KISS-GS: 3D Gaussian Splatting Compression Kept Simple

Wieland Morgenstern[1,2], Friedrich Elias Branschke[1,3], Florian Fleischmann[1,3], Adrian Szatmari[1,3], Paul Schlack[1], Florian Barthel[1,2], Peter Eisert[1,2], and Anna Hilsmann[1,2]

[1] Fraunhofer HHI, Germany
{first}.{last}@hhi.fraunhofer.de
[2] Humboldt-Universität zu Berlin, Germany
[3] Technische Universität Berlin, Germany

Abstract. Scene reconstruction with 3D Gaussian Splatting (3DGS) has become common, however deployment remains painful as the uncompressed file sizes can be massive. Current 3DGS compression systems combine multiple strategies for file size reduction, which can obscure where gains come from and limit component reuse across training pipelines. To make the gains more transparent, we propose KISS-GS, a modular compression pipeline named after the principle of keeping things simple, designed to decouple compression entirely from training. Given a 3DGS scene reconstructed with vanilla 3DGS, we are able to reduce it through compaction by 15.7x using a combination of state-of-the-art pruning schemes. Then we encode it into an image-based format designed for simple, ubiquitous decoding. With the SOG-XT format, we propose a novel extension to Self-Organizing Gaussians with two main contributions: (i) Self-organizing 2D Codebooks and (ii) Parallel Representative Assignment Smoothing (PRAS), which leverages the symmetry of quaternion and scale parameterizations to produce 2D attribute grids more amenable to encoding. This encoding reduces scene size by 6.6x. We show that optional encoding-aware fine-tuning yields a further 2.2x. Across standard 3DGS benchmarks, our simple and modular approach thus achieves a total of 85× to 319× reductions in the size of the scene over uncompressed vanilla 3DGS, setting new benchmarks for real-world scenes and surpassing tightly integrated methods in rate-distortion. Decoding relies solely on web-native image formats, and the modular design makes each stage easy to combine with future advances in reconstruction and compaction.
Code and project page: `https://fraunhoferhhi.github.io/KISS-GS/`

1 Introduction

"Simplicity is a great virtue but it requires hard work to achieve it and education to appreciate it. And to make matters worse: complexity sells better."
— *Edsger W. Dijkstra (1984),* On the nature of Computing Science *[6]*

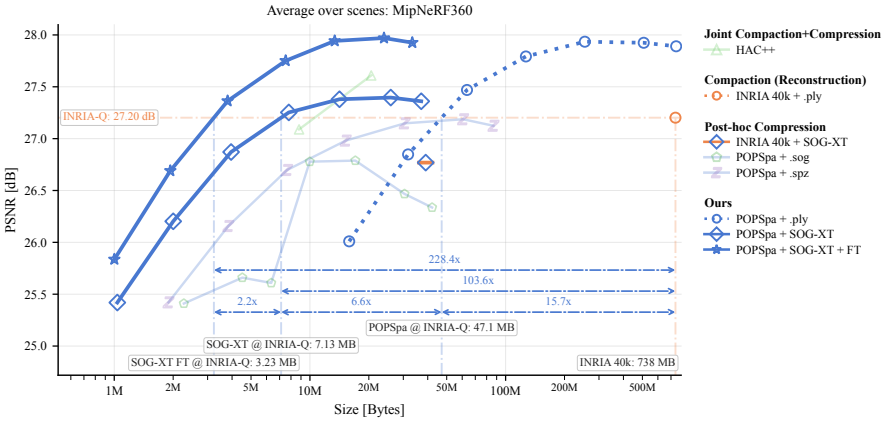

Fig. 1: Rate distortion on *Mip-NeRF 360* for KISS-GS, with quality measured by peak signal-to-noise ratio (PSNR). At the INRIA 3DGS quality reference (INRIA-Q), KISS-GS reduces file size by 228× from 738 MB to 3.2 MB. From the uncompressed INRIA `.ply` baseline, POPSpa matches INRIA-Q with a 15.7× reduction, our SOG-XT encoding adds 6.6×, and encoding- aware fine-tuning adds a further 2.2×, outperforming HAC++ [5].

In practice, 3D Gaussian Splatting (3DGS) scenes are still commonly exchanged as raw `.ply` exports. These files are easy to produce, but large and inefficient for storage, streaming, and deployment. As 3DGS moves beyond research prototypes into hardware-constrained web viewers, mobile renderers, and AR/VR applications, file size, decoding cost, and memory footprint become obstacles.

Storage efficiency was not a focus of the original 3DGS formulation [16], leading to a rapidly growing body of work on compression. The recent 3DGS.zip survey [1] distinguishes two storage-reduction strategies, *compaction*, which reduces the number of primitives, and *compression*, which reduces the number of bits per primitive. To avoid overloading the term *compression*, we refer to this bits-per-primitive component as *encoding*. We add *adaptation* as a third strategy, which uses encoding-aware training or fine-tuning to steer parameters toward configurations that are friendly to quantization and coding. KISS-GS turns these three strategies into explicit pipeline stages, whose separate rate-distortion gains are previewed in Fig. 1. Many methods combine multiple strategies to achieve their final rate-distortion scores. While effective, this obscures where gains actually originate, creating an *attribution gap*. Without clear attribution, it is difficult to compare methods fairly and reuse components across pipelines. Compaction alone can already yield large gains. Densification and pruning improvements can reduce primitive counts by an order of magnitude in some settings without large drops in reconstruction quality, as demonstrated by Mini-Splatting [7, 8]. Some approaches, such as anchor-based families of methods, restructure the representation so fundamentally that compression becomes inseparable from the

reconstruction regime and data format. This *training-format coupling* effectively locks the codec to a particular 3DGS training pipeline: when new reconstruction methods or parameterizations emerge, their improvements cannot be adopted without reworking the compression scheme itself.

We call our approach *KISS-GS*, inspired by the KISS principle ("keep it simple, stupid" [37]) and restrict ourselves to a deliberately simple, browser-compatible decoding pipeline backed by widely supported image codecs.

Guided by this decoder-first stance, we ask three research questions (RQs):

1. (**RQ1**) Can we close the **attribution gap** by modularizing the compression pipeline and measuring the contribution of each stage in turn?
2. (**RQ2**) Can a **simple image-based encoding format** match more complex, integrated methods like HAC++ [5] across real-world 3DGS compression benchmarks?
3. (**RQ3**) Does **encoding-aware fine-tuning** still deliver gains when constrained to a simple, web-friendly codec?

To close the **attribution gap** (RQ1), we propose KISS-GS, a modular compression framework that separates compaction, encoding, and adaptation into distinct stages, enabling each stage to be replaced, ablated, or analyzed independently. We instantiate the compaction stage with POPSpa, a reconstruction-independent module that combines state-of-the-art pruning schemes after the initial reconstruction, ensuring that gains from fewer primitives are never conflated with gains from better encoding. To test whether a **simple image-based encoding format** can match more integrated methods (RQ2), we introduce SOG-XT, an image-based codec and storage format designed for web-friendly decoding. We extend the Self-Organizing Gaussians (SOG) [27] approach with (i) Self-organizing 2D codebooks for spherical harmonics and (ii) Parallel Representative Assignment Smoothing (PRAS) to improve compression rates while keeping the decoder simple. To isolate **encoding-aware fine-tuning** under the constrained decoder model (RQ3), we systematically evaluate it with encoding-in-the-loop as a separate stage.

2 Related work

The original 3DGS paper [16] focused on reconstruction quality and real-time rendering performance, without explicitly addressing storage or memory constraints. Due to its strong reconstruction quality with explicit primitives (in contrast to the implicit nature of NeRF-based approaches [26, 28]), the community quickly adopted 3DGS. Many follow-up works have since investigated storage reduction, memory efficiency, and rendering performance, introducing modifications at different pipeline stages which can broadly be categorized into *compaction*, *encoding*, and *adaptation* [1].

Compaction reduces the number of active Gaussians while preserving or improving reconstruction quality, directly impacting storage footprint and rendering cost. Many methods perform sensitivity-based pruning after or during training by estimating the importance of individual Gaussians. Various criteria have

been proposed, including strict primitive budgets via score-based sampling [25], reconstruction error [11, 12, 20, 33], as well as primitive transmittance, opacity, or rendering contribution [9, 19, 22]. Another class of methods controls the Gaussian population through sampling-based strategies. For example, MCMC-GS [17] removes low-opacity primitives and relocates them to high-opacity regions to improve reconstruction under a fixed primitive budget. Finally, some approaches formulate compaction as an explicit sparsity-constrained optimization problem: RDO-GS [41] learns binary pruning masks jointly with the reconstruction objective using sparsity-inducing regularization, whereas GaussianSpa [47] introduces an auxiliary sparsity variable and uses an augmented-Lagrangian formulation that alternates between standard reconstruction optimization and a projection step that enforces a target sparsity level.

Encoding reduces the number of bits required to represent each primitive without necessarily changing the primitive count. A common strategy is numerical compression of Gaussian attributes, often using vector quantization (VQ), particularly for view-dependent spherical harmonics [29, 30, 32]. These approaches reduce bitwidth without altering primitive layout through quantization and entropy coding.

Another class of methods replaces explicit attribute storage with structured representations that exploit spatial correlations between primitives. These include anchors, codebooks, multi-layer perceptrons (MLPs), octrees, triplanes, or hash grids to encode Gaussian attributes [3, 21, 24, 36, 44]. Scaffold-GS [24], for example, predicts attributes from neighboring anchors via a lightweight MLP. Extensions include learning sparse primary and dense secondary anchors [23], hierarchical pyramids where coarser anchors predict finer ones [42], and querying hash-grid features at anchor locations [3, 5].

Another direction organizes Gaussian attributes into **structured 2D layouts** to leverage standard image codecs. Mapping 3D primitives to 2D grids via Morton curves or Self-Organizing Gaussians (SOG) [27] produces locally smooth attribute maps optimized for conventional encoders. Although encoding requires an initial sorting step, decoding reduces to standard image decompression, which is widely hardware-accelerated and readily available in browser and mobile environments. This decoding simplicity has enabled **practical deployment** on mobile devices, AR/VR platforms, and browser-based viewers, leading to formalization of the `.sog` format [35], which combines Self-Organizing Gaussians with VQ strategies for spherical harmonics. The format has been adopted in research and practical web pipelines [15, 34, 39]. Alternative formats like `.spz` [31] use structured quantization with a custom binary format.

Adaptation modifies scene parameters to make them more compatible with the target encoding. This can be achieved either during training or through post-training fine-tuning. RDO-GS [41] adds VQ-loss and rate-loss to the objective function, FCGS [4] incorporates entropy loss accounting for bit count, while HAC [3] and HAC++ [5] combine rendering fidelity, entropy, masking, and hash-grid size losses.

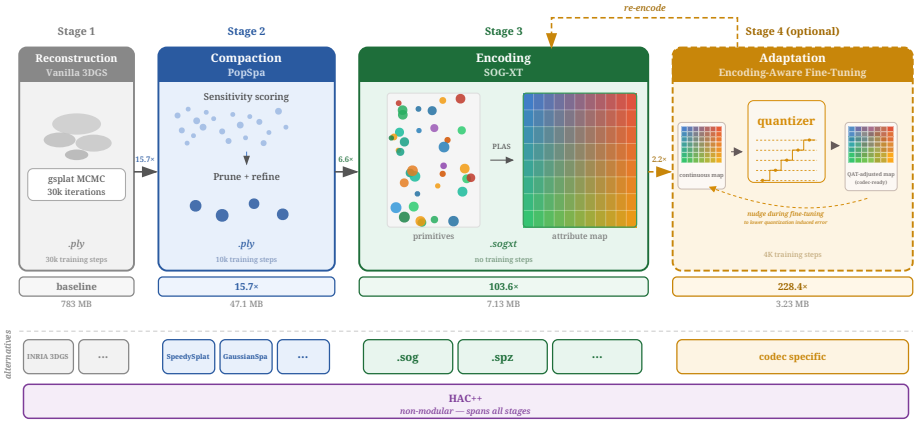

Fig. 2: The KISS-GS pipeline with four distinct stages. (1) A standard 3DGS reconstruction yields a `.ply`, with primitive count setting the rate-distortion point. (2) POPSpa compacts the scene. (3) SOG-XT encodes the compacted primitives into attribute images. (4) Optional codec-aware fine-tuning improves rate-distortion without changing the output format. Below the dashed separator, we show per-stage alternatives to the KISS-GS stages and contrast them with integrated methods like HAC++ spanning all stages as a single non-modular unit.

3 Modules of the KISS-GS pipeline

Figure 2 shows how the modular KISS-GS pipeline takes any off-the-shelf 3DGS scene as input, applies compaction, encodes the result with SOG-XT, and optionally fine-tunes it with the codec in the loop. When competing methods differ in both file size and reconstruction quality, individual operating points listed in tables are difficult to compare. We therefore use rate-distortion curves to compare this tradeoff across operating points. When one curve dominates another in the Pareto sense (higher quality at the same file size, or smaller file size at the same quality) the dominant method is strictly better at every operating point. We control the rate-distortion tradeoff primarily through primitive count, which has the additional benefit of reducing rendering cost and the work required for encoding and decoding. Using more primitives with heavier compression to reach the same operating point would instead increase all three, with no gain in rate or quality. We describe each stage in turn.

3.1 3D Reconstruction

KISS-GS accepts any standard 3DGS `.ply` file as input, regardless of which implementation produced it. This is a deliberate design choice: by decoupling compression from reconstruction, we avoid training-format coupling and ensure that improvements in reconstruction quality can be adopted without changes

to the compression pipeline. The standard 3DGS parameterization (means, covariances represented as quaternions and scales, spherical harmonic colors up to degree 3, and scalar opacity) is shared across all major implementations, and KISS-GS preserves this representation throughout. Importantly, this parameterization relies on no MLPs, hash grids, or anchor structures, keeping the decoder simple. The original training pipelines apply non-linear mappings to scales and opacities to constrain parameter ranges. We use the same mappings because the resulting values are well suited to uniform quantization.

The 3DGS reconstruction ecosystem is actively evolving, with implementations varying in speed, quality, and license, including INRIA [16], gsplat [45], Lichtfeld Studio [40], and Hahlbohm et al. [10]. Being decoupled from any specific implementation means KISS-GS can benefit from future reconstruction advances without modification. Our main experiments use the gsplat MCMC reconstruction [17] with default parameters for 30k iterations; compatibility with alternative implementations is demonstrated in Sec. 4.

3.2 Compaction with POPSpa

Densely trained 3DGS scenes typically contain substantial redundancy. Since storage scales approximately linearly with the primitive count, compaction is essential to achieve high compression ratios. We instantiate this stage with *POPSpa*, a reconstruction-independent compaction recipe that starts from a trained 3DGS scene and combines score-based pruning from GaussianPOP [20] with the alternating optimize-sparsify scheme of GaussianSpa [47], augmented by effective-rank regularization. Its role is not to introduce a new pruning objective, but to provide a strong, replaceable module whose primitive-count gains can be measured separately from encoding gains.

Effective Rank Regularization. Although all covariance matrices maintain a full rank of 3, they exhibit widely different eigenvalue spectra, dictating the shape of the Gaussians. To quantify these shapes during training, we utilize the effective rank (erank) as a differentiable relaxation that measures the dimensions effectively occupied by the distribution. Formally, for a 3D Gaussian with ordered scale attributes $s_1 \geq s_2 \geq s_3$, the normalized eigenvalues of its covariance matrix Σ are defined as $p_i = \frac{s_i^2}{\sum_{j=1}^{3} s_j^2}$. The effective rank is formulated as the exponential of the Shannon entropy of the probability distribution induced by the eigenvalues of the covariance matrix: $\mathrm{erank}(\Sigma) = \exp\left(-\sum_{i=1}^{3} p_i \cdot \log p_i\right)$ [38]. We employ the effective rank regularization (Eq. (1)) proposed by Hyung *et al.* [14] to discourage the formation of extremely needle-shaped Gaussians, characterized by $\mathrm{erank}(\Sigma_k) \approx 1$, which can overfit training views and produce artifacts from novel viewpoints. At the same time, this regularization favors disk-shaped Gaussians exhibiting an $\mathrm{erank}(\Sigma_k) \approx 2$, which are better suited for surface reconstruction.

$$\mathcal{L}_{\mathrm{erank}} = \sum_{k} \lambda_{\mathrm{erank}} \max(-\log(\mathrm{erank}(\Sigma_k) - 1 + \epsilon), 0) + s_3 \tag{1}$$

First Pruning Stage. For each Gaussian primitive k, we use the error criterion of GaussianPOP [20] to estimate how much the rendered image changes when k is removed from the scene representation. Let C_{render} be the original rendered color and C'_{render} the color obtained after omitting Gaussian k; the per-primitive error $\Delta SE_k = \|C_{\text{render}} - C'_{\text{render}}\|^2$ then quantifies the change in pixel colors, accumulated over all training views. We compute ΔSE_k for all primitives using the efficient single-pass algorithm of [20] and prune a fixed fraction with the lowest scores.

Optimize-Sparsify Refinement. To counteract potential degradation caused by large pruning ratios, we refine the remaining primitives using the alternating optimize-sparsify scheme of GaussianSpa [47]. GaussianSpa formulates primitive selection as a constrained optimization problem with an ℓ_0 sparsity objective on primitive opacities:

$$\min_{\mathbf{a},\boldsymbol{\Theta}} \mathcal{L}(\mathbf{a}, \boldsymbol{\Theta}), \qquad \text{s.t. } \|\mathbf{a}\|_0 \leq \kappa \tag{2}$$

where $\boldsymbol{\Theta}$ denotes all primitive parameters except opacity $\mathbf{a}$, and κ is the primitive budget. The optimization alternates between a standard reconstruction step and a sparsification step that gradually reduces the opacity of expendable primitives. GaussianSpa applies this scheme during training; we repurpose it as a post-training refinement stage of 5k iterations. This allows the remaining primitives to compensate for the reduced primitive set by redistributing opacity and geometry among nearby Gaussians. In this stage, we additionally apply the effective rank regularization term $\mathcal{L}_{\text{erank}}$ defined in Eq. (1).

Second Pruning and Fine-Tuning. For a second pruning pass, we recompute ΔSE_k to remove primitives that have become redundant. Finally, another 5k iterations of fine-tuning recover quality lost in the previous steps, for a total of 10k post-processing iterations.

3.3 Encoding with SOG-XT

Following compaction with POPSpa, the remaining primitives are encoded by SOG-XT, an image-based codec extending the Self-Organizing Gaussians (SOG) approach [27] with two novel contributions described below.

The core idea of SOG [27] is to arrange the N primitives into a $\lceil\sqrt{N}\rceil \times \lceil\sqrt{N}\rceil$ 2D grid using PLAS, a parallel assignment algorithm that preserves spatial locality in the grid layout. The resulting attribute maps are locally smooth, making them well-suited for standard image codecs. SOG stores the individual per-attribute slices as lossless JPEG-XL images, with quantization applied post-training. In the original SOG method, densification parameters, training-time smoothness regularization, and quantization are coupled, which makes its encoding stage difficult to apply in isolation. The `.sog` format keeps the image-based storage idea of SOG [27], but makes it usable as a post-training format by using

Morton-order curves to arrange primitives and k-means clustering for spherical harmonics dimensionality reduction, with centroids sorted lexicographically.

Like `.sog`, SOG-XT decouples encoding from reconstruction and compaction entirely, operating as a pure post-training step on any compacted scene. SOG-XT retains the PLAS-based grid layout for all attributes but replaces the 48 raw SH slices of SOG [27] with vector quantization: k-means clustering reduces the high-dimensional SH representation to a compact codebook, with only per-primitive cluster indices stored. Rather than sorting these centroids lexicographically as in `.sog`, SOG-XT sorts the 2D cluster centroids with PLAS, yielding a codebook that is itself locally smooth in 2D. Cluster assignments are stored as two 8-bit image channels (u, v) indexing the PLAS-sorted centroid grid. This deliberately caps the SH codebook at $256 \times 256 = 65{,}536$ entries, which was the point where increasing the codebook size no longer improved PSNR in our experiments. This 2D structure additionally enables fine-tuning to refine the codebook and provides a natural foundation for future vector quantization methods that exploit spatial coherence more aggressively.

3.4 Covariance Symmetry and PRAS

The mapping from a 3D covariance matrix to the parameterization of tuples of quaternions and scales (q, s) is one-to-many. A unique covariance matrix maps to an entire equivalence class of discrete parameter choices that all yield the identical spatial distribution. This symmetry stems from three independent sources of redundancy in the decomposition of $\mathbf{\Sigma}$: **Scale Permutations**: The three principal radii can be assigned to the coordinate axes in $3! = 6$ ways, provided they are compensated by 90° rotations around principal axes to arrive at the identical covariance. **Eigenvector Orientations**: Flipping eigenvector directions yields 2^3 combinations, of which 4 represent valid rotations satisfying $\det(R) = 1$. **Quaternion Double-Cover**: For every valid rotation matrix R, there exist two quaternions, q and $-q$, which produce that exact rotation.

Combined, this results in $6 \cdot 4 \cdot 2 = 48$ distinct parameter combinations within an equivalence class that describe a single Gaussian. To exploit this symmetry, we propose **Parallel Representative Assignment Smoothing (PRAS)**. Similarly to *PLAS*, *PRAS* employs coarse to fine optimization over 2D parameter grids. First, to ensure comparable magnitudes during optimization, the floating-point quaternions and log-scales are mapped to 8-bit quantized proxy grids. The algorithm then iterates over a sequence of exponentially decreasing blur radii. At each scale, a low-pass filter is applied to the current proxies to generate spatially smoothed target grids. For every primitive, *PRAS* evaluates all candidates within its equivalence class and assigns the representative that minimizes the sum of absolute channel differences, i.e. the ℓ_1 distance, to the smoothed targets. Because the search space is strictly restricted to valid equivalence classes, *PRAS* effectively removes artificial variance to improve the compressibility of scale and quaternion grids while leaving the resulting 3D covariances and grid coordinates unchanged.

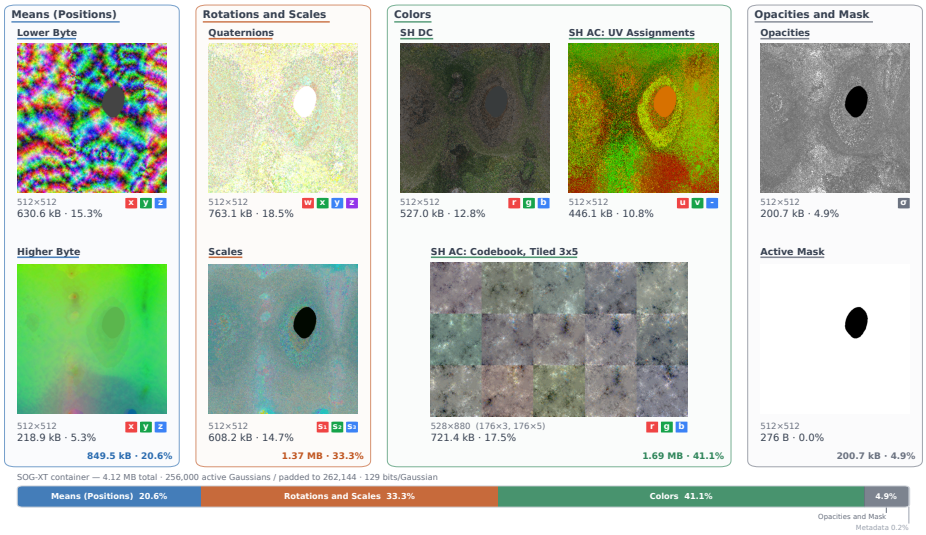

Fig. 3: Encoding overview for SOG-XT. PLAS sorts primitives into 2D attribute grids using both mean byte planes. View dependent color is reduced with a k-means codebook whose centroids are PLAS sorted again, and assignments are stored as UV indices. Bars show the relative on-disk size contribution of each component.

3.5 SOG-XT format

Figure 3 shows an example of an encoded SOG-XT container with the file size and share of the total for each stored component.

Following SOG [27], we arrange primitives into a 2D grid with PLAS and store the attribute grids as images. SOG uses JPEG-XL with higher bit depth support, while we restrict storage to browser-decodable 8-bit images. We avoid truncation by padding primitives to a square grid with side length aligned to a multiple of 8 and storing an *active mask*. Padded primitives use fixed defaults and are dropped at decoding time, which adds little metadata overhead.

We apply per-channel min-max quantization with uniform step sizes and store each channel's minimum and maximum as float32 values in a YAML metadata file. In the example container shown in Fig. 3, all metadata including these ranges accounts for only 0.2% of the total size on disk. These values are therefore negligible for compression, but necessary for exact dequantization.

For the Gaussian means, which define primitive positions, we follow SOG and apply a signed log remapping for space contraction before 16-bit quantization. As in browser-oriented `.sog`, we store the quantized positions as two 8-bit byte planes. The upper byte captures coarse spatial structure, while the lower byte contains finer residual variation and is less smooth under image coding. Our contribution is to expose this split to the layout optimization: we run PLAS on both byte planes, with a stronger weight on the upper byte and a weaker

weight on the lower byte, and apply the resulting permutation to all remaining attributes.

For view-dependent color, we split spherical harmonics into the DC term f_{dc} and the remaining coefficients f_{rest}. We store f_{dc} directly as an 8-bit image. We compress f_{rest} with a k-means codebook of size K that scales with the number of active primitives. In our experiments, we choose the SH codebook size as $K = S^2$. We set the codebook side length S from $\sqrt{N/8}$, clamp it to $[8, 256]$, and round it down to a multiple of 8. As a further enhancement, we PLAS-sort the SH codebook centroids to obtain a 2D centroid grid and store assignments as two 8-bit UV channels that index this grid. We quantize the SH codebook centroids with 100 levels using per-channel min-max ranges. Each centroid contains 15 non-DC SH coefficients with RGB values. We store the resulting centroid grid as a single RGB image by arranging the 15 coefficient images in a 3×5 grid of tiles.

Scales use a log mapping and 8-bit quantization, and opacities use an inverse sigmoid mapping and 8-bit quantization. We optimize quaternions jointly with scales using PRAS, which enables quaternion quantization with $q = 99$. Without PRAS, we find that $q = 255$ is needed for comparable quality. The encoder does not require camera parameters or a renderer.

3.6 Adapting the SOG-XT Format in Encoding-aware Fine-tuning

Optionally, we fine-tune with SOG-XT in the loop. At each training step, we map the current Gaussian attributes to the quantized image tensors used by SOG-XT, reconstruct a Gaussian set from these tensors, render it, and backpropagate the rendering loss through this round trip. The discrete codec structure is fixed at the start, including the active mask, PLAS order, SH assignments, and codebook layout. During training, SH assignments stay fixed while centroid values are recomputed from the current f_{rest} coefficients. Rounding is handled with a straight-through estimator [2].

We optimize the standard rendering objective and add lightweight smoothness losses on the quantized images. We also re-run PRAS during fine-tuning to preserve the compressibility benefits of covariance symmetry resolution as the parameters change. The output format is identical to SOG-XT without fine-tuning, so the decoder remains unchanged. Implementation details are provided in the appendix.

4 Evaluation

Let us demonstrate the effectiveness of keeping things simple: we will show how POPSpa improves reconstruction quality with a reduced number of primitives compared to INRIA 3DGS and competing compaction methods. Then we'll look into our individual contributions that make up SOG-XT: 2D sorted codebooks and our PRAS covariance smoothing. Finally, we show how our results compare to the previous best-in-class, HAC++ [5], over the standard datasets for benchmarking 3DGS compression.

4.1 Evaluation protocol for 3DGS compression

The original 3DGS paper [16] established a de-facto evaluation protocol through its public codebase, which is the baseline for many follow-up works. As explicitly stated in the 3DGS.zip survey [1], this protocol specifies the datasets and splits (21 scenes across *Tanks and Temples*, *Mip-NeRF 360*, *Deep Blending*, and *Synthetic NeRF*), scene-dependent training resolutions (4× downsampling for *Mip-NeRF 360* outdoor scenes, 2× for indoor), a fixed evaluation resolution (images downscaled to 1600 px on the longest side), and evaluation with peak signal-to-noise ratio (PSNR), structural similarity index measure (SSIM) [43], and learned perceptual image patch similarity (LPIPS) [46], using the VGG backbone for LPIPS and images in their $[0, 1]$ range. Full details are provided in the appendix.

Deviating from this evaluation protocol can lead to considerable differences in reported metrics. The effects of training and test resolutions are discussed in 3DGS.zip survey [1], while a discussion of the incorrect LPIPS range being used in 3DGS literature can be found in the NeRF baselines benchmark [18].

In all tables that have color-highlighted values, the best results in each category are highlighted first, second, and third.

4.2 Baselines: INRIA for Quality and HAC++ for Compression

For an uncompressed quality reference, we use INRIA 3DGS [16], which we denote INRIA-Q. The original 3DGS protocol trains for 30k iterations, which matches the reconstruction stage of our pipeline. Our compaction stage then adds 10k refinement iterations before compression. We therefore train the INRIA quality reference for 40k iterations as well, matching the total pre-compression optimization budget of our method.

As the compression baseline, we use HAC++ [5], the highest ranked method in the 3DGS.zip survey. Recomputing all published 3DGS compression methods under a single protocol is infeasible, but HAC++ is the strongest available reference point in the survey. Only the preprint HEMGS [22] achieves higher PSNR on several datasets, while HAC++ is more balanced across metrics and has public code available. We therefore recompute HAC++ under our evaluation protocol and compare to the remaining methods using their self-reported values from the survey. We run HAC++ for 44k iterations as a compute-matched baseline. This matches the total budget of the full KISS-GS pipeline, whose stages use 30k iterations for reconstruction, 10k iterations for POPSpa compaction, and 4k iterations for optional encoding-aware fine-tuning. We use the parameters provided by the authors for their high- and low-rate versions, but follow our evaluation protocol throughout, which leads to slight differences from the numbers reported in their paper. The appendix ablates HAC++ across the reported 30k setting and our 30k, 40k, and 44k recomputations under both evaluation protocols.

4.3 Evaluating Compaction with POPSpa

Figure 1 shows results for POPSpa using seven Gaussian primitive budgets. For the smallest model, we set the target to 64 k Gaussians for all datasets, except for *Synthetic NeRF*, where the lower scene complexity motivates a target of 16 k. To obtain an intended pruning ratio of approximately 80%, we first train an over-parameterized 3DGS model with *gsplat-MCMC* [45] using 320 k primitives (80 k for *Synthetic NeRF*).

For each subsequent point in Fig. 1, we double both the compacted and the corresponding initial budget. The initial (uncompacted) models are capped by a dataset-specific maximum of 3000 k Gaussians for *Tanks and Temples* and *Mip-NeRF 360*, 2048 k for *Deep Blending*, and 640 k for *Synthetic NeRF*. The right-most point corresponds to the maximal model size without pruning and only applies 10 k fine-tuning iterations.

Table 1: Comparison of 3DGS compaction methods across four standard benchmark datasets. Ranks represent average rankings across all available datasets, with model size and overall quality weighted equally: size contributes half the weight, and PSNR, SSIM, and LPIPS each contribute one-sixth. In the top half, post-training pruning is applied to a 30k-step vanilla 3DGS checkpoint, removing 90% of Gaussians. The bottom half compares against compaction-aware training methods, where each method is trained for 10k steps beyond its original implementation, resulting in 40k steps for all methods except Mini-Splatting 2 which was originally trained for only 18k steps. Our method is post-training only, applied to a checkpoint trained with gsplat's MCMC implementation.

	Method	Rank	Tanks and Temples				Mip-NeRF 360				Deep Blending				Synthetic NeRF			
			PSNR↑	SSIM↑	LPIPS↓	k Gauss	PSNR↑	SSIM↑	LPIPS↓	k Gauss	PSNR↑	SSIM↑	LPIPS↓	k Gauss	PSNR↑	SSIM↑	LPIPS↓	k Gauss
post training	Inria [16] SIGGRAPH '23 (30k)		23.77	.847	.170	1,563	27.28	.807	.222	2,723	29.69	.905	.242	2,463	33.46	.970	.030	259
	+ Ours (+10k)	1.1	23.37	.819	.231	156	26.78	.781	.289	272	29.51	.903	.265	246	32.46	.964	.041	25
	+ Speedy-Splat [11] CVPR '25 (+10k)	1.5	23.52	.813	.245	156	26.87	.772	.305	272	29.45	.902	.267	246	32.40	.962	.045	25
	+ PUP 3D-GS [12] CVPR '25 (+10k)	1.8	22.62	.798	.251	156	25.58	.767	.300	272	29.26	.903	.264	246	31.49	.958	.049	25
compaction aware	gsplat-MCMC [45] (30k)		24.54	.869	.150	1,280	27.95	.829	.196	2,560	30.08	.907	.244	1,280	33.82	.972	.028	320
	+ Ours (+10k)	1.7	24.47	.856	.179	256	27.79	.811	.239	512	29.94	.904	.262	256	33.41	.969	.032	64
	gsplat-MCMC [45] (40k)	2.5	23.90	.841	.205	256	27.52	.809	.243	512	29.43	.899	.270	256	32.73	.967	.036	64
	GaussianSpa [47] CVPR '25 (40k)	2.8	23.63	.849	.171	349	27.40	.817	.227	561	30.09	.913	.245	457	33.34	.968	.029	212
	Mini-Splatting2 [8] (28k)	3.3	23.57	.843	.178	357	27.33	.815	.223	619	30.05	.911	.241	658	32.69	.964	.037	54
	Taming 3DGS [25] SIGGRAPH Asia '24 (40k)	3.9	24.10	.836	.203	318	27.33	.795	.258	684	29.77	.903	.273	294				

Table 1 presents results for both post-training pruning and compaction-aware training. In the post-training setting, our method consistently outperforms all competing approaches across all benchmarks.

The compaction-aware comparison is more nuanced: since most methods cannot easily predetermine the final scene size or quality, the compared samples sit at slightly different positions on each method's quality-size trade-off curve. Despite this, our method achieves the best average rank of 1.7 while operating at significantly fewer Gaussians than competing methods across most datasets. The exception is *Synthetic NeRF*, where model sizes are more comparable. In *Deep Blending*, we achieve competitive quality using only 56% of the primitives of the best performing method. Full quality-size trade-off curves for all datasets are

provided in the appendix. Across these curves, our method consistently achieves higher quality than competing methods at a given size.

4.4 Evaluating Compression with SOG-XT

As shown in Fig. 1 for *Mip-NeRF 360* and in the appendix for other datasets, our encoding method SOG-XT is able to compress the test scenes to single-digit Megabyte sizes. With a single parameter choice, as presented in Sec. 3.3, our method works well across a large range of primitive counts, from several thousand to millions. The same encoding method works on the INRIA 40k baseline, highlighting the modularity of the approach. Two popular post-training encoding methods in `.spz` and `.sog` are benchmarked against our method using their official implementations (`splat-transform` v1.6.0 [35] for `.sog`, the official `spz` PyPI package v0.0.6 [31] for `.spz`). With `.sog`, the files are even larger, and this format appears sensitive to the number of primitives, producing a jagged rate-distortion curve.

Table 2: Ablation of contributions to the SOG-XT format on the Bicycle scene with 256 k primitives. The first row contains the absolute values after encoding. The following rows disable individual features of SOG-XT, ordered by size penalty. Most removals increase the final size considerably at small quality gains. Using image encoding, our novel codebook sorting and its UV label packing bring clean size wins without incurring any quality loss. Find additional rows in the appendix.

		Effect of Leaving Feature Out			
	Size [MB]	Δ Size [Bytes] ↓	Δ PSNR ↑	Δ SSIM ↑	Δ LPIPS ↓
SOG-XT (Full Pipeline)	3.878	(3,877,944)	(24.2281)	(0.68176)	(0.35404)
w/o SH AC codebooks	7.313	+3,434,816	+0.2320	+0.00790	-0.00579
w/o WebP image compression	6.908	+3,030,148	+0.0000	+0.00000	+0.00000
w/o quaternion u8 quantization	6.427	+2,548,718	+0.0317	+0.00147	-0.00067
w/o primitive PLAS sorting	4.493	+614,851	+0.0059	+0.00022	-0.00036
w/o centroid PLAS sorting	3.920	+42,416	+0.0000	+0.00000	+0.00000
w/o label UV packing	3.890	+12,142	+0.0000	+0.00000	+0.00000
w/o PRAS	3.866	-11,810	-0.1104	-0.00538	+0.00273
w/o means signed-log remapping	3.359	-519,272	-7.5508	-0.37864	+0.22151

In Table 2, we show how the individual components of SOG-XT contribute to its high compression ratio. In these leave-one-out experiments, we can see that most components provide size wins against small quality losses, like the quantization operations. Our 2D codebooks (centroids PLAS sorting and label UV packing) bring free size wins without any quality loss. Our PRAS method incurs a small size hit, but provides large quality gains. The highest quality loss occurs from building codebooks for SH AC, but otherwise, the files would nearly triple in size. Without the log-space contraction of means, file sizes would shrink. However, with 16-bit quantization for the means, reconstruction quality would degrade severely.

4.5 Adaptation and Comparison with State of the Art

In Figure 1, we show that encoding-aware fine-tuning clearly improves the rate-distortion curve over post-training compression, achieving another 2.2× reduction at INRIA quality for the *Mip-NeRF 360* dataset. While SOG-XT by itself can already achieve great results without feedback from the renderer, running some adaptation can be worth it if a GPU and training views are available at encoding time. A sensitivity analysis in the appendix shows that the default fine-tuning setting is not a fragile optimum; the SH codebook size is the clearest rate-control knob, while additional fine-tuning steps and stronger TV regularization do not improve the trade-off.

Table 3: File-size reduction factors over INRIA 3DGS after 40k iterations. Each entry is the interpolated point where a method matches INRIA-Q for the corresponding metric. KISS-GS achieves the highest reductions on *Tanks and Temples* and *Mip-NeRF 360*, and on two of the three *Synthetic NeRF* metrics.

Method	Tanks and Temples			Mip-NeRF 360			Deep Blending			Synthetic NeRF		
	PSNR	SSIM	LPIPS	PSNR	SSIM	LPIPS	PSNR	SSIM	LPIPS	PSNR	SSIM	LPIPS
CodecGS [21] ICCV '25 †	53*	-	-	72*	72*	-	76*	76*	-	n/a	n/a	n/a
ContextGS [42] NeurIPS '24 †	42*	42*	-	55*	55*	-	187*	187*	-	n/a	n/a	n/a
gsplat-1M [13] arXiv '24 †	60*	44	26	52	57	28	n/a	n/a	n/a	n/a	n/a	n/a
HAC [3] ECCV '24 †	49*	48	-	46*	46*	-	150*	129	-	47	-	-
HEMGS [22] arXiv '24 †	69*	69*	-	59*	59*	-	229*	229*	-	57*	-	-
HAC++ [5] TPAMI '25	77*	68	-	70	36	-	156	156	-	-	-	-
Ours (SOG-XT)	227	104	60	104	73	40	-	-	-	10	-	-
Ours+FT	319	131	72	228	109	64	85	98	-	21	14	14

* smallest available point already reaches INRIA-Q, so the value is a lower bound.
\- INRIA-Q not reached within available points.
† self-reported result from 3DGS.zip, not recomputed under our protocol.

In Table 3, each entry is the interpolated file-size reduction at which a method matches the INRIA 3DGS 40k quality reference. We use recomputed HAC++ as the main baseline, since it is the strongest reproducible published method in the 3DGS.zip survey. HEMGS reports higher reductions in some cases, but it is a preprint without public code, so we include it only as self-reported context. On *Tanks and Temples* and *Mip-NeRF 360*, KISS-GS reaches INRIA-Q at substantially higher reduction factors than HAC++ across all reported metrics. On *Deep Blending*, the gap is already visible before SOG-XT is applied: compressed HAC++ exceeds uncompressed gsplat-MCMC and INRIA 3DGS on *Playroom* (30.63 vs. 30.42 and 30.09) and *Dr. Johnson* (29.60 vs. 29.48 and 29.43). This points to HAC++'s anchor-based learned representation and regularization rather than to SOG-XT quantization or POPSpa pruning. To keep the comparison compact, we include literature methods only if they provide at least four finite entries in this table.

Notably, many competing methods from the literature do not reach INRIA quality in the perceptual LPIPS metric at any of their available operating points. This may point to overfitting the PSNR metric for the training views and losing generalization capabilities in rendering novel views or novel resolutions (for the *Mip-NeRF 360* dataset).

Runtime measurements in the appendix show that post-hoc SOG-XT encoding is inexpensive compared with reconstruction and compaction. Encoding-aware fine-tuning is more expensive, but it can be omitted. A reference Python decoder reconstructs models with up to 1024k Gaussians in well under a second on CPU, supporting our claim that deployment remains simple.

5 Have we kept it simple?

The title of this paper, together with the Dijkstra quote in the introduction, makes a deliberately cheeky promise. We set out to keep 3DGS compression simple. With respect to simplicity, we fell short in one place: the encoder is a sophisticated modular pipeline with modules such as POPSpa and novel components such as PRAS. This deliberate complexity lets KISS-GS separate reconstruction, compaction, encoding, and **encoding-aware fine-tuning**, close the **attribution gap**, and make the source of each gain measurable.

The payoff is substantial: KISS-GS reaches **single-digit megabytes** at vanilla 3DGS reconstruction quality. With 319× size reduction on *Tanks and Temples* in PSNR and 228× on *Mip-NeRF 360*, it sets a new benchmark for 3DGS compression on these two real-world datasets. A pipeline built around a **simple image-based encoding format** can exceed the strongest published baselines on the main real-world benchmarks while keeping deployment simple. SOG-XT reconstructs standard 3D Gaussian Splatting attributes through standard image decoding and small, deterministic rescaling and reshaping operations. This keeps implementation effort low and makes decoders easy to port across languages and platforms. Optional **encoding-aware fine-tuning** improves rate-distortion without changing this decoder. *Deep Blending* remains the main boundary, where anchor-based learned representations better absorb capture inconsistencies than the standard-splat reconstructions used as input to KISS-GS. Its modular design makes future advances in reconstruction, compaction, and vector quantization directly useful for compression. Still, KISS-GS shows that state-of-the-art 3DGS compression does not require a complex deployment stack: the strongest results can come from a sophisticated encoder while keeping the decoder deliberately *simple*.

Acknowledgements

This work has partly been funded by the European Union's Horizon Europe research and innovation program (Luminous, grant no. 101135724), the German Research Foundation (3DIL, grant no. 502864329), and the Fraunhofer society (Self Structured Gaussians).

References

1. Bagdasarian, M.T., Knoll, P., Li, Y., Barthel, F., Hilsmann, A., Eisert, P., Morgenstern, W.: 3dgs.zip: A survey on 3d gaussian splatting compression methods. Computer Graphics Forum **44**(2), e70078 (2025). https://doi.org/https://doi.org/10.1111/cgf.70078, https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.70078, https://w-m.github.io/3dgs-compression-survey/
2. Bengio, Y., Léonard, N., Courville, A.: Estimating or propagating gradients through stochastic neurons for conditional computation (2013)
3. Chen, Y., Wu, Q., Cai, J., Harandi, M., Lin, W.: Hac: Hash-grid assisted context for 3d gaussian splatting compression. In: ECCV. pp. 422–438 (2024), https://yihangchen-ee.github.io/project_hac/
4. Chen, Y., Wu, Q., Li, M., Lin, W., Harandi, M., Cai, J.: Fast feedforward 3d gaussian splatting compression. In: ICLR. pp. 28308–28321 (2025), https://arxiv.org/pdf/2410.08017
5. Chen, Y., Wu, Q., Lin, W., Harandi, M., Cai, J.: Hac++: Towards 100x compression of 3d gaussian splatting. IEEE TPAMI (2025), https://arxiv.org/pdf/2501.12255
6. Dijkstra, E.W.: On the nature of computing science. EWD896 (manuscript) (Jan 1984), https://www.cs.utexas.edu/users/EWD/ewd08xx/EWD896.PDF, eWD896
7. Fang, G., Wang, B.: Mini-splatting: Representing scenes with a constrained number of gaussians. In: ECCV. pp. 165–181. Springer (2024)
8. Fang, G., Wang, B.: Mini-splatting2: Building 360 scenes within minutes via aggressive gaussian densification (2024), https://arxiv.org/abs/2411.12788
9. Girish, S., Gupta, K., Shrivastava, A.: Eagles: Efficient accelerated 3d gaussians with lightweight encodings. In: ECCV. pp. 54–71. Springer (2024)
10. Hahlbohm, F., Franke, L., Eisemann, M., Magnor, M.: Faster-gs: Analyzing and improving gaussian splatting optimization (2026), https://arxiv.org/abs/2602.09999
11. Hanson, A., Tu, A., Lin, G., Singla, V., Zwicker, M., Goldstein, T.: Speedy-splat: Fast 3d gaussian splatting with sparse pixels and sparse primitives. In: CVPR. pp. 21537–21546 (June 2025), https://speedysplat.github.io/
12. Hanson, A., Tu, A., Singla, V., Jayawardhana, M., Zwicker, M., Goldstein, T.: Pup 3d-gs: Principled uncertainty pruning for 3d gaussian splatting. In: CVPR. pp. 5949–5958 (June 2025), https://pup3dgs.github.io/
13. Hu, J., Li, R., Ye, V., Kanazawa, A.: gsplat compression (2024), https://github.com/w-m/3dgs-compression-survey/pull/7, accessed: March 5, 2026
14. Hyung, J., Hong, S., Hwang, S., Lee, J., Choo, J., Kim, J.H.: Effective rank analysis and regularization for enhanced 3d gaussian splatting. In: Globerson, A., Mackey, L., Belgrave, D., Fan, A., Paquet, U., Tomczak, J., Zhang, C. (eds.) NeurIPS. pp. 110412–110435. Curran Associates, Inc. (2024). https://doi.org/10.52202/079017-3505, https://proceedings.neurips.cc/paper_files/paper/2024/file/c73052b864497e078db340c9f6fa4ab5-Paper-Conference.pdf
15. Kellogg, M.: 3d Gaussian Splatting with Three.js (2023), https://github.com/mkkellogg/gaussian-splats-3d, self-contained renderer for 3D Gaussian Splatting in web environments. Accessed: March 5, 2026.
16. Kerbl, B., Kopanas, G., Leimkühler, T., Drettakis, G.: 3d gaussian splatting for real-time radiance field rendering. ACM TOG **42**(4) (July 2023), https://repo-sam.inria.fr/fungraph/3d-gaussian-splatting/

17. Kheradmand, S., Rebain, D., Sharma, G., Sun, W., Tseng, Y.C., Isack, H., Kar, A., Tagliasacchi, A., Yi, K.M.: 3d gaussian splatting as markov chain monte carlo. NeurIPS **37**, 80965–80986 (2024)
18. Kulhanek, J., Sattler, T.: NerfBaselines: Consistent and reproducible evaluation of novel view synthesis methods. In: NeurIPS (2025)
19. Lee, J.C., Rho, D., Sun, X., Ko, J.H., Park, E.: Compact 3d gaussian representation for radiance field (2024), https://maincold2.github.io/c3dgs/
20. Lee, S., Kim, Y.G., Sasse, S., Borges, T.M., Sanchez, Y., Ryu, E.S., Schierl, T., Hellge, C.: Gaussianpop: Principled simplification framework for compact 3d gaussian splatting via error quantification (2026), https://arxiv.org/abs/2602.06830
21. Lee, S., Shu, F., Sanchez, Y., Schierl, T., Hellge, C.: Compression of 3d gaussian splatting with optimized feature planes and standard video codecs. In: ICCV. pp. 25496–25505 (2025)
22. Liu, L., Chen, Z., Xu, D.: Hemgs: A hybrid entropy model for 3d gaussian splatting data compression (2024), https://arxiv.org/pdf/2411.18473
23. Liu, X., Wu, X., Zhang, P., Wang, S., Li, Z., Kwong, S.: Compgs: Efficient 3d scene representation via compressed gaussian splatting. In: ACM MM. pp. 2936–2944 (2024), https://www.arxiv.org/abs/2404.09458
24. 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. pp. 20654–20664 (2024), https://city-super.github.io/scaffold-gs/
25. Mallick, S.S., Goel, R., Kerbl, B., Carrasco, F.V., Steinberger, M., Torre, F.D.L.: Taming 3dgs: High-quality radiance fields with limited resources. In: SIGGRAPH Asia Conference Papers (2024). https://doi.org/10.1145/3680528.3687694, https://humansensinglab.github.io/taming-3dgs/
26. Mildenhall, B., Srinivasan, P.P., Tancik, M., Barron, J.T., Ramamoorthi, R., Ng, R.: Nerf: Representing scenes as neural radiance fields for view synthesis. In: ECCV (2020)
27. Morgenstern, W., Barthel, F., Hilsmann, A., Eisert, P.: Compact 3d scene representation via self-organizing gaussian grids (2024), https://fraunhoferhhi.github.io/Self-Organizing-Gaussians/
28. Müller, T., Evans, A., Schied, C., Keller, A.: Instant neural graphics primitives with a multiresolution hash encoding. ACM TOG **41**(4) (Jul 2022). https://doi.org/10.1145/3528223.3530127
29. Navaneet, K., Meibodi, K.P., Koohpayegani, S.A., Pirsiavash, H.: Compact3d: Smaller and faster gaussian splatting with vector quantization (2023), https://ucdvision.github.io/compact3d/
30. Navaneet, K., Pourahmadi Meibodi, K., Abbasi Koohpayegani, S., Pirsiavash, H.: Compgs: Smaller and faster gaussian splatting with vector quantization. In: ECCV. pp. 330–349. Springer (2024)
31. Niantic Labs: SPZ: Open-sourcing the JPG for 3D Gaussian Splats (2024), https://github.com/nianticlabs/spz, open-source file format for compressed 3D Gaussian Splatting. Accessed: March 5, 2026.
32. Niedermayr, S., Stumpfegger, J., Westermann, R.: Compressed 3d gaussian splatting for accelerated novel view synthesis. In: CVPR. pp. 10349–10358 (2024), https://keksboter.github.io/c3dgs/
33. Papantonakis, P., Kopanas, G., Kerbl, B., Lanvin, A., Drettakis, G.: Reducing the memory footprint of 3d gaussian splatting. Proceedings of the ACM on Computer Graphics and Interactive Techniques **7**(1) (May 2024), https://repo-sam.inria.fr/fungraph/reduced_3dgs/

34. PlayCanvas Team: PlayCanvas: Web-First 3d Engine (2024), https://playcanvas.com/, open-source JavaScript run-time with native 3D Gaussian Splatting support. Accessed: March 5, 2026.
35. PlayCanvas Team: The SOG Format - PlayCanvas Developer Site (2025), https://developer.playcanvas.com/user-manual/gaussian-splatting/formats/sog/, technical specification for Spatially Ordered Gaussians (SOG). Accessed: March 5, 2026.
36. Ren, K., Jiang, L., Lu, T., Yu, M., Xu, L., Ni, Z., Dai, B.: Octree-gs: Towards consistent real-time rendering with lod-structured 3d gaussians. PAMI (2025). https://doi.org/10.1109/TPAMI.2025.3568201
37. Rich, B.R.: Clarence leonard (kelly) johnson 1910-1990: a biographical memoir. Biographical Memoirs **67**, 221–241 (1995)
38. Roy, O., Vetterli, M.: The effective rank: A measure of effective dimensionality. In: European Signal Processing Conference. pp. 606–610 (2007)
39. SparkJS contributors: Sparkjs. https://sparkjs.dev/ (2025), https://sparkjs.dev/, accessed: March 5, 2026
40. Studio, L.: A high-performance c++ and cuda implementation of 3d gaussian splatting (2025), https://github.com/MrNeRF/LichtFeld-Studio, accessed: March 5, 2026
41. Wang, H., Zhu, H., He, T., Feng, R., Deng, J., Bian, J., Chen, Z.: End-to-end rate-distortion optimized 3d gaussian representation. In: ECCV. pp. 76–92 (2024), https://rdogaussian.github.io/
42. Wang, Y., Li, Z., Guo, L., Yang, W., Kot, A.C., Wen, B.: Contextgs: Compact 3d gaussian splatting with anchor level context model. In: NeurIPS. pp. 51532–51551 (2024), https://github.com/wyf0912/ContextGS
43. Wang, Z., Bovik, A.C., Sheikh, H.R., Simoncelli, E.P.: Image quality assessment: from error visibility to structural similarity. TIP (2004)
44. Wu, M., Tuytelaars, T.: Implicit gaussian splatting with efficient multi-level triplane representation (2024), https://www.arxiv.org/abs/2408.10041
45. Ye, V., Li, R., Kerr, J., Turkulainen, M., Yi, B., Pan, Z., Seiskari, O., Ye, J., Hu, J., Tancik, M., Kanazawa, A.: gsplat: An open-source library for gaussian splatting. Journal of Machine Learning Research **26**(34), 1–17 (2025)
46. Zhang, R., Isola, P., Efros, A.A., Shechtman, E., Wang, O.: The unreasonable effectiveness of deep features as a perceptual metric. In: CVPR (2018)
47. Zhang, Y., Jia, W., Niu, W., Yin, M.: Gaussianspa: An" optimizing-sparsifying" simplification framework for compact and high-quality 3d gaussian splatting. In: CVPR. pp. 26673–26682 (2025)

6 Appendix to KISS-GS: 3D Gaussian Splatting Compression Kept Simple

In the main paper, we have shown how to build a modular pipeline for 3DGS compression whose output is a compact SOG-XT container that remains simple to decode while still reaching record-setting compression levels. With the additional material in this appendix, we strengthen the paper's claims with more detailed comparisons across datasets and metrics, additional ablations, a visual explanation of the covariance symmetry, and qualitative comparisons of rendered views.

6.1 Runtime Performance

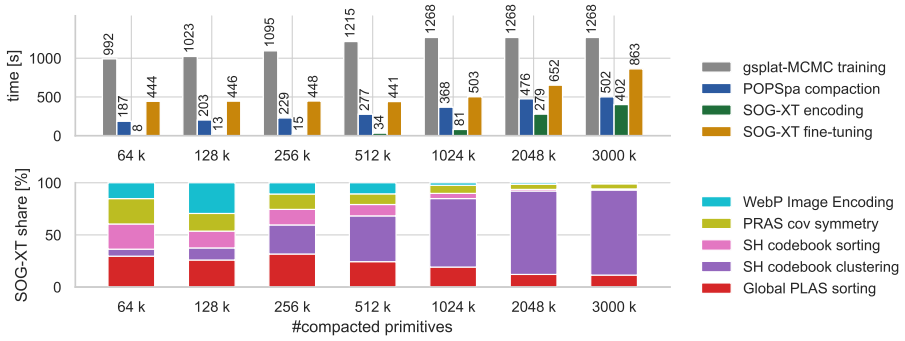

Fig. 4: Runtime breakdown on the Bicycle scene. The top plot shows wall-clock time per KISS-GS stage and the bottom plot breaks down post-hoc SOG-XT encoding. Bicycle belongs to *Mip-NeRF 360*, and 512k is the smallest Bicycle point exceeding INRIA-40k PSNR. At that point, reconstruction takes 1215 s, POPSpa compaction takes 277 s, post-hoc SOG-XT encoding takes 34 s, and optional encoding-aware fine-tuning adds 441 s. For context, Fig. 1 reports stage-wise size reductions of 15.7x, 6.6x, and 2.2x on *Mip-NeRF 360* at INRIA-Q. Encoding costs are balanced between components for small scenes, while SH codebook clustering dominates at high primitive counts.

Figure 4 shows wall time for **encoding** with our KISS-GS pipeline. Timings were measured using 16 AMD EPYC 7200 CPU cores and one Nvidia A100 GPU. When comparing the wall time of different stages of the KISS-GS pipeline in the top view, we show that encoding with the SOG-XT format is inexpensive compared with reconstruction and compaction. The optional encoding-aware fine-tuning is more costly because the codec is evaluated in the training loop. In deployment, choosing between post-hoc and fine-tuned encoding is a trade-off between encoding time and bitrate, with the latter being more expensive but yielding better compression. Both variants decode identically, so the adaptation

improves bitrate without increasing deployment complexity. Within post-hoc SOG-XT encoding (bottom view), the individual components have similar cost for small models. For larger models, SH codebook clustering starts to dominate and is the clearest target for future encoder optimization.

For **decoding** our format, we benchmark a reference Python decoder that reconstructs SOG-XT tensors using only standard WebP image decoding plus lightweight dequantization and table lookups. On a Threadripper PRO 5955WX CPU, our end-to-end decoder takes 108 ms for 256k Gaussians, 192 ms for 512k, and 347 ms for 1024k. These timings include file reads but exclude PLY writing. Peak memory usage is 0.37 GB, 0.67 GB, and 1.24 GB, respectively. The timing values are medians over the seven kept runs after one warmup run, with the two slowest of nine recorded runs discarded. These measurements are intended as a conservative baseline because image decoding could be parallelized and handed off to hardware decoders, while per-field processing could be done more efficiently on the GPU. They demonstrate that this unoptimized code can decode large 3DGS models in well under a second on commodity hardware, and that the memory footprint is small enough to fit comfortably in GPU memory for real-time rendering.

6.2 Encoding-aware Fine-tuning Details

In all experiments with encoding-aware fine-tuning, we run 4000 steps with a batch size of 8. Our implementation is based on PyTorch and uses gsplat for rendering. We set a global step offset of 40 k to continue the learning-rate schedule after the reconstruction and compaction stages, which strongly reduces updates to the 3D positions. The discrete structure of the codec is refreshed once at the start, including padding and the *active mask*, PLAS ordering, the SH codebook assignments with $K \approx N/8$ clamped to $[64, 65{,}536]$, and PLAS sorting of the codebook. We align the grid side length to a multiple of 8 and use a PLAS minimum block size of 8. Quantization follows SOG-XT, including signed log for means with a 16-bit proxy split into byte planes, $q = 99$ for quaternions, and 100-level quantization for SH codebook centroids. Quantization ranges are computed once at initialization and kept fixed during training, except after PRAS updates. We optimize a weighted combination of ℓ_1 and SSIM with $\lambda_{\text{SSIM}} = 0.2$. We add masked ℓ_1 total variation on opacities, scales, both mean byte planes, and f_{dc}, and we use a masked quaternion TV loss. We set the TV weights to $1\text{e}{-2}$ for quaternions and $1\text{e}{-4}$ for all other terms. These auxiliary losses are linearly warmed up from 0.35 to 0.60 of the run, corresponding to steps 1400 to 2400. We run PRAS once at initialization and again at 0.6 and 0.8 of the run, with PLAS as the driver.

6.3 Encoding-aware Fine-tuning Sensitivity

Figure 5 varies one parameter at a time around the default setting used in the paper. The sweep shows a stable rate-distortion region rather than a fragile optimum on the Bicycle scene. Disabling TV regularization or increasing quaternion

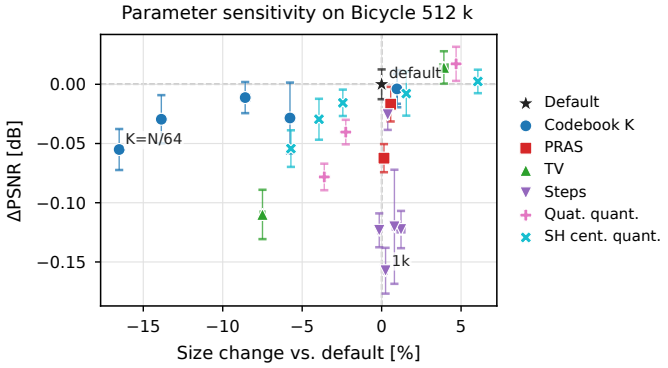

Fig. 5: Parameter sensitivity of encoding-aware fine-tuning on the Bicycle scene with 512k Gaussians. We vary one parameter at a time around the default setting used in the paper: SH codebook size ($K \in N/\{2, 4, 12, 16, 32, 64\}$ compared with the default $N/8$), PRAS timing, TV regularization (off, stronger), fine-tuning steps from 0.5k to 12k, quaternion quantization range 47–255, and SH-centroid quantization range 31–255. Points are means over seven runs, with error bars showing one standard deviation.

precision gives only small PSNR gains while increasing file size. Stronger TV regularization, very short fine-tuning, and additional fine-tuning steps do not improve the trade-off. Changing the PRAS schedule stays close to or below the default, so we keep the fixed schedule described above. The SH codebook size is the clearest optional rate-control knob, but its best setting is scene dependent. We therefore use $K = N/8$ as a conservative default and use POPSpa primitive count as the primary rate-distortion control.

6.4 Compaction Ablation

Table 4: Ablation on the components of our compaction method, POPSpa. Adding the components individually, the results improve for two out of three of the real-world datasets and stagnate for the synthetic data.

Method	Tanks and Temples				Mip-NeRF 360				Deep Blending				Synthetic NeRF			
	PSNR↑	SSIM↑	LPIPS↓	k Gauss	PSNR↑	SSIM↑	LPIPS↓	k Gauss	PSNR↑	SSIM↑	LPIPS↓	k Gauss	PSNR↑	SSIM↑	LPIPS↓	k Gauss
gsplat-MCMC [45] arXiv '24 (30k)	24.74	.875	.134	2,560	27.95	.829	.196	2,560	29.97	.907	.239	2,048	33.98	.972	.027	640
+ GaussianPOP [20] (10k)	24.68	.864	.162	512	27.69	.802	.249	512	29.92	.906	.249	512	33.83	.970	.029	128
+ Sparsification	24.77	.870	.149	512	27.78	.809	.240	512	29.85	.905	.248	512	33.83	.971	.029	128
+ Effective Rank Regularization	24.77	.871	.148	512	27.79	.811	.239	512	29.84	.906	.250	512	33.73	.970	.029	128

Table 4 presents an ablation study of our compaction method. Starting with a scene trained with gsplat-MCMC for 30k iterations, we apply two rounds of prune-refine using GaussianPOP's score. Adding Sparsification to the first refinement stage generally improves the results, with the exception of the *Deep Blending* dataset, which shows a slightly lower PSNR. After adding the Effective Rank

Regularization, our compaction still achieves competitive results, indicating that preparing the compacted data for compression introduces no quality loss. Although this regularization is not intended to improve quality, it effectively reduces the spikiness of the Gaussians without degrading performance.

6.5 Covariance Symmetries

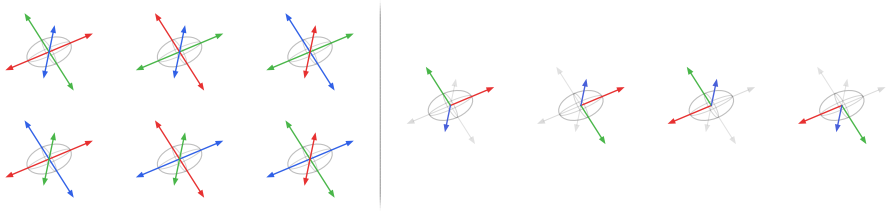

Fig. 6: Covariance symmetries
Left: Scale permutations (6). Each permutation is indicated by a unique assignment of the local coordinate axes (red, green, and blue) to the principal axes of the ellipsoid.
Right: Valid eigenvector orientations (4) for a fixed scale permutation.

Figure 6 illustrates the discrete symmetries inherent in the decomposition of the 3D covariance matrix $\Sigma = RSS^TR^T$, where $S = \mathrm{diag}(s_1, s_2, s_3)$ are the scales and $R \in SO(3)$ are the rotation matrices. As discussed in the main paper in Sec. 3.3, these symmetries give rise to 48 equivalent parameterizations of the same Gaussian covariance. As shown on the left, there are $3! = 6$ possible scale permutations, each defined by a unique assignment of the local coordinate axes to the ellipsoid's principal axes. For any fixed permutation, we further identify four valid eigenvector orientations on the right. While there are $2^3 = 8$ possible sign configurations for the eigenvectors, only four satisfy the $SO(3)$ constraint $\det(R) = 1$. Together with the quaternion double cover, this yields the full set of 48 equivalent choices used by PRAS. PRAS exploits this freedom by evaluating the candidates within each equivalence class and selecting the one that improves the compressibility of the quaternion and scale grids without changing the represented covariance.

6.6 HAC++ Results

Table 5 shows the original HAC++ numbers alongside our own runs on *Mip-NeRF 360*. We first list the reported 30k results from the HAC++ paper. This 30k setting is also the default training budget for 3DGS, which makes it the natural reference point. We then report our recomputed 30k, 40k, and 44k runs with training resolution equal to evaluation resolution. Finally, we report the same runs under our full evaluation protocol from Sec. 6.7, including 1600 px test resolution. We

Table 5: Scores from the original HAC++ paper [5] and our own runs on *Mip-NeRF 360*. We list the reported 30k numbers, then our recomputed results with training resolution equal to evaluation resolution, and finally our recomputed results under the full evaluation protocol with 1600 px test resolution. Scene size decreases up to 40k iterations and stabilizes thereafter, indicating convergence.

Method	Mip-NeRF 360, train: 2x/4x, test: matched train resolution.							
	Low-rate				High-rate			
	PSNR↑	SSIM↑	LPIPS↓	Size	PSNR↑	SSIM↑	LPIPS↓	Size
HAC++ (30k, reported)	27.60	0.803	0.253	8.34	27.82	0.811	0.231	18.48
HAC++ (30k, recomputed)	27.55	0.805	0.247	8.84	27.80	0.814	0.225	19.80
HAC++ (40k, recomputed)	27.33	0.802	0.250	8.41	27.80	0.813	0.225	19.68
HAC++ (44k, recomputed)	27.20	0.800	0.251	8.42	27.79	0.813	0.226	19.67
	Mip-NeRF 360, train: 2x/4x, test: 1600 px.							
HAC++ (30k, recomputed)	27.43	0.800	0.256	8.82	27.57	0.806	0.235	19.76
HAC++ (40k, recomputed)	27.21	0.796	0.260	8.38	27.62	0.806	0.236	19.62
HAC++ (44k, recomputed)	27.09	0.795	0.261	8.39	27.61	0.806	0.236	19.62

include 44k because our own pipeline adds 10k steps in the compaction stage and 4k steps in the fine-tuning stage, so this gives HAC++ the same overall compute budget in this comparison. Our 30k recomputation in the matched train and eval setting is close to the reported values. Moving to the full evaluation protocol lowers the quality metrics slightly, while file size remains nearly unchanged. Extending the compute budget to 40k and 44k iterations demonstrates size convergence: scene size reduces slightly at 40k in the low-rate regime, but additional steps do not improve rendering quality. This is consistent with the HAC++ loss:

$$\mathcal{L} = \mathcal{L}_{\text{Scaffold}} + \lambda \frac{1}{N(D_a + 6 + 3K)} \left(\mathcal{L}_{\text{entropy}} + \mathcal{L}_{\text{hash}}\right) \tag{3}$$

where λ controls the rate-distortion trade-off. At high λ (low-rate), extended training forces further compression without recovering quality. At low λ (high-rate), this pressure is relaxed and quality remains stable regardless of iteration count.

6.7 Evaluation Protocol

For our evaluation experiments, we follow the evaluation protocol established in the codebase of the original 3DGS paper [16]. The 3DGS.zip survey [1] states the scene selection, the training and evaluation resolutions, and the training and test split. Here, we add details on LPIPS and the background color.

– 21 scenes from four datasets (two real-world scenes in *Tanks and Temples*, a mix of nine indoor and outdoor scenes in *Mip-NeRF 360*, two in *Deep Blend-*

ing and eight synthetic scenes in the *Synthetic NeRF* dataset) show a range of tiny synthetic scenes to medium-large real-world scenes.
- Training resolution is 4x downsampled for *Mip-NeRF 360* outdoor scenes, 2x downsampled for *Mip-NeRF 360* indoor scenes, and otherwise 1x the provided images.
- Evaluation uses images downsampled to 1600 px on the longest side rather than the training resolution.
- Image downsampling is handled with PIL bicubic resampling.
- The background color is white for *Synthetic NeRF*, and black otherwise.
- Metrics are PSNR, SSIM, and LPIPS, where for LPIPS the VGG backbone is used, and the images are not normalized to $[-1, 1]$ but are instead kept in their $[0, 1]$ range [18].
- Metrics are evaluated on a hold-out validation set, which comprises every 8th image of the real-world datasets and the designated test split of *Synthetic NeRF*.

Because metrics are computed on held-out images, this protocol evaluates *novel view generalization* rather than fidelity on the training set alone. This distinction is particularly important in the context of compression, where a compact representation should preserve not only the appearance of the observed views but also the scene structure required to render unseen viewpoints accurately. The evaluation thus helps ensure that bitrate reductions are not achieved by compressing away the generalization capacity.

In practice, the evaluation-resolution mismatch affects only *Mip-NeRF 360*, as the remaining datasets have image resolutions below 1600 pixels on the longest side. *Mip-NeRF 360* is therefore the most informative benchmark in this protocol: it covers a diverse set of real-world indoor and outdoor scenes, contains multiple scenes rather than isolated examples, and jointly tests novel view generalization and robustness to differing training and evaluation resolutions.

6.8 Compression Ablation

Additional ablations of our compression format SOG-XT are reported in Tab. 6. Across nearly all attributes, uniform min–max quantization is highly effective: values are stored using a small fixed number of discrete levels, while the original minimum and maximum are retained as metadata for dequantization. Depending on the attribute, we use 100, 256, or 65,536 quantization levels, denoted in the table as `q100_u8`, `u8`, and `u16`, respectively. This reduces size substantially while causing only minor quality loss.

For quaternions, SOG-XT uses particularly aggressive min–max quantization with only 100 levels, referred to as `q100_u8` in the table. This is enabled by PRAS: when PRAS is removed, the same quantization becomes too coarse and causes a pronounced quality drop. Recovering the quality requires relaxing the quantization to a denser 256-level representation, denoted as `u8`, as shown in the row "w/o PRAS, quaternion full u8 quant".

PLAS sorting requires the primitives to fit into square 2D grids. In overcomplete representations, this can be handled by pruning surplus primitives, for

Table 6: Ablation of contributions to the SOG-XT format on the Bicycle scene with 256 k primitives. The first row contains the absolute values after encoding. The following rows disable individual features of SOG-XT, ordered by size penalty. Most removals increase the final size considerably at small quality gains. Using image encoding, our novel codebook sorting and its UV label packing bring clean size wins without incurring any quality loss. Compared with the main paper version, additional rows included only in the appendix are marked by a colored bar in the first column.

		Effect of Leaving Feature Out			
	Size [MB]	Δ Size [Bytes] ↓	Δ PSNR ↑	Δ SSIM ↑	Δ LPIPS ↓
SOG-XT (Full Pipeline)	3.878	(3,877,944)	(24.2281)	(0.68176)	(0.35404)
w/o centroid q100_u8 quantization	7.920	+4,042,523	+0.0097	+0.00016	-0.00032
w/o SH AC codebooks	7.313	+3,434,816	+0.2320	+0.00790	-0.00579
w/o WebP image compression	6.908	+3,030,148	+0.0000	+0.00000	+0.00000
w/o quaternion u8 quantization	6.427	+2,548,718	+0.0317	+0.00147	-0.00067
w/o f_dc u8 quantization	5.962	+2,084,291	+0.0063	+0.00012	-0.00009
w/o scales u8 quantization	5.844	+1,966,415	+0.0040	+0.00023	-0.00013
w/o means u16 quantization	5.271	+1,392,843	+0.0121	+0.00049	-0.00020
w/o primitive PLAS sorting	4.493	+614,851	+0.0059	+0.00022	-0.00036
w/o opacities u8 quantization	4.487	+609,117	+0.0006	+0.00007	-0.00006
w/o PRAS, quaternion full u8 quant	4.040	+161,748	+0.0101	+0.00040	-0.00008
w/o centroid PLAS sorting	3.920	+42,416	+0.0000	+0.00000	+0.00000
w/o label UV packing	3.890	+12,142	+0.0000	+0.00000	+0.00000
w/o grid mask (prune to square)	3.881	+3,344	-0.1150	-0.00576	+0.00279
w/o PRAS	3.866	-11,810	-0.1104	-0.00538	+0.00273
w/o means signed-log remapping	3.359	-519,272	-7.5508	-0.37864	+0.22151

example, by removing the lowest-opacity ones as in SOG [27]. After POPSpa, however, the remaining Gaussians are already valuable, making further pruning undesirable. SOG-XT therefore pads the grid with dummy primitives and uses an active mask to discard them again at decode time. Replacing this with opacity-based pruning provides only a tiny size benefit but leads to a clear quality loss ("w/o grid mask (prune to square)").

6.9 Simple Decoding of the SOG-XT Format

In the paper, we argue that although the encoding pipeline is sophisticated, decoding can remain simple. To illustrate this point, we provide a compact Python reference implementation that decodes one of our output files and writes the reconstructed scene to an INRIA-style '.ply' file.

The script first defines a small set of helpers for reading quantization ranges from the metadata, dequantizing stored attributes, and unpacking the tiled spherical harmonics centroid table into the 45 AC components.

It then decodes the attribute images, applies the corresponding inverse transforms, reconstructs the per-primitive attributes, and writes the result as a '.ply' file.

```
#!/usr/bin/env -S uv run --script
# /// script
# requires-python = ">=3.12"
# dependencies = [
#     "numpy",
#     "pillow",
#     "pyyaml",
# ]
```

```
# ///

"""Decode a SOG-XT container directory into a 3DGS-INRIA `.ply`."""

from __future__ import annotations

import argparse
from pathlib import Path

import numpy as np
import yaml
from PIL import Image


# ============================================================================
# Imports and Helpers
# ============================================================================

def read_image_uint8(path: Path) -> np.ndarray:
    image = np.array(Image.open(path))
    if image.ndim == 2:
        image = image[..., None]
    return image.astype(np.uint8, copy=False)


def normalize_to_01(values: np.ndarray) -> np.ndarray:
    values_f32 = values.astype(np.float32, copy=False)
    return (values_f32 - values_f32.min()) / (values_f32.max() - values_f32.min() + 1e-8)


def dequantize(
    quantized: np.ndarray,
    min_values: np.ndarray | float,
    max_values: np.ndarray | float,
) -> np.ndarray:
    return normalize_to_01(quantized) * (max_values - min_values) + min_values


def extract_ranges(container_meta: dict) -> dict[str, tuple[object, object]]:
    ranges: dict[str, tuple[object, object]] = {}
    for op in container_meta["ops"]:
        fields = op.get("input_fields") or []
        if len(fields) != 1:
            continue
        for transform in op.get("transforms", []):
            quant = transform.get("simple_quantize")
            if quant and "min_values" in quant:
                ranges[fields[0]] = (quant["min_values"], quant["max_values"])
    return ranges


def untile_feature_grid_3x5(tiled_hw3: np.ndarray) -> np.ndarray:
    tile_rows, tile_cols = 3, 5
    total_height, total_width, channels = tiled_hw3.shape
    height = total_height // tile_rows
    width = total_width // tile_cols
    tiles = tiled_hw3.reshape(tile_rows, height, tile_cols, width, channels)
    return tiles.transpose(1, 3, 4, 0, 2).reshape(height, width, channels * tile_rows *
     tile_cols)


# ============================================================================
# Main Decoding Path
# ============================================================================

def decode_container_to_columns(container_dir: Path) -> list[tuple[str, np.ndarray]]:
    container_meta = yaml.safe_load((container_dir / "container_meta.yaml").read_text())
    grid_side = int(container_meta["meta"]["grid_side"])
```

```
centroid_grid_side = int(container_meta["meta"]["f_rest_centroids_side"])
image_codec = str(container_meta["meta"]["image_codec"])
ranges = extract_ranges(container_meta)

def p(field: str) -> Path:
    return container_dir / f"{field}.{image_codec}"

active_mask = read_image_uint8(p("active_mask"))[..., 0].reshape(-1) > 0

opacities = dequantize(
    read_image_uint8(p("opacities"))[..., 0],
    float(ranges["opacities"][0]),
    float(ranges["opacities"][1]),
)
opacities = (1.0 / (1.0 + np.exp(-opacities))).reshape(grid_side * grid_side, 1)

scales_min = np.asarray(ranges["scales"][0], dtype=np.float32).reshape(1, 1, 3)
scales_max = np.asarray(ranges["scales"][1], dtype=np.float32).reshape(1, 1, 3)
scales = np.exp(dequantize(read_image_uint8(p("scales")), scales_min, scales_max)).
 reshape(grid_side * grid_side, 3)

means_low = read_image_uint8(p("means_bytes_0")).astype(np.float32)
means_high = read_image_uint8(p("means_bytes_1")).astype(np.float32)
means_signed_log = dequantize(
    means_low + 256.0 * means_high,
    float(ranges["means"][0]),
    float(ranges["means"][1]),
)
means = (np.sign(means_signed_log) * np.expm1(np.abs(means_signed_log))).reshape(
 grid_side * grid_side, 3)

quaternions_min = np.asarray(ranges["quaternions"][0], dtype=np.float32).reshape(1, 1, 4)
quaternions_max = np.asarray(ranges["quaternions"][1], dtype=np.float32).reshape(1, 1, 4)
quaternions = dequantize(
    read_image_uint8(p("quaternions")),
    quaternions_min,
    quaternions_max,
).reshape(grid_side * grid_side, 4)

f_dc_min = np.asarray(ranges["f_dc"][0], dtype=np.float32).reshape(1, 1, 3)
f_dc_max = np.asarray(ranges["f_dc"][1], dtype=np.float32).reshape(1, 1, 3)
f_dc = dequantize(read_image_uint8(p("f_dc")), f_dc_min, f_dc_max).reshape(grid_side *
 grid_side, 3)

labels_uv = read_image_uint8(p("f_rest_labels")).astype(np.int64, copy=False)
centroid_indices = (labels_uv[..., 1] * centroid_grid_side + labels_uv[..., 0]).reshape
 (-1)

centroid_min = np.asarray(ranges["f_rest_centroids"][0], dtype=np.float32)
centroid_max = np.asarray(ranges["f_rest_centroids"][1], dtype=np.float32)
if centroid_min.size != 45:
    raise ValueError("Expected 45-channel centroids (SOG-XT).")

centroid_grid = untile_feature_grid_3x5(normalize_to_01(read_image_uint8(p("
 f_rest_centroids"))))
centroids = dequantize(
    centroid_grid.reshape(centroid_grid_side, centroid_grid_side, 45),
    centroid_min.reshape(1, 1, 45),
    centroid_max.reshape(1, 1, 45),
).reshape(centroid_grid_side * centroid_grid_side, 45)

f_rest = centroids[centroid_indices].reshape(grid_side * grid_side, 3, 15).transpose(0,
 2, 1)
sh = np.concatenate([f_dc[:, None, :], f_rest], axis=1)

means = means[active_mask]
scales = scales[active_mask]
opacities = opacities[active_mask]
```

```
    quaternions = quaternions[active_mask]
    sh = sh[active_mask]

    normals = np.zeros_like(means, dtype=np.float32)
    scales_log = np.log(scales)
    opacities_logit = np.log(opacities) - np.log1p(-opacities)
    f_rest_flat = sh[:, 1:, :].transpose(0, 2, 1).reshape(sh.shape[0], 45)

    return [
        ("x", means[:, 0]),
        ("y", means[:, 1]),
        ("z", means[:, 2]),
        ("nx", normals[:, 0]),
        ("ny", normals[:, 1]),
        ("nz", normals[:, 2]),
        ("f_dc_0", sh[:, 0, 0]),
        ("f_dc_1", sh[:, 0, 1]),
        ("f_dc_2", sh[:, 0, 2]),
        *[(f"f_rest_{i}", f_rest_flat[:, i]) for i in range(45)],
        ("opacity", opacities_logit[:, 0]),
        ("scale_0", scales_log[:, 0]),
        ("scale_1", scales_log[:, 1]),
        ("scale_2", scales_log[:, 2]),
        ("rot_0", quaternions[:, 0]),
        ("rot_1", quaternions[:, 1]),
        ("rot_2", quaternions[:, 2]),
        ("rot_3", quaternions[:, 3]),
    ]


# ============================================================================
# PLY Writer and CLI Entry Point
# ============================================================================

def write_ply(path: Path, *, columns: list[tuple[str, np.ndarray]]) -> None:
    path.parent.mkdir(parents=True, exist_ok=True)
    num_vertices = int(columns[0][1].shape[0])
    matrix = np.empty((num_vertices, len(columns)), dtype="<f4")
    for column_index, (_name, values) in enumerate(columns):
        matrix[:, column_index] = values.astype(np.float32, copy=False).reshape(num_vertices)

    header = "\n".join([
        "ply",
        "format binary_little_endian 1.0",
        f"element vertex {num_vertices}",
        *[f"property float {name}" for name, _ in columns],
        "end_header",
        "",
    ]).encode("ascii")

    with path.open("wb") as f:
        f.write(header)
        f.write(matrix.tobytes())


def main() -> None:
    parser = argparse.ArgumentParser(description=__doc__)
    parser.add_argument("--input", type=Path, required=True, help="Input container directory.
     ")
    parser.add_argument("--output", type=Path, required=True, help="Output INRIA `.ply` path.
     ")
    args = parser.parse_args()
    write_ply(args.output, columns=decode_container_to_columns(args.input))


if __name__ == "__main__":
    main()
```

6.10 Comparison with State of the Art

Figure 7 places our *Mip-NeRF 360* results in the broader context of recent 3DGS compression methods from the literature. *Mip-NeRF 360* is the most informative benchmark in our evaluation protocol because it contains multiple real-world scenes, is evaluated on held-out views, and exposes the effect of using different training and evaluation resolutions. Even with this broader comparison, our reproduced operating points remain competitive across the rate-distortion range, with particularly strong results in LPIPS and file size.

6.11 Rate-Distortion Curves

The full curves make the dataset differences more explicit. On *Tanks and Temples* and *Mip-NeRF 360*, our method benefits from a clear quality margin over INRIA after reconstruction and compaction, which can then be traded for bitrate during compression. *Deep Blending* is the exception, where this margin is largely absent. The gap to HAC++ is already present before SOG-XT encoding, since compressed HAC++ exceeds uncompressed gsplat-MCMC and INRIA 3DGS on both *Playroom* and *Dr. Johnson*. This points to the underlying representation and its regularization rather than to SOG-XT quantization or POPSpa pruning. Our per-view inspection indicates that the mean gap is driven by a small number of views where high-capacity standard splats form isolated floaters. Reducing the primitive budget with POPSpa removes many of these weakly constrained primitives, which explains why the 512k models remain close to HAC++ on the full scenes. We therefore view *Deep Blending* as a limitation of the standard-splat reconstruction used as input to KISS-GS under these captures, rather than as a limitation of the image-based encoder alone.

6.12 Qualitative Comparison

Table 7 complements the quantitative tables and rate-distortion curves with a view-level comparison on the Bicycle scene. The examples support the same trend as the quantitative results. At 128k primitives and only 1.9 MB, our method already reconstructs the main foreground structure well and still reaches competitive PSNR, but it loses fine background detail. This can be seen in the grass behind the pedals in the 4x crop, the gray label on the bicycle frame on the left in the 8x crop, and the background behind the wheels in the 12x crop. At 1024k primitives, most of this missing background structure is recovered while the file remains slightly smaller than HAC++ low-rate, which still misses these details at a similar size. At 2048k primitives, the reconstruction also surpasses the INRIA baseline in LPIPS while remaining far smaller on disk, at 25.0 MB versus 1334.9 MB.

6.13 Full-Page Figures and Tables

For convenience, we collect the large comparison figures and tables here in the same order in which they were discussed above.

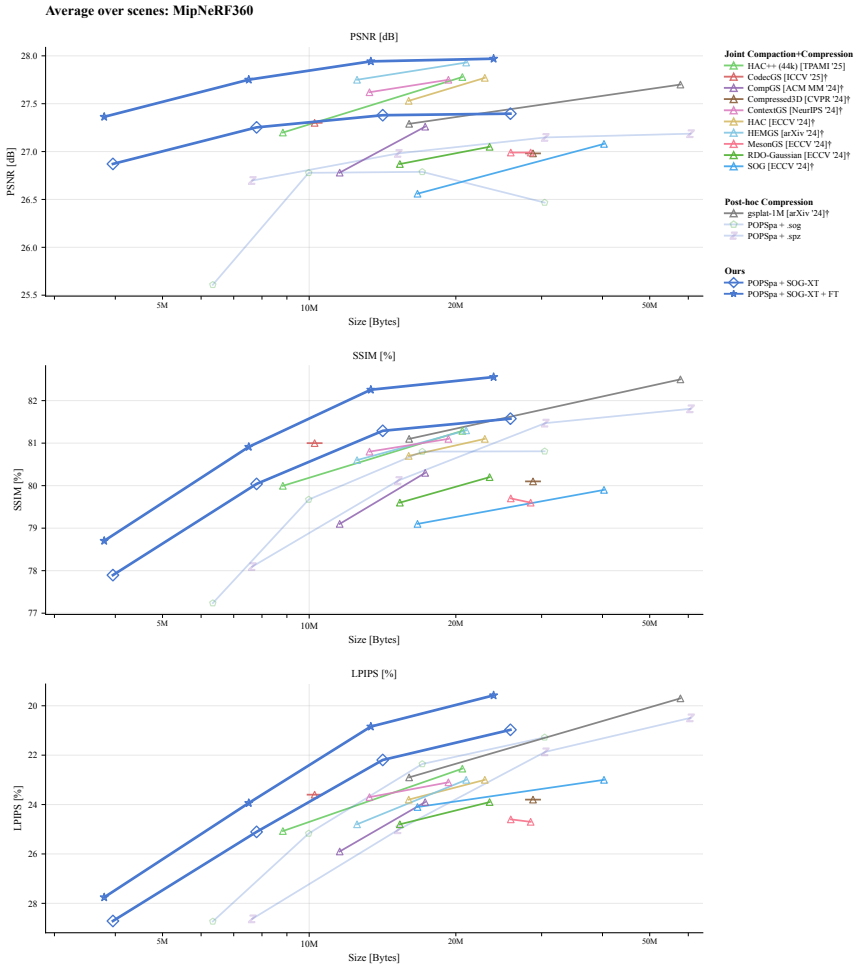

Fig. 7: Rate-distortion performance of KISS-GS and related work on the *Mip-NeRF 360* scenes. Methods marked with a dagger † are self-reported by the authors, as collected by the 3DGS.zip survey [1], and may not follow our evaluation protocol from Sec. 6.7.

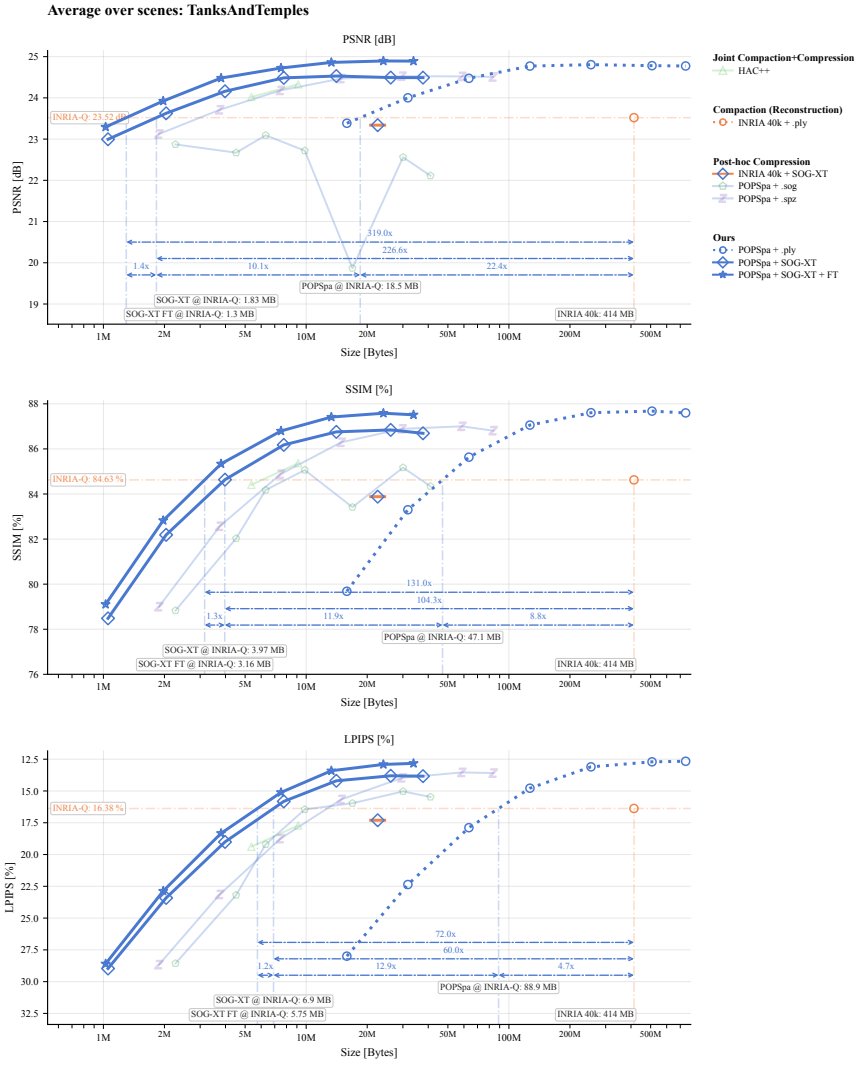

Fig. 8: Rate-distortion performance of KISS-GS on the *Tanks and Temples* scenes. Our reconstruction and compaction pipeline substantially surpasses the INRIA 40k baseline, which creates headroom for compression. As a result, KISS-GS can match the INRIA baseline at file sizes as low as 1.3 MB in PSNR and 5.75 MB in LPIPS.

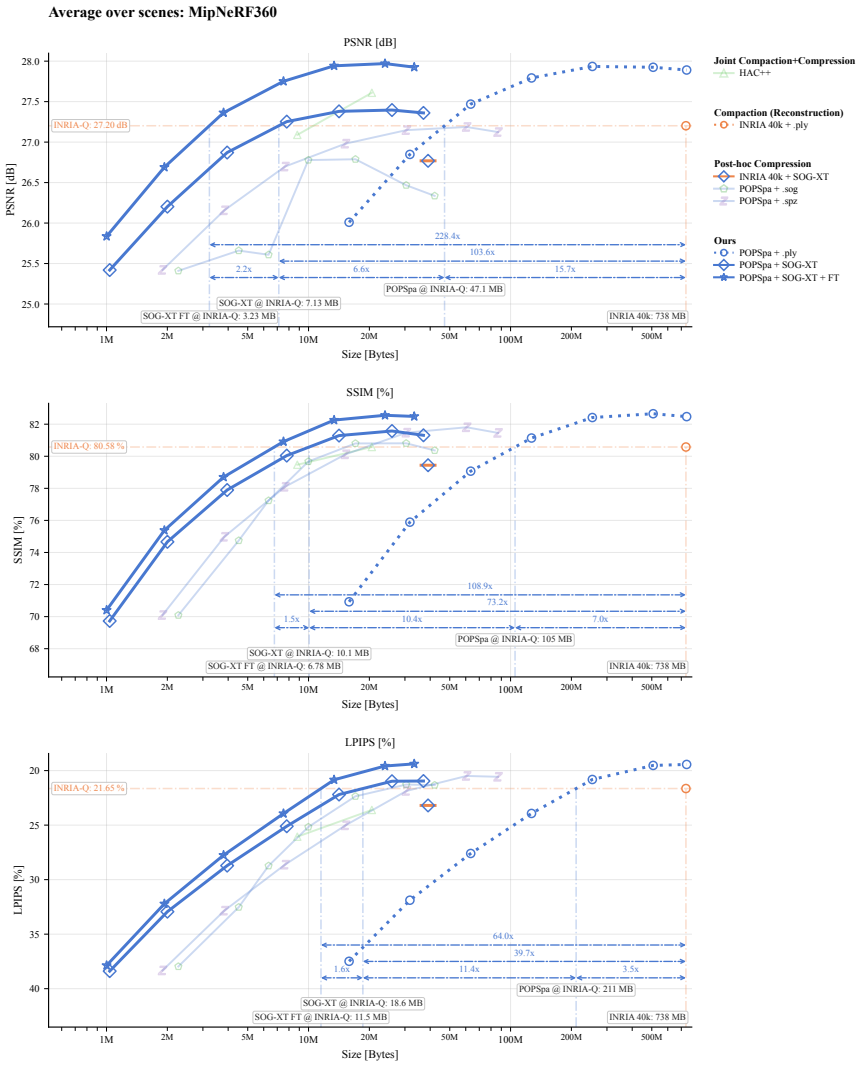

Fig. 9: Rate-distortion performance of KISS-GS on the *Mip-NeRF 360* scenes: In addition to the PSNR plot in Fig. 1, we report SSIM and LPIPS. The ordering is similar across metrics, although the gaps are closer for SSIM and LPIPS than for PSNR.

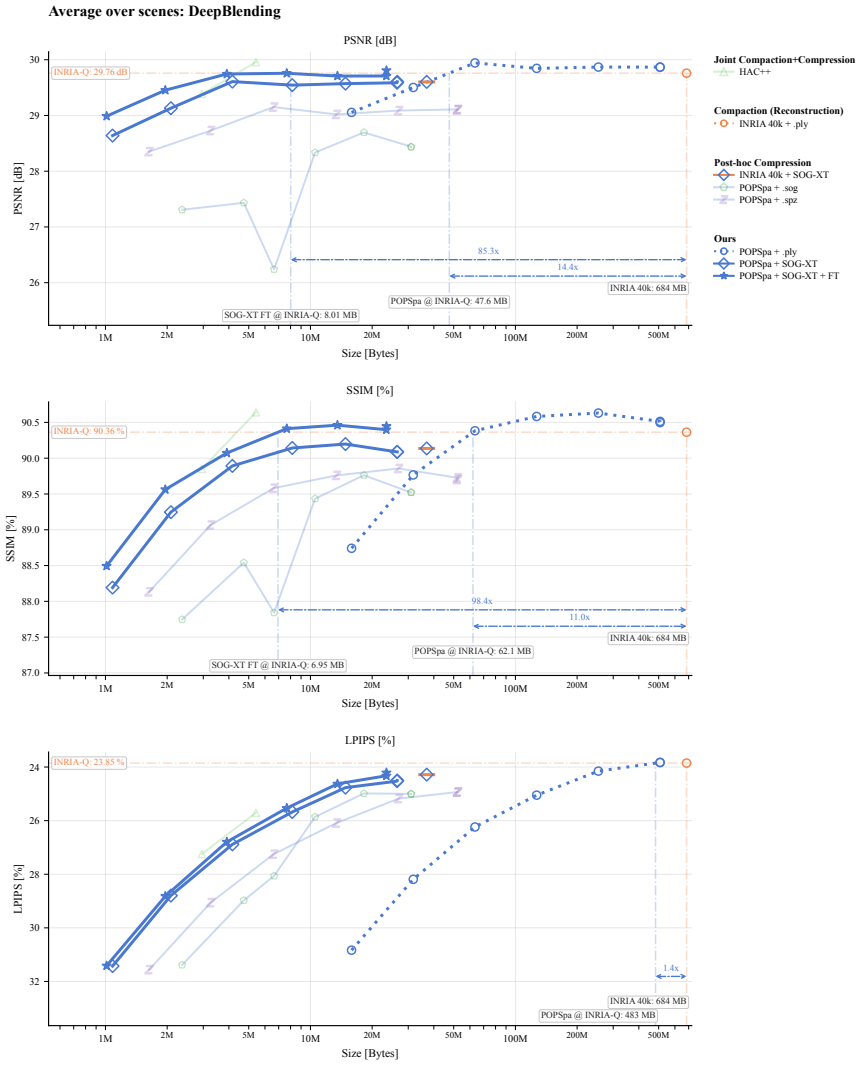

Fig. 10: Rate-distortion performance of KISS-GS on the *Deep Blending* scenes. Unlike on the other datasets, our reconstruction and compaction pipeline does not substantially improve over the INRIA baseline. The gap to HAC++ is already present before SOG-XT encoding, which indicates that the limiting factor is the standard-splat reconstruction under these captures rather than the image-based compression stage alone. As a result, none of the operating points of our compressed curve exceeds INRIA quality, while HAC++ [5] remains above INRIA for part of the range.

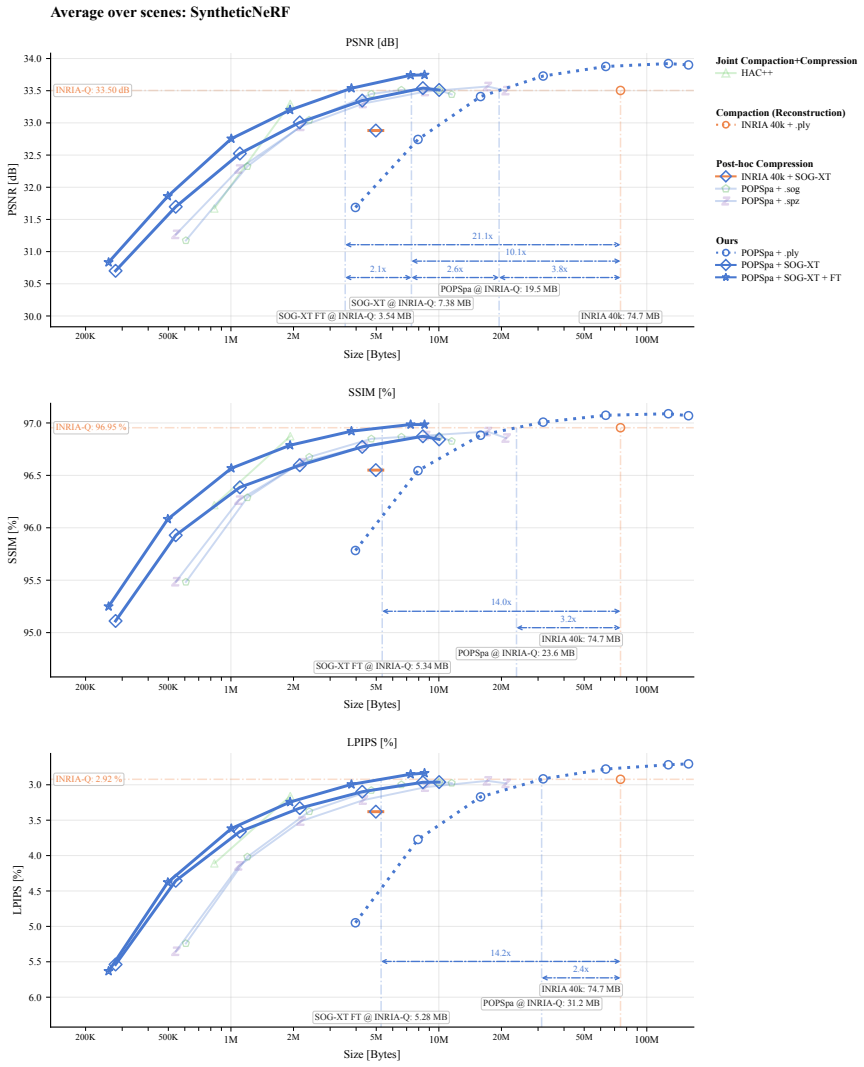

Fig. 11: Rate-distortion performance of KISS-GS on the *Synthetic NeRF* scenes. Because these scenes are synthetic, spatially smaller, and free of real-world capture effects, absolute quality metrics are higher for all methods.

Table 7: Qualitative comparison on evaluation view 17 of the Bicycle scene. We compare INRIA 40k, HAC++-44k, and our method (POPSpa + SOG-XT + Finetuning) at five primitive budgets. The Metrics column reports PSNR, LPIPS, and file size for this view. Informative regions to inspect are the grass behind the pedals in the 4x crop, the gray label on the bicycle frame on the left in the 8x crop, and the background behind the wheels in the 12x crop.

Method	Metrics	1x	4x	8x	12x
Ground truth					
INRIA	PSNR 24.806				
40k	LPIPS 0.1662				
	Size 1334.9 MB				
HAC++	PSNR 25.223				
low-rate	LPIPS 0.2533				
	Size 13.8 MB				
HAC++	PSNR 25.023				
high-rate	LPIPS 0.2174				
	Size 33.5 MB				
POPSpa-128k	PSNR 24.949				
SOG-XT-FT	LPIPS 0.3474				
(Ours)	Size 1.9 MB				
POPSpa-256k	PSNR 25.613				
SOG-XT-FT	LPIPS 0.2925				
(Ours)	Size 3.7 MB				
POPSpa-512k	PSNR 25.973				
SOG-XT-FT	LPIPS 0.2380				
(Ours)	Size 7.4 MB				
POPSpa-1024k	PSNR 25.990				
SOG-XT-FT	LPIPS 0.1812				
(Ours)	Size 13.5 MB				
POPSpa-2048k	PSNR 25.663				
SOG-XT-FT	LPIPS 0.1515				
(Ours)	Size 25.0 MB				