, View Counts, and Image Resolutions. We evaluate the generalization capability of our model, which is trained exclusively on DL3DV at 512 × 960 resolution with 8 input views. First, when generalizing to the unseen RealEstate10K dataset (Fig. 5a), the improvement yielded by our recurrent model is more significant since, unlike single-step feed-forward models (iteration 0), our model adapts to the test data via rendering error, thus mitigating the domain gap. Second, we evaluate our initial and recurrent models with varying input view counts in Fig. 5b, and observe that our recurrent



(a) Cross-dataset. **(b) Cross-view.** **(c) Cross-resolution.**

Fig. 5: Generalization to unseen datasets, view counts, and resolutions. Our recurrent model demonstrates robust generalization capabilities, despite being trained solely on DL3DV at a fixed resolution (512×960) with 8 input views.

Table 2: Single-step *vs*. recurrent models. Our recurrent ReSplat-Small (77M) outperforms all single-step baselines, including the significantly larger non-recurrent ReSplat-Large (559M), AnySplat (886M) and WorldMirror (1263M). This demonstrates that the benefits of recurrent error correction cannot be matched by simply increasing model parameters.

	Method	Params	PSNR ↑	SSIM ↑	LPIPS ↓
Single-step	AnySplat [21]	886M	20.79	0.643	0.241
	WorldMirror [31]	1263M	23.54	0.789	0.193
	ReSplat-Small (init)	62M	26.77	0.865	0.142
	ReSplat-Base (init)	209M	27.37	0.877	0.130
	ReSplat-Large (init)	559M	27.86	0.886	0.121
Recurrent	ReSplat-Small (recurrent 1)	77M	28.17	0.890	0.118
	ReSplat-Small (recurrent 2)	77M	28.73	0.898	0.110
	ReSplat-Small (recurrent 3)	77M	28.96	0.901	0.107
	ReSplat-Small (recurrent 4)	77M	**29.07**	**0.902**	**0.105**

model benefits more from the additional input views, while the initial model saturates. This indicates that our rendering error-informed recurrent model exploits the additional information more effectively. Third, existing single-step feed-forward models usually exhibit significant performance degradation when the testing image resolution deviates from training. However, our recurrent model significantly improves the robustness to different testing resolutions (Fig. 5c). For example, our recurrent model improves by 5dB PSNR when generalizing from 512×960 to 320×640. These experiments demonstrate that our recurrent model effectively adapts to out-of-distribution scenarios using the rendering error as a feedback signal, thus substantially enhancing robustness.

Single-Step *vs*. Recurrent Models. In Tab. 2, we compare our recurrent model with significantly larger single-step baselines. Our recurrent ReSplat-Small model comprises only 77M parameters, just 15M beyond its single-step initialization counterpart, yet it consistently surpasses all single-step baselines regardless of scale. Even with a single refinement iteration, ReSplat-Small (recurrent 1) achieves 28.17 dB PSNR, outperforming ReSplat-Large (init) which has 559M parameters and is 7× larger. After four iterations, the gap widens to 1.21 dB

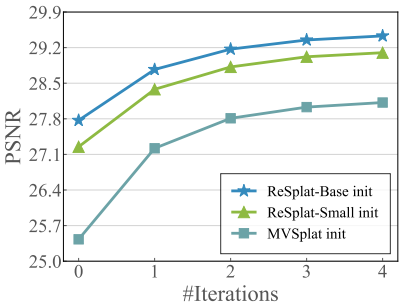


Fig. 6: ReSplat consistently improves across different initializations.

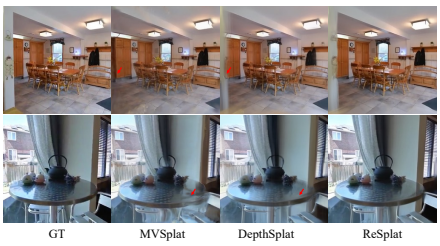


Fig. 7: Visual comparisons on RealEstate10K. ReSplat produces sharper structures than MVSplat and DepthSplat.

Table 3: Evaluation of 16 input views (540×960) **on DL3DV.** Our ReSplat reconstructs 4× fewer Gaussians than Long-LRM but still outperforms it.

Method	#Iterations	PSNR ↑	SSIM ↑	LPIPS ↓	Recon. Time	#Gaussians
3DGS [24]	30000	21.20	0.708	0.264	13min	-
Mip-Splatting [60]	30000	20.88	0.712	0.274	13min	-
Scaffold-GS [34]	30000	22.13	0.738	**0.250**	16min	-
Long-LRM [10]	0	22.66	0.740	0.292	**0.4sec**	2073K
ReSplat	0	22.69	0.742	0.307	0.7sec	**518K**
	1	23.23	0.758	0.291	1.2sec	**518K**
	2	**23.51**	**0.766**	0.284	1.7sec	**518K**

PSNR over ReSplat-Large (init) and 5.53 dB over WorldMirror [31], which is 16× larger with 1263M parameters. This demonstrates that recurrent refinement is fundamentally more parameter efficient than scaling a single-step model: the gains from iterative error correction cannot be matched by simply increasing model capacity. Rather than committing to a single feed-forward prediction, our model progressively corrects its estimates via the rendering-error feedback loop, allowing a compact network to surpass much larger single-step counterparts.

Different Initializations. As shown in Fig. 6, our recurrent model consistently improves rendering quality across various initializations: MVSplat [6], ReSplat-Small, and ReSplat-Base. Performance improves monotonically over successive iterations regardless of the starting point, with stronger initializations yielding higher final quality. Furthermore, because MVSplat predicts per-pixel Gaussians, it produces 16× more Gaussians than ReSplat, making subsequent refinement 13× slower. In contrast, our compact initialization simultaneously provides a superior starting point and enables highly efficient recurrent updates.

16 Views at 540×960 **Resolution.** We follow Long-LRM [10] for this evaluation setup such that a direct comparison is possible. The results of 3DGS [24], Mip-Splatting [60], and Scaffold-GS [34] are borrowed from Long-LRM paper. This experiment aims to reconstruct the full DL3DV scene from 16 input views, which is particularly challenging due to the expansive spatial coverage of the DL3DV dataset. However, our ReSplat still outperforms previous optimization and feed-forward methods, as shown in Tab. 3. Notably, Long-LRM uses Gaussian pruning

Table 4: Evaluation of two input views on RealEstate10K.

Method	w/ 3DGS	PSNR ↑	SSIM ↑	LPIPS ↓
pixelSplat [4]	✓	25.89	0.858	0.142
MVSplat [6]	✓	26.39	0.869	0.128
DepthSplat [55]	✓	27.47	0.889	0.114
GS-LRM [61]	✓	28.10	0.892	0.114
Long-LRM [10]	✓	28.54	0.895	0.109
LVSM (enc-dec) [22]	✗	28.58	0.893	0.114
LVSM (dec-only) [22]	✗	29.67	0.906	**0.098**
ReSplat	✓	**29.75**	**0.912**	0.100

Table 5: RealEstate10K to ACID cross-dataset generalization.

Method	PSNR ↑	SSIM ↑	LPIPS ↓
pixelSplat [4]	27.64	0.830	0.160
MVSplat [6]	28.15	0.841	0.147
DepthSplat [55]	28.37	0.847	0.141
GS-LRM [61]	28.84	0.849	0.146
ReSplat	**29.87**	**0.864**	**0.135**

based on opacity during training and evaluation, resulting in a $\sim 4\times$ reduction in the number of Gaussians. In contrast, we compress the Gaussians by $16\times$, thus our final reconstruction has $4\times$ fewer Gaussians than Long-LRM while still outperforming it. Our reconstruction time is slower than Long-LRM, mainly due to the kNN operation. Further implementation-level optimizations could potentially improve our reconstruction speed.

2 Views at 256×256 Resolution. Since redundancy is less prevalent in two-view, low-resolution (256×256) scenarios, we employ $4\times$ spatial subsampling in the 3D space and decode 4 Gaussians from each subsampled 3D point in our initial reconstruction model. Consequently, the total number of Gaussians remains consistent with previous per-pixel methods. The recurrent process remains the same as the many-view setups. Table 4 shows that our ReSplat outperforms previous feed-forward 3DGS models (*e.g.*, DepthSplat [55], GS-LRM [61] and Long-LRM [10]) by significant margins. Compared to the 3DGS-free method LVSM [22], we outperform its encoder-decoder architecture by 1.1dB PSNR, and our results are competitive with its best-performing decoder-only model variant. However, our method offers the benefits of an explicit 3D Gaussian representation, enabling a $20\times$ increase in rendering speed. We present visual comparisons in Fig. 7, where our ReSplat produces sharper structures than MVSplat and DepthSplat. In Tab. 5, we show the zero-shot generalization results on the unseen ACID [29] dataset, where our ReSplat again outperforms previous methods by clear margins.

Comparison with SplatFormer. We note that our ReSplat has several crucial differences with SplatFormer [9]. First, we identify the rendering error as an informative feedback signal for improving Gaussian reconstruction, which is missing in SplatFormer. Second, ReSplat is a recurrent model which supports multi-step refinement with a weight-sharing architecture, while SplatFormer is a non-recurrent network designed for single-step refinement. Third, ReSplat is a pure feed-forward model with feed-forward initialization and feed-forward refinement. In contrast, SplatFormer relies on lengthy optimization-based 3DGS to get initial Gaussians. Fourth, SplatFormer is designed for object-centric datasets, while ReSplat can handle diverse scene-level datasets where the complexity is much higher than objects. Despite these differences, we tried to conduct a comparison with SplatFormer on our scene-level datasets. We found it particularly

Table 6: Ablations.

(a) Ablation of the rendering error.

Method	PSNR ↑	SSIM ↑	LPIPS ↓
Initialization	26.77	0.865	0.142
w/o rendering error	27.19	0.873	0.137
RGB error only	27.90	0.882	0.130
Feature error only	28.77	0.897	0.110
Concat (RGB & feature errors)	28.93	0.900	0.106
Add (RGB & feature errors)	**29.07**	**0.902**	**0.105**

(b) Ablation of the coordinate system.

Method	PSNR ↑	SSIM ↑	LPIPS ↓
Initialization	26.77	0.865	0.142
COLMAP	28.14	0.886	0.116
First view	28.66	0.896	0.109
Last view	28.59	0.895	0.110
Middle view	**29.07**	**0.902**	**0.105**

(c) Ablation of the initial reconstruction model.

Method	PSNR ↑	SSIM ↑	LPIPS ↓	#Gaussians
DepthSplat [55]	25.79	0.861	**0.134**	918K
Initialization	**26.77**	**0.865**	0.142	**57K**
w/o kNN attn	25.30	0.833	0.178	**57K**
w/o global attn	26.33	0.856	0.150	**57K**
w/o kNN, w/o global	24.50	0.814	0.200	**57K**

(d) Ablation of the recurrent model.

Method	PSNR ↑	SSIM ↑	LPIPS ↓
Initialization	26.77	0.865	0.142
Full	**29.07**	**0.902**	**0.105**
w/o state	27.79	0.878	0.125
w/o kNN attn	28.58	0.894	0.111
w/o global attn	28.96	0.900	0.107

challenging to make it work properly for scene-level datasets, since it relies on Point Transformer V3 [53] where a proper grid size is required to serialize the point cloud. This can be done for object-centric datasets where normalizing the objects to $[-1, 1]$ is possible. However, for unbounded scene-level datasets, this would be very challenging. We tried different normalizations and did grid search for the grid size, and the best results we obtained with SplatFormer are clearly worse than ReSplat across all metrics (ReSplat *vs*. SplatFormer, PSNR: 29.07 *vs*. 27.03, SSIM: 0.902 *vs*. 0.868, LPIPS: 0.105 *vs*. 0.140).

In the appendix, we provide additional comparisons with optimization-based 3DGS [24] across 8, 16, and 32 views, and depth-regularized sparse-view 3DGS optimization methods [11, 25].

4.2 Analysis and Ablation

We conduct several experiments to analyze the behavior of our architecture and validate our design choices. To save compute, all experiments in this section are performed using 8 input views at 256×448 resolution on the DL3DV dataset.

Rendering Error. We evaluate the impact of the rendering error in Tab. 6a. Removing the rendering error in our recurrent model results in a significant performance drop (-1.9dB PSNR). The feature-space errors are more effective than the pixel-space errors, and the best performance is obtained by combining both (addition is slightly better than concatenation).

Coordinate System. In Tab. 6b, we observe that aligning to the middle input view's camera pose performs significantly better (+0.9dB PSNR) than using the default global coordinate system provided by COLMAP. We attribute this to the spatial distribution of the views, where anchoring to the central view acts as a pivot, balancing the spatial distribution of the 3D Gaussians and facilitating the learning of 3D spatial relationships.

Ablation of the Initial Model. As shown in Tab. 6c, kNN attention is crucial for maintaining performance when compressing the Gaussian count by 16×.

Global attention also yields moderate gains, indicating that both local and global 3D contexts are essential for learning compact 3D representations. Together, these components enable our initial model to outperform DepthSplat, despite using 16× fewer Gaussians. The visual results are in the appendix.

Ablation of the Recurrent Model. The state input (Eq. (3)) is critical to our recurrent network (Tab. 6d). Unlike the raw, low-level Gaussian attributes, it encodes rich latent features derived from our initialization model. Both kNN attention and global attention contribute to the performance. Corresponding visual ablations are provided in the appendix.

In the appendix, we provide further evaluations of different feature types for computing the rendering error, recurrent *vs*. non-recurrent architectures, as well as different compression factors (4×, 16×, and 64×) in the initialization model, model profiling, and qualitative results across varying iteration counts.

5 Conclusion

We presented ReSplat, a feed-forward recurrent Gaussian splatting model that enables efficient and high-quality view synthesis. By leveraging the rendering error as a feedback signal and operating in a compact subsampled 3D space, our method significantly reduces the number of Gaussians while improving performance and generalization across datasets, view counts, and resolutions.

Limitations. Our current model maintains a fixed Gaussian count during refinement. Integrating adaptive pruning and densification strategies [24] could potentially further improve performance. In addition, ReSplat currently saturates after four iterations. We hypothesize that the recurrent network rapidly extracts the most informative rendering error signals during the earliest few steps, and performance gradually plateaus due to the diminishing returns of local refinement. Exploring more informative feedback mechanisms to effectively scale test-time compute remains a promising future direction. Furthermore, ReSplat currently operates on static view sets; it would be interesting to extend it to streaming scenarios using a sliding-window approach. In this setup, Gaussians could be initialized independently per window, and the recurrent module could continuously refine the scene, either locally or globally, as new frames arrive. We view these directions as promising avenues for future research.

Acknowledgments. We thank Naama Pearl, Xudong Jiang, Stefano Esposito, and Ata Celen for the insightful comments, and Yung-Hsu Yang and Kashyap Chitta for the fruitful discussions. We thank the anonymous reviewers for their constructive feedback. Andreas Geiger was supported by the ERC Starting Grant LEGO-3D (850533) and the DFG EXC number 2064/1 - project number 390727645. This work was supported as part of the Swiss AI Initiative by a grant from the Swiss National Supercomputing Centre (CSCS) under project ID a144 on Alps.

References

1. Andrychowicz, M., Denil, M., Gomez, S., Hoffman, M., Pfau, D., Schaul, T., Shillingford, B., de Freitas, N.: Learning to learn by gradient descent by gradient descent. In: NeurIPS (2016)
2. Ba, J.L., Kiros, J.R., Hinton, G.E.: Layer normalization. arXiv preprint arXiv:1607.06450 (2016)
3. Bai, S., Kolter, J.Z., Koltun, V.: Deep equilibrium models. NeurIPS (2019)
4. Charatan, D., Li, S., Tagliasacchi, A., Sitzmann, V.: pixelsplat: 3d gaussian splats from image pairs for scalable generalizable 3d reconstruction. In: CVPR (2024)
5. Chen, Y., Wu, Q., Lin, W., Harandi, M., Cai, J.: Hac: Hash-grid assisted context for 3d gaussian splatting compression. In: ECCV (2024)
6. Chen, Y., Xu, H., Zheng, C., Zhuang, B., Pollefeys, M., Geiger, A., Cham, T.J., Cai, J.: Mvsplat: Efficient 3d gaussian splatting from sparse multi-view images. In: ECCV (2024)
7. Chen, Y., Zheng, C., Xu, H., Zhuang, B., Vedaldi, A., Cham, T.J., Cai, J.: Mvsplat360: Feed-forward 360 scene synthesis from sparse views. In: NeurIPS (2024)
8. Chen, Y., Wang, J., Yang, Z., Manivasagam, S., Urtasun, R.: G3r: Gradient guided generalizable reconstruction. In: ECCV (2024)
9. Chen, Y., Mihajlovic, M., Chen, X., Wang, Y., Prokudin, S., Tang, S.: Splatformer: Point transformer for robust 3d gaussian splatting. In: ICLR (2025)
10. Chen, Z., Tan, H., Zhang, K., Bi, S., Luan, F., Hong, Y., Fuxin, L., Xu, Z.: Long-lrm: Long-sequence large reconstruction model for wide-coverage gaussian splats. In: ICCV (2025)
11. Chung, J., Oh, J., Lee, K.M.: Depth-regularized optimization for 3d gaussian splatting in few-shot images. In: CVPR (2024)
12. Contributors, P.: Pointcept: A codebase for point cloud perception research. `https://github.com/Pointcept/Pointcept` (2023)
13. Deng, J., Dong, W., Socher, R., Li, L.J., Li, K., Fei-Fei, L.: Imagenet: A large-scale hierarchical image database. In: CVPR (2009)
14. Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., et al.: An image is worth 16x16 words: Transformers for image recognition at scale. arXiv (2020)
15. Flynn, J., Broxton, M., Debevec, P., DuVall, M., Fyffe, G., Overbeck, R., Snavely, N., Tucker, R.: Deepview: View synthesis with learned gradient descent. In: CVPR (2019)
16. Geiping, J., McLeish, S., Jain, N., Kirchenbauer, J., Singh, S., Bartoldson, B.R., Kailkhura, B., Bhatele, A., Goldstein, T.: Scaling up test-time compute with latent reasoning: A recurrent depth approach. arXiv preprint arXiv:2502.05171 (2025)
17. Godard, C., Mac Aodha, O., Brostow, G.J.: Unsupervised monocular depth estimation with left-right consistency. In: CVPR (2017)
18. Godard, C., Mac Aodha, O., Firman, M., Brostow, G.J.: Digging into self-supervised monocular depth estimation. In: ICCV (2019)
19. Harrison, J., Metz, L., Sohl-Dickstein, J.: A closer look at learned optimization: Stability, robustness, and inductive biases. In: NeurIPS (2022)
20. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: CVPR (2016)
21. Jiang, L., Mao, Y., Xu, L., Lu, T., Ren, K., Jin, Y., Xu, X., Yu, M., Pang, J., Zhao, F., et al.: Anysplat: Feed-forward 3d gaussian splatting from unconstrained views. TOG **44**(6), 1–16 (2025)

22. Jin, H., Jiang, H., Tan, H., Zhang, K., Bi, S., Zhang, T., Luan, F., Snavely, N., Xu, Z.: Lvsm: A large view synthesis model with minimal 3d inductive bias. In: ICLR (2025)
23. Johnson, J., Alahi, A., Fei-Fei, L.: Perceptual losses for real-time style transfer and super-resolution. In: ECCV (2016)
24. Kerbl, B., Kopanas, G., Leimkühler, T., Drettakis, G.: 3d gaussian splatting for real-time radiance field rendering. ACM TOG (2023)
25. Li, J., Zhang, J., Bai, X., Zheng, J., Ning, X., Zhou, J., Gu, L.: Dngaussian: Optimizing sparse-view 3d gaussian radiance fields with global-local depth normalization. In: CVPR (2024)
26. Li, Z., Tucker, R., Cole, F., Wang, Q., Jin, L., Ye, V., Kanazawa, A., Holynski, A., Snavely, N.: Megasam: Accurate, fast, and robust structure and motion from casual dynamic videos. In: CVPR (2025)
27. Ling, L., Sheng, Y., Tu, Z., Zhao, W., Xin, C., Wan, K., Yu, L., Guo, Q., Yu, Z., Lu, Y., et al.: Dl3dv-10k: A large-scale scene dataset for deep learning-based 3d vision. arXiv (2023)
28. Lipson, L., Teed, Z., Deng, J.: Raft-stereo: Multilevel recurrent field transforms for stereo matching. In: 3DV (2021)
29. Liu, A., Tucker, R., Jampani, V., Makadia, A., Snavely, N., Kanazawa, A.: Infinite nature: Perpetual view generation of natural scenes from a single image. In: ICCV (2021)
30. Liu, T., Wang, G., Hu, S., Shen, L., Ye, X., Zang, Y., Cao, Z., Li, W., Liu, Z.: Mvsgaussian: Fast generalizable gaussian splatting reconstruction from multi-view stereo. In: ECCV (2024)
31. Liu, Y., Min, Z., Wang, Z., Wu, J., Wang, T., Yuan, Y., Luo, Y., Guo, C.: Worldmirror: Universal 3d world reconstruction with any-prior prompting. arXiv preprint arXiv:2510.10726 (2025)
32. Liu, Y.C., Höllein, L., Nießner, M., Dai, A.: Quicksplat: Fast 3d surface reconstruction via learned gaussian initialization. In: ICCV (2025)
33. Loshchilov, I., Hutter, F.: Decoupled weight decay regularization. arXiv (2017)
34. Lu, T., Yu, M., Xu, L., Xiangli, Y., Wang, L., Lin, D., Dai, B.: Scaffold-gs: Structured 3d gaussians for view-adaptive rendering. In: CVPR (2024)
35. Lucas, B.D., Kanade, T.: An iterative image registration technique with an application to stereo vision. In: IJCAI (1981)
36. Ma, W.C., Wang, S., Gu, J., Manivasagam, S., Torralba, A., Urtasun, R.: Deep feedback inverse problem solver. In: ECCV (2020)
37. Metz, L., Harrison, J., Freeman, C.D., Merchant, A., Beyer, L., Bradbury, J., Agrawal, N., Poole, B., Mordatch, I., Roberts, A., Sohl-Dickstein, J.: Velo: Training versatile learned optimizers by scaling up. In: NeurIPS (2022)
38. Oquab, M., Darcet, T., Moutakanni, T., Vo, H., Szafraniec, M., Khalidov, V., Fernandez, P., Haziza, D., Massa, F., El-Nouby, A., et al.: Dinov2: Learning robust visual features without supervision. arXiv (2023)
39. Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al.: Pytorch: An imperative style, high-performance deep learning library. In: NeurIPS (2019)
40. Schonberger, J.L., Frahm, J.M.: Structure-from-motion revisited. In: CVPR (2016)
41. Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556 (2014)
42. Sun, D., Roth, S., Black, M.J.: Secrets of optical flow estimation and their principles. In: CVPR (2010)

43. Szymanowicz, S., Insafutdinov, E., Zheng, C., Campbell, D., Henriques, J.F., Rupprecht, C., Vedaldi, A.: Flash3d: Feed-forward generalisable 3d scene reconstruction from a single image. In: 3DV (2025)
44. Szymanowicz, S., Rupprecht, C., Vedaldi, A.: Splatter image: Ultra-fast single-view 3d reconstruction. In: CVPR (2024)
45. Teed, Z., Deng, J.: Raft: Recurrent all-pairs field transforms for optical flow. In: ECCV (2020)
46. Teed, Z., Deng, J.: Droid-slam: Deep visual slam for monocular, stereo, and rgb-d cameras. NeurIPS (2021)
47. Teed, Z., Deng, J.: Raft-3d: Scene flow using rigid-motion embeddings. In: CVPR (2021)
48. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł., Polosukhin, I.: Attention is all you need. NeurIPS (2017)
49. Wang, F., Galliani, S., Vogel, C., Pollefeys, M.: Itermvs: Iterative probability estimation for efficient multi-view stereo. In: CVPR (2022)
50. Wang, Y., Huang, T., Chen, H., Lee, G.H.: Freesplat: Generalizable 3d gaussian splatting towards free view synthesis of indoor scenes. NeurIPS (2024)
51. Wen, B., Trepte, M., Aribido, J., Kautz, J., Gallo, O., Birchfield, S.: Foundationstereo: Zero-shot stereo matching. In: CVPR (2025)
52. Wen, J., Schwing, A.G., Wang, S.: Life-gom: Generalizable human rendering with learned iterative feedback over multi-resolution gaussians-on-mesh. In: ICLR (2025)
53. Wu, X., Jiang, L., Wang, P.S., Liu, Z., Liu, X., Qiao, Y., Ouyang, W., He, T., Zhao, H.: Point transformer v3: Simpler, faster, stronger. In: CVPR (2024)
54. Xu, H., Chen, A., Chen, Y., Sakaridis, C., Zhang, Y., Pollefeys, M., Geiger, A., Yu, F.: Murf: Multi-baseline radiance fields. In: CVPR (2024)
55. Xu, H., Peng, S., Wang, F., Blum, H., Barath, D., Geiger, A., Pollefeys, M.: Depthsplat: Connecting gaussian splatting and depth. In: CVPR (2025)
56. Yang, L., Kang, B., Huang, Z., Zhao, Z., Xu, X., Feng, J., Zhao, H.: Depth anything v2. NeurIPS (2024)
57. Ye, B., Chen, B., Xu, H., Barath, D., Pollefeys, M.: Yonosplat: You only need one model for feedforward 3d gaussian splatting. arXiv preprint arXiv:2511.07321 (2025)
58. Ye, B., Liu, S., Xu, H., Li, X., Pollefeys, M., Yang, M.H., Peng, S.: No pose, no problem: Surprisingly simple 3d gaussian splats from sparse unposed images. In: ICLR (2025)
59. Ye, V., Li, R., Kerr, J., Turkulainen, M., Yi, B., Pan, Z., Seiskari, O., Ye, J., Hu, J., Tancik, M., et al.: gsplat: An open-source library for gaussian splatting. JMLR pp. 1–17 (2025)
60. Yu, Z., Chen, A., Huang, B., Sattler, T., Geiger, A.: Mip-splatting: Alias-free 3d gaussian splatting. In: CVPR (2024)
61. Zhang, K., Bi, S., Tan, H., Xiangli, Y., Zhao, N., Sunkavalli, K., Xu, Z.: Gs-lrm: Large reconstruction model for 3d gaussian splatting. In: ECCV (2024)
62. Zhang, S., Fei, X., Liu, F., Song, H., Duan, Y.: Gaussian graph network: Learning efficient and generalizable gaussian representations from multi-view images. NeurIPS (2024)
63. Zhao, H., Jiang, L., Jia, J., Torr, P.H., Koltun, V.: Point transformer. In: ICCV (2021)
64. Zhou, T., Tucker, R., Flynn, J., Fyffe, G., Snavely, N.: Stereo magnification: learning view synthesis using multiplane images. ACM TOG (2018)

Appendix

This appendix provides further details and comprehensive evaluations to complement the main manuscript. In Sec. A, we present extended evaluations, including comparisons with optimization-based 3DGS [24] across varying input views and depth-regularized 3DGS [11, 25]. We also provide detailed ablations on our recurrent architecture, feature extraction choices, compression factors, and our efficient local kNN implementation, alongside comprehensive model profiling. In Sec. B, we detail our specific training schedules and hardware configurations. In Sec. C, we supply extensive additional qualitative results, featuring visual comparisons with state-of-the-art baselines, and progressive refinement visualizations across iterations.

A Additional Evaluations

Comparison with 3DGS Across 8, 16, and 32 Views. In Tab. S1, we compare with optimization-based 3DGS [24] across 8, 16 and 32 input views. The quality gap becomes smaller when given 32 views for optimization-based 3DGS. However, our ReSplat is more than 200× faster in terms of the reconstruction speed. In this setup, we expand the sampling region as the number of input views increases to enlarge scene coverage. Consequently, the test views differ across configurations to account for this larger spatial extent.

Table S1: Comparison with optimization-based 3DGS across 8, 16, and 32 input views. The quality gap becomes smaller when given 32 views for optimization-based 3DGS. However, our ReSplat is more than 200× faster in terms of the reconstruction speed. The image resolution is 256×448.

#Views	Method	Category	#Iterations	PSNR ↑	SSIM ↑	LPIPS ↓	#Gaussians	Recon. Time (s)
8	3DGS	Optimization	4000	26.44	0.841	0.134	250K	49
	ReSplat	Feed-Forward	**4**	**29.20**	**0.904**	**0.104**	**57K**	**0.21**
16	3DGS	Optimization	4000	27.38	0.864	0.119	395K	70
	ReSplat	Feed-Forward	**4**	**29.01**	**0.900**	**0.105**	**114K**	**0.34**
32	3DGS	Optimization	4000	27.86	0.879	**0.113**	522K	160
	ReSplat	Feed-Forward	**4**	**28.30**	**0.891**	0.114	**229K**	**0.75**

Recurrent *vs*. Non-recurrent Architecture. We demonstrate the effectiveness of our recurrent model by comparing with non-recurrent variants in Tab. S2. In particular, we first compare with non-weight-sharing multi-step stacked networks where different iterations have different model weights and all the other components are the same. We can observe that non-weight-sharing not only leads

to 4× more parameters, but also results in worse view synthesis results. We further compare with non-weight-sharing single-step deeper networks by increasing the number of attention blocks for a single-step refinement, where the results are clearly worse than our multi-step recurrent network. The weight-sharing design in our recurrent network implicitly regularizes training, which is not only more parameter-efficient but also leads to better results.

Table S2: Comparison of recurrent and non-recurrent architectures. Compared to non-weight-sharing multi-step stacked networks and non-weight-sharing single-step deeper networks, our weight-sharing multi-step recurrent network is not only more parameter-efficient, but also leads to better results. We also note that our single recurrent network can support different numbers of iterations thanks to weight-sharing.

Configuration	#Params	PSNR ↑	SSIM ↑	LPIPS ↓
weight-sharing				
Recurrent (iter 1, block 4)	13.8M	28.17	0.890	0.118
Recurrent (iter 2, block 4)	13.8M	28.73	0.898	0.110
Recurrent (iter 3, block 4)	13.8M	28.96	0.901	0.107
Recurrent (iter 4, block 4)	13.8M	**29.07**	**0.902**	**0.105**
non-weight-sharing, multi-step, stacked				
Non-recurrent (stack 1)	13.8M	28.17	0.890	0.118
Non-recurrent (stack 2)	27.6M	28.74	0.898	0.109
Non-recurrent (stack 3)	41.4M	28.72	0.898	0.110
Non-recurrent (stack 4)	55.2M	28.71	0.897	0.110
non-weight-sharing, single-step, deeper				
Non-recurrent (stack 1, block 4)	13.8M	28.17	0.890	0.118
Non-recurrent (stack 1, block 8)	27.6M	28.30	0.891	0.116
Non-recurrent (stack 1, block 12)	41.4M	28.36	0.893	0.115
Non-recurrent (stack 1, block 16)	55.2M	28.40	0.893	0.115

Comparison with Sparse-View Optimization Methods. We additionally compare with optimization-based 3DGS methods that are specifically designed for sparse input views. These sparse-view optimization methods [11, 25] usually rely on additional depth losses to regularize the optimization process. To compare with them, we perform 3DGS optimization with an additional depth loss between the rendered depth map and the estimated monocular depth map from Depth Anything V2 (Large) [56]. The results are reported in Tab. S3. With the additional depth loss, the 3DGS optimization results are improved by 1dB PSNR. However, the gap with our ReSplat is still significant (3dB PSNR). The additionally introduced depth loss also makes the optimization slower due to the additional time for depth rendering and monocular depth estimation. In contrast, our model doesn't rely on any additional supervision from an external monocular depth model and it's 94× faster thanks to our feed-forward nature.

Features for Computing the Rendering Error. In Tab. S4, we compare ResNet [20] features with those from DINOv2 [38]. We observed no improvement

Table S3: Comparison with depth-regularized 3DGS optimization method. Despite with an additional depth loss, the optimization-based method is still 3dB PSNR worse than our ReSplat and runs 94× slower.

Method	PSNR ↑	SSIM ↑	LPIPS ↓	Recon. Time (s)
3DGS (w/o depth loss)	23.46	0.770	0.224	70.0
3DGS (w/ depth loss)	24.54	0.796	0.204	75.4
ReSplat (w/o depth loss)	**27.70**	**0.868**	**0.160**	**0.8**

Table S4: ResNet *vs*. DINOv2 features for computing the rendering error. We observed no improvement when using the larger, more recent feature extractor. We attribute this to the patch-based architecture of DINOv2, which may result in coarser spatial information. In contrast, convolutional networks maintain local structural fidelity, which is critical for high-quality view synthesis.

Features	#Parameters	PSNR ↑	SSIM ↑	LPIPS ↓
ResNet	0.7M	29.07	0.902	0.105
DINOv2	86.6M	29.00	0.901	0.107

when using the larger, more recent feature extractor. We attribute this to the patch-based architecture of DINOv2, which may result in coarser spatial information. In contrast, convolutional networks maintain local structural fidelity, which is critical for high-quality pixel-accurate view synthesis.

Compression Factor. By default, we compress the number of Gaussians by 16× using depth maps at 1/4 resolution. In Tab. S5 and Fig. S1, we compare with 64× (with 1/8 depth maps) and 4× (with 1/2 depth maps) compression factors. We observe that less compression leads to higher quality. However, 4× compression is 2× slower than 16× at 256 × 448 resolution. It would be more expensive when handling higher resolution images (*e.g.*, 512 × 960). Thus, we choose 16× compression as a good speed-accuracy trade-off.

Table S5: Different compression factors. 16× compression represents a good speed-accuracy trade-off and thus is used in our model.

Compression	PSNR ↑	SSIM ↑	LPIPS ↓	Time (s)
64×	24.77	0.797	0.226	**0.096**
16×	26.77	0.865	0.142	0.104
4×	**28.36**	**0.900**	**0.103**	0.206

Efficient Local kNN Implementation. The standard kNN attention implementation [12] used in this paper relies on a global brute-force search over all N points, imposing an $O(N^2)$ complexity bottleneck for high-resolution point cloud. To address this, we introduce a drop-in $O(N)$ local kNN module that exploits known multi-view grid structure. Because nearby pixels on a surface project to nearby 3D points, the true k-nearest neighbors are almost exclusively found at spatially adjacent pixels in the source view, or at corresponding projected pixels in other views. We therefore constrain our search spatially while remaining

Fig. S1: Different compression factors. 16× compression represents a good speed-accuracy trade-off.

comprehensive across the multi-view dimension. For each point, we generate same-view candidates using a $(2r_s + 1)^2$ spatial window. Simultaneously, we gather cross-view candidates by projecting the 3D query point into the 2D image planes of all other available cameras. Specifically, we use each target camera's pose to locate the point relative to that camera, and its camera intrinsic matrix to pinpoint the exact 2D pixel where the point would be visible. We then extract the 3D points located within a $(2r_c + 1)^2$ spatial window centered around this projected pixel coordinate. For our implementation, we set both spatial radii r_s and r_c to 3. Second, we select the top-k neighbors from these candidates using 3D Euclidean distance. By vectorizing the cross-view reprojections (with camera intrinsic and extrinsic parameters), our method reduces complexity to $O(N)$, significantly reducing distance computations with negligible quality loss. As demonstrated in Tab. S6, our local kNN implementation improves inference speed over the global baseline when evaluated on 8 views at 512×960 resolution, empirically validating its computational efficiency.

Table S6: Global *vs*. Local kNN. Our local kNN implementation improves inference speed over the default global baseline. The results are evaluated on 8 views at 512×960 resolution (with 246K Gaussians).

kNN	PSNR ↑	SSIM ↑	LPIPS ↓	Time (s)
Global	**27.70**	**0.868**	**0.160**	0.816
Local	27.65	0.867	0.163	**0.591**

Model Profiling. In Tab. S7, we report the total runtime and individual component latency. In the initial reconstruction model, the depth prediction module constitutes the majority of the runtime. For the recurrent model, the kNN attention mechanism consumes the most time. These results highlight potential areas for future optimization.

Table S7: Model Profiling. Inference time (s) measured on 8 input views at varying resolutions. We report both total runtime and individual component latency.

(a) Initial Model Profiling. The depth module constitutes the majority of the runtime.

Resolution	Total	Depth pred.	kNN attn	Global attn	Gaussian head
256×448	0.149	0.111	0.024	0.013	0.001
512×960	0.311	0.197	0.094	0.018	0.002

(b) Recurrent Model Profiling. The kNN attention mechanism consumes the most time.

Resolution	Total	Render error	kNN attn	Global attn	Update head
256×448	0.022	0.003	0.015	0.002	0.002
512×960	0.126	0.016	0.092	0.008	0.010

Per-Attribute Update. Our per-attribute analysis shows that the recurrent module primarily refines geometry and base appearance (Gaussian centers, scales, and base colors (0-th order)), whereas higher-order SH adjustments remain negligible.

B Additional Details

Training Details. We train our model with cosine learning rate schedule. For experiments on DL3DV, we adopt a progressive training strategy with gradually increased image resolutions and number of input views for better efficiency. More specifically, we first train our model with 8 input views at 256×448 resolution, and then we fine-tune the model with 8 input views at 512×960 resolution, and finally we fine-tune the model with 16 input views at 512×960 resolution. For each stage, we train with 16 GH200 GPUs for 80K steps, with 50K steps for the initial reconstruction model and 30K steps for the recurrent model. For experiments on RealEstate10K at 256×256 resolution, we first train the initial model with 16 GH200 GPUs for 200K steps and then train the recurrent model for 100K steps. More details are provided at `https://github.com/cvg/resplat`.

C Additional Visualizations

In Fig. S2, we show the visual results with different numbers of iterations.

In Fig. S3, we show more visual comparisons with 3DGS [24], MVSplat [6] and DepthSplat [55] on the DL3DV dataset.

In Fig. S4, we show the visual ablation results of our initial model.

In Fig. S5, we show the visual ablation results of our recurrent model.

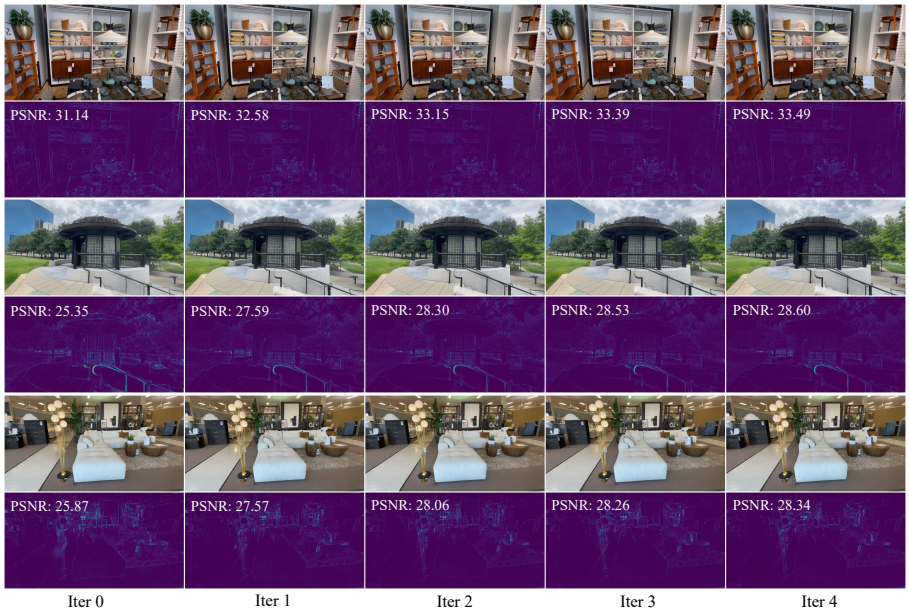


Fig. S2: Results with different numbers of iterations.

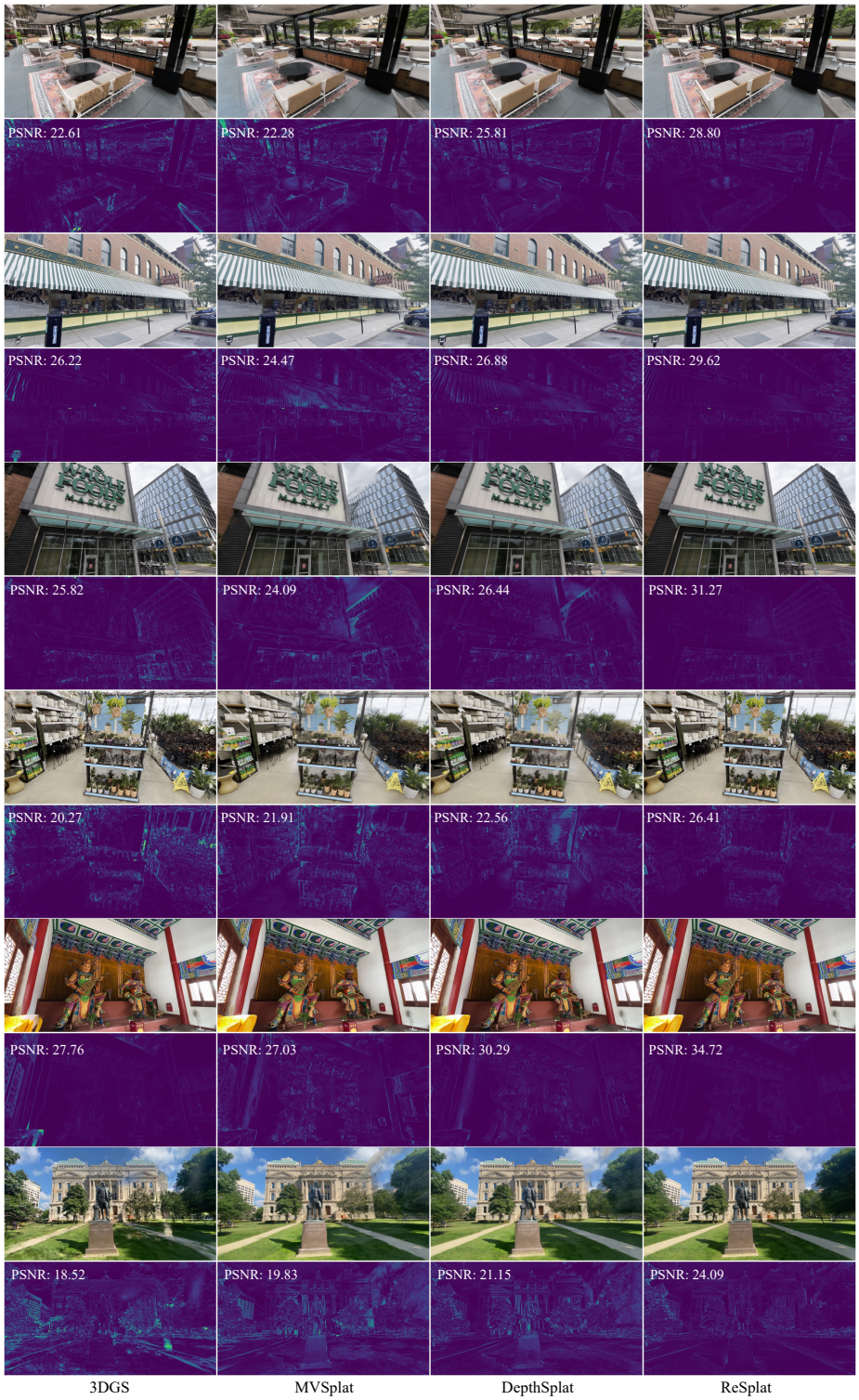


Fig. S3: More comparisons of novel view synthesis on DL3DV.

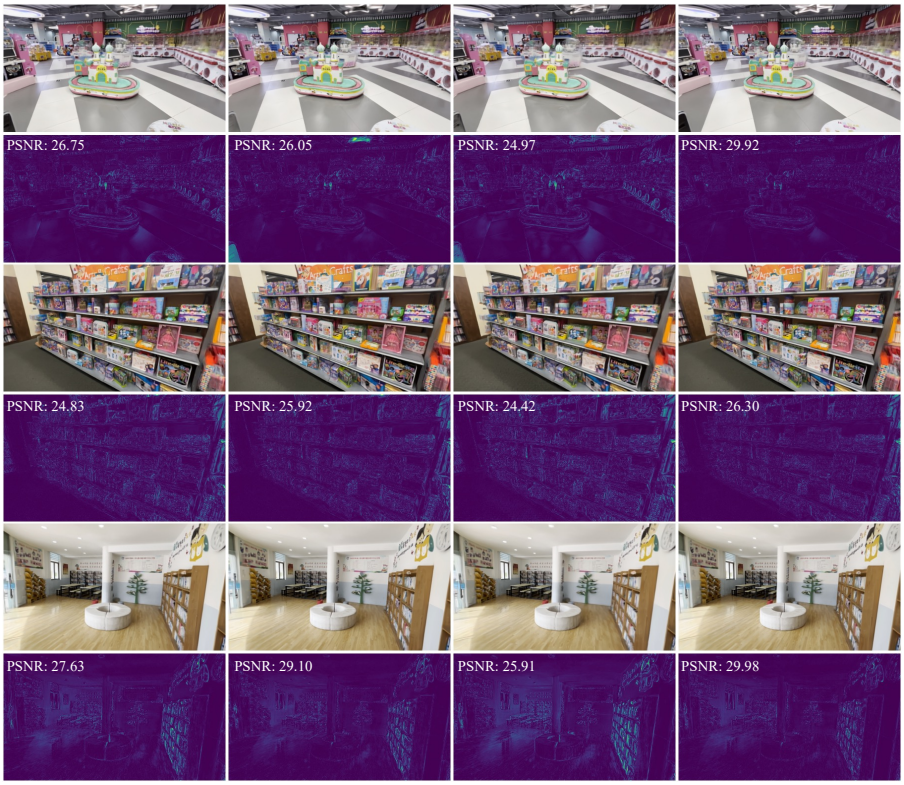


Fig. S4: Ablation of the initial model.

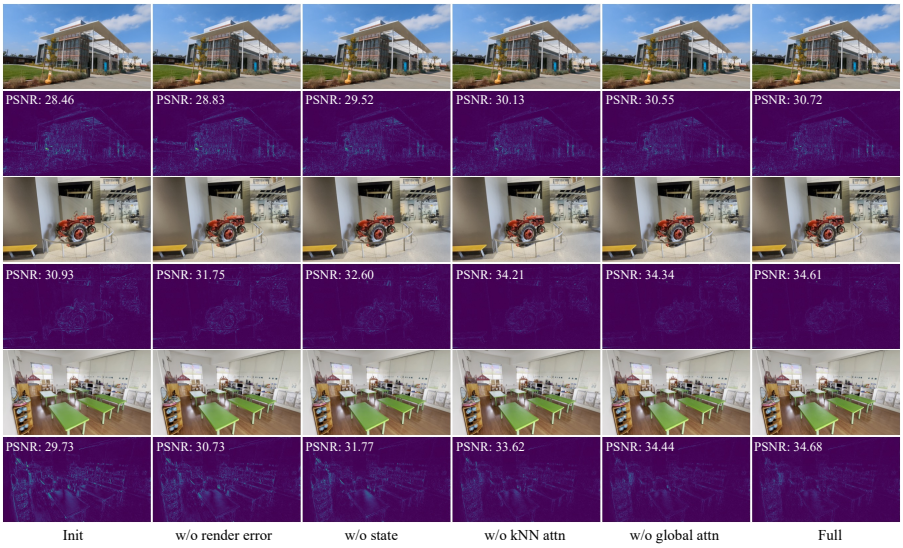


Fig. S5: Ablation of the recurrent model.