CL-Splats: Continual Learning of Gaussian Splatting with Local Optimization

Supplementary Material

In this **Supplementary Material**, we provide additional details and discussions to complement the main paper. In Sec. 7, we elaborate on the implementation details of the method introduced in Sec. 3 of the main paper. Section 8 describes the datasets used in Sec. 4 of the main paper. In Sec. 9, we present further insights into the design choices of our method. As highlighted in Sec. 5 of the main paper, we demonstrate our method's ability to handle multiple time steps with dynamic changes in Sec. 11.2. Finally, Sec. 12 includes the full quantitative results of the experiments discussed in Sec. 4 of the main paper. Please also check our supplementary video for more visualizations.

7. Implementation Details

In our experiments, we utilize the official 3D Gaussian Splatting [19] implementation as our basis, adopting all default hyperparameters as provided in 3DGS [19].

Detecting Changes in 2D. We start by obtaining a vanilla 3DGS representation (optimized on time T_a) and rendering it from the same viewpoints as the new captures (time $T_b; b > a$). In our implementation, we use DINOv2 as the feature extractor. In particular, we use the Dinov2-small-14 configuration, which enables real-time processing on an NVIDIA Quadro RTX 6000, facilitating dynamic change detection while exploring a scene. DINOv2 model expects image height and width to be divisible by the patch size 14. To achieve this, we perform a center crop with box width $w_{14} = w - (w \mod 14)$ and height $h_{14} = h - (h \mod 14)$. After this preprocessing step, we normalize the inputs using the statistics from imagenet [14]. The normalized result is fed into DINOv2, and we extract the last layer patch token activations of shape $(\lfloor w/14 \rfloor, \lfloor h/14 \rfloor)$. We compute the DINOv2 features of the 3DGS T_a rendering and the ground-truth images at T_b and compare them using cosine similarity, giving us a down-scaled changed map. To scale the result to the original resolution, we first resize the change map to (w_{14}, h_{14}) using bilinear interpolation and then pad it with 0s. Now, we translate this soft mask to a binary mask by thresholding the cosine similarity at $\tau_1 = 0.5$. Finally, we dilate the mask by 2% of the image width. We implement this via a convolution with a kernel having only 1s as weights and then thresholding at 0.

Identifying Changed 3D Regions via Majority Voting. Given the 2D binary masks for each render-capture pair, we now lift these masks into 3D, giving us the set of 3DGS that are affected by the change. For each frame, we use the camera-projection-matrix to project [18] the center of each Gaussian into 2D. We count the number of times each Gaussian g_i projects into the 2D mask with c_i and how often it projects outside of the image with o_i. Based on these counts, we perform a majority voting strategy that independently assigns each Gaussian either to the set of changed Gaussians $\mathcal{O}^t$ or to the rest:

$$g_i \in \mathcal{O}^t \quad \textit{iff.} \quad \frac{4}{3} o_i < |\mathbf{I}^t| < 2c_i. \tag{3}$$

Here, $\mathbf{I}^t$ is the set of new images described in Sec. 3 of the main paper. This guarantees that the changed region is well observed while admitting less robust detection in 2D.

Given this set of Gaussians, we split them into clusters (k_i) using HDBSCAN [26] with a minimum cluster size of 1000. For each k_i, we compute the mean m_i and the 98th quantile of the Euclidean distance d_i of all points in k_i to m_i. We then define a sphere s_i with center m_i and radius $d_i \cdot 1.1$ for each k_i.

Sampling New Points. As described in the main paper, to guarantee that there will be Gaussians to optimize, we sample points in the area of change. If there are already points close to the area of change, then we can sample in that local area. To do that, we fit the K-Means clustering algorithm on the xyz coordinates of the values with $K = 10$ and then reinterpret these as a GaussianMixture and sample new points from there. By sampling few points each round, we gradually expand the 3D region where points are sampled without sampling too many points that will be far away. In contrast if there are no Gaussians in the area of change, then we start by sampling everywhere. In any case, we only keep the sampled points that project into enough 2D masks according to the majority voting. In Alg. 2 and Alg. 3 we describe how we sample new points across the scene and around existing points. The function *Random* samples points uniformly distributed in the given interval, the function *K-Means* computes a K-Means clustering with K clusters, the function *InitGMMFromClusters* constructs a Gaussians Mixture model with mean being the cluster center and the covariance being the a diagonal matrix containing the vector from the center to the furthest point on its diagonal. Finally *MajorityVote* filters the points according to the voting algorithm described in the main paper. While it is a viable approach to always sample in the entire scene, we found that sampling in the region can help to get points faster.

Algorithm 2 Full-Scene Point Sampling

Input: $\mathcal{G}^{t-1}$,$(\mathcal{M}_i^t)_{i\leq N}$, n
Output: pointcloud $\mathcal{P}$
function RANDOMSAMPLE($\mathcal{G}^{t-1}$,$(\mathcal{M}_i^t)_{i\leq N}$, n)
    $min = \min(\mathcal{G}^{t-1}.\text{xyz})$
    $max = \max(\mathcal{G}^{t-1}.\text{xyz})$
    $\mathcal{S} = \text{RANDOM}(min, max, n)$
    $\mathcal{S}_{\text{in}} = \text{MAJORITYVOTE}((\mathcal{M}_i^t)_{i\leq N}, \mathcal{S})$
    return $\mathcal{S}_{\text{in}}$
end function

Algorithm 3 Region Point Sampling

Input: $\mathcal{O}$, $(\mathcal{M}_i^t)_{i\leq N}$, n
Output: pointcloud $\mathcal{P}$
function SAMPLEREGION($\mathcal{O}$,$(\mathcal{M}_i^t)_{i\leq N}$, n)
    $\mathcal{K} = \text{K-MEANS}(\mathcal{O}.\text{xyz}, K = 10)$
    $\text{GMM} = \text{INITGMMFROMCLUSTERS}(\mathcal{K})$
    $\mathcal{S} = \text{SAMPLE}(\text{GMM}, n/5)$
    $\mathcal{S}_{\text{in}} = \text{MAJORITYVOTE}((\mathcal{M}_i^t)_{i\leq N}, \mathcal{S})$
    return $\mathcal{S}_{\text{in}}$
end function

3DGS Optimization in Changed Regions. We are able to only perform local updates by only applying gradients to Gaussians in $\mathcal{O}^t$. We achieve this by masking the gradients for each Gaussian not in that set and only applying optimizer update steps to the Gaussians in this set. This also prevents unwanted movement due to first and second moment from Adam [20]. In addition to this, we perform pruning of Gaussians in $\mathcal{O}^t$ that are not inside any sphere s_i – every 15 iterations, we check if a Gaussian $g_i \in \bigcup_j s_j$ and if not we remove it. We found that on average less than 1 Gaussian leaves the spheres each iteration which leads to a smooth optimization. On the other hand, if they are not pruned then a significant amount of Gaussians will not be in the area of the change making the optimization with the local kernel inefficient.

Local Optimization Kernel. We modify the 3DGS CUDA kernel to exploit the spatial locality inherent to our optimization approach. We conducted a preliminary experiment using a real-world scene to justify these modifications. In this experiment, we measured the time spent in the rasterization routines under two conditions: (a) the original complete 3DGS optimization and (b) an artificially constrained version where rendering is restricted to a central crop occupying 30% of the image.

Table 6 presents the results of these measurements, detailing the timings for individual steps during 3DGS optimization. The results demonstrate that localized optimization achieves speedups proportional to the reduced number of tiles rendered.

Given this information, we make three modifications to the diff-gaussian-rasterization kernel:

Routing / Method	Full 3DGS	3DGS 30%
Forward::Preprocess	0.70ms	0.61ms
Forward::Render	0.92ms	0.21ms
Backward::Preprocess	0.32ms	0.21ms
Backward::Render	10.99ms	3.37ms
Total	12.93ms	4.40ms

Table 6. **Comparison of Individual Routine Times.** We artificially restrict the optimization to a center crop making up 30% of the tiles and compare the times of the routines.

1. We compute the dynamic rendering mask by projecting all Gaussians from $\mathcal{O}^t$ onto a boolean matrix of size $\lceil w/16 \rceil \times \lceil h/16 \rceil$, setting each entry to *true* if at least one Gaussian projects onto the corresponding tile. To avoid additional overhead, we reuse the computed projection to create the per-tile depth ordering.
2. During forward and backward rendering, we utilize the generated mask to terminate threads associated with tiles marked as *false* in the boolean matrix. We can compute the entry associated with a thread by checking the threats group index for x and y.
3. Finally, we terminate threads performing the backward pass of the preprocessing step for Gaussians that are not part of $\mathcal{O}^t$.

Baselines.

1. **3DGS** [19]: In our experiments we use the official implementation of 3DGS with unchanged hyperparameters.
2. **GaussianEditor** [10]: We used the local optimization code from GaussianEditor [10] in combination with our masks. We leave the hyperparameters mostly unchanged – the only change we make is that we set the number of iterations from 1000 to 30000 for a fair comparison.
3. **CL-NeRF** [49]: We use the official implementation with the default hyperparameters. Additionally, we make sure that the parameters work robustly on new data as well.
4. **CLNeRF** [6]: We use the official implementation with the default hyperparameters.

Camera Pose Estimation. The first step in creating photo-realistic reconstructions is obtaining camera positions. For the synthetic part of the *CL-Splats* dataset, we export the camera positions from Blender, eliminating any influence camera pose estimation has on the reconstruction quality. For the in-the-wild images, we follow existing methods [6, 49] and use COLMAP to recover the camera positions of all the captured images at all times together. We found that this works well and that COLMAP does not have problems with scene changes encountered in these datasets. We expected this since COLMAP was designed to handle even images

from internet collections, which vary in style and geometry from frame to frame and not only between time steps. At the same time, this allows us to conduct our experiments and focus on the reconstruction part; in practice, reconstructing from scratch after each record is not feasible. COLMAP already includes the necessary functionality to match existing images to a reconstruction. In particular, the following steps can be taken to add new images to the model.

1. Extract features of the new images with *feature_extractor*.
2. Match the new images against the existing ones, preferably using *vocab_tree_matcher*. This matcher only scales linearly in the number of new images.
3. Use *image_registrator* to register the new images to the reconstruction.

The authors of COLMAP suggest using *bundle_adjuster* to improve the accuracy of the reconstruction, but currently, they only support adjusting over all cameras. A valuable addition for the continual setting would be to allow bundle adjusting only over the new images.

Finally, if too many changes have accumulated over time, it is possible to render the existing 3DGS reconstruction from some viewpoints and use that to start a new reconstruction.

We have experimented with this incremental model and compared it to the jointly estimated model on scenes from the real-world dataset. We observed that incrementally building the model only led to an average PSNR drop of 1.2 while enabling speedups of up to 20 times.

8. Details on *CL-Splats* Dataset

In this section, we provide additional details for the datasets that we introduced in Sec. 4 of the main paper.

Synthetic Data. We constructed a synthetic dataset comprising three scenes of varying complexity using Blender, incorporating objects sourced from Objaverse [13] and BlenderKit [5]. Each scene features multiple camera trajectories inspired by Mip-NeRF-360 [3]. Notably, the training and test trajectories are designed to be distinct.

Each level of complexity incorporates the following types of changes: (1) addition of a new object, (2) removal of an existing object, (3) repositioning of an existing object, and (4) combinations of multiple operations simultaneously. Fig. 9 illustrates the modifications introduced in Level-3.

The levels progressively increase in complexity, with each level encompassing the previous ones, as illustrated in Fig. 8. The first scene shows a shelf that contains multiple objects. This shelf is found in the bottom left corner of the room-scale level. Finally, this room is part of the Level-3 configuration that is comprised of four different rooms with a large variety of objects and textures. Below, we outline the key characteristics of each scene.

1. **Level-1:** The area of the scene is $1m^2$. The camera is $3m$ away from the scene center and the focal length of the camera is $50mm$. The changing objects make up 10-20% of the scene.
2. **Level-2:** The area of the scene is $100m^2$. The camera is $5m$ away from the scene center and the focal length of the camera is $50mm$. The changing objects make up $1-2\%$ of the scene.
3. **Level-3:** The area of the scene is $400m^2$. The camera is $10m$ away from the scene center and the focal length of the camera is $50mm$. The changing objects make up $< 1\%$ of the scene.

For each level, we use 200 frames for the initial reconstruction and 25 for training and testing the changed scene. Since 3DGS [19] requires a sparse reconstruction, we generate a point cloud using COLMAP [33]. Given that we already have accurate camera poses and intrinsics from Blender, we adopt the pipeline from Tetra-NeRF [21] to produce a sparse point cloud.

Real-World Data. We casually captured five different scenes using an iPhone 13 at 60 FPS. Four of these scenes were recorded indoors, and one was captured outdoors. The outdoor scene and one indoor scene followed Mip-NeRF-360-style trajectories, while the remaining three indoor scenes utilized Zip-NeRF [4]-like trajectories. For each scene, we extract every 20th frame, resulting in 100-200 images for the initial reconstruction and 10-30 images for the changed parts.

Statistics. In addition to the scene descriptions, we have gathered statistics about the number of pixels and Gaussians constituting the changes in all scenes of all datasets. We have obtained these numbers by counting the inliers in the 2D masks that we predict as described in Sec. 7 and the number of Gaussians in $\mathcal{O}^t$. Fig. 10 shows the number of pixels that have changed compared to the total number of pixels in each dataset. We can see the variety of change complexities in the datasets. A similar trend can be seen in Fig. 11 but for the number of Gaussians.

9. Discussions

9.1. Local Optimization Without Static Elements

In GaussianEditor [10], the authors propose rendering only the modified parts of the scenes. Specifically, their renderer considers only the changed Gaussians, entirely omitting the background. While they leverage this approach to refine an existing Gaussian model, we use it to generate new objects from randomly sampled Gaussians. Additionally, their method optimizes for only 1,000 iterations, whereas we optimize for 30,000.

In our main paper, we present the results of their optimization using our masks in Sec. 4.1 of the leading paper.

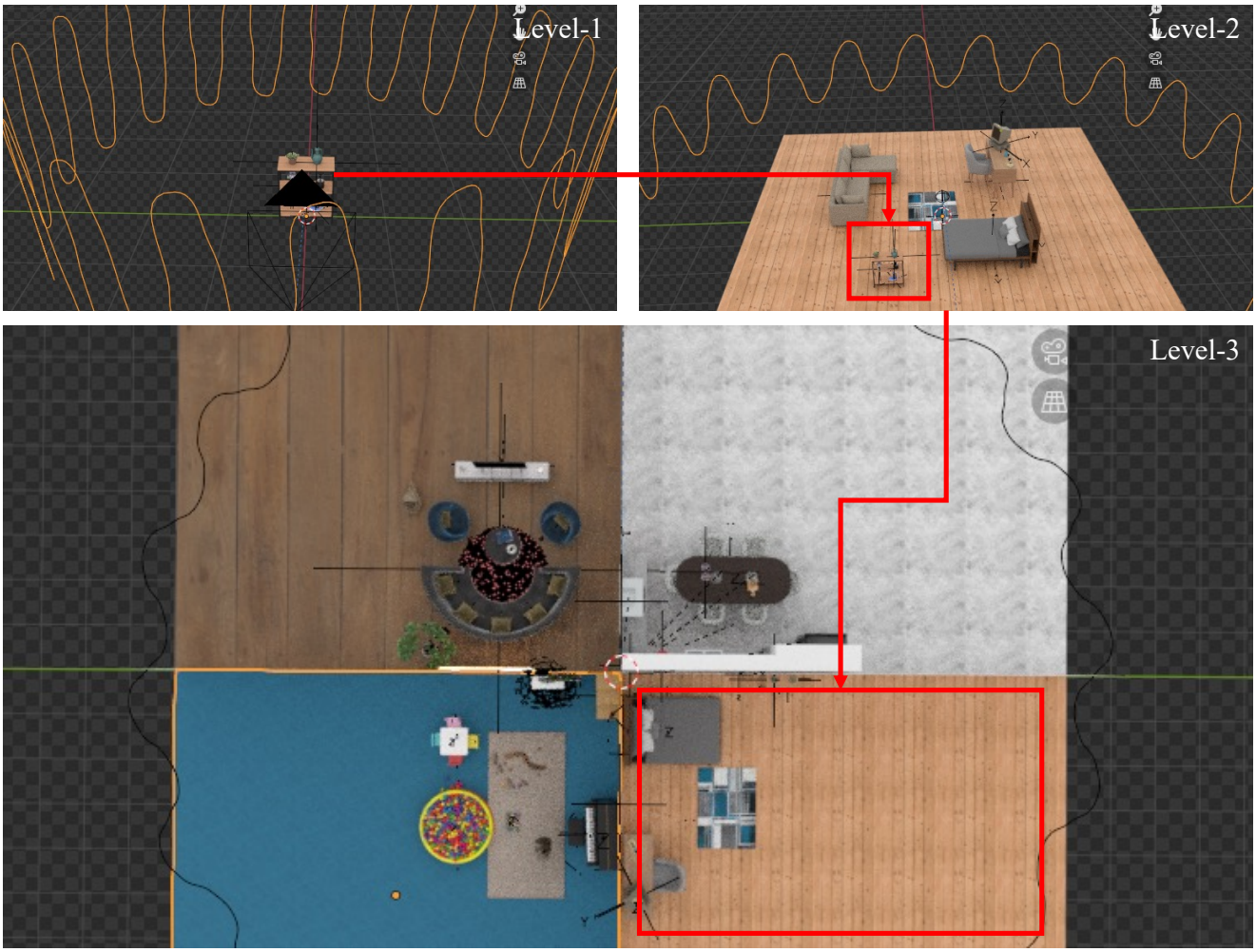

Figure 8. **Overview of the Synthetic Levels.** Each level is part of the next larger level. The location of the smaller scene is marked by red rectangles in the larger scenes. The curves that are visible in each scene, are the trajectories of the camera for the initial reconstruction.

However, we also explored how GaussianEditor would perform if tasked with pruning points during optimization. We observed that even for small objects, the optimization failed to converge. It diverged to the point where all points were pruned, causing the optimization to crash.

In contrast, our method works seamlessly with the same set of masks. This highlights the importance of accounting for existing structures during local optimization.

9.2. Failure Case

As discussed in Sec. 4.2, achieving a high recall in the changed area is critical for reconstruction quality. Existing structures cannot be effectively optimized without sufficient recall, and new structures are constrained too tightly. Fig. 12 illustrates an example where the estimated area is too small, resulting in an inability to modify the scene during optimization properly. We observed that this happens when some structures are extremely thin. Despite DinoV2 being a semantic model, we found that we can reliably segment changes even when objects are replaced by new ones with similar semantics. We also experimented with naïve sampling methods which a) samples a fixed number of points in the entire scene or b) samples a fixed number of points in a fixed radius of the inliers and found that these lead to up to 6 PSNR lower scores.

10. Batched Updates

To obtain a unified reconstruction $\mathcal{G}^t$ that reflects both sets of changes, we track the indices of the modified Gaussians, $\mathcal{I}_1^{t-1}$ and $\mathcal{I}_2^{t-1}$, in the original representation. The final scene is constructed by combining the updated Gaussians $\mathcal{O}_1^t$ and $\mathcal{O}_2^t$ with the unchanged Gaussians from the original scene:

$$\mathcal{G}^t = \mathcal{O}_1^t \cup \mathcal{O}_2^t \cup \mathcal{G}^{t-1} \setminus \mathcal{G}^{t-1}[\overline{\mathcal{I}_1^{t-1}} \cap \overline{\mathcal{I}_2^{t-1}}]. \quad (4)$$

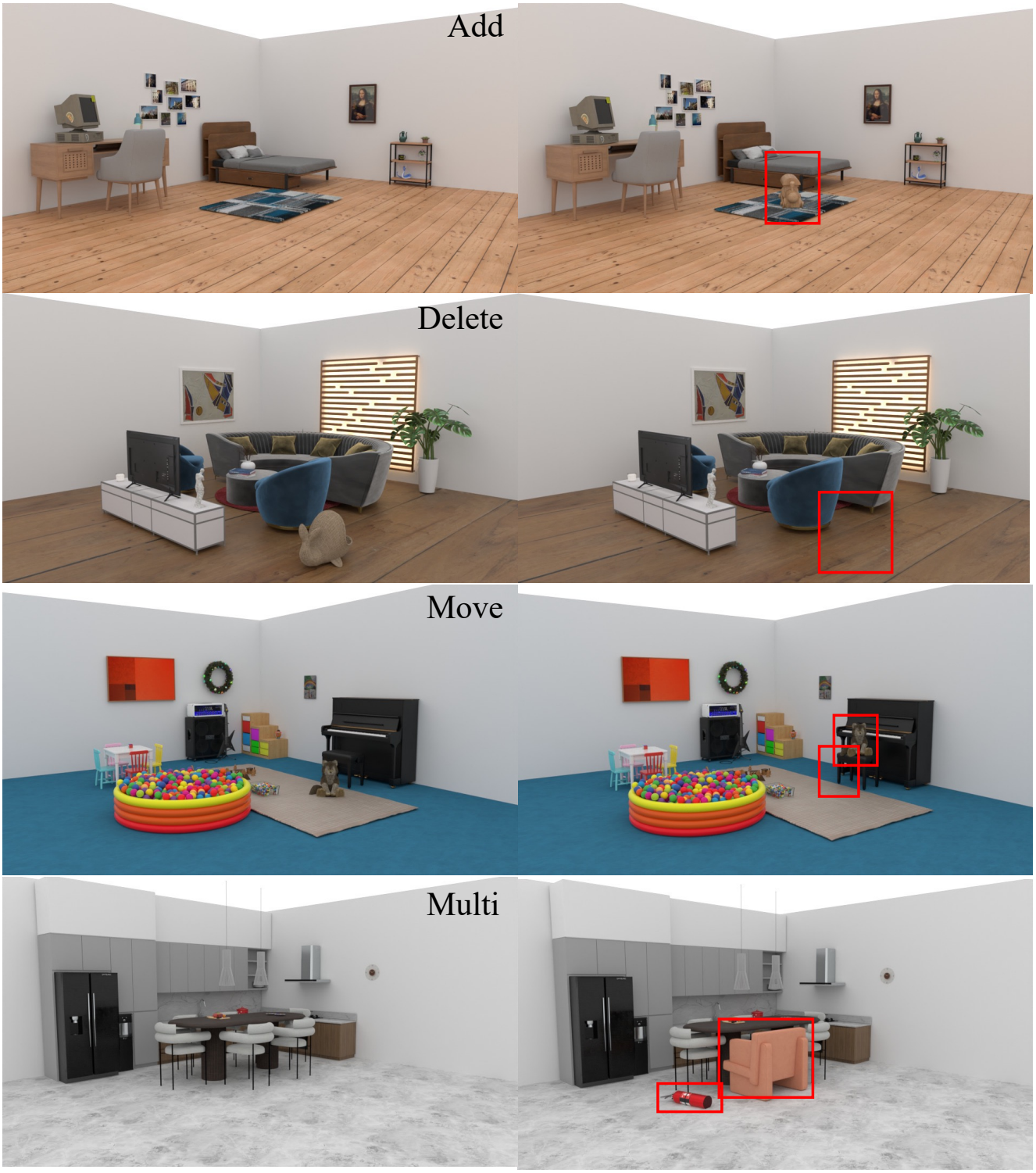

Figure 9. **Changes in Level-3.** Each column shows the effect of one change on the Level-3 dataset. The changed areas are highlighted by red rectangles. 1) a wooden train is added onto the carpet. 2) The whale basket is removed. 3) The husky plushy gets moved. 4) The white chair gets replaced by a red armchair and a fire extinguisher appears on the floor.

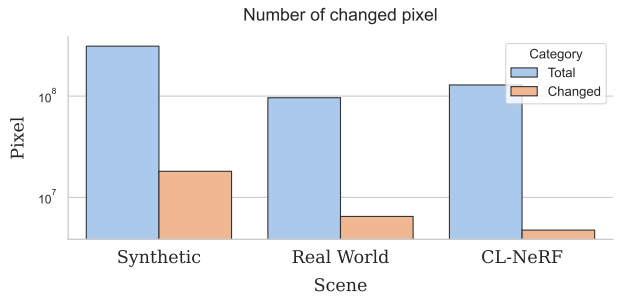

Figure 10. **Number of Pixels in Changes.** Shows the number of pixels that have changed according to our 2D masks compared to the total number of pixels. The y-axis is in log-scale.

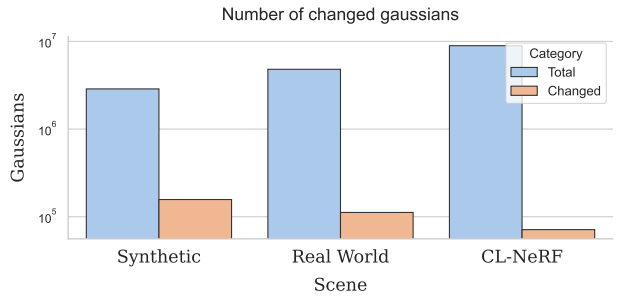

Figure 11. **Number of Gaussians in Changes.**Shows the number of Gaussians that have changed according to our 3D masks compared to the total number of Gaussians. The y-axis is in log-scale.

For ease of notation, we treat Gaussians and indices as indexable sets. This formulation ensures that all unchanged Gaussians are retained, while the updated ones are seamlessly integrated.

By leveraging this approach, our method successfully merges scene updates, enabling efficient reconstruction of complex dynamic environments without the need for full re-optimization.

11. Multi Day

11.1. History Recovery

Algorithm. To efficiently recover any past reconstruction $\mathcal{G}^i$, we introduce an algorithm that stores only the changing Gaussians $\mathcal{O}^t$ and their corresponding indices $\mathcal{I}^t$. This allows us to reconstruct past states without storing redundant information. We assume that in $\mathcal{G}^t$, the first

$$\#^t_{\text{static}} := |\mathcal{G}^{t-1} \setminus \mathcal{O}^t| \tag{5}$$

Gaussians correspond to the static parts of the previous state:

$$\mathcal{G}^{t-1}_{\text{static}} := \mathcal{G}^{t-1} \setminus \mathcal{O}^t. \tag{6}$$

To satisfy this requirement, we store $\mathcal{O}^t$ and $\mathcal{I}^t$ immediately after computing $\mathcal{O}^t$, before any optimization. Additionally, before optimization, we rearrange the Gaussian order to ensure:

$$\mathcal{G}^t[:\#^t_{\text{static}}] = \mathcal{G}^{t-1}_{\text{static}}. \tag{7}$$

Since duplication and pruning only affect $\mathcal{O}^t$, this condition remains invariant throughout optimization. To reconstruct $\mathcal{G}^{t-1}$ from $\mathcal{G}^t$, we: Extract $\mathcal{G}^{t-1}_{\text{static}}$ by accessing the first:

$$\#^t_{\text{static}} = |\{i \mid \forall i \in \mathcal{I}^t, i = 0\}|. \tag{8}$$

2. Retrieve the stored $\mathcal{O}^t$ and reconstruct the full scene as:

$$\mathcal{G}^{t-1} := \mathcal{O}^t \cup \mathcal{G}^{t-1}_{\text{static}}. \tag{9}$$

3. Re-arrange the Gaussians to maintain consistency: $\mathcal{G}^{t-1}_{\text{static}}$ is placed where $\mathcal{I}^t = 0$ and changing Gaussians $\mathcal{O}^t$ are placed where $\mathcal{I}^t = 1$. Alg. 4 shows how this process can be repeated to recover $\mathcal{G}^k$ from $\mathcal{G}^t$, enabling multi-step history recovery. Our algorithm allows for efficient and exact scene recovery while drastically reducing storage requirements. As shown in the main paper, it achieves identical reconstructions to storing the full scene while using significantly less memory.

Algorithm 4 History Recovery

Input: $\mathcal{G}^T$, $(\mathcal{O}^k)_k$,$(\mathcal{I}^k)_k$, n
Output: $\mathcal{G}^n$
function RECOVERSTATE($\mathcal{G}^T$,$(\mathcal{O}^k)_k$, $(\mathcal{I}^k)_k$, n)
  if $n = T$ **then**
    return $\mathcal{G}^T$
  else
    $\mathcal{N} = \mathcal{G}^T[:|\neg\mathcal{I}^T|] + \mathcal{O}^T$ ▷ Array of target size
    $\mathcal{N}[\mathcal{I}^T] = \mathcal{O}^T$ ▷ Set indices to dynamic
    $\mathcal{N}[\neg\mathcal{I}^T] = \mathcal{C}$ ▷ Set complement to static
    return RECOVERSTATE($\mathcal{N}$, $(\mathcal{O}^k)_k$, $(\mathcal{I}^k)_k$, n)
  end if
end function

11.2. Multi Day Reconstruction

In Sec. 5 of the main paper, we discussed the shortcomings of existing methods to scale to multiple days and how our method can elegantly solve this problem due to the explicit representation of the changes. This section showcases our method's prowess over multiple days (4). As a basis, we use the second level of our blender dataset.

Setup. We perform three different operations leading to four reconstructions based on our Level-2 synthetic dataset. For each change, we provide 25 images for training focusing on the changed region.

Figure 12. **Illustration of a Failure Case.** Due to the underestimation of the changed area, the optimization can not possibly make the appropriate changes to the scene. The area of interest is marked by a red rectangle.

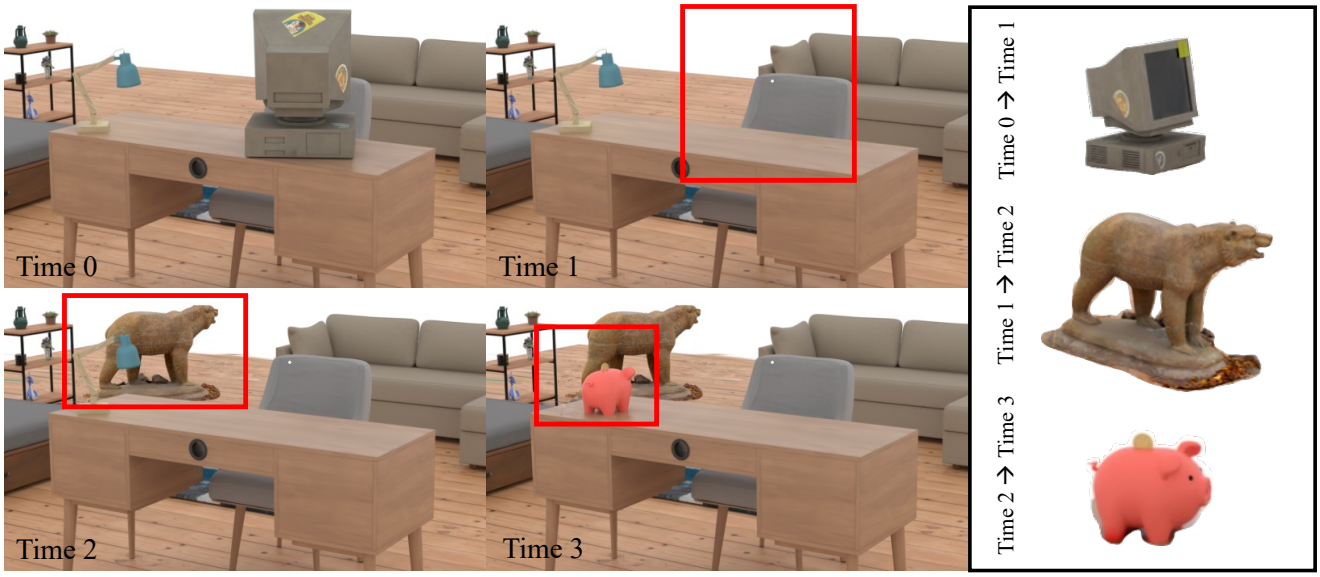

Figure 13. **Reconstruction Results over Multiple Days.** In this figure we show the same scene and viewpoint at four different points in time. The changed region is highlighted in red. On the right side are the changed objects that can be used to efficiently recover any point in time. From 0 to 1, the computer on the desk gets removed. From 1 to 2, a statue of a bear gets added in the back. From 2 to 3, the desk lamp gets replaced by a piggy bank.

Results. Fig. 13 shows some qualitative results of the reconstructions in each time step. We can see that unchanged regions stay the same over the time steps and that no error accumulates due to forgetting. Additionally, the figure separately shows the objects involved in the change. Our method has the advantage that saving a new day's reconstruction is as simple as saving the Gaussians of the changed object and their original indices. With just the base scene and the changed objects, we can go back in time, reconstructing any time steps scene immediately and at no loss in quality.

12. Complete Results

In this section, we present the extended results to the CL-Splats dataset and the CL-NeRF dataset from Sec. 4.1.

12.1. *CL-Splats* Dataset

Level-1. Table 8 presents the per-operation performance of all methods on our Level-1 dataset. Our method demonstrates a clear advantage, outperforming all baselines substantially. Among the baseline methods, 3DGS shows competitive performance in this scenario, benefiting from the relatively high scene coverage compared to other levels. However, the local optimization of 3DGS within 2D masks (3DGS+M) proves to be particularly challenging, leading to

a noticeable drop in overall performance despite the simplicity of the setting.

GaussianEditor, which also employs local optimization, demonstrates clear limitations due to its reliance on 2D masks and the absence of background consideration during optimization. This drawback is particularly evident in the PSNR scores achieved during object deletion, where GaussianEditor performs noticeably worse than other methods. Unlike other approaches that can effectively handle object removal, GaussianEditor's failure to account for the background significantly impacts its metrics.

Additionally, we observed that CL-NeRF struggles to converge on this dataset. Even when the training duration was doubled to 400,000 steps, no significant improvement in results was achieved. This limitation appears to stem from the inherent characteristics of the underlying NeRF representation.

Level-2. As shown in Table 9, the performance of all methods is evaluated on the Level-2 dataset. Compared to Level-1, the baseline methods—3DGS, 3DGS+M, and GaussianEditor—demonstrate a noticeable decline in performance, indicating increased difficulty at this level. In contrast, our method remains highly consistent, underscoring its robustness across varying levels of complexity. Interestingly, the challenges associated with object deletion also persist at this level.

CL-NeRF achieves improved reconstruction quality compared to Level-1, though it still lags significantly behind our method.

Level-3. We can see in Table 10 that Level-3 continues to follow the established trends, supporting our synthetic data design. The baseline methods show a further decline in performance compared to Level-2, while our method delivers stable and reliable results. Interestingly, CL-NeRF exhibits improved performance on this dataset, likely due to the greater object distances, which make PSNR less sensitive to the smoother reconstructions it generates.

Real-World Data. Table 11 presents the complete results on real-world data. While the performance gap between our method and the baseline approaches narrows in this setting, our method outperforms the others on average. Unlike CL-NeRF and GaussianEditor, our approach demonstrates reliability, avoiding catastrophic failures.

12.2. CL-NeRF Dataset

Whiteroom. As shown in Table 12, despite the Whiteroom's sparse textures and challenging lighting conditions, our method achieves performance comparable to CL-NeRF. In contrast, the baseline 3DGS performs significantly worse, mainly due to the limited coverage of the scene's unchanged regions.

Method/Ops	**Add**			**Delete**		
	PSNR↑	LPIPS↓	SSIM↑	PSNR↑	LPIPS↓	SSIM↑
3DGS [19]	10.688	0.445	0.313	6.697	0.499	0.345
CL-NeRF [49]	26.416	0.123	0.807	23.364	0.154	0.761
Ours	34.474	0.009	0.979	34.961	0.009	0.979

Table 7. **Reconstruction Quality on the Rome Dataset from the CL-NeRF dataset.**

Additionally, we observe that move-and-replace operations result in poorer reconstruction quality than add-and-delete operations. While this can be partly attributed to the inherent complexity of these operations, the primary challenge stems from the placement of the objects involved. These objects are often situated in distant corners, where reduced view coverage adversely impacts reconstruction accuracy.

Kitchen. Table 13 provides the complete results for the Kitchen scene from the CL-NeRF dataset. On this dataset, our method achieves results comparable to those of CL-NeRF. However, CL-NeRF faces significant challenges in accurately reconstructing the scene after object deletion, movement, or replacement operations. In contrast, our approach produces reliable reconstructions, demonstrating its robustness.

Rome. Table 7 highlights our method's ability to reconstruct large-scale scenes with remarkable accuracy and ease, such as the Colosseum in Rome. Comparatively, CL-NeRF faces significant challenges in delivering detailed reconstructions for such complex scenarios.

Method/Operation	Add			Delete			Move			Multi		
	PSNR↑	LPIPS↓	SSIM↑	PSNR↑	LPIPS↓	SSIM↑	PSNR↑	LPIPS↓	SSIM↑	PSNR↑	LPIPS↓	SSIM↑
3DGS [19]	23.018	0.051	0.956	27.707	0.021	0.977	25.897	0.030	0.972	25.947	0.030	0.972
3DGS+M	18.802	0.101	0.899	18.669	0.108	0.894	18.927	0.116	0.889	19.682	0.096	0.907
GaussianEditor [10]	25.548	0.047	0.973	22.863	0.065	0.957	24.193	0.051	0.959	25.254	0.044	0.967
CL-NeRF [49]	26.541	0.051	0.947	26.918	0.045	0.953	24.902	0.053	0.946	25.203	0.054	0.947
CL-Splats (ours)	41.923	0.001	0.998	44.018	0.001	0.998	37.481	0.007	0.994	36.310	0.007	0.994

Table 8. **Reconstruction Quality on Level-1 of the Synthetic Data from the CL-Splats Dataset.**

Method/Operation	Add			Delete			Move			Multi		
	PSNR↑	LPIPS↓	SSIM↑	PSNR↑	LPIPS↓	SSIM↑	PSNR↑	LPIPS↓	SSIM↑	PSNR↑	LPIPS↓	SSIM↑
3DGS [19]	17.319	0.245	0.761	26.081	0.125	0.854	26.496	0.115	0.874	25.674	0.113	0.881
3DGS+M	13.731	0.330	0.715	14.118	0.322	0.639	14.845	0.339	0.679	13.581	0.382	0.635
GaussianEditor [10]	22.386	0.109	0.926	19.521	0.162	0.887	24.792	0.072	0.942	26.913	0.074	0.939
CL-NeRF [49]	28.845	0.077	0.918	31.157	0.068	0.928	31.076	0.070	0.927	28.134	0.109	0.894
CL-Splats (ours)	39.528	0.025	0.978	41.647	0.016	0.982	41.838	0.017	0.9815	40.329	0.015	0.979

Table 9. **Reconstruction Quality on Level-2 of the Synthetic Data from the CL-Splats Dataset.**

Method/Operation	Add			Delete			Move			Multi		
	PSNR↑	LPIPS↓	SSIM↑	PSNR↑	LPIPS↓	SSIM↑	PSNR↑	LPIPS↓	SSIM↑	PSNR↑	LPIPS↓	SSIM↑
3DGS [19]	16.509	0.389	0.696	15.602	0.395	0.678	18.010	0.349	0.756	15.653	0.402	0.680
3DGS+M	12.638	0.437	0.682	12.397	0.403	0.693	12.767	0.505	0.601	11.366	0.500	0.611
GaussianEditor [10]	15.581	0.378	0.725	10.443	0.434	0.732	10.951	0.481	0.718	9.191	0.446	0.718
CL-NeRF [49]	34.560	0.043	0.951	34.326	0.045	0.952	34.772	0.045	0.951	34.326	0.045	0.952
CL-Splats (ours)	39.902	0.023	0.979	39.842	0.024	0.979	40.027	0.023	0.979	38.669	0.025	0.978

Table 10. **Reconstruction Quality on Level-3 of the Synthetic Data from the CL-Splats Dataset.**

Method/Operation	Cone			Shoe			Shelf			Room			Desk		
	PSNR↑	LPIPS↓	SSIM↑	PSNR↑	LPIPS↓	SSIM↑	PSNR↑	LPIPS↓	SSIM↑	PSNR↑	LPIPS↓	SSIM↑	PSNR↑	LPIPS↓	SSIM↑
3DGS [19]	13.449	0.471	0.136	6.592	0.409	0.326	20.211	0.120	0.860	10.061	0.471	0.326	8.506	0.409	0.345
3DGS+M	8.900	0.578	0.117	5.976	0.462	0.226	12.177	0.496	0.599	8.359	0.385	0.184	7.515	0.381	0.226
GaussianEditor [10]	26.277	0.140	0.850	25.969	0.053	0.921	10.211	0.434	0.671	27.944	0.039	0.950	30.268	0.050	0.944
CL-NeRF [49]	21.482	0.422	0.511	25.583	0.192	0.829	25.731	0.154	0.844	15.125	0.551	0.574	28.419	0.132	0.867
Ours	27.163	0.114	0.886	28.144	0.046	0.926	25.256	0.064	0.944	31.203	0.0375	0.965	29.477	0.066	0.929

Table 11. **Reconstruction Quality on Real-World Data from the CL-Splats Dataset.**

Method/Ops	Add			Delete			Move			Replace		
	PSNR↑	LPIPS↓	SSIM↑	PSNR↑	LPIPS↓	SSIM↑	PSNR↑	LPIPS↓	SSIM↑	PSNR↑	LPIPS↓	SSIM↑
3DGS [19]	11.503	0.275	0.733	14.839	0.233	0.773	11.717	0.284	0.731	11.524	0.282	0.728
CL-NeRF [49]	31.736	0.056	0.935	34.021	0.053	0.938	29.121	0.089	0.906	29.917	0.082	0.912
Ours	34.014	0.053	0.953	31.821	0.068	0.935	27.889	0.078	0.909	27.805	0.078	0.909

Table 12. **Reconstruction Quality on Whiteroom from the CL-NeRF dataset.**

Method/Ops	Add			Delete			Move			Replace		
	PSNR↑	LPIPS↓	SSIM↑	PSNR↑	LPIPS↓	SSIM↑	PSNR↑	LPIPS↓	SSIM↑	PSNR↑	LPIPS↓	SSIM↑
3DGS [19]	19.486	0.4021	0.511	8.487	0.602	0.4074	7.339	0.5573	0.4025	8.444	0.583	0.425
CL-NeRF [49]	27.199	0.263	0.789	24.290	0.316	0.751	24.610	0.304	0.759	23.241	0.334	0.736
Ours	27.546	0.296	0.695	26.927	0.323	0.6738	27.291	0.319	0.681	27.151	0.322	0.679

Table 13. **Reconstruction Quality on Kitchen from the CL-NeRF dataset.**