PACE: A REINFORCEMENT-LEARNING AGENT FOR ACCELERATED 3D GAUSSIAN SPLATTING

Anonymous authors
Paper under double-blind review

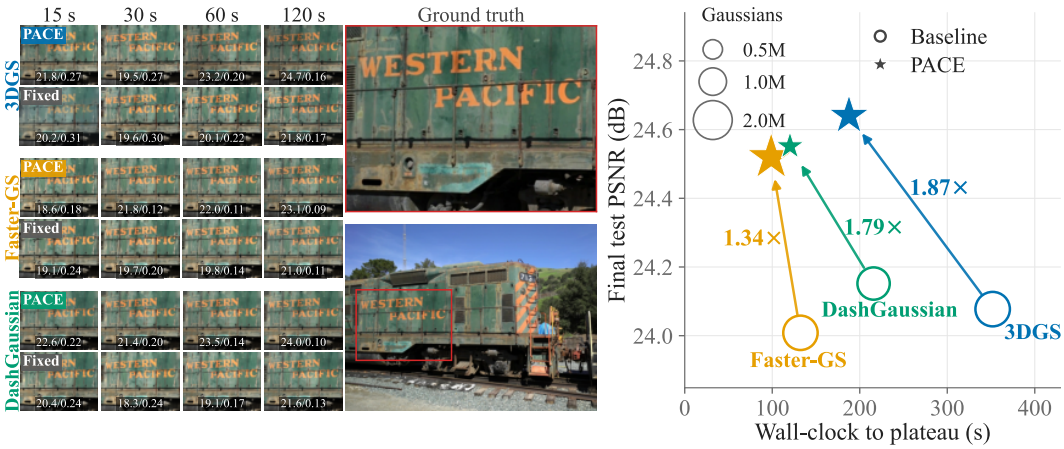

Figure 1: **PACE accelerates 3DGS training**. *Left:* test view at four early wall-clock instants; metrics are PSNR (dB) / LPIPS. *Right:* wall-clock time to train a test scene to its plateau against the PSNR finally reached, averaged over the six test scenes. The arrows are labeled with the speed-up factor. Up and to the left is better.

ABSTRACT

Accelerating 3D Gaussian Splatting (3DGS) training is a central open problem for real-world deployment: per-scene optimization still costs minutes of GPU time, which highly depends on a fixed, hand-tuned schedule of densification, pruning, opacity resets, and learning-rate decay. Heuristic alternatives require significant domain expertise and manual tuning, while existing learned controllers are trained online per scene, increasing training time. To address these challenges, we propose Policy for Adaptive Compute-Efficient (PACE) 3DGS training, a reinforcement-learning agent that accelerates training and generalizes across scenes and 3DGS training backends. We cast 3DGS training as a block-wise Markov decision process over a pluggable training backend. At each 3DGS training block, the agent observes the optimization state (a compact summary of training progress) and selects controls (Gaussian thresholds, learning rates, etc.) for the next block. PACE uses a lightweight MLP trained offline on a small set of 3D scenes, with a time-value reward that favors achieving higher quality more quickly. At deployment, PACE only requires feedforward evaluation of the trained policy. Evaluated on four representative backends (3DGS, Faster-GS, DashGaussian, and LeGS) across different scene types, PACE's acceleration has a consistent mechanism in the Gaussians optimization: where optimizing a Gaussian is expensive (e.g., 3DGS, DashGaussian), it densifies sparingly and freezes growth early, matching the fixed schedule's quality with half the primitives; where it is cheap (Faster-GS), it densifies aggressively, converting capacity into quality during early training stages. The result is up to $2.1\times$ faster training (e.g., 832 to 476 seconds to reach 21 dB on 3DGS), faster than the fixed schedule in all scene$\times$backend cases, up to $2.9\times$ smaller models, up to $1.3\times$ lower peak training VRAM, and up to $1.7\times$ faster rendering. The learned schedules also transfer across backends without retraining in eight of the nine policy–backend pairs we evaluate, retaining 1.2–$1.8\times$, and on LeGS reaching $2.02\times$. The exception is predicted by the discovered mechanism, where a policy trained such that compaction is desired (DashGaussian) underperforms on the one backend where growth is optimal (Faster-GS).

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

1 INTRODUCTION

3D Gaussian Splatting (3DGS) (Kerbl et al., 2023) has become a dominant representation for novel-view synthesis, combining high visual fidelity with real-time rendering (Fei et al., 2024). It now underpins applications well beyond view synthesis such as immersive AR/VR, telepresence, and photorealistic avatars (Jiang et al., 2024; Qian et al., 2024), dynamic and physically animated scenes for virtual production and visual effects (Wu et al., 2024b; Xie et al., 2024), digital twins of buildings and entire cities (Kerbl et al., 2024; Lin et al., 2024; Liu et al., 2024), dense SLAM and mapping for robots (Keetha et al., 2024; Matsuki et al., 2024), simulators in which manipulation and driving policies are trained and tested (Qureshi et al., 2024; Yan et al., 2024; Zhou et al., 2024), and compact models for mobile and web delivery (Fan et al., 2024). In most of these applications, the reconstruction is not the end product but rather a step in a loop where a scene is captured, a learned 3D model (by a 3DGS trainer) is produced, and it is immediately used. The time taken by 3DGS training determines the delay between capture and use, which is essential for applications that require fast workflows. For example, when 1) multiple scene captures are performed sequentially with human evaluation in the loop, or 2) 3D training is performed on a laptop or edge device with limited compute. In general, 3DGS training can require minutes of GPU time, creating a bottleneck on the performance of the ultimate task. Thus, accelerating 3DGS training is a core problem in the field and has attracted considerable attention in recent years.

Most acceleration work focuses on speeding up the trainer itself, including the kernels, fused rasterizers, backward passes, and optimizers (Hahlbohm et al., 2026; Mallick et al., 2024; Hanson et al., 2025), while keeping the original 3DGS training process. DashGaussian (Chen et al., 2025) schedules rendering resolution and primitive growth to cut wasted computations. These approaches are complementary and effective, and we build on them. However, they still use fixed 3DGS training controls: densification at a preset interval and threshold, pruning and opacity resets at preset iterations, and a user-defined learning-rate decay schedule. Additionally, the cost of a run is set by the number of Gaussians optimized and the number of optimization steps. A fixed schedule adds Gaussians regardless of whether they improve the reconstruction, and keeps updating those that no longer do. Usually, early in training, schedules grow the model with Gaussians that never get optimized. To reach a target quality faster, an optimized schedule would allocate compute only to Gaussians that improve quality and spend less compute on those that are useless or computationally expensive. Such an allocation would depend on the state of the optimization, the scene, and the solver.

Thus, in this paper, we propose *Policy for Adaptive Compute-Efficient (PACE) 3DGS training*. The core of PACE is to train a lightweight MLP on a small set of 3D scenes, so that it controls the optimization dynamics of the 3DGS trainer backend, reducing the wall-clock time required to achieve plateau performance. This policy network generalizes to unseen scenes and adapts the selected control depending on the scene and the 3DGS trainer used. Importantly, the training of PACE (offline via reinforcement learning) is independent of the 3DGS training (using gradient descent), so after PACE is optimized, it remains frozen and is used only for a few feedforward evaluations during the 3DGS training. Particularly, PACE turns the schedule of the 3DGS training into a learned decision problem. We model 3DGS optimization as a block-wise Markov decision process over a pluggable training backend: at the end of each short block, PACE observes a compact, normalized 45-dimensional vector summary of the optimization (progress, loss and quality trends, compute usage, Gaussian population statistics) and selects the controls of the next block (its length, whether to densify and prune and how aggressively to do so, whether to reset opacities, how to scale the per-parameter learning rates). The reward is a time-value-discounted quality gain, for which maximizing return means reaching every quality target in the least wall-clock time. The policy is a lightweight MLP trained offline with proximal policy optimization (PPO) (Schulman et al., 2017) on a small pool of scenes and fixed for deployment (using the trained policy for unseen scenes), i.e., without per-scene or per-backend tuning.

Because state, action set, transition, and optimizer carry no backend-specific element, the same formulation controls a wide range of 3DGS training backends. We tested our approach on the base 3DGS trainer Kerbl et al. (2023), Faster-GS Hahlbohm et al. (2026), which provides highly efficient 3DGS training, DashGaussian Chen et al. (2025), which includes a coarse-to-fine resolution schedule of the views for faster convergence and an online learned control approach targeted only towards quality improvement, and LeGS Ning et al. (2026). Although we select these trainer backends, our approach can be easily adapted to other efficient 3DGS approaches Fan et al. (2024); Mallick

et al. (2024); Hanson et al. (2025); Kheradmand et al. (2024). We show that PACE adapts the controls based on the 3DGS trainer properties and the 3D scene, leading to faster convergence, smaller models in most cases, and better performance. Also, PACE shows remarkable generalization to unseen scenes and in cross-backend testing (policy training on one backend and testing on a different one). Our approach introduces a new method for fast, efficient 3DGS training that leverages existing training methods.

2 Related Work

Accelerating 3DGS Most work on faster 3DGS speeds up the kernels or the primitives while keeping the training schedule fixed: Faster-GS (Hahlbohm et al., 2026) consolidates rasterizer, sorting, backward-pass, and optimizer improvements into a 2–5$\times$ faster trainer with the original optimization outcome, Taming-3DGS (Mallick et al., 2024) parallelizes the backward pass, Speedy-Splat (Hanson et al., 2025) prunes primitives by an importance score, 3DGS-MCMC (Kheradmand et al., 2024) recasts densification as sampling toward a target count; DashGaussian (Chen et al., 2025) schedules rendering resolution and primitive growth, and LeGS (Ning et al., 2026) learns per-Gaussian densification decisions with RL. These methods accelerate the *backend*; our policy controls the *schedule* and is agnostic to the backend, so we run it on top of the official rasterizer, Faster-GS, DashGaussian, and LeGS itself.

Feed-forward methods Feed-forward reconstruction amortizes optimization entirely: pixelSplat (Charatan et al., 2024), MVSplat (Chen et al., 2024), Splatter Image (Szymanowicz et al., 2024) and SHARP (Mescheder et al., 2026) regress a Gaussian scene from one or a few views in a single pass. Their fidelity is bounded by the training distribution and the sparse-input regime, so for dense captures, per-scene optimization remains the quality reference. Feed-forward prediction could be used for model initialization, where our agent then schedules it until convergence.

Learned control of optimization Learning to control optimization has a long history in learning-to-learn (Thrun & Pratt, 1998; Hospedales et al., 2021): learned optimizers replace the update rule, trained by gradient descent (Andrychowicz et al., 2016; Wichrowska et al., 2017; Metz et al., 2022) or RL (Li & Malik, 2016; Bello et al., 2017; Chen et al., 2022); hyperparameter methods keep a standard optimizer and search *static* configurations (Snoek et al., 2012; Li et al., 2018) or adapt them *online* during a run (Jaderberg et al., 2017; Daniel et al., 2016; Hansen, 2016; Xu et al., 2019; Jomaa et al., 2019), the RL-as-controller view that also underlies neural architecture search (Zoph & Le, 2016; Baker et al., 2017). We follow the online, RL-as-controller line (Daniel et al., 2016; Xu et al., 2019). Within 3DGS, LeGS (Ning et al., 2026) applies it to density control, deciding *which* primitives to grow or remove with a per-scene policy rewarded for reconstruction quality, while the outer recipe stays fixed. PACE has three main design principles: it learns the schedule (when to densify, prune and reset, threshold and learning-rate multipliers, block length), it is trained offline across scenes and deployed zero-shot, and it is rewarded for time to quality. To our knowledge, we are the first to develop and offline learned optimization control to the 3DGS training schedule and to optimize it directly for time to quality.

3 Problem Formulation

3D Gaussian Splatting A 3D scene is represented with a set of N anisotropic 3D Gaussians with parameters $\mathbf{\Omega} = \{(\boldsymbol{\mu}_i, \boldsymbol{s}_i, \boldsymbol{q}_i, o_i, \boldsymbol{c}_i)\}_{i=1}^{N}$ (mean, scale, rotation, opacity, and spherical-harmonic color). Rendering projects them to the image plane and alpha-composites them front-to-back (Kerbl et al., 2023); the parameters are fit to posed images by gradient descent on

$$\mathcal{L}(\mathbf{\Omega}) = (1-\lambda)\,\mathcal{L}_1(\mathbf{\Omega}) + \lambda\,\mathcal{L}_{\text{D-SSIM}}(\mathbf{\Omega}). \tag{1}$$

Because N must itself be learned, training interleaves gradient steps with adaptive density control (ADC). Particularly, it controls the Gaussians every few hundred iterations, where those with large positional gradients are cloned or split, low-opacity or oversized ones are pruned, opacities are periodically reset, and the position learning rate decays exponentially. The optimizer and rasterizer that performs this loop are swappable backend functions, providing different effects on the resulting 3D scene representation and rendering quality and wall-clock time. Our work does not focus on

developing a new trainer backend but on developing a plug-in model that controls the underlying optimizer to achieve the highest quality in less wall-clock time.

Training as a controlled block-wise process We model 3DGS training as a controlled process by partitioning it into blocks (a set of gradient descent steps). Let the solver state $\mathcal{G}_k = (\boldsymbol{\Omega}_k, \boldsymbol{m}_k)$ collect, at the start of block k, everything training carries forward: the Gaussians $\boldsymbol{\Omega}_k = (\boldsymbol{\omega}_i)_{i\in I_k}$ over an index set I_k of cardinality N_k, with $\boldsymbol{\omega}_i = (\boldsymbol{\mu}_i, \boldsymbol{s}_i, \boldsymbol{q}_i, o_i, \boldsymbol{c}_i)$ as above, and the optimizer bookkeeping $\boldsymbol{m}_k = \big((\boldsymbol{u}_i, \boldsymbol{v}_i, \boldsymbol{b}_i)_{i\in I_k},\ \boldsymbol{\zeta}_k\big)$, where $\boldsymbol{u}_i, \boldsymbol{v}_i \in \mathbb{R}^{59}$ are the optimizer first and second moments of $\boldsymbol{\theta}_i$, $\boldsymbol{b}_i \in \mathbb{R}^3$ holds the statistics ADC thresholds (the accumulated view-space position-gradient norm, the number of views it accumulates over, and the largest screen-space radius observed so far), and $\boldsymbol{\zeta}_k = (k_{\mathrm{it}}, \ell_k) \in \mathbb{Z}^2$ the global iteration index and active SH degree, respectively. Then, block k runs a given number of gradient-descent steps through a training operator $\mathcal{T}_{a_k}$, so the state evolves as

$$\mathcal{G}_{k+1} = \mathcal{T}_{a_k}(\mathcal{G}_k; \mathcal{D}), \qquad k = 0, 1, \ldots, K \tag{2}$$

where a_k are the controls of the 3DGS iterations and $\mathcal{D}$ is the scene. The controls set the block length, whether and how aggressively to densify and prune, whether to reset opacities, and multipliers on the learning rates and ADC thresholds (Section 4.3). Block k costs wall-clock Δt_k, which depends on the backend $\mathcal{B}$ and the model size, with $t_k = \sum_{j=1}^{k} \Delta t_j$.

Let q map a solver state to a scalar reconstruction-quality metric, so $q(\mathcal{G}_k) \in \mathbb{R}$ is the quality of the model at the start of block k. Under this setting, we formally define our acceleration.

Acceleration objective

Choose the actions $a_0, a_1, \ldots a_K$ that deliver the higher quality improvement in the least wall-clock time

$$\max_{a_0, a_1, \ldots} \sum_{k\geq 0} \nu(t_{k+1}) \big[q(\mathcal{G}_{k+1}) - q(\mathcal{G}_k)\big], \qquad \nu(t) = e^{-t/\tau}, \tag{3}$$

with τ being a time-scale parameter that sets how strongly the objective prioritizes performance in the first blocks.

4 PACE FOR 3DGS TRAINING

In general, PACE is designed as a lightweight neural network that takes as input an observation vector of the 3DGS training dynamics and outputs per-scene/per-backend actions that control the next 3DGS block. The network is trained offline with an acceleration objective with a time-to-quality reward. The optimization of the network given by the designed reward is formulated via PPO. Particularly, PPO is quite suitable for this problem since it optimizes the discrete and continuous parts of the action under a single objective, and because it changes the policy only slightly at each update, since highly abrupt changes will lead to unstable algorithm control. The overall pipeline of PACE is shown in Figure 2.

4.1 MARKOV DECISION PROCESS FORMULATION

We define a Markov Decision Process (MDP) by the 5-tuple $(\mathcal{S}, \mathcal{A}, P, r, \gamma)$. $\mathcal{S} \subseteq [0,1]^d$ is the space of compact solver observations (Section 4.2); $\mathcal{A} = \mathcal{A}_{\mathrm{d}} \times \mathcal{A}_{\mathrm{c}}$ is the mixed discrete/continuous action set (Section 4.3); P is the transition kernel induced by executing one block of training (Section 4.4); $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is the reward (Section 4.5); and $\gamma \in [0,1]$ is the discount. Each episode (single-scene 3DGS training) initializes $\mathcal{G}_0$ from the scene's structure-from-motion point cloud Irschara et al. (2009) and runs to a fixed number of iterations.

4.2 STATE VECTOR

We define the state $s_k \in [0,1]^{45}$ as a compact, normalized, scene- and resolution-agnostic summary of the solver, built only from quantities available at block boundaries. It concatenates six feature groups:

$$s_k = \Big[\, \boldsymbol{\phi}_k^{\mathrm{prog}},\ \boldsymbol{\phi}_k^{\mathrm{loss}},\ \boldsymbol{\phi}_k^{\mathrm{val}},\ \boldsymbol{\phi}_k^{\mathrm{comp}},\ \boldsymbol{\phi}_k^{\mathrm{pop}},\ \boldsymbol{\phi}_{k-1}^{\mathrm{act}} \,\Big], \tag{4}$$

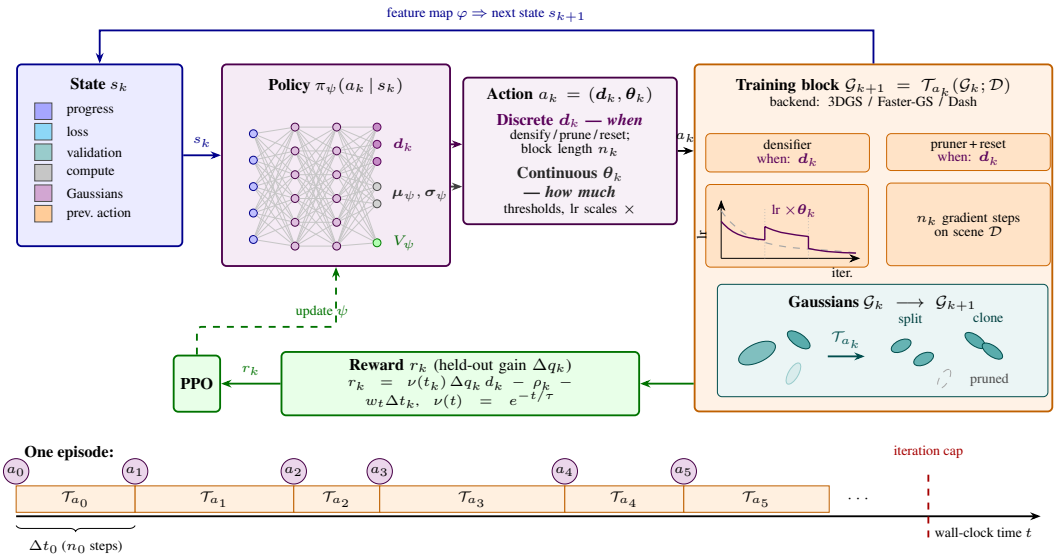

Figure 2: **Overview of PACE.** At every block boundary, the policy observes the solver state s_k and emits a mixed action $a_k = (\boldsymbol{d}_k, \boldsymbol{\theta}_k)$ that configures one block of otherwise-standard training on a pluggable backend; PPO updates the policy from a reward whose time value $\nu(t)$ credits early quality gains more. An episode is a sequence of variable-length blocks on the wall clock.

where the groups are defined as: (i) *progress* $\boldsymbol{\phi}_k^{\text{prog}} = [\,k_{\text{it}}/K,\ t_k/T,\ 1 - t_k/T,\ k/K_{\text{blk}}\,]$, the normalized iteration, the elapsed and remaining fractions of the episode's time cap T, and the block index; (ii) *loss* $\boldsymbol{\phi}_k^{\text{loss}}$, exponential moving averages of the $\mathcal{L}_1$, D-SSIM and total losses and a recent loss slope; (iii) *validation* $\boldsymbol{\phi}_k^{\text{val}}$, PSNR, SSIM and their improvement, evaluated on a small farthest-point-sampled subset of the *training* cameras so that test views never enter the state; (iv) *compute* $\boldsymbol{\phi}_k^{\text{comp}}$, time per iteration, VRAM fraction and active resolution; (v) *population* $\boldsymbol{\phi}_k^{\text{pop}}$, the Gaussian count relative to a reference, growth rate, opacity and scale quantiles, the near-transparent and visible fractions, anisotropy and active SH degree; and (vi) the decoded *previous action* $\boldsymbol{\phi}_{k-1}^{\text{act}}$. All entries are clipped to $[0, 1]$. The state is backend-agnostic: all trainer backends are driven through this same 45-dimensional vector, with no backend-specific features—which is what lets a single formulation control 3DGS, Faster-GS, and DashGaussian, and transfer unchanged to LeGS.

4.3 Action Set

An action $a_k = (\boldsymbol{d}_k, \boldsymbol{\theta}_k) \in \mathcal{A}_{\text{d}} \times \mathcal{A}_{\text{c}}$ combines five categorical decisions with a vector of seven continuous multipliers, which decode into the controls executed by $\mathcal{T}_{a_k}$.

Discrete controls The discrete action $\boldsymbol{d}_k \in \mathcal{A}_{\text{d}}$ is the product of five categorical variables,

$$\mathcal{A}_{\text{d}} = \underbrace{\mathcal{A}_{\text{d}}^{n}}_{\text{block length } n_k} \times \underbrace{\mathcal{A}_{\text{d}}^{\text{dm}}}_{\text{densify mode}} \times \underbrace{\mathcal{A}_{\text{d}}^{\text{di}}}_{\text{densify interval}} \times \underbrace{\mathcal{A}_{\text{d}}^{\text{pm}}}_{\text{prune mode}} \times \underbrace{\mathcal{A}_{\text{d}}^{\text{or}}}_{\text{opacity reset}}, \tag{5}$$

with $|\mathcal{A}_{\text{d}}| = 3 \cdot 4 \cdot 4 \cdot 3 \cdot 3 = 432$ joint choices. Table 1 lists the options of each set and decodes them into the concrete controls applied to the block: the block length sets the control horizon, the densify mode and interval set how hard and how often the model grows, and the prune mode and opacity reset set what is removed. The neutral choice $(100, \texttt{def}, 100, \texttt{op+sz}, \texttt{plt})$ with all multipliers at 1 is the fixed-schedule baseline used throughout: the backend's stock densification, pruning and learning-rate settings, with resets gated on a plateau.

Continuous controls The continuous action $\boldsymbol{\theta}_k \in \mathcal{A}_{\text{c}} \subset \mathbb{R}^7$ comprises multiplicative factors on the densification gradient threshold and the pruning opacity threshold, and per-parameter learning-rate multipliers for the positions, SH features, opacities, scales, and rotations. The policy emits

Table 1: **The five discrete actions of Eq. (5)**, **Bold** marks the neutral baseline action. Every option is defined relative to that stock behaviour; the thresholds, rates and limits it stands for are listed in Section C.2.

Control	Option	Effect on the GS training
Block length n_k	50/**100**/200	Gradient steps run before the policy is queried again.
Densify mode	`off`	No cloning or splitting; pruning still runs, so the model can only shrink.
	`cons`	Raises the clone/split gradient threshold: only the most under-reconstructed Gaussians are refined.
	`def`	Default threshold: a Gaussian above it is cloned if small, split in two if large.
	`aggr`	Lowers the threshold: many more Gaussians qualify, so the model grows fastest.
Densify interval	50/**100**/200/400	Iterations between densify–prune events: how often the modes above fire.
Prune mode	`off`	Nothing is removed.
	`op`	Removes Gaussians whose opacity has fallen below the prune threshold.
	`op+sz`	Also removes Gaussians that are too large in screen or world space.
Opacity reset	`none`	Never resets.
	`plt`	Resets only if the previous block failed to improve held-out quality.
	`force`	Resets whenever the safety rails allow, improvement or not.

unbounded reals $\tilde{\boldsymbol{\theta}}_k$ that are constrained into bounded (mostly log-uniform) ranges $[\ell_j, u_j]$,

$$[\boldsymbol{\theta}_k]_j = \ell_j \left(u_j/\ell_j\right)^{\sigma([\tilde{\boldsymbol{\theta}}_k]_j)}, \qquad \sigma(z) = \tfrac{1}{1+e^{-z}}, \tag{6}$$

so each factor can be damped or amplified around the default schedule (e.g., the position learning rate over $[1/4, 4]$). We keep the standard 3DGS optimizer, densifier, and pruner fixed; the policy only *modulates* them through $(\boldsymbol{d}_k, \boldsymbol{\theta}_k)$.

4.4 Transition function

The transition is defined operationally by executing one block of training. Given s_k and $a_k = (\boldsymbol{d}_k, \boldsymbol{\theta}_k)$, the environment (i) decodes the controls via Eqs. (5) and (6), (ii) advances the solver state $\mathcal{G}_{k+1} = \mathcal{T}_{a_k}(\mathcal{G}_k; \mathcal{D})$ by running n_k gradient steps with the chosen densification, pruning, opacity-reset and learning-rate settings, (iii) updates the clock $t_{k+1} = t_k + \Delta t_k$, and (iv) extracts the next observation $s_{k+1} = \varphi(\mathcal{G}_{k+1}, t_{k+1})$ through the feature map φ of Eq. (4). Compactly,

$$s_{k+1} = P(s_k, a_k) \;\triangleq\; \varphi\big(\mathcal{T}_{a_k}(\mathcal{G}_k; \mathcal{D}),\, t_k + \Delta t_k\big). \tag{7}$$

4.5 Reward

We instantiate the quality metric q of Eq. (3) on a set $\mathcal{V}$ of validation views drawn from the *training* split by farthest-point sampling over camera centers and excluded from the gradient steps. Scoring view v by $q^{(v)}(\mathcal{G}) = \mathrm{PSNR}_v(\mathcal{G}) + w_{\mathrm{ssim}}\,\mathrm{SSIM}_v(\mathcal{G})$, the model score blends the average view with the worst one,

$$q(\mathcal{G}) = (1 - w_{\min})\,\frac{1}{|\mathcal{V}|}\sum_{v\in\mathcal{V}} q^{(v)}(\mathcal{G}) \;+\; w_{\min}\min_{v\in\mathcal{V}} q^{(v)}(\mathcal{G}), \tag{8}$$

with w_{ssim} and $w_{\min}$ being hyperparameters on the SSIM and the weight on the worst view. The minimum-view term matters because the mean alone lets a policy buy quality on well-covered views by neglecting a poorly covered one, which the full test set then exposes. Note that q is the *training-time* objective, computed on $|\mathcal{V}|$ training views; thus, testing views are only used at evaluation time,

Algorithm 1 Inference

Require: scene $\mathcal{D}$, backend $\mathcal{B}$, trained policy $\pi_{\psi^\star}$, iteration cap K

init $\mathcal{G}_0$, $t_0 \leftarrow 0$, $k \leftarrow 0$
while iteration $< K$ **do**
  $s_k \leftarrow \varphi(\mathcal{G}_k, t_k)$; $a_k \leftarrow \arg\max_a \pi_{\psi^\star}(a \mid s_k)$
  $\mathcal{G}_{k+1} \leftarrow \mathcal{T}_{a_k}(\mathcal{G}_k; \mathcal{D})$; $t_{k+1} \leftarrow t_k + \Delta t_k$; $k \leftarrow k+1$
return trained model $\mathcal{G}_k$

and the policy never sees them. Let $q_k = q(\mathcal{G}_k)$, $\Delta q_k = q_k - q_{k-1}$ the per-block gain, $d_k \in (0, 1]$ a compactness discount that shrinks the gain when the Gaussian count exceeds a reference N_{ref}. Particularly, we define $d_k = 1/(1 + e_k)$ with $e_k = \max(0, N_k/N_{\text{ref}} - 1)$, the relative excess of the Gaussian count over a reference $N_{\text{ref}} = 8N_0$ (N_0 is Structure from Motion points), so necessary densification is not penalized in excess. We define $\rho_k \geq 0$ as a small stability, VRAM and model-size penalties (all constants and the exact terms of d_k and ρ_k in Section C.2).We time-discount the per-block quality gain and charge a mild time cost:

$$r_k \;=\; \nu(t_k)\,\Delta q_k\,d_k \;-\; \rho_k \;-\; w_t\,\Delta t_k, \qquad \nu(t_k) = e^{-t_k/\tau}, \tag{9}$$

where the discount $\nu(t_k)$ makes early quality worth more than late quality, and the time cost $w_t \Delta t_k$ charges every block for the wall-clock time it consumes. Thus, with horizon K (the number of blocks in the episode), the optimization of the policy π_ψ is given by

$$R^\gamma(\xi) = \sum_{k=0}^{K-1} \gamma^k\, r_k, \qquad \pi_\psi^\star = \arg\max_\psi\; J(\psi), \qquad J(\psi) = \mathbb{E}_{\pi_\psi}\big[R^\gamma(\xi)\big], \tag{10}$$

where $\xi == (s_0, a_0, r_0, \ldots, s_{K-1}, a_{K-1}, r_{K-1}, s_K)$ is the trajectory, the expectation is over scenes, initializations, and the stochastic transition. The value function is $V^\pi(s) = \mathbb{E}_\pi[\sum_{t \geq k} \gamma^{t-k} r_t \mid s_k = s]$.

4.6 Policy parameterization and optimization

The policy and value function share a small MLP encoder $z_k = \mathcal{E}_\psi(s_k)$ (architecture and sizes in Section C). Five categorical heads produce logits $\boldsymbol{w}_k^{(m)} = \mathcal{D}_\psi^{(m)}(z_k)$ for the discrete factors m of Eq. (5); a continuous head produces a mean $\boldsymbol{\mu}_\psi(z_k)$ and a state-independent log-scale $\log \boldsymbol{\sigma}_\psi$ of a diagonal Gaussian over $\tilde{\boldsymbol{\theta}}_k$; and a value head outputs $V_\psi(z_k)$. The action likelihood factorizes as

$$\pi_\psi(a_k \mid s_k) = \Big[\textstyle\prod_{m=1}^{5} \mathrm{Cat}\big([\boldsymbol{d}_k]_m;\, \boldsymbol{w}_k^{(m)}\big)\Big]\, \mathcal{N}\big(\tilde{\boldsymbol{\theta}}_k;\, \boldsymbol{\mu}_\psi(z_k),\, \mathrm{diag}\,\boldsymbol{\sigma}_\psi^2\big). \tag{11}$$

For the optimization of the policy $\pi_\psi(a_k|s_k)$ we employ a PPO formulation Schulman et al. (2017), with an actor-critic scheme Schulman et al. (2015). We provide the full derivation of the PPO objective for the proposed approach in Section A.

5 Experiments

Each controller is trained with PPO on ten real COLMAP scenes from Tanks&Temples (Knapitsch et al., 2017), Deep Blending (Hedman et al., 2018) and Mip-NeRF 360 (Barron et al., 2022) (bounded indoor and unbounded outdoor captures; the Faster-GS controller on photometric augmentations of the pool) and evaluated zero-shot on held-out scenes, Tanks&Temples Knapitsch et al. (2017) and Mip-NeRF 360 Barron et al. (2022); hyperparameters and training budget are in Section C. The baseline is each backend's fixed-schedule trainer, run with the same rasterizer on the same GPU. *Time-to-target* is the wall-clock time to first cross a PSNR target on the full-resolution test cameras; the speed-up is baseline/PACE per target, summarized by the geometric mean (*gmean*) over the targets both reach.

5.1 Training dynamics of PACE

We first show the effect of PACE during the Gaussian optimization at Figure 3 for three backends. All plots are defined with respect to the wall-clock 3DGS training time. Plots at Figure 3 (a) show the Gaussian count, Figure 3 (b) the cost of an iteration, and Figure 3 (c) the test reconstruction quality

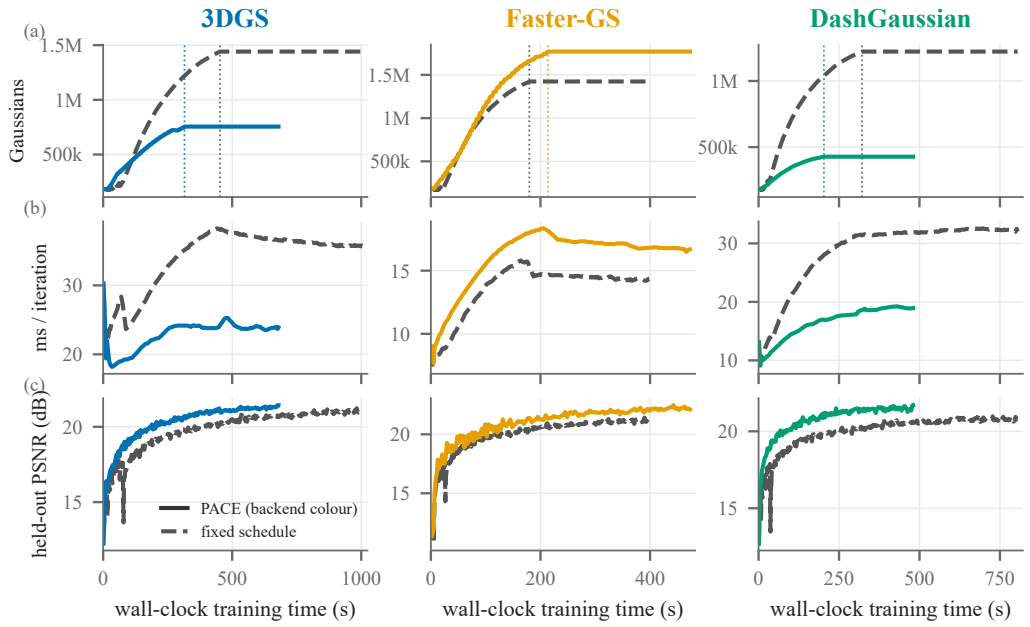

Figure 3: **Training dynamics of PACE** (`train` scene, full 30k rollouts, one column per backend, shared wall-clock axis; PACE in the backend's colour, fixed schedule in grey). (a) Gaussian count (dotted verticals: last block that added Gaussians), (b) wall-clock cost of one iteration, (c) Test PSNR.

Table 2: **Acceleration on eight held-out scenes** (Tanks&Temples and Mip-NeRF 360; none in the training pool). For each backend: geometric-mean time-to-target speed-up of PACE over that backend's fixed schedule, test PSNR (dB) and Gaussian count at the end of the run, PACE / fixed. **Teal**: speed-up $\geq 1.4\times$ or PACE model at least 20% smaller; **orange**: PACE at least 0.3 dB ahead at the end of the run.

		3DGS			Faster-GS			DashGaussian		
Scene	**Data**	**Speed-up** (gmean)	**PSNR (dB)** PACE / fixed	**#G (k)** PACE / fixed	**Speed-up** (gmean)	**PSNR (dB)** PACE / fixed	**#G (k)** PACE / fixed	**Speed-up** (gmean)	**PSNR (dB)** PACE / fixed	**#G (k)** PACE / fixed
`train`	T&T	**1.80×**	21.3/21.3	**720**/1447	1.37×	**22.7**/21.2	1713/1408	**2.11×**	**21.5**/21.2	**427**/1226
`barn`	T&T	1.26×	28.4/28.5	**680**/1140	1.05×	**29.1**/28.5	1215/1111	1.32×	28.0/28.5	**368**/889
`caterpillar`	T&T	1.28×	23.8/24.1	**855**/1330	1.38×	**24.9**/23.6	1669/1319	**1.43×**	23.5/23.9	**392**/1062
`ignatius`	T&T	**2.03×**	**23.8**/23.3	**1704**/2562	**1.45×**	23.1/23.3	2774/2552	**1.81×**	**24.5**/23.3	**702**/1924
`meetingroom`	T&T	1.19×	25.7/26.4	**856**/1371	1.16×	**27.7**/26.8	1489/1434	**1.41×**	26.1/26.9	**481**/1118
`courthouse`	T&T	**1.97×**	**22.8**/21.8	**583**/909	1.29×	**23.5**/22.6	1131/916	**1.98×**	**22.9**/22.2	**620**/818
`bicycle`	Mip-360	1.21×	**24.7**/23.2	**2194**/2888	1.28×	22.9/23.2	2426/2817	**1.41×**	**24.6**/23.6	**1210**/2790
`stump`	Mip-360	**1.44×**	**25.8**/24.1	2157/2620	1.22×	**24.5**/24.2	**2097**/2725	1.14×	**25.1**/24.3	**927**/2654

(c); the per-block decisions that produce them and the opacity and scale statistics of the population are in Fig. 4 (Section D.2), the final populations in Fig. 5 (Section D.3). The learned schedules are interpretable and differ by backend in a way that tracks the cost of a Gaussian. Where computing/optimizing a Gaussian is expensive, PACE densifies sparingly, freezes growth early, skips the periodic opacity resets (whose dips mark the fixed schedule's opacity and PSNR curves) and spends its iterations optimizing the primitives it already has: the 3DGS policy stretches the densification interval to its longest setting, prunes more aggressively and ends at half the fixed schedule's size with a compact set of opaque splats where the fixed schedule assembles twice as many small, semi-transparent ones; the DashGaussian policy densifies steadily but far more slowly than the schedule, letting the inherent coarse-to-fine resolution schedule of the backend supply the early quality and adding Gaussians only as fast as the rising render resolution can use them. The smaller models make every remaining iteration cheaper (by roughly a third on 3DGS and a half on Dash), which is where the wall-clock is won, and PACE-3DGS still ends at higher quality. Where a Gaussian is cheap (Faster-GS) PACE does the opposite: it lowers the densification threshold, prunes less, raises the appearance learning rates while damping position and opacity, accepts a modestly higher cost per iteration and converts the extra capacity into quality at matched wall-clock.

5.2 TIME-TO-TARGET ACCELERATION

Table 2 reports the time-to-target speed-up, final quality, and model size of PACE against each backend's fixed schedule on the eight held-out scenes (per-target times in Section D.1). PACE reaches every shared quality target faster than the fixed schedule in all 24 scene×backend cases, by at least $1.2\times$ in 20 of them and by $1.43\times$ in geometric mean. The gains are largest where the fixed schedule is least efficient: on `train` and `courthouse` PACE reaches the same quality about twice as fast on 3DGS and DashGaussian (up to $2.11\times$), and on `ignatius`, where the fixed schedules plateau, PACE-3DGS and PACE-Dash arrive twice as fast and outperform in reconstruction quality. Where a Gaussian is expensive (3DGS, DashGaussian) PACE builds smaller models so every remaining iteration is cheaper, and on the unbounded scenes, where the fixed 3DGS schedule runs into the Gaussian/VRAM cap and stops early, the compact models also win on quality ($+0.8$ to $+1.7$ dB). Where a Gaussian is cheap (Faster-GS) PACE does the opposite and grows past the schedule, converting the rasterizer's headroom into quality: it ends ahead on five of the six Tanks&Temples scenes, by up to 1.5 dB. The speed-up thus comes with a smaller model or a better one, and often both; the one exception is `meetingroom`, where the compact 3DGS and Dash models trail the schedule by under a dB at the end of the run. More detailed timing analysis of the proposed approach can be found at (Section C.7). Performance on some of the training scenes is shown in Section D, showing similar trends, thus confirming that PACE does not memorize on the training set. Experiments with different numbers of training views are at (Section D.8).

5.3 CROSS-BACKEND TRANSFER

Because all policies share the base state and action set, a controller trained on one backend can be run *unchanged* on another. The most demanding case is LeGS (Ning et al., 2026), whose learned per-Gaussian controller takes over the role of our densify and prune thresholds while our policy keeps the densification interval, learning rates and opacity resets.

Table 3 runs the three trained policies on LeGS on held-out `train`. *All three* accelerate LeGS over its native schedule, by 1.6–2.0× in geometric mean and at every target for the 3DGS and Dash policies; the 3DGS policy does so while matching the native quality with a third fewer Gaussians, the Faster-GS policy additionally unlocks a quality level the native schedule never reaches, and the Dash policy is the fastest of the three ($2.02\times$) while ending 0.5 dB ahead. Block-level scheduling and primitive-level learned density control therefore compose rather than conflict, and the composition is robust to which backend the schedule was learned on. Transfers between the three heuristic-densification backends are shown in Table Table 16, Appendix Section D.6. Deployment-side consequences of the learned schedules (model size on disk, peak training VRAM, render speed, LPIPS) and the cost of training and running the controller are reported in Section D.9.

Table 3: **Zero-shot transfer onto LeGS**: each trained policy run unchanged on LeGS vs. LeGS's native controller and schedule; Gaussians and test PSNR at 30k (PACE/LeGS), time-to-target speed-up per target and in geometric mean. Colour code as in Table 2: **teal** for a geometric-mean speed-up $\geq 1.4\times$ or a PACE model $\geq 20\%$ smaller, **orange** for PACE ≥ 0.3 dB ahead.

Policy (trained on)	3DGS	Faster-GS	Dash
At 30k iterations (PACE/LeGS)			
Gaussians (k)	**399**/609	720/642	**482**/607
test PSNR (dB)	**20.86**/20.83	**21.35**/20.87	**20.86**/20.34
Speed-up vs. LeGS's native schedule			
16 dB	1.54×	3.05×	3.31×
17 dB	1.84×	2.27×	2.40×
18 dB	2.10×	1.27×	1.82×
19 dB	2.41×	1.00×	1.75×
20 dB	1.52×	1.27×	1.33×
Geometric mean	**1.86×**	**1.62×**	**2.02×**

6 CONCLUSION

We presented PACE, a reinforcement-learning agent that treats the 3DGS training schedule as a block-wise control problem over a pluggable trainer backend. One backend-agnostic formulation accelerates every backend we tested, reaching each quality target faster than the backend's own fixed schedule, and it does so by learning a different schedule for each: it compacts where a Gaussian is expensive and grows the model where a Gaussian is cheap. PACE transfers zero-shot to held-out scenes, including unbounded ones, and across backends, composing also with learned per-Gaussian density control. We plan future work on budget-conditioned applications such as streaming Ren et al. (2026) and dynamic scenes (4DGS) Wu et al. (2024a). To meet the efficiency requirements for real-time 3DGS, model compression is a key problem, where the policy could include actions on parameter quantization or SH degree Ramlot et al. (2026); Chen et al. (2026).

REFERENCES

Marcin Andrychowicz, Misha Denil, Sergio Gomez, Matthew W Hoffman, David Pfau, Tom Schaul, Brendan Shillingford, and Nando De Freitas. Learning to learn by gradient descent by gradient descent. *Advances in neural information processing systems*, 29, 2016.

Bowen Baker, Otkrist Gupta, Nikhil Naik, and Ramesh Raskar. Designing neural network architectures using reinforcement learning. In *ICLR*, 2017.

Jonathan T Barron, Ben Mildenhall, Dor Verbin, Pratul P Srinivasan, and Peter Hedman. Mip-nerf 360: Unbounded anti-aliased neural radiance fields. In *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 5460–5469. IEEE, 2022.

Irwan Bello, Barret Zoph, Vijay Vasudevan, and Quoc V Le. Neural optimizer search with reinforcement learning. In *ICML*, 2017.

David Charatan, Sizhe Lester Li, Andrea Tagliasacchi, and Vincent Sitzmann. pixelsplat: 3d gaussian splats from image pairs for scalable generalizable 3d reconstruction. In *2024 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 19457–19467. IEEE, 2024.

Jiarui Chen, Yikeng Chen, Yingshuang Zou, Ye Huang, Peng Wang, Yuan Liu, Yujing Sun, and Wenping Wang. Megsˆ{2}: Memory-efficient gaussian splatting via spherical gaussians and unified pruning. In *International Conference on Learning Representations*, volume 2026, pp. 79917–79940, 2026.

Tianlong Chen, Xiaohan Chen, Wuyang Chen, Howard Heaton, Jialin Liu, Zhangyang Wang, and Wotao Yin. Learning to optimize: A primer and a benchmark. *JMLR*, 23(189):1–59, 2022.

Youyu Chen, Junjun Jiang, Kui Jiang, Xiao Tang, Zhihao Li, Xianming Liu, and Yinyu Nie. Dashgaussian: Optimizing 3d gaussian splatting in 200 seconds. In *2025 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 11146–11155. IEEE, 2025.

Yuedong Chen, Haofei Xu, Chuanxia Zheng, Bohan Zhuang, Marc Pollefeys, Andreas Geiger, Tat-Jen Cham, and Jianfei Cai. Mvsplat: Efficient 3d gaussian splatting from sparse multi-view images. In *European conference on computer vision*, pp. 370–386. Springer, 2024.

Christian Daniel, Jonathan Taylor, and Sebastian Nowozin. Learning step size controllers for robust neural network training. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 30, 2016.

Zhiwen Fan, Kevin Wang, Kairun Wen, Zehao Zhu, Dejia Xu, and Zhangyang Wang. Lightgaussian: Unbounded 3d gaussian compression with 15x reduction and 200+ fps. *Advances in neural information processing systems*, 37:140138–140158, 2024.

Ben Fei, Jingyi Xu, Rui Zhang, Qingyuan Zhou, Weidong Yang, and Ying He. 3d gaussian splatting as a new era: A survey. *IEEE Transactions on Visualization and Computer Graphics*, 31(8): 4429–4449, 2024.

Florian Hahlbohm, Linus Franke, Martin Eisemann, and Marcus Magnor. Faster-gs: Analyzing and improving gaussian splatting optimization. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 18946–18957, June 2026.

Samantha Hansen. Using deep q-learning to control optimization hyperparameters. *arXiv preprint arXiv:1602.04062*, 2016.

Alex Hanson, Allen Tu, Geng Lin, Vasu Singla, Matthias Zwicker, and Tom Goldstein. Speedysplat: Fast 3d gaussian splatting with sparse pixels and sparse primitives. In *2025 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 21537–21546. IEEE, 2025.

Peter Hedman, Julien Philip, True Price, Jan-Michael Frahm, George Drettakis, and Gabriel Brostow. Deep blending for free-viewpoint image-based rendering. *ACM Transactions on Graphics (ToG)*, 37(6):1–15, 2018.

Timothy Hospedales, Antreas Antoniou, Paul Micaelli, and Amos Storkey. Meta-learning in neural networks: A survey. *IEEE transactions on pattern analysis and machine intelligence*, 44(9): 5149–5169, 2021.

Arnold Irschara, Christopher Zach, Jan-Michael Frahm, and Horst Bischof. From structure-from-motion point clouds to fast location recognition. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 2599–2606. IEEE, 2009.

Max Jaderberg, Valentin Dalibard, Simon Osindero, Wojciech M Czarnecki, Jeff Donahue, Ali Razavi, Oriol Vinyals, Tim Green, Iain Dunning, Karen Simonyan, Chrisantha Fernando, and Koray Kavukcuoglu. Population based training of neural networks. *arXiv preprint arXiv:1711.09846*, 2017.

Ying Jiang, Chang Yu, Tianyi Xie, Xuan Li, Yutao Feng, Huamin Wang, Minchen Li, Henry Lau, Feng Gao, Yin Yang, et al. Vr-gs: A physical dynamics-aware interactive gaussian splatting system in virtual reality. In *ACM SIGGRAPH 2024 conference papers*, pp. 1–1, 2024.

Hadi S Jomaa, Josif Grabocka, and Lars Schmidt-Thieme. Hyp-RL: Hyperparameter optimization by reinforcement learning. *arXiv preprint arXiv:1906.11527*, 2019.

Nikhil Keetha, Jay Karhade, Krishna Murthy Jatavallabhula, Gengshan Yang, Sebastian Scherer, Deva Ramanan, and Jonathon Luiten. SplaTAM: Splat, track & map 3d gaussians for dense RGB-D SLAM. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2024.

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

Bernhard Kerbl, Andreas Meuleman, Georgios Kopanas, Michael Wimmer, Alexandre Lanvin, and George Drettakis. A hierarchical 3d gaussian representation for real-time rendering of very large datasets. *ACM Transactions on Graphics*, 43(4), 2024.

Shakiba Kheradmand, Daniel Rebain, Gopal Sharma, Weiwei Sun, Yang-Che Tseng, Hossam Isack, Abhishek Kar, Andrea Tagliasacchi, and Kwang Moo Yi. 3d gaussian splatting as markov chain monte carlo. *Advances in Neural Information Processing Systems*, 37:80965–80986, 2024.

Arno Knapitsch, Jaesik Park, Qian-Yi Zhou, and Vladlen Koltun. Tanks and temples: Benchmarking large-scale scene reconstruction. *ACM Transactions on Graphics (ToG)*, 36(4):1–13, 2017.

Ke Li and Jitendra Malik. Learning to optimize. *arXiv preprint arXiv:1606.01885*, 2016.

Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *JMLR*, 18(185):1–52, 2018.

Jiaqi Lin, Zhihao Li, Xiao Tang, Jianzhuang Liu, Shiyong Liu, Jiayue Liu, Yangdi Lu, Xiaofei Wu, Songcen Xu, Youliang Yan, and Wenming Yang. VastGaussian: Vast 3d gaussians for large scene reconstruction. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2024.

Yang Liu, He Guan, Chuanchen Luo, Lue Fan, Junran Peng, and Zhaoxiang Zhang. CityGaussian: Real-time high-quality large-scale scene rendering with gaussians. In *European Conference on Computer Vision (ECCV)*, 2024.

Saswat Subhajyoti Mallick, Rahul Goel, Bernhard Kerbl, Markus Steinberger, Francisco Vicente Carrasco, and Fernando De La Torre. Taming 3dgs: High-quality radiance fields with limited resources. In *SIGGRAPH Asia 2024 Conference Papers*, pp. 1–11, 2024.

Hidenobu Matsuki, Riku Murai, Paul H. J. Kelly, and Andrew J. Davison. Gaussian splatting SLAM. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2024.

Lars Mescheder, Wei Dong, Shiwei Li, Xuyang Bai, Marcel Santos, Peiyun Hu, Bruno Lecouat, Mingmin Zhen, Amaël Delaunoy, Tian Fang, et al. Sharp monocular view synthesis in less than a second. In *International Conference on Learning Representations*, volume 2026, pp. 24192–24230, 2026.

Luke Metz, James Harrison, C Daniel Freeman, Amil Merchant, Lucas Beyer, James Bradbury, Naman Agrawal, Ben Poole, Igor Mordatch, Adam Roberts, and Jascha Sohl-Dickstein. VeLO: Training versatile learned optimizers by scaling up. *arXiv preprint arXiv:2211.09760*, 2022.

Zhenhua Ning, Xin Li, Jun Yu, Guangming Lu, Yaowei Wang, and Wenjie Pei. Beyond heuristics: Learnable density control for 3d gaussian splatting. *arXiv preprint arXiv:2605.00408*, 2026.

Shenhan Qian, Tobias Kirschstein, Liam Schoneveld, Davide Davoli, Simon Giebenhain, and Matthias Nießner. Gaussianavatars: Photorealistic head avatars with rigged 3d gaussians. In *2024 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 20299–20309. IEEE, 2024.

Mohammad Nomaan Qureshi, Sparsh Garg, Francisco Yandun, David Held, George Kantor, and Abhisesh Silwal. SplatSim: Zero-shot sim2real transfer of RGB manipulation policies using Gaussian splatting. *arXiv preprint arXiv:2409.10161*, 2024.

Bert Ramlot, Martijn Courteaux, Peter Lambert, and Glenn Van Wallendael. Potr: Post-training 3dgs compression. *IEEE Transactions on Circuits and Systems for Video Technology*, 2026.

Shiwei Ren, Tianci Wen, Yongchun Fang, and Biao Lu. Fastgs: Training 3d gaussian splatting in 100 seconds. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 26094–26103, 2026.

John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, 2015.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical Bayesian optimization of machine learning algorithms. In *NeurIPS*, 2012.

Stanislaw Szymanowicz, Chrisitian Rupprecht, and Andrea Vedaldi. Splatter image: Ultra-fast single-view 3d reconstruction. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 10208–10217, 2024.

Sebastian Thrun and Lorien Pratt. Learning to learn: Introduction and overview. In *Learning to learn*, pp. 3–17. Springer, 1998.

Olga Wichrowska, Niru Maheswaranathan, Matthew W Hoffman, Sergio Gomez Colmenarejo, Misha Denil, Nando Freitas, and Jascha Sohl-Dickstein. Learned optimizers that scale and generalize. In *International conference on machine learning*, pp. 3751–3760. PMLR, 2017.

Guanjun Wu, Taoran Yi, Jiemin Fang, Lingxi Xie, Xiaopeng Zhang, Wei Wei, Wenyu Liu, Qi Tian, and Xinggang Wang. 4d gaussian splatting for real-time dynamic scene rendering. In *2024 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 20310–20320. IEEE, 2024a.

Guanjun Wu, Taoran Yi, Jiemin Fang, Lingxi Xie, Xiaopeng Zhang, Wei Wei, Wenyu Liu, Qi Tian, and Xinggang Wang. 4d gaussian splatting for real-time dynamic scene rendering. In *2024 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 20310–20320. IEEE, 2024b.

Tianyi Xie, Zeshun Zong, Yuxing Qiu, Xuan Li, Yutao Feng, Yin Yang, and Chenfanfu Jiang. Physgaussian: Physics-integrated 3d gaussians for generative dynamics. In *2024 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 4389–4398. IEEE, 2024.

Zhen Xu, Andrew M Dai, Jonas Kemp, and Luke Metz. Learning an adaptive learning rate schedule. *arXiv preprint arXiv:1909.09712*, 2019.

Yunzhi Yan, Haotong Lin, Chenxu Zhou, Weijie Wang, Haiyang Sun, Kun Zhan, Xianpeng Lang, Xiaowei Zhou, and Sida Peng. Street Gaussians: Modeling dynamic urban scenes with Gaussian splatting. In *European Conference on Computer Vision (ECCV)*, 2024.

Xiaoyu Zhou, Zhiwei Lin, Xiaojun Shan, Yongtao Wang, Deqing Sun, and Ming-Hsuan Yang. DrivingGaussian: Composite Gaussian splatting for surrounding dynamic autonomous driving scenes. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2024.

Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.

AI USE STATEMENT

In this work, we used generative AI tools (LLM-based coding and writing assistants) to help implement and debug the evaluation, plotting, and data-processing scripts and the accompanying demo website, to draft and edit prose, and to typeset tables and figures. We have not used generative AI tools to generate research ideas, to design the method or the experiments, or to produce, select, or alter experimental results; every controller, rollout, and measurement reported here was produced by the code and hardware described in Section C. We have reviewed all AI-assisted work: LLM-generated code was checked and tested by the authors, and all AI-edited text was verified against the underlying results. We take responsibility for the final content of this work, including text, claims, or artifacts produced with the aid of generative AI.

A POLICY OPTIMIZATION AND ALGORITHM

We optimize $J(\psi)$ equation 10 with PPO (Schulman et al., 2017). Given rollouts under $\pi_{\psi_{\text{old}}}$, let $\varrho_k(\psi) = \pi_\psi(a_k \mid s_k)/\pi_{\psi_{\text{old}}}(a_k \mid s_k)$ be the likelihood ratio. The advantage is the generalized advantage estimate (Schulman et al., 2015), computed backwards over the blocks of a rollout from the value baseline V_ψ, let $\delta_k = r_k + \gamma V_\psi(s_{k+1})(1-d_k) - V_\psi(s_k)$, be a one-step temporal difference error, the generalized advantage is

$$\hat{A}_k = \delta_k + \gamma\lambda_{\text{GAE}}(1-g_k)\,\hat{A}_{k+1} = \sum_{l=0}^{K-k} (\gamma\lambda_{\text{GAE}})^l \delta_{k+l}, \tag{12}$$

where $g_k = 1$ if the episode ends at block k and $\hat{A}_K = 0$; the trace parameter λ_{GAE} trades the variance of the empirical return against the bias of the critic, and a rollout cut mid-episode bootstraps from V_ψ of its last state rather than being treated as terminal. The value target is $\hat{R}_k = \hat{A}_k + V_\psi(s_k)$, and advantages are standardized within each batch before the update, which keeps the clip range meaningful as the reward scale changes. PPO maximizes the clipped surrogate with value and entropy terms,

$$\mathcal{L}(\psi) = \mathbb{E}_k\Big[\min\big(\varrho_k(\psi)\hat{A}_k,\ \text{clip}(\varrho_k(\psi), 1-\epsilon, 1+\epsilon)\hat{A}_k\big) - c_v\big(V_\psi(s_k) - \hat{R}_k\big)^2 + c_e\,\mathcal{H}\big[\pi_\psi(\cdot \mid s_k)\big]\Big], \tag{13}$$

with clip range ϵ, target-KL early stopping, and rewards normalized per scene by a running z-score to stabilize learning across scenes of very different difficulty; all optimizer, network, and environment settings are listed in Section C. Training algorithm is summarized in Algorithm 2

Algorithm 2 Training the acceleration controller (PPO)

Require: scenes $\{\mathcal{D}\}$, backend $\mathcal{B}$, time-value scale τ, discount γ, λ_{GAE}, weights c_v, c_e, w_t

1: initialize policy/value parameters ψ
2: **for** update $= 1, \ldots, U$ **do**
3: **for** each rollout step (collect a batch of blocks) **do**
4: on episode start: sample a scene $\mathcal{D}$, init $\mathcal{G}_0$, set $t_0 = 0$
5: build s_k via Eq. (4); sample $a_k \sim \pi_\psi(\cdot \mid s_k)$ ▷ Eq. (11)
6: $\mathcal{G}_{k+1} \leftarrow \mathcal{T}_{a_k}(\mathcal{G}_k; \mathcal{D})$; $t_{k+1} \leftarrow t_k + \Delta t_k$; compute r_k ▷ Eqs. (7) and (9)
7: terminate episode at the iteration cap (or at the stop gate, Section C.2)
8: compute $\hat{A}_k, \hat{R}_k$ (GAE); update ψ by maximizing Eq. (13) for several epochs
9: select $\psi^\star$ by the held-out probe score of Section C.4 (test PSNR at fixed wall-clocks)

B TRAINER BACKENDS AND THE TRANSITION FUNCTION

The training operator $\mathcal{T}_{a_k}$ of Eq. (2) is implemented once, in the environment, on top of a small backend interface; a backend supplies the model, the rasterizer and the optimizer, and the environment supplies everything else (block loop, controls, safety rails, timing, state and reward). This appendix gives the operator step by step, the interface a backend must implement, and what each of the four backends contributes.

B.1 THE BLOCK TRANSITION, STEP BY STEP

Given the solver state $\mathcal{G}_k = (\boldsymbol{\Theta}_k, \boldsymbol{m}_k)$ and the decoded controls $a_k = (n_k, g_k, c_k, p_k, \omega_k, \boldsymbol{\theta}_k)$ (Eq. (5) and Table 1), one block proceeds as follows.

1. *Opacity reset* (before the timer starts). If ω_k is `force`, or `plt` and the previous block's held-out quality gain was below the plateau tolerance, all opacities are set to $\min(o_i,\, 0.01)$ (the stock 3DGS `reset_opacity`), subject to the rate limits of Section C.2.
2. n_k *gradient steps.* For each iteration i of the block:
   (a) *Learning rates.* The backend's own schedule is evaluated (the position learning rate follows its exponential decay $\eta_{\text{pos}}(i)$), then the block's multipliers are applied: $\eta_{\text{pos}} \leftarrow \theta_{\text{pos}}\, \eta_{\text{pos}}(i)$ scales the scheduled value, and the colour (DC and higher SH bands share one factor), opacity, scale and rotation rates are set to $\theta_j\, \eta_j$ with η_j the stock constants. Multipliers are re-applied at every iteration of the block and do not carry over to the next block.
   (b) *SH schedule.* One SH band is unlocked every 1000 iterations up to degree 3 (Faster-GS's optional `sh_unlock` head can hold this, but it is not used by the paper's controllers).
   (c) *Render and loss.* A training camera is drawn without replacement from a reshuffled stack; the backend renders it; the loss is $(1-\lambda)\mathcal{L}_1 + \lambda\, \mathcal{L}_{\text{D-SSIM}}$ (Eq. (1)) against the ground truth (on DashGaussian at the scheduled resolution, Section B.5) and back-propagated.
   (d) *Densification statistics.* While $i < i_{\text{until}}$ (15k), the backend accumulates its per-Gaussian growth statistic (the view-space positional gradient norm for 3DGS/Dash, the rasterizer's 2-D mean gradient for Faster-GS) and the maximum screen-space radius.
   (e) *ADC event.* If $i > i_{\text{from}}$ (500) and $i \bmod c_k = 0$, the backend's densify-and-prune step runs with the controlled thresholds: clone (small Gaussians) and split (large Gaussians, into two, scales divided by 1.6, parents removed) for Gaussians whose statistic exceeds $2\times10^{-4}\cdot m(g_k)\cdot\theta_\nabla$ with $m(\texttt{cons}, \texttt{def}, \texttt{aggr}) = (2, 1, \frac{1}{2})$, skipped entirely when $g_k = \texttt{off}$; then pruning according to p_k: none, opacity $< \theta_{\text{op}}$, or additionally screen radius > 20 px (after iteration 3000) or world extent > 0.1 of the scene radius. The prune floor of Section C.2 (at most 25% of the population per event, never below $\max(50\text{k}, 0.5N_0)$) replaces the mask by "keep the most opaque" if it would be violated, and θ_{op} is capped at 0.005 for 300 iterations after a reset.
   (f) *Optimizer step* on the backend's optimizer (Adam variants, Table 4), then gradients are zeroed.
3. *Clock.* Δt_k is the wall-clock of step 2 alone, measured with a CUDA synchronization at the end of the block; $t_{k+1} = t_k + \Delta t_k$. The opacity reset, the validation renders that build s_{k+1} and the reward, and any evaluation renders are outside the timer for the agent and the fixed schedule alike.
4. *Observation and reward.* The validation cameras are rendered at full resolution, the population statistics (opacity and scale quantiles, visible fraction, ...) are computed on the full model, and s_{k+1} and r_k follow Eqs. (4) and (9). The block ends early only if the iteration cap is reached, the loss becomes non-finite, or the Gaussian/VRAM hard caps are exceeded.

The stochasticity of the kernel is the camera order (and, on LeGS, the sampled per-Gaussian decisions); everything else is a deterministic function of $(\mathcal{G}_k, a_k)$.

B.2 BACKEND INTERFACE

A backend is a class that names its repository and rasterizer and implements six hooks: `make_scene` and `make_gaussians` (construct the scene, cameras and the Gaussian model from the SfM points), `render_training` (render one camera and return the image, per-Gaussian radii and visibility, and whatever tensor the densifier needs), `accumulate_densification_stats` (fold that tensor into the model's growth statistics), `render_image` (render for validation and evaluation), and the densify-and-prune entry point (the environment patches the stock 3DGS one onto models that lack it). Optional attributes tell the environment that the backend brings its own resolu-

tion schedule (DashGaussian) or its own density controller (LeGS). Everything in Section B.1 except the hooks is shared, which is what makes the state, action, reward and policy backend-agnostic. Exactly one backend is active per process (its repository is placed first on the import path).

Table 4: The four backends: what they contribute to $\mathcal{T}$, and which controls of Eqs. (5) and (6) are effective on each (✓ effective, (✓) effective with the noted restriction, × no-op).

	3DGS	Faster-GS	DashGaussian	LeGS
code base	official `gaussian-splatting`	Faster-GS fork of 3DGS with its CUDA kernels	3DGS code + our re-implementation of the Dash schedules	`AaronNZH/LeGS`
rasterizer	stock `diff-gaussian-rasterization`	faster rasterizer (fused kernels)	as 3DGS	LeGS's `fastgs` rasterizer
optimizer	Adam	fused Adam	Adam	Adam + separate lazy SH Adam
growth statistic	view-space $\|\nabla_{xy}\|$	kernel 2-D mean gradient	view-space $\|\nabla_{xy}\|$	LeGS per-Gaussian state features
density control	stock clone / split / prune	stock clone / split / prune	stock, capped by the momentum budget	learned actor (keep / clone / split) + prune estimator
training resolution	fixed	fixed	coarse-to-fine schedule	fixed
block length	✓	✓	✓	✓
densify mode	✓	✓	(✓) budget may veto growth	(✓) on/off gate only
densify interval	✓	✓	✓	✓
densify threshold ×	✓	✓	✓	✓ scales LeGS's candidate thresholds
prune mode / threshold	✓	(✓) size test uses visibility counts	✓	× LeGS owns deletion
opacity reset	✓	✓	✓	✓
learning-rate multipliers	✓	✓	✓	✓

B.3 Official 3DGS

The reference backend wraps the official implementation (commit `54c035f`) unchanged: the `diff-gaussian-rasterization` kernels, dense Adam, the stock `densify_and_clone` / `densify_and_split` rules (clone if the largest scale is below `percent_dense`× scene extent, otherwise split), and the stock prune. Our only additions are the controlled entry point that receives the block's thresholds and modes, and the prune floor. With the neutral action of Section 4.3 the operator is the stock training loop.

B.4 Faster-GS

Faster-GS replaces the rasterizer and the optimizer by the fused kernels of Hahlbohm et al. (2026) (fork of the 3DGS code base, commit `f87071e`); the rest of the loop is the 3DGS loop, so each Gaussian is cheaper to render and update, which is what changes the cost structure the controller adapts to. Two details matter for the transition: (i) the growth statistic is the rasterizer's per-Gaussian 2-D mean gradient (`densification_info`) rather than the autograd view-space gradient, and the visibility count plays the role of the radii, so the screen-size prune criterion is a visibility test on this backend; (ii) in the fork used, the positional autograd gradient is anomalous on synthetic Blender scenes (both agent and fixed schedule diverge there), whereas real COLMAP scenes train normally; all Faster-GS results in the paper are on COLMAP scenes. Sparse Adam and the optional anti-aliasing and SH-unlock heads are not used by the paper's Faster-GS controller, which drives the base action set of Eq. (5).

B.5 DashGaussian

DashGaussian (Chen et al., 2025) is itself a training accelerator: it renders training views at a coarse-to-fine resolution and caps densification with a momentum primitive budget. Its released code is not vendored (licence); the two schedules are re-implemented on the 3DGS backend from the paper's description and drive the same rasterizer, optimizer and ADC. *Resolution schedule.* From the training images a radial power spectrum gives the fraction f of low-frequency energy, which sets $\gamma = \mathrm{clip}(1 + 2f,\, 1,\, 3)$. Scales s_j are spaced linearly from $s_{\min} = 1/8$ to 1 over 32 levels, level j being reached at iteration $i_j = 0.9\,K\,(j/31)^{\gamma}$ with K the curriculum horizon, pinned to 30k in training and evaluation (Section C.6); between levels the inverse squared scale is interpolated linearly, so the render scale at iteration i is $s(i) = \big(\mathrm{lerp}(s_j^{-2}, s_{j+1}^{-2};\, i)\big)^{-1/2}$, clamped to $[s_{\min}, 1]$. Training cameras and ground truth are downsampled to $s(i)$ (bilinear, alpha masks included); validation and evaluation always render at full resolution, and $s(i)$ is exposed to the state as the active resolution. *Momentum budget.* With N_i the current count and $m_0 = N_0$, the budget evolves every iteration of the densification window as $m_{i+1} = \max\big(m_i,\ 0.98\,m_i + \min(10^6,\ 0.2\,N_i\,s(i)^2)\big)$ and growth is permitted only while $N_i < m_i$; because the allowance scales with $s(i)^2$, growth is strongly suppressed in the low-resolution phase. The policy cannot override the resolution or the budget: when the budget binds, the ADC event prunes but does not grow whatever the densify mode says. Block length, ADC cadence and thresholds, prune mode, opacity resets and learning rates remain the policy's. The fixed-schedule baseline on this backend is the neutral action run under the same two schedules, i.e. DashGaussian's own accelerated trainer.

B.6 LeGS

LeGS (Ning et al., 2026) (commit `8eb120b`) replaces the ADC heuristic by a learned per-Gaussian controller: at a densification event it computes per-Gaussian state features (ten sampled cameras, one forward/backward pass each, plus a per-Gaussian error attribution), an actor samples keep/clone/split for every candidate whose gradient statistic passes LeGS's candidate thresholds, and a prune estimator decides deletions against a cosine-annealed opacity floor. Its own reward is delayed: 50 iterations after an event the per-Gaussian metric is recomputed on the same cameras, children are scatter-added onto their parents, and the controller is updated once two events have been collected (or kept frozen in inference mode). We keep this machinery intact and only change when it is invoked. Our `densify` head becomes an on/off gate of the event (`cons`/`aggr` have no extra effect), our interval sets its cadence, and the continuous threshold multiplier rescales LeGS's two candidate thresholds (2×10^{-4} and 1.2×10^{-3}); when the Gaussian hard cap is hit the thresholds are set to infinity so only pruning survives. Because LeGS's estimator owns deletion, our prune mode and prune threshold are no-ops on this backend; block length, learning-rate multipliers (LeGS's separate SH optimizer receives the colour factor) and opacity resets are effective. Two implementation details were needed: LeGS's delayed reward assumes a fixed event cadence (it fires on a modular-arithmetic condition), which our variable, policy-chosen cadence breaks, so events are tracked explicitly in a queue keyed on their firing iteration and settled when they mature or when a new event arrives early; and its `torch_scatter` dependency is replaced by an `index_add`-based shim. The LeGS baseline in Table 16 is LeGS's controller under its native cadence (neutral action), so the transfer results measure only what our scheduling adds on top of a learned density controller.

B.7 Fixed-schedule baselines and timing

For every backend the baseline executes the neutral action of Section 4.3 in the same environment: blocks of 100 iterations, stock densification every 100 iterations with unit thresholds from iteration 500 to 15k, opacity-and-size pruning at 0.005, plateau-gated opacity resets, unit learning rates. Agent and baseline therefore share the rasterizer, optimizer, ADC code, cameras, resolution, safety rails and clock; the only difference is the sequence of controls. Wall-clock is the sum of block training times of Section B.1, measured on the same otherwise idle GPU.

C IMPLEMENTATION DETAILS

This appendix lists every setting behind the three agents of the paper (the 3DGS, Faster-GS and DashGaussian agents), the environment they were trained in, the training data and its augmentation, checkpoint selection, and the evaluation protocol. Unless a column says otherwise, all three agents share a value.

C.1 POLICY NETWORK

The actor and the critic share one trunk: the 45-dimensional state (Eq. (4)) is mapped by three fully connected layers of width 256 with GELU activations to a feature z_k. Six categorical heads (one linear layer each) produce the logits of the discrete factors—block length (3 classes), densify mode (4), densify interval (4), prune mode (3) and opacity reset (3)—plus a binary stop head that is not part of the method as evaluated (see below). A linear layer outputs the 7-dimensional mean of the continuous action; the log standard deviation is a state-independent parameter vector initialized at -0.5 (initial $\sigma = 0.61$) and clamped to $[-5, 2]$ at every forward pass. The distribution over $\tilde{\boldsymbol{\theta}}_k$ is an unsquashed diagonal Gaussian; squashing happens only in the decoding of Eq. (6), with the ranges of Table 5. A linear value head on the same trunk outputs V_ψ. The network has 150,306 parameters (143,360 in the trunk). At inference (probes and all evaluations) actions are taken greedily: arg-max logits and the Gaussian mean.

Stop head (present but not used) The network also carries a sixth binary head, `continue`/`stop`, intended to let the policy end an episode once further training no longer paid for its wall-clock. It was trained with the rest of the policy (a stop was honoured only after 3000 iterations, and the reward's time cost then made the trained policies terminate almost always right at that gate), but it did not learn a useful, state-dependent criterion: on held-out scenes the head emits `stop` at every block from the first one, so where an episode ends is set entirely by the gate. We therefore report it only for completeness: in every experiment of the paper the head is overridden and the policy is run to the iteration cap, exactly like the fixed schedule, and none of the results depends on it.

Table 5: Continuous controls: decoding ranges and defaults (Eq. (6)).

Multiplier	**range** $[\ell_j, u_j]$	**scale**	**fixed-schedule value**
densification gradient threshold	$[0.25,\ 4]$	log	1
pruning opacity threshold (absolute)	$[0.001,\ 0.02]$	linear	0.005
position learning rate	$[0.25,\ 4]$	log	1
SH / colour, opacity, scale, rotation learning rates	$[0.25,\ 2]$ each	log	1

State normalization No running statistics are used; every entry is divided by a fixed reference so that it lies in $[0, 1]$ and is then clipped: iteration by the episode cap (7000), PSNR by 50 dB, time per iteration by 0.2 s, validation seconds per view by 0.5 s, VRAM by the 20 GB target, Gaussian counts by the 3 M hard cap, the block index by (cap/100); signed quantities (loss slope, quality change) pass through $\frac{1}{2} + \frac{1}{2}\tanh(\cdot)$. Three channels are constant in the acceleration setting and therefore uninformative: the LPIPS slot is fixed to 0 (LPIPS is not computed during training) and the elapsed/remaining wall-clock fractions saturate because no episode time cap is used (T disabled).

C.2 ENVIRONMENT AND TRAINING EPISODES

Episodes run on the backend's own optimizer with the stock 3DGS hyperparameters (Table 7); the policy only modulates them through the action. An episode initializes the Gaussians from the scene's SfM points, runs at most 7000 iterations or ends earlier when the (unused) stop head fires, which is allowed from iteration 3000. No wall-clock cap is imposed. Twelve validation cameras are drawn from the training cameras by farthest-point sampling over camera centres (at most one fifth of the training cameras) and excluded from training; they feed the state and the reward, so test cameras are never seen. In practice training episodes ended almost always right after the 3000-iteration gate (median 13 blocks, end at 3000–3100 iterations, range 2550–3950), because the stop head fires as soon as it is allowed (Section C.1).

Table 6: PPO settings (Eq. (13)) and training budget of the three agents.

	3DGS	DashGaussian	Faster-GS
optimizer	Adam, learning rate 3×10^{-4} (constant), $\beta = (0.9, 0.999)$, $\epsilon_{\text{Adam}} = 10^{-8}$		
clip range ϵ	0.2		
value loss weight c_v / value clip	0.5 / 0.2 (max of clipped and unclipped value loss)		
entropy weight c_e	0.003 (one coefficient on the summed entropy of all heads)		
discount γ / GAE λ_{GAE}	0.99 / 0.95		
target KL (early stopping of the epochs)	0.03		
gradient-norm clip	0.5		
rollout length (blocks per update) / environments	32 / 1 (single sequential environment; episodes span updates)		
epochs per update / minibatch	6 (often cut by the KL criterion) / 32 (one minibatch)		
advantage normalization	batch z-score		
reward normalization	per-scene running z-score (Welford), standard-deviation floor 0.01		
bootstrap value at rollout end	0 if the rollout ended with a terminal block, else V_ψ		
seed	7 (Python, NumPy, PyTorch, CUDA, environment and scene RNGs)		
PPO updates	300	300	600
environment steps (blocks)	9,600	9,600	19,200
episodes completed	538	592	1,181
wall-clock (one RTX 3090, probes included)	$\approx$11.6 h	$\approx$7.5 h	$\approx$11.3 h

Table 7: Environment settings (identical for the three agents, except that the DashGaussian agent pins its backend's curriculum horizon to 30k, Section C.6) and the underlying 3DGS optimizer hyperparameters.

Episode	iteration cap 7000; wall-clock cap disabled; stop head honoured after 3000 iterations during training only; SH degree 3, one band unlocked every 1000 iterations
Cameras	12 validation cameras by farthest-point sampling ($\leq$ one fifth of the training views), test split `llffhold`= 8, training at native resolution of the pool images
Photometric loss	$(1-\lambda)\mathcal{L}_1 + \lambda\,\mathcal{L}_{\text{D-SSIM}}$, $\lambda = 0.2$; black background; no depth or exposure terms
Learning rates (fixed schedule)	position $1.6 \times 10^{-4} \to 1.6 \times 10^{-6}$ (exponential decay over the episode cap, delay multiplier 0.01), SH DC 2.5×10^{-3} (higher bands /20), opacity 0.025, scale 5×10^{-3}, rotation 10^{-3}; dense Adam
Densification (fixed schedule)	from iteration 500 to 15000, every 100 iterations, view-space gradient threshold 2×10^{-4}, `percent_dense` 0.01; screen-size pruning threshold 20 px after iteration 3000; opacity reset only when the policy asks (no periodic reset)

Reward constants The held-out quality is $q_i = \text{PSNR}_i + 10\,\text{SSIM}_i$ per validation view and $Q = 0.75\,\text{mean}_i q_i + 0.25\,\min_i q_i$ (the min-view term). The time value is $\nu(t) = e^{-t/\tau}$ with $\tau = 45$ s and multiplies only positive quality gains (losses are charged in full). The compactness discount is $d_k = 1/(1+\kappa\,e_k)$ with $\kappa = 2$ and excess $e_k = \max(0, N_k/N_{\text{ref}} - 1)$, $N_{\text{ref}} = 8\,N_0$ (N_0: SfM points). The time cost is $w_t \Delta t_k$ with $w_t = 0.05$ per second. The penalty term ρ_k collects: a count-excess penalty $1.0\,e_k$; a growth penalty $1.0\max(0, N_k - \max(N_{k-1}, N_{\text{ref}}))/N_{\text{ref}}$; a compaction bonus $-1.0\max(0, N_{k-1} - \max(N_k, N_{\text{ref}}))/N_{\text{ref}}$ granted only when the block did not lose quality; a VRAM penalty of 0.01 per GB of peak memory above 20 GB; an instability penalty $0.1\max(0, -\Delta q_k)$; and a fixed 2.0 penalty if the loss becomes non-finite. Episodes that reach the iteration cap receive a terminal bonus $0.1\,(Q_{\text{final}} - Q_0)\,d_K\,\nu(t_K) - 0.5\,e_K$. Rewards are then normalized per scene as in Table 6.

Safety rails Inside the training operator: hard cap of 3 M Gaussians and 24 GB VRAM (episode abort); a prune event may remove at most 25% of the Gaussians and never below $\max(50{,}000,\ 0.5\,N_0)$; opacity resets are allowed from iteration 500, at most once every 3 blocks and at most 5 per episode, and `plt` triggers only if the previous block's quality gain was ≤ 0.02; for 300 iterations after a reset the pruning opacity threshold is capped at the stock 0.005 (post-reset prune guard), which prevents a reset followed by an aggressive prune from wiping the model.

C.3 Training data and augmentation

All three agents are trained on the same ten real COLMAP scenes at the resolution of the released images: `truck` (Tanks & Temples), `drjohnson`, `playroom` (Deep Blending), `garden`, `kitchen`, `counter`, `room`, `bonsai`, `flowers`, `treehill` (Mip-NeRF 360). Scenes are vis-

ited one episode at a time; the 3DGS and DashGaussian agents cycle through the pool in a fixed round-robin order (≈ 54 and 59 passes), the Faster-GS agent draws from a shuffled pool that is reshuffled after every pass. The probe and test scene `train` is never in a pool.

Photometric augmentation (Faster-GS agent only) The Faster-GS agent's pool is enlarged to 410 scenes: the ten originals plus 50 photometric variants of each of eight scenes (all but `drjohnson` and `playroom`). A variant re-renders every image of a scene with one shared parameter draw, so it is multi-view consistent, and keeps the original camera intrinsics and SfM point cloud (the `sparse/` model is linked, not copied), so only appearance changes. Variant $v \in \{0, \dots, 49\}$ uses strength $s = 0.25 + 0.75\,v/49$, and each factor is drawn once per variant from a range scaled by s: per-channel white-balance gain $1 + \mathcal{U}(-0.09, 0.09)$, brightness $1 + \mathcal{U}(-0.16, 0.19)$, gamma $\exp \mathcal{U}(-0.29, 0.29)$, contrast $1 + \mathcal{U}(-0.19, 0.26)$, S-curve strength $\mathcal{U}(0, 0.32)$, saturation $1+\mathcal{U}(-0.26, 0.33)$, hue rotation $\mathcal{U}(-8°, 8°)$, vignetting $\mathcal{U}(0, 0.26)$, sharpening/blur $\mathcal{U}(-0.65, 0.65)$ (unsharp mask or Gaussian blur of radius ≤ 2), and additive Gaussian noise with $\sigma = \mathcal{U}(0, 2.5)/255$ per image; they are applied in this order and the result is stored as JPEG (quality 95). The sampled parameters of every variant are recorded in a manifest. Geometric augmentation was not used, since the state is (almost) invariant to rigid transformations of the scene.

C.4 Checkpoint selection

Every 10 PPO updates (15 for the Faster-GS agent) the current policy is rolled out greedily on the held-out scene `train` and its test PSNR (evenly spaced 30-view subset of the test cameras) is recorded the first time training wall-clock crosses 15, 30 and 60 s; the mean of the three values is the acceleration score, and the checkpoint with the highest score is kept (`best.pth`). For the 3DGS and DashGaussian agents a second stage re-ran the best few probe candidates to 30k iterations on `train` (agent only, stop head overridden) and picked the operating point with the best combination of time-to-target, quality and model size at 30k; for 3DGS this selected update 289 over the probe winner 279. The Faster-GS agent uses its probe winner (update 494) without a second stage. The DashGaussian agent needs a different rule, for a reason particular to that backend (Section C.6): its acceleration score is measured in the first 60 s, which on a coarse-to-fine substrate is entirely within the low-resolution phase, where withholding densification is genuinely the fastest way to raise early PSNR. The score therefore ranks a policy that never grows the model above one that does, even though the second is better at 30k. We consequently select the DashGaussian checkpoint by the second stage alone, over the candidates whose probe rollout actually densifies; this selected update 109, whose probe score (19.51) is within noise of the probe winner (19.55, update 119) but which reaches all six PSNR targets and ends 0.9 dB higher. The selection protocol is therefore not identical across agents, and it never uses the test cameras of any evaluation scene other than `train`, on which we report results as well (marked held-out throughout).

C.5 Evaluation protocol

Time-to-target runs use the same environment with the iteration cap raised to 30k (50k for the long-horizon curves), the stop head overridden (Section C.1) so that the policy, like the fixed schedule, runs to the cap and the full quality-vs-time curve is traced, and the position learning-rate decay stretched to the cap. Test PSNR/SSIM are measured on an evenly spaced 12-camera subset of the held-out test split at iterations $\{200, 500, 1000, 1500, 2000, 3000, 4000, 5000, 7000, 10000, 15000, 20000, 25000, 30000\}$; the time to a target is linearly interpolated between bracketing samples. Two clocks are recorded per block, both written to every run's `curve.csv`: the *train-loop* clock, which accumulates the gradient-step blocks alone, and the *deployment-inclusive* clock, which additionally charges every per-block cost a deployed run pays—the opacity reset, the feedback renders that build the next observation, the population statistics, the observation assembly and the actor forward pass. Test-curve renders are excluded from both and for both methods: they are measurement, not training. Section C.7 reports the breakdown and shows that the choice of clock changes the speed-ups by at most a few percent. The fixed-schedule baseline is the neutral action of Section 4.3 executed by the same backend. Evaluation resolutions: Tanks & Temples scenes at half resolution (`train` from its released 980×545 images; `barn`, `caterpillar`, `ignatius` downsampled by 2), Mip-NeRF 360

scenes (`stump`, `bicycle`) at the released quarter resolution (`images_4`). Timings are measured on one NVIDIA RTX 3090 with nothing else running on the GPU.

C.6 THE DASHGAUSSIAN AGENT: SCHEDULE HORIZON AND SELECTION

DashGaussian differs from the other backends in that its substrate is itself a schedule: training views are rendered coarse-to-fine, and the momentum primitive budget allows per-step growth in proportion to the square of the current render scale, so densification is suppressed while the render is coarse. Both curricula are defined over a training horizon. Keying that horizon to the RL episode length, as the other backends' settings implicitly do, makes the substrate change shape between training and evaluation: with the 7k episode cap used during policy training the resolution ramp completes at 6.3k, so an episode ending at 3k sits at 48% of the ramp, whereas under the 30k evaluation cap the ramp completes at 27k and iteration 3k is only 11% of the way up. Because the budget scales with the square of the scale, the growth headroom at the end of a training episode differs by about $6\times$ between the two settings. A policy trained that way is optimizing a different problem from the one it is scored on, and the symptom is unambiguous: it emits `densify=off` on every block of every held-out scene and never leaves the SfM initialization, which looks like an aggressive compaction strategy but is an artifact.

We therefore pin the DashGaussian curriculum to a fixed 30k horizon in both training and evaluation, so that a training episode is the first tenth of the deployment curriculum rather than half of a shorter one. This is the only setting that differs from the 3DGS and Faster-GS agents, and it is a property of the backend, not of the method: the other two have no resolution or budget curriculum to mismatch. With it the agent densifies, and reaches $2.11\times$ on `train` at $+0.4$ dB from a model $2.9\times$ smaller, against $1.43\times$ at $+0.1$ dB for the mismatched policy; on `stump` it moves from -2.3 dB to $+0.8$ dB against the fixed schedule. Two further observations are worth recording. Neither the training pool nor the complexity weights explain the collapse: retraining on the augmented pool used for Faster-GS, and relaxing the growth, count and compaction penalties together with the time-value horizon, both leave the behaviour unchanged, because those penalties are gated above a reference count the policy never approaches. And a policy that has already collapsed cannot be repaired by continued training—fine-tuning it under the corrected schedule leaves the model size at exactly the initialization while the action entropy keeps falling—so the corrected agent is trained from scratch.

C.7 AGENT OVERHEAD AND DEPLOYMENT-INCLUSIVE TIMING

Choosing an action is not free, and the cost is part of the method: the state of Eq. (4) is built from feedback renders on the held-out validation views and from statistics over the whole Gaussian population, and the action itself requires a forward pass through the actor. We therefore measure every per-block cost outside the gradient steps, with a CUDA synchronization around each, on held-out `train` (Table 8).

Table 8: **Individual time costs**, held-out `train` over the first 6k iterations, measured per backend for PACE and for the fixed schedule run through the same environment. "feedback" is the 12-view validation render that builds the next observation, "stats" the population statistics plus observation assembly, "policy" the actor forward pass (zero for the fixed schedule, which has no policy). The last column is the total of these as a fraction of the gradient-step time. PACE selects 200-step blocks, so it queries the agent half as often as the 100-step cadence of the fixed schedule and pays *less* overhead in both absolute and relative terms.

		Per block (ms)					Per run	
Backend	**Schedule**	**Steps**	**Reset**	**Feedback**	**Stats**	**Policy**	**Blocks**	**Overhead**
3DGS	PACE	3487	0.01	56.9	4.1	8.2	31	2.0%
	fixed	2468	0.13	73.7	4.0	0.0	60	3.2%
Faster-GS	PACE	2227	0.71	52.4	3.9	8.6	30	2.9%
	fixed	941	0.21	46.6	4.3	0.0	60	5.4%
DashGaussian	PACE	3221	0.01	63.2	4.4	6.9	30	2.3%
	fixed	1910	0.04	74.8	4.6	0.0	60	4.2%

The feedback render dominates (47–75 ms per block); the population statistics and observation assembly cost ≈ 4 ms and the actor forward pass 7–9 ms, so the policy itself is well under 1% of a

block. In total a deployed PACE run pays 2.0–2.9% on top of its gradient-step time, against 3.2–5.4% for the fixed schedule in the same harness. Block length is itself an action, so the agent could in principle have inflated this cost by querying itself more often, and nothing stopped it from doing so: the time cost of Eq. (9) charges gradient time only, so policy training never penalised a feedback render. It settled on the longest available block on every backend regardless, which is why its overhead is the smaller of the two.

Table 9 recomputes every time-to-target under three accountings: the *train-loop* clock used in Table 2; a *symmetric* clock charging each method its own overhead; and a *deployment* clock that charges PACE its full overhead while charging the fixed schedule nothing, since a deployed fixed schedule renders no feedback views and evaluates no policy. The last is the least favourable accounting available to us and is the one the main claim should survive. It does: the overall geometric mean over the 24 scene×backend pairs of Table 2 moves from $1.43\times$ to $1.39\times$, and the headline 21 dB crossing on `train` goes from $832 \to 476$ s ($1.75\times$) to $832 \to 485$ s ($1.71\times$) on 3DGS. No per-scene conclusion changes sign: PACE is faster than the fixed schedule in all 24 pairs under each of the three accountings, and the weakest cases, Faster-GS on `barn` and DashGaussian on `stump`, stay above unity ($1.05\times \to 1.02\times$ and $1.14\times \to 1.08\times$). The count of pairs at or above $1.2\times$ falls from 20 to 18. Only the non-render part of the overhead is extrapolated from Table 8: each block enters the accounting with the duration of its *own* feedback render, as logged during that rollout, so the growth of the render cost with model size (up to 116 ms per block on `bicycle`) is captured exactly. The remainder is the ≈ 12 ms of statistics, observation assembly, actor forward and opacity reset, and the result is insensitive to it: in the six-scene audit, inflating that term $5\times$ moved the deployment geometric mean only from $1.29\times$ to $1.26\times$, at which point one further pair (Faster-GS on `barn`) reaches $0.99\times$.

Table 9: **Speed-up under three timing accountings.** *train-loop*: gradient-step time only, as in Table 2. *symmetric*: each method additionally charged its own per-block overhead (Table 8). *deployment*: PACE charged its overhead, the fixed schedule charged none. Geometric mean over each scene's PSNR ladder; "blocks" is the number of agent queries each method made over the 30k-iteration run.

		blocks		gmean speed-up		
scene	**backend**	**PACE**	**fixed**	**train-loop**	**symmetric**	**deployment**
`train`	3DGS	151	300	1.80×	1.82×	1.76×
	Faster-GS	150	300	1.37×	1.40×	1.33×
	DashGaussian	151	300	2.11×	2.15×	2.04×
`barn`	3DGS	151	300	1.26×	1.27×	1.24×
	Faster-GS	150	300	1.05×	1.08×	1.02×
	DashGaussian	151	300	1.32×	1.35×	1.28×
`caterpillar`	3DGS	151	300	1.28×	1.29×	1.25×
	Faster-GS	150	300	1.38×	1.41×	1.33×
	DashGaussian	151	300	1.43×	1.45×	1.37×
`ignatius`	3DGS	151	300	2.03×	2.05×	1.99×
	Faster-GS	56	300	1.45×	1.48×	1.40×
	DashGaussian	151	300	1.81×	1.84×	1.74×
`bicycle`	3DGS	151	72	1.21×	1.22×	1.18×
	Faster-GS	28	74	1.28×	1.31×	1.24×
	DashGaussian	151	76	1.40×	1.44×	1.33×
`stump`	3DGS	151	62	1.44×	1.45×	1.40×
	Faster-GS	30	64	1.22×	1.25×	1.18×
	DashGaussian	151	64	1.14×	1.17×	1.08×
`meetingroom`	3DGS	151	300	1.19×	1.20×	1.17×
	Faster-GS	150	300	1.16×	1.19×	1.12×
	DashGaussian	151	300	1.41×	1.44×	1.35×
`courthouse`	3DGS	151	300	1.97×	1.99×	1.94×
	Faster-GS	150	300	1.29×	1.33×	1.25×
	DashGaussian	151	300	1.98×	2.02×	1.92×
overall (geometric mean)				1.43×	1.45×	1.38×

Two costs are reported separately rather than folded into the clock. Policy *learning* is paid once per backend, offline, and amortized over every future scene (Section C.1); it is not a per-scene

training cost. Scene loading and point-cloud initialization precede the first block and are identical for both methods; the single feedback render that builds the initial state s_0 also precedes it, and is the only agent cost outside the per-block accounting, worth at most 0.1% of a run. The reward is also excluded, since it is evaluated only during training and never at deployment; the deployment-inclusive clock recorded online by the evaluation protocol nevertheless includes it, making that column a slight over-estimate of the true deployment cost.

D Additional results

D.1 Detailed time-to-target results

Tables 10 to 15 give, for every held-out scene summarized in Table 2, the wall-clock time at which PACE and the fixed schedule first reach each PSNR target and the per-target speed-up ("—": target crossed before the curve's first sample or never reached). The PSNR ladder is chosen per scene to span the range both methods traverse.

Table 10: **Time-to-target wall-clock, `train` scene.** Seconds to first reach each PSNR target (agent vs. fixed schedule) and the per-target speed-up; protocol in Section 5.

	3DGS			Faster-GS			DashGaussian		
Target	**PACE (s)**	**Base (s)**	**Speed-up**	**PACE (s)**	**Base (s)**	**Speed-up**	**PACE (s)**	**Base (s)**	**Speed-up**
16 dB	24.4	41.5	**1.70×**	14.3	16.0	1.12×	8.7	20.9	**2.40×**
17 dB	44.9	87.0	**1.94×**	18.4	33.7	**1.83×**	17.4	42.5	**2.44×**
18 dB	66.7	122.0	**1.83×**	38.6	45.9	1.19×	31.2	70.3	**2.25×**
19 dB	92.9	174.8	**1.88×**	65.6	67.8	1.03×	69.7	101.9	1.46×
20 dB	223.3	380.7	**1.70×**	82.0	125.8	**1.53×**	126.8	237.4	**1.87×**
21 dB	476.1	831.7	**1.75×**	185.1	314.4	**1.70×**	209.7	518.0	**2.47×**
gmean		**1.80×**			1.37×			**2.11×**	

Table 11: **Time-to-target wall-clock on the unbounded `stump` scene**.

	3DGS			Faster-GS			DashGaussian		
Target	**PACE (s)**	**Base (s)**	**Speed-up**	**PACE (s)**	**Base (s)**	**Speed-up**	**PACE (s)**	**Base (s)**	**Speed-up**
19 dB	—	—	—	5.9	9.4	**1.59×**	—	—	—
20 dB	—	—	—	8.3	11.5	1.38×	—	—	—
21 dB	9.9	9.9	0.99×	10.4	15.4	1.48×	2.8	3.2	1.14×
22 dB	26.2	45.3	**1.73×**	27.1	23.3	0.86×	9.4	12.4	1.33×
23 dB	48.5	78.5	**1.62×**	32.6	37.7	1.16×	24.5	27.2	1.11×
24 dB	85.3	130.8	**1.53×**	63.0	65.5	1.04×	65.0	65.4	1.01×
gmean		1.44×			1.22×			1.14×	

Table 12: **Time-to-target wall-clock on the unbounded `bicycle` scene.**

	3DGS			Faster-GS			DashGaussian		
Target	**PACE (s)**	**Base (s)**	**Speed-up**	**PACE (s)**	**Base (s)**	**Speed-up**	**PACE (s)**	**Base (s)**	**Speed-up**
16 dB	—	—	—	3.7	7.1	**1.95×**	—	—	—
17 dB	—	—	—	5.8	8.5	1.46×	—	—	—
18 dB	—	—	—	8.0	9.9	1.25×	—	—	—
19 dB	—	—	—	9.4	11.3	1.21×	—	—	—
20 dB	13.0	10.4	0.80×	10.8	14.6	1.36×	2.9	3.1	1.07×
21 dB	38.9	51.6	1.33×	28.9	28.2	0.98×	9.9	15.8	**1.60×**
22 dB	89.5	115.3	1.29×	58.5	57.5	0.98×	30.1	44.2	1.47×
23 dB	139.2	216.4	**1.55×**	—	101.9	—	66.7	103.8	**1.56×**
24 dB	282.8	—	—	—	—	—	182.2	—	—
gmean		1.21×			1.28×			1.41×	

Table 13: **Time-to-target wall-clock on the `barn` scene.**

	3DGS			Faster-GS			DashGaussian		
Target	**PACE (s)**	**Base (s)**	**Speed-up**	**PACE (s)**	**Base (s)**	**Speed-up**	**PACE (s)**	**Base (s)**	**Speed-up**
16 dB	—	7.2	—	6.3	5.2	0.83×	—	2.7	—
17 dB	8.2	9.6	1.17×	6.7	6.0	0.89×	3.6	3.5	0.98×
18 dB	10.3	11.9	1.16×	7.1	6.7	0.94×	4.4	4.4	0.99×
19 dB	12.4	14.5	1.17×	7.5	7.5	1.00×	5.2	5.2	1.00×
20 dB	14.5	17.4	1.20×	7.9	8.2	1.04×	6.6	11.1	**1.67×**
21 dB	17.4	20.3	1.17×	8.3	9.8	1.18×	8.3	12.7	**1.53×**
22 dB	22.7	23.2	1.03×	10.9	11.6	1.07×	10.0	14.3	1.43×
23 dB	32.9	35.1	1.07×	18.8	14.6	0.78×	11.7	15.9	1.36×
24 dB	47.2	76.8	**1.63×**	24.1	36.2	**1.50×**	23.9	35.3	1.47×
25 dB	83.6	144.5	**1.73×**	42.6	53.0	1.25×	47.0	74.4	**1.58×**
26 dB	138.1	204.9	1.48×	60.3	81.0	1.34×	81.7	120.3	1.47×
gmean		1.26×			1.05×			1.32×	

Table 14: **Time-to-target wall-clock on the `caterpillar` scene.**

	3DGS			Faster-GS			DashGaussian		
Target	**PACE (s)**	**Base (s)**	**Speed-up**	**PACE (s)**	**Base (s)**	**Speed-up**	**PACE (s)**	**Base (s)**	**Speed-up**
16 dB	9.6	9.2	0.96×	7.1	7.5	1.06×	3.5	3.3	0.95×
17 dB	13.5	11.4	0.84×	7.7	8.2	1.08×	4.3	4.3	0.98×
18 dB	19.3	20.5	1.06×	8.2	10.4	1.26×	6.6	10.8	**1.64×**
19 dB	35.6	52.3	1.47×	17.6	22.1	1.26×	11.6	27.0	**2.32×**
20 dB	55.1	80.7	1.46×	21.9	33.0	**1.51×**	24.2	40.1	**1.66×**
21 dB	86.7	138.3	**1.60×**	26.3	58.7	**2.23×**	42.7	76.2	**1.79×**
22 dB	139.0	234.3	**1.69×**	72.5	113.0	**1.56×**	101.3	139.4	1.38×
23 dB	348.0	492.9	1.42×	180.0	244.7	1.36×	268.3	316.5	1.18×
24 dB	—	828.2	—	311.9	—	—	—	—	—
25 dB	—	—	—	—	—	—	—	—	—
26 dB	—	—	—	—	—	—	—	—	—
gmean		1.28×			1.38×			1.43×	

Table 15: **Time-to-target wall-clock on the `ignatius` scene.**

	3DGS			Faster-GS			DashGaussian		
Target	**PACE (s)**	**Base (s)**	**Speed-up**	**PACE (s)**	**Base (s)**	**Speed-up**	**PACE (s)**	**Base (s)**	**Speed-up**
16 dB	—	6.6	—	6.9	7.5	1.09×	—	2.5	—
17 dB	7.6	8.7	1.14×	7.4	8.1	1.10×	3.7	3.5	0.93×
18 dB	11.1	21.8	**1.95×**	7.9	13.2	**1.67×**	4.9	4.4	0.90×
19 dB	17.7	25.3	1.42×	8.3	18.2	**2.18×**	7.7	15.3	**1.99×**
20 dB	24.1	42.6	**1.77×**	20.4	25.9	1.27×	15.5	23.1	1.48×
21 dB	32.9	58.9	**1.79×**	38.9	39.6	1.02×	30.1	65.5	**2.17×**
22 dB	59.3	166.6	**2.81×**	47.3	87.7	**1.85×**	78.6	184.6	**2.35×**
23 dB	161.0	798.1	**4.96×**	125.5	227.9	**1.82×**	151.9	754.2	**4.96×**
24 dB	—	—	—	—	—	—	352.6	—	—
25 dB	—	—	—	—	—	—	—	—	—
26 dB	—	—	—	—	—	—	—	—	—
gmean		**2.03×**			1.45×			**1.81×**	

D.2 Per-block decisions and population statistics

Figure 4 shows, for the same rollouts as Fig. 3, what the three controllers decided at every block and how the Gaussian population responded: when the densifier was on and opacities were reset, the threshold and learning-rate multipliers chosen relative to each backend's default, and the mean opacity and median scale of the Gaussians over training.

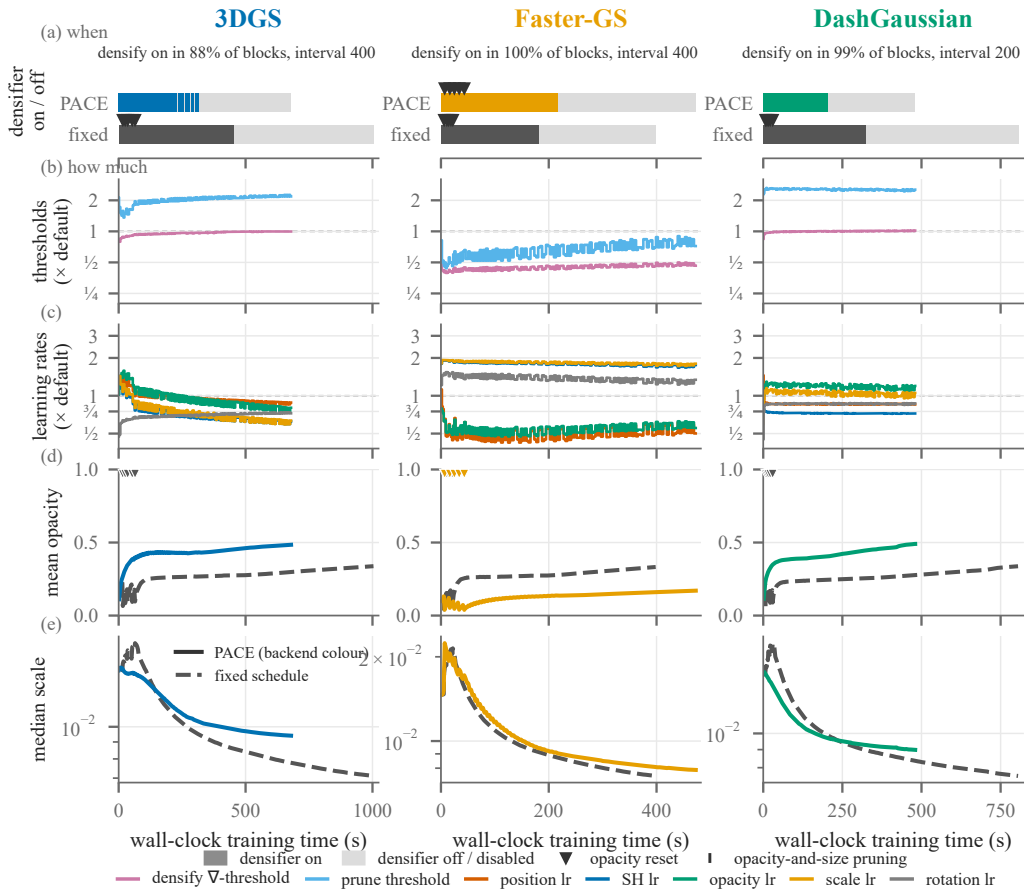

Figure 4: **PACE's decisions and the Gaussian population** (`train` scene, the rollouts of Fig. 3; one column per backend; PACE in the backend's colour, fixed schedule in grey). *(a) When:* blocks with the densifier on (coloured) or off (grey) for PACE and for the fixed schedule, which densifies until iteration 15k; ▼ opacity resets (a $>30\%$ drop of the mean opacity between blocks), | blocks with opacity-and-size pruning. *(b, c) How much:* the densification- and prune-threshold multipliers and the five learning-rate multipliers chosen per block, relative to the backend's default (dashed line at 1; log scale). *(d, e):* mean opacity and median Gaussian scale (log scale).

D.3 Final Gaussian populations

Figure 5 complements Figs. 3 and 4 with the distributions of per-Gaussian opacity and scale at the end of the same rollouts.

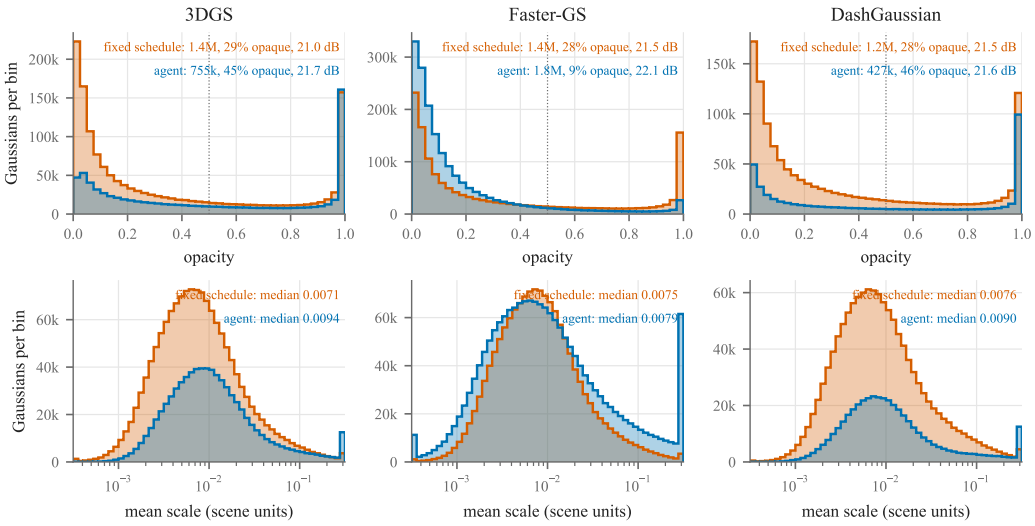

Figure 5: **Final populations at** 30**k** (`train` scenes): histograms of per-Gaussian opacity (top) and mean scale (bottom, log scale; the last bin collects everything beyond the axis), PACE vs. fixed schedule, in absolute counts so that areas compare model sizes; the dotted line marks opacity 0.5. Annotations give each model's Gaussian count, share of Gaussians above 0.5 opacity, test PSNR, and median scale; analysis in the text.

D.4 THE SAME MECHANISM ON A SECOND HELD-OUT SCENE

Figures 6 and 7 repeat Figs. 3 and 4 on the held-out Tanks&Temples scene `ignatius`, with the same protocol and the same logging; they are separate single-seed rollouts from those behind Tables 2 and 15, so end-of-run counts and PSNR differ slightly from that table. The DashGaussian traces in this subsection, and in Sections D.5 and D.9, were produced with the earlier DashGaussian controller, whose curriculum was keyed to the episode cap and which therefore never densifies (Section C.6); they are retained as the analysis of that policy. The 3DGS and Faster-GS traces, and every number in Tables 2 and 16, are unaffected. The three regularities of Section 5.1 reappear unchanged, and again differ by backend in the way the cost of a Gaussian predicts. On 3DGS, PACE turns the densifier on in 63% of the blocks before iteration 15k where the fixed schedule densifies in every one of them, and only 28 of its blocks actually add Gaussians against the fixed schedule's 144; it never resets opacities against the fixed schedule's four resets, and stops growing the model after 239 s against 498 s; it ends at 1.70M Gaussians against 2.61M, buys an iteration for 31 instead of 41 ms, and is ahead by 0.9–1.8 dB at every instant of the run. On DashGaussian, whose densifier is the most expensive of the three, the policy again never densifies at all: it keeps the 155k initialization, pays 15 instead of 33 ms per iteration, and is 1.5–2.1 dB ahead over the first two minutes. On Faster-GS, where a Gaussian is cheap, the policy again grows the model harder than the fixed schedule does — here to the point that the episode ends early, as discussed next.

Faster-GS on `ignatius` is the one bounded scene in this paper where PACE meets the environment's safety rail (on the unbounded `bicycle` and `stump` both PACE-FasterGS and the fixed schedules reach it, Table 9): the policy densifies in every block and crosses the 3M-Gaussian cap (Section C) at iteration 12k, which terminates the episode at 173 s. Its curve in Fig. 6 therefore stops where the marker sits, and the run is not comparable at 30k with the others. It is still ahead of the fixed schedule on the wall clock it covers: it reaches 23.30 dB at 173 s, above the 23.07 dB the fixed schedule reaches at 458 s. We report the run as it happened rather than raising the cap, because the cap is part of the deployment protocol used everywhere else in the paper.

Figures 8 to 10 show what the wall-clock gap looks like, on two held-out views of the same scene at four instants of the same rollouts. The zoom window is not chosen by hand: for each view it is the window of fixed aspect that maximizes the drop in squared error from the fixed schedule to PACE at 60 s, which is where the two reconstructions differ most. On 3DGS the gap is 0.9–2.2 dB over the whole view and 1.6–5.3 dB inside the window, and it is large enough to cost the fixed schedule more than the factor of four in wall clock that separates the columns: measured on the window, PACE at 30 s is already ahead of where the fixed schedule stands at 120 s, on both views (17.2 against 15.5 dB and 18.4 against 17.1 dB). On DashGaussian the same holds from a model nine times smaller. On Faster-GS the two are closer, consistent with the smaller wall-clock gap that backend shows on this scene.

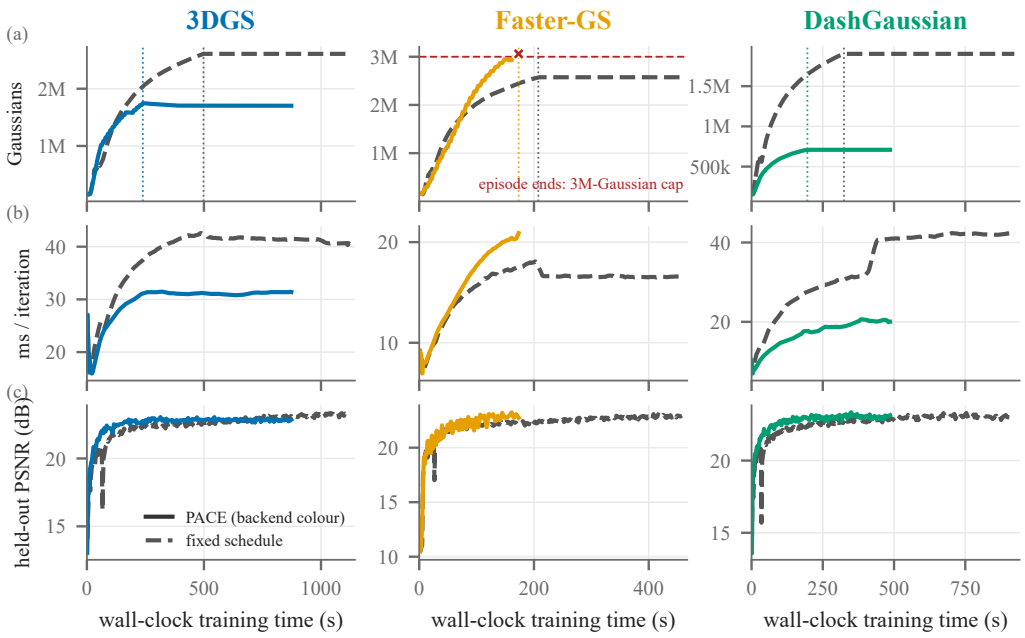

Figure 6: **What PACE does to model size, compute and quality on held-out `ignatius`** (full 30k rollouts, one column per backend, shared wall-clock axis; PACE in the backend's colour, fixed schedule in grey). (a) Gaussian count (dotted verticals: last block that added Gaussians; dashed red line: the 3M-Gaussian safety cap, $\times$ the block at which it ends the Faster-GS episode), (b) wall-clock cost of one iteration (smoothed over 7 blocks), (c) held-out PSNR.

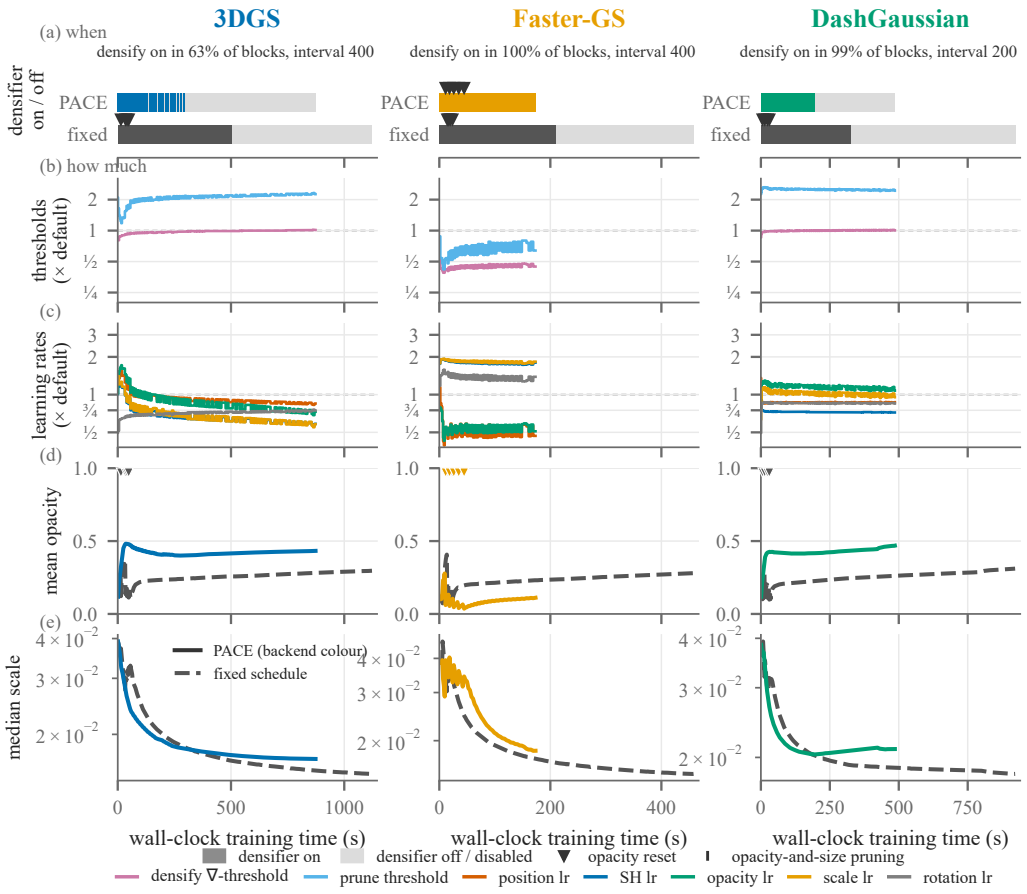

Figure 7: **PACE's decisions and the Gaussian population on held-out `ignatius`** (the rollouts of Fig. 6; one column per backend; PACE in the backend's colour, fixed schedule in grey). *(a) When:* blocks with the densifier on (coloured) or off (grey); ▼ opacity resets, | blocks with opacity-and-size pruning. *(b, c) How much:* threshold and learning-rate multipliers relative to the backend's default (dashed line at 1; log scale). *(d, e):* mean opacity and median Gaussian scale (log scale).

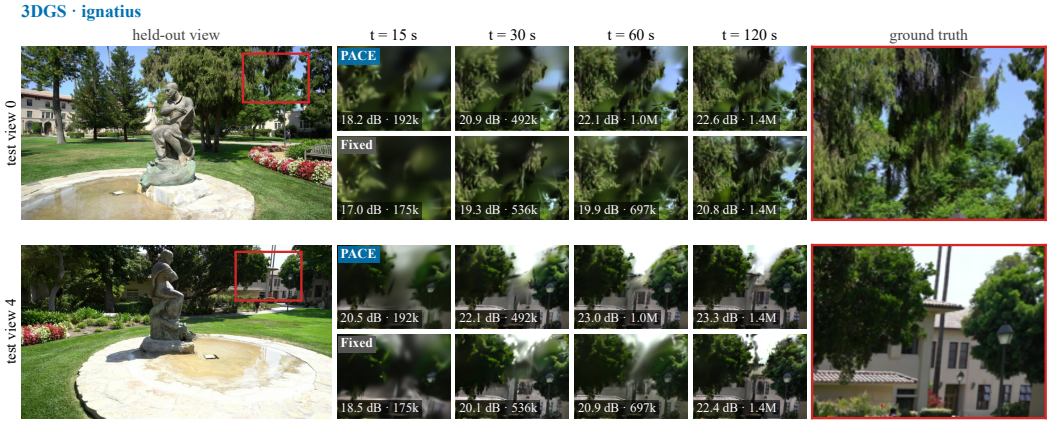

Figure 8: **Zoomed visual quality against wall-clock on `ignatius`, 3DGS.** Two held-out views (rows), PACE above the fixed schedule at four instants of the rollouts of Fig. 6. Left: the full view with the zoom window; right: the window in the ground truth. Annotations give the test PSNR of the whole view and the Gaussian count.

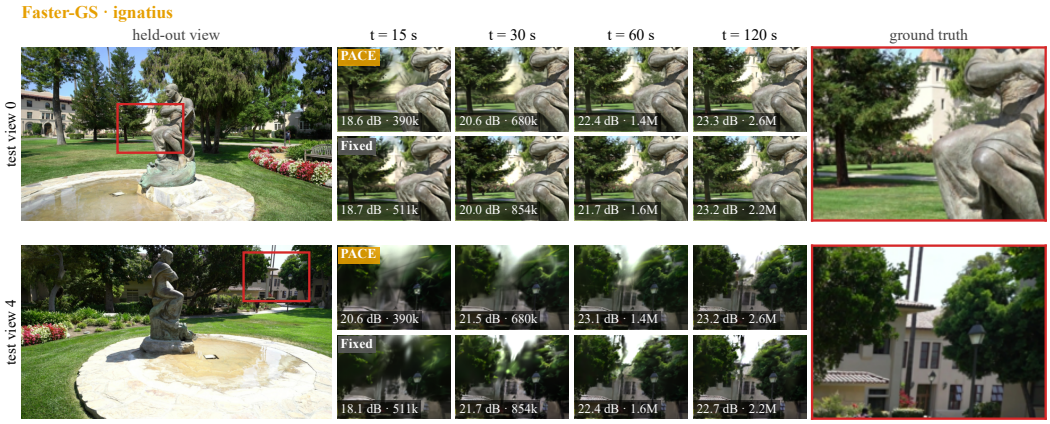

Figure 9: **Zoomed visual quality against wall-clock on `ignatius`, Faster-GS**; layout as in Fig. 8.

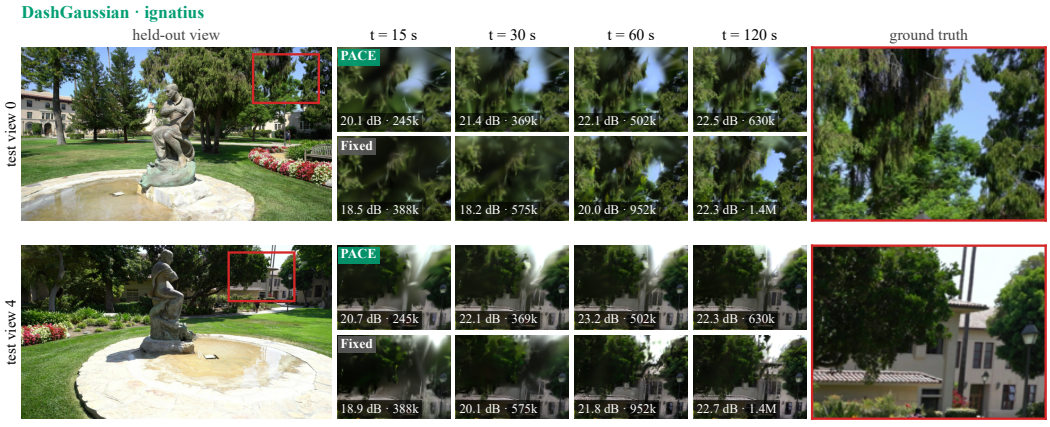

Figure 10: **Zoomed visual quality against wall-clock on `ignatius`, DashGaussian**; layout as in Fig. 8. PACE keeps the 155k initialization throughout.

D.5 A LONGER HORIZON: 50K ITERATIONS ON CATERPILLAR

The controllers are trained with an episode cap of 7k iterations and evaluated at 30k everywhere else in this paper. To see how far the learned schedules carry beyond that, we repeat the study of Section D.4 on held-out `caterpillar` with the cap raised to 50k — 7 times the training horizon — and with 16 wall-clock snapshots instead of six, so the early part of the run is sampled densely. As in the 30k protocol, the iteration cap scales the learning-rate decay and the minimum-stop iteration; the backends' densification cutoff stays at iteration 15k. Figures 11 and 12 show the rollouts and the decisions, Figs. 13 to 15 the views at six of the instants.

The mechanism is the one described in Section 5.1: on 3DGS only 43 of PACE's blocks add Gaussians against the fixed schedule's 144, it never resets opacities against five resets, it freezes growth at 340 s against 407 s, and it reaches 50k in 1250 s with 969k Gaussians at 25.7 ms per iteration where the fixed schedule needs 1570 s and 1.54M at 32.6 ms. On DashGaussian the policy again refuses to densify, and here it also prunes the initialization in half (188k to 94k Gaussians at 120 s), reaching 50k in 539 s against 1216 s at less than half the cost per iteration.

The quality outcome, however, depends on the backend, and the longer horizon is where PACE's front-loaded reward shows its cost. PACE leads at every early instant on all three backends ($+1.5$, $+1.7$ and $+1.1$ dB at 15 s on 3DGS, Faster-GS and DashGaussian). On Faster-GS the lead survives the whole run: PACE is $+1.1$ dB ahead at 300 s and still $+0.37$ dB ahead at 50k (24.59 against 24.21 dB), and it reaches the fixed schedule's best held-out quality in 284 s against 673 s, while the fixed schedule never reaches PACE's. On 3DGS the gap closes as the budget grows — $+1.3$ dB at 60 s, $+0.24$ dB at 300 s — and at 50k the fixed schedule ends 0.11 dB ahead from a 59% larger model. On DashGaussian the ranking reverses: the pruned 94k model saturates between 22.6 and 22.8 dB from 300 s onwards, and the fixed schedule ends 1.25 dB ahead. The controllers optimize a time-discounted return over 7k-iteration episodes, so they buy early quality with model capacity; at 7 times that horizon the purchase is still favourable where a Gaussian is cheap (Faster-GS), roughly break-even on 3DGS, and a loss on the backend whose policy compacts hardest (DashGaussian). The policies evaluated here were not retrained for this horizon; a controller intended for long runs would have to be trained at, or conditioned on, the longer budget, and we regard that as the natural next step rather than a property of the schedules reported in this paper.

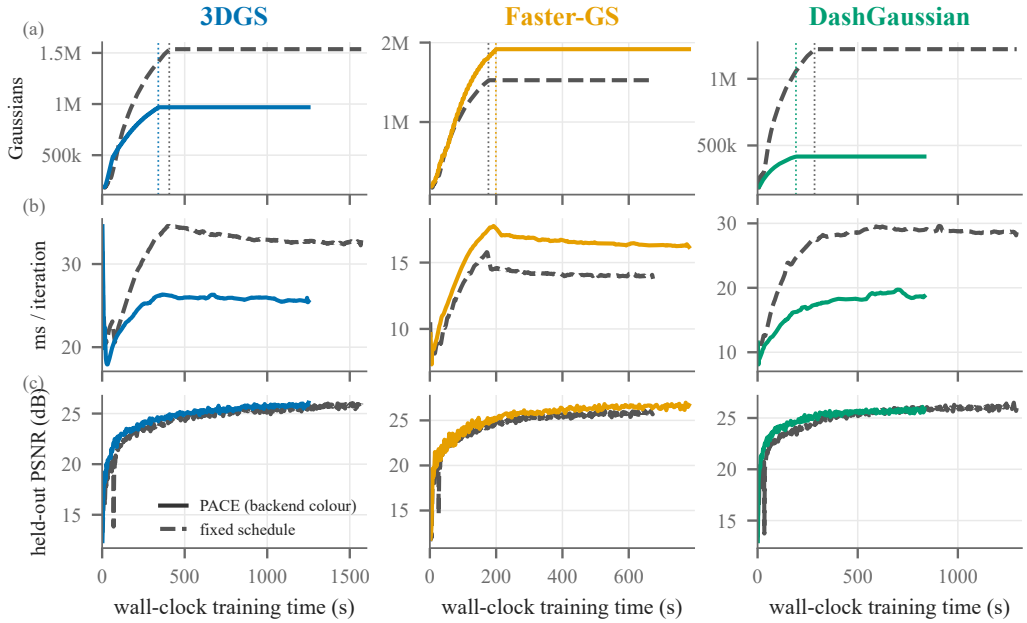

Figure 11: **Model size, compute and quality over a 50k-iteration horizon on held-out `caterpillar`** (one column per backend, shared wall-clock axis; PACE in the backend's colour, fixed schedule in gray). (a) Gaussian count (dotted verticals: last block that added Gaussians), (b) wall-clock cost of one iteration (smoothed over 7 blocks), (c) held-out PSNR.

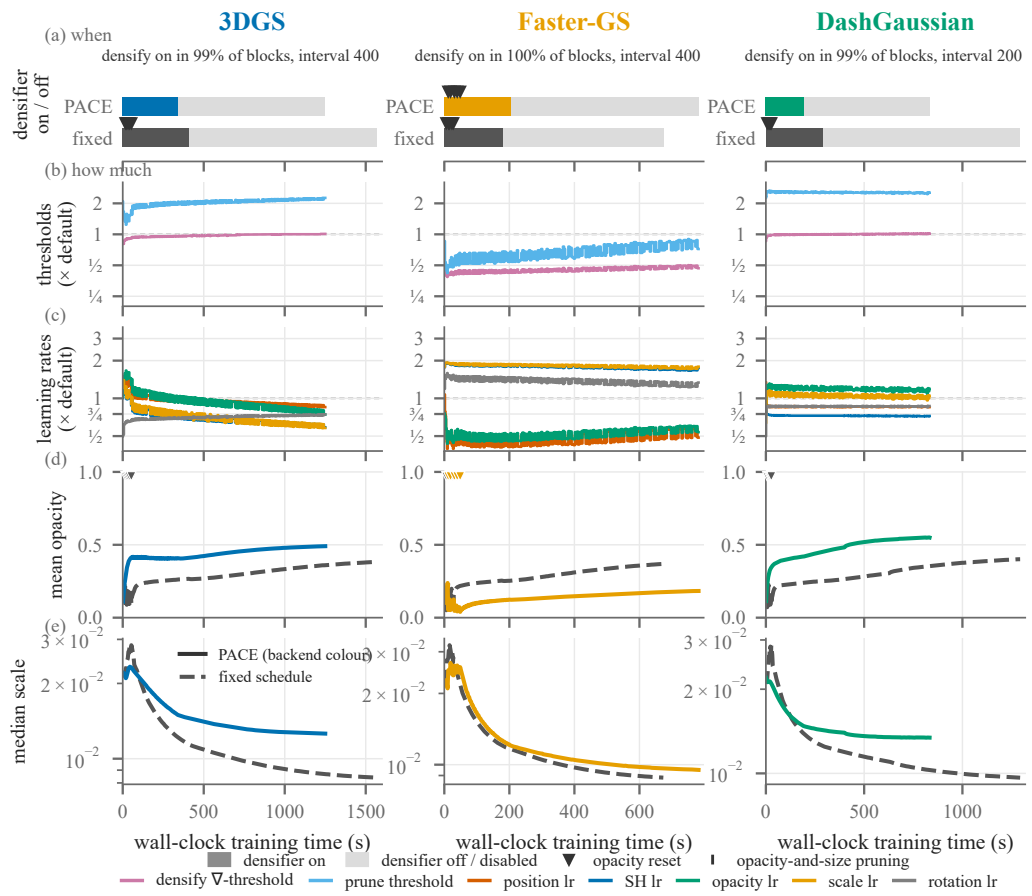

Figure 12: PACE's decisions and the Gaussian population over the 50k horizon on held-out `caterpillar` (the rollouts of Fig. 11); panels as in Fig. 7.

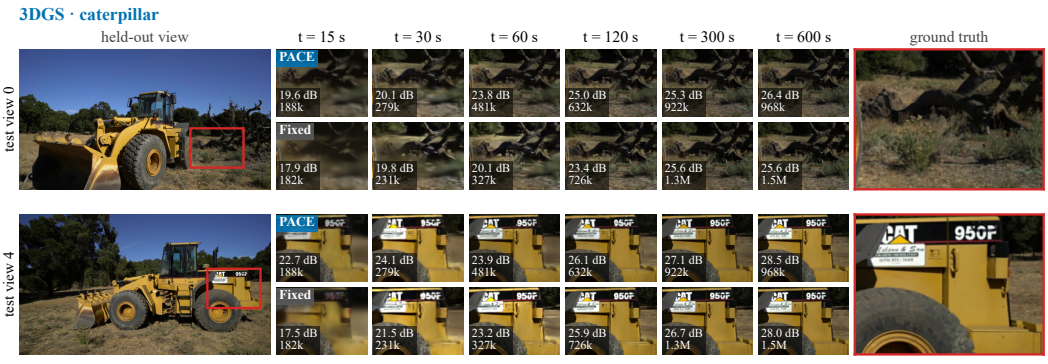

Figure 13: **Zoomed visual quality against wall-clock over the** 50**k horizon, 3DGS**; layout as in Fig. 8, at six instants.

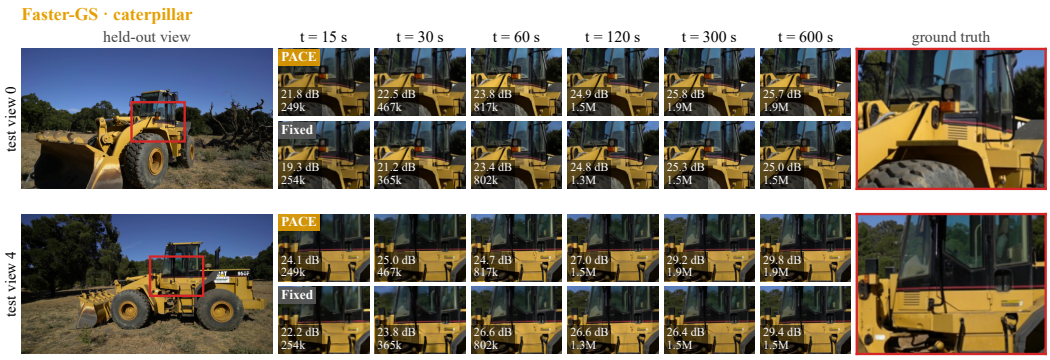

Figure 14: **Zoomed visual quality against wall-clock over the** 50**k horizon, Faster-GS**; layout as in Fig. 8.

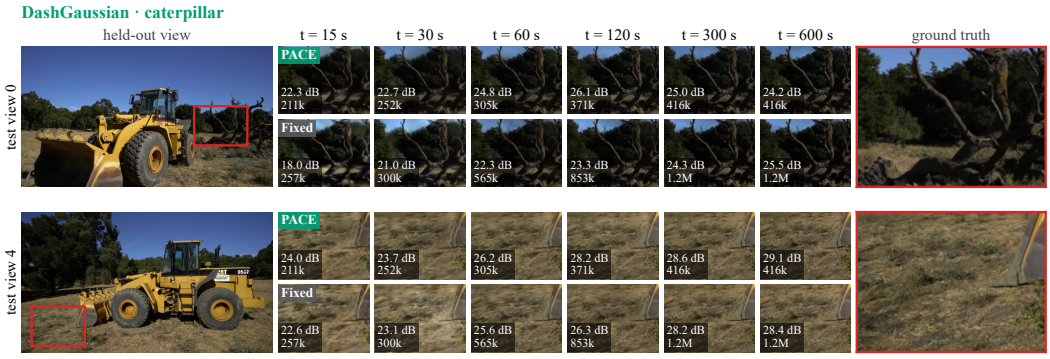

Figure 15: **Zoomed visual quality against wall-clock over the** 50**k horizon, DashGaussian**; layout as in Fig. 8. The last column is $t = 500$ s because PACE finishes 50k at 539 s. PACE leads early and is overtaken once the fixed schedule's larger model matures.

D.6 CROSS-BACKEND TRANSFER, FULL TABLE

Table 16 lists every policy×backend pair behind Section 5.3: model size and quality at 30k, the wall-clock time to 18 and 20 dB, and the geometric-mean speed-up over the targets both runs reach.

Table 16: **Cross-backend transfer** (`train` scene). Each trained policy run unchanged on each backend vs. that backend's fixed schedule: Gaussians and test PSNR at 30k, wall-clock to 18 and 20 dB (agent / baseline), and geometric-mean speed-up over the targets both runs reach (16–21 dB; 16–20 dB on LeGS, Table 3). Bold: best speed-up per backend and agent models $\geq 20\%$ smaller than the baseline.

Policy (train → test backend)	**#G @ 30k (k)** agent / base	**PSNR @ 30k (dB)** agent / base	$t_{\to 18}$ **(s)** agent / base	$t_{\to 20}$ **(s)** agent / base	**Speed-up** (gmean)
Target backend: 3DGS					
3DGS → 3DGS (native)	**720**/1447	21.28/21.30	66.7/122.0	223.3/380.7	**1.80×**
Faster-GS → 3DGS (transfer)	1681/1457	22.49/21.30	161.7/125.7	298.2/321.1	1.24×
Dash → 3DGS (transfer)	**477**/1448	21.10/21.29	82.0/131.4	192.7/337.1	1.78×
Target backend: Faster-GS					
Faster-GS → Faster-GS (native)	1713/1408	22.68/21.21	38.6/45.9	82.0/125.8	**1.37×**
3DGS → Faster-GS (transfer)	**678**/1461	21.71/21.31	37.1/43.4	109.1/128.8	1.23×
Dash → Faster-GS (transfer)	**185**/1427	19.27/21.54	185.0/50.9	− − −/151.0	0.23×
Target backend: DashGaussian					
Dash → DashGaussian (native)	**427**/1226	21.54/21.16	31.2/70.3	126.8/237.4	**2.11×**
3DGS → DashGaussian (transfer)	**678**/1239	21.51/21.91	40.6/59.8	145.6/172.7	1.40×
Faster-GS → DashGaussian (transfer)	1273/1221	21.86/21.45	74.2/61.2	143.7/231.1	1.46×
Target backend: LeGS					
3DGS → LeGS (transfer)	**399**/609	20.86/20.83	32.2/67.8	124.5/189.8	1.86×
Faster-GS → LeGS (transfer)	720/642	21.35/20.87	42.2/53.7	151.4/192.7	1.62×
Dash → LeGS (transfer)	**482**/607	20.86/20.34	34.4/62.8	140.9/187.1	**2.02×**

D.7 SCENES THE POLICY WAS TRAINED ON

Every result in the main text is zero-shot, which leaves open how much of the speed-up is specific to the scenes the policy saw during training. Table 17 repeats the protocol on four of them, one per dataset family in the pool. The speed-ups, 1.21–$1.50\times$ in geometric mean, sit inside the range measured on the held-out scenes (1.21–$2.03\times$ on 3DGS, Table 2): the controller is no faster on scenes it was trained on than on scenes it has never seen, so the held-out results are not explained by it having memorized its pool. The per-target structure is the same one visible throughout: PACE is neutral or slightly behind at the earliest targets and pulls ahead as they rise (on `drjohnson`, $0.94\times$ at 20 dB against $2.24\times$ at 29 dB), which is what freezing growth early implies—the advantage accrues once the fixed schedule's model has grown and its iterations have slowed.

Table 17: **Time-to-target on scenes from the training pool.** Four scenes the policy was trained on (`truck`: Tanks&Temples, `garden` and `room`: Mip-NeRF 360, `drjohnson`: Deep Blending), under the protocol of Section 5 on the 3DGS backend at 30k iterations. The ladder runs from the first target both methods cross to the last, so it differs per scene and from Table 2; the geometric mean is over the targets shown

Scene	Target	PACE (s)	Base (s)	Speed-up
`truck` (T&T)	18 dB	7.9	9.4	1.18×
	19 dB	11.3	11.2	0.99×
	20 dB	16.0	16.9	1.05×
	21 dB	28.6	26.2	0.92×
	22 dB	46.2	52.1	1.13×
	23 dB	74.1	114.2	**1.54×**
	24 dB	126.4	205.6	**1.63×**
	25 dB	381.6	547.3	1.43×
	gmean			1.21×
`garden` (Mip-360)	21 dB	10.1	9.0	0.89×
	22 dB	19.1	27.4	1.43×
	23 dB	31.9	44.9	1.41×
	24 dB	59.3	84.3	1.42×
	25 dB	102.5	160.2	**1.56×**
	26 dB	215.0	298.0	1.39×
	gmean			1.33×
`room` (Mip-360)	24 dB	3.8	5.2	1.35×
	25 dB	6.7	6.1	0.91×
	26 dB	9.8	14.1	1.44×
	27 dB	13.3	17.8	1.33×
	28 dB	21.4	37.3	**1.75×**
	29 dB	27.8	49.4	**1.78×**
	30 dB	37.5	75.2	**2.01×**
	31 dB	64.6	125.0	**1.94×**
	32 dB	172.2	250.0	1.45×
	33 dB	437.4	614.0	1.40×
	gmean			**1.50×**
`drjohnson` (DB)	20 dB	13.0	12.2	0.94×
	21 dB	18.1	14.9	0.82×
	22 dB	27.4	34.3	1.25×
	23 dB	37.7	41.9	1.11×
	24 dB	51.0	61.6	1.21×
	25 dB	76.8	110.4	1.44×
	26 dB	105.0	185.3	**1.77×**
	27 dB	135.6	268.6	**1.98×**
	28 dB	231.0	461.2	**2.00×**
	29 dB	429.8	963.1	**2.24×**
	gmean			1.40×

D.8 FEWER TRAINING VIEWS

Capture setups rarely provide hundreds of images. We repeat the time-to-target protocol on the 3DGS backend with the frozen 3DGS controller while keeping only 25, 50 or 100 of the training cameras of the four held-out Tanks&Temples scenes (a farthest-point subset of the capture with a fixed seed, identical for both methods; the controller's validation cameras are drawn from the reduced set, at most one fifth of it; the test cameras and the SfM initialization are unchanged). Table 18 and Fig. 16 report the outcome. Two trends stand out. First, fewer views lower the quality both methods can reach—the ceiling moves, not the ordering: in the ten reduced-view cases where at least one target of the 16–28 dB ladder is reached, PACE is faster in nine (geometric-mean speed-ups of 1.2–1.6×), ends with a 20–50% smaller model in every case, and matches or exceeds the fixed schedule's peak PSNR in eight. The exception is `barn` with 25 views, where the compacted model under-builds and is both slower and 0.6 dB worse; on `train` with 25 and 50 views neither method reaches 16 dB (both plateau near 15 dB, the fixed schedule degrading late), so no speed-up is defined there. Second, the learned schedule itself is nearly independent of the number of views (densification on in a third to a half of the blocks at the longest interval, learning-rate multipliers unchanged): the controller does not adapt its schedule to view sparsity, and its gain shrinks at the sparsest settings only because both methods saturate within seconds and little remains to be accelerated.

Table 18: **Training-view ablation** (3DGS backend, 3DGS controller). Geometric-mean time-to-target speed-up over the 16–28 dB targets both methods reach (*Targets*: their number; "–": none reached), best test PSNR within 30k iterations, Gaussians at the end, and the wall-clock to the highest common target (PACE / fixed). Bold: speed-up $\geq 1.5\times$. The ladder is the same for every view count, so the all-view gmeans differ slightly from the per-scene ladders of Table 2.

Scene	Views	Speed-up (gmean)	Targets	PSNR agent / fixed (dB)	#G agent / fixed	$t_{\to\max}$ agent / fixed (s)
train	25	–	0	14.74 / 14.63	0.73M / 1.15M	12 / 16 (to 14 dB)
	50	–	0	15.31 / 15.38	0.75M / 1.23M	52 / 57 (to 15 dB)
	100	**1.64×**	3	18.86 / 18.81	0.71M / 1.36M	127 / 232 (to 18 dB)
	all (263)	**1.80×**	6	21.28 / 21.30	0.72M / 1.45M	476 / 832 (to 21 dB)
ignatius	25	1.20×	1	18.23 / 17.79	1.22M / 1.55M	8 / 10 (to 17 dB)
	50	**1.54×**	3	20.18 / 19.10	1.50M / 1.91M	18 / 49 (to 19 dB)
	100	1.49×	4	22.03 / 20.76	1.67M / 2.26M	25 / 47 (to 20 dB)
	all (230)	**2.03×**	7	23.85 / 23.28	1.70M / 2.56M	161 / 798 (to 23 dB)
caterpillar	25	1.25×	2	17.30 / 17.59	0.75M / 0.97M	17 / 22 (to 17 dB)
	50	1.49×	3	18.42 / 18.29	0.85M / 1.12M	51 / 91 (to 18 dB)
	100	**1.53×**	5	20.73 / 20.48	0.85M / 1.23M	64 / 207 (to 20 dB)
	all (335)	1.28×	8	23.81 / 24.10	0.86M / 1.33M	348 / 493 (to 23 dB)
barn	25	0.86×	5	20.20 / 20.78	0.50M / 0.77M	64 / 25 (to 20 dB)
	50	1.21×	6	23.18 / 22.80	0.58M / 0.85M	26 / 42 (to 22 dB)
	100	1.31×	9	25.64 / 25.20	0.65M / 0.99M	173 / 333 (to 25 dB)
	all (358)	1.29×	12	28.44 / 28.50	0.68M / 1.14M	440 / 655 (to 28 dB)

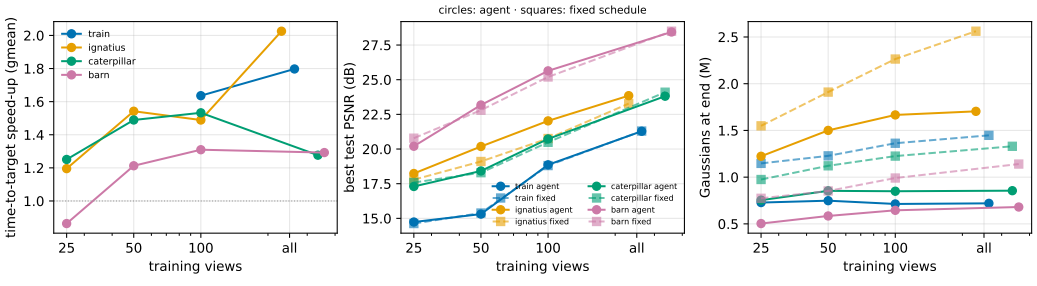

Figure 16: **Training-view ablation.** Speed-up, best test PSNR and final model size as a function of the number of training views ("all": the full capture), one colour per scene; circles PACE, squares fixed schedule.

D.9 Deployment efficiency, perceptual quality, and cost

Where the controller compacts, the compaction turns into concrete deployment savings at the 30k reference (Table 19; single-seed measurements on held-out `train`, each model evaluated with its own rasterizer). PACE-3DGS and PACE-Dash store a $2.0\times$ and $2.8\times$ smaller model on disk at matched PSNR (within 0.05 dB) and train with 0.8–1.5 GB less peak VRAM; on 3DGS the compact model also renders $1.7\times$ faster (144 vs. 84 FPS), while on DashGaussian render speed is unchanged. The consistent cost of compaction is perceptual: LPIPS rises by 0.013 on 3DGS and 0.031 on Dash and SSIM drops by 0.005–0.013, because a smaller population resolves less fine texture even when PSNR is matched, a natural target for a perceptual term in the reward. PACE-FasterGS sits at the other end of the trade-off: it spends its speed advantage on capacity rather than compactness, storing a 26% larger model that is better on every quality metric ($+0.8$ dB, LPIPS 0.208 vs. 0.239) and still renders 20% faster; its savings are wall-clock and quality, not storage.

Cost of learning and inference The policy is trained once (6–11 GPU-hours per backend on one RTX 3090, Section C) and reused zero-shot, so its learning cost is amortized over every future scene; at inference the actor adds about 5 ms per block, 0.1–0.2% of the block's training time (the full per-block overhead including the feedback renders is accounted in Section C.7). The advantage also holds throughout training, not only at the end: the time-averaged test PSNR over the agent's training window is 0.6–0.9 dB above the fixed schedule's on the three backends ($+0.87$ dB on 3DGS).

Table 19: **Deployment efficiency at the 30k reference**, `train` scene, PACE vs. the backend's fixed schedule (each model evaluated with its own rasterizer on 40 test views at the evaluation resolution). Gaussian count, on-disk model size, test PSNR, LPIPS (VGG), peak training VRAM and synchronized render FPS. Colour code as in Table 2: **teal** marks the better of each pair on the efficiency columns, **orange** on the quality columns; differences inside the measurement noise (0.1 dB, 0.005 LPIPS, 0.3 GB, 5% FPS) are left unmarked.

Backend	Schedule	Gauss. (k)	Size (MB)	PSNR	LPIPS	Train VRAM (GB)	FPS
3DGS	PACE	**727**	**172**	22.01	0.251	**5.2**	**144**
	fixed	1444	342	21.97	**0.238**	6.7	84
Faster-GS	PACE	1798	425	**22.91**	**0.208**	7.5	**138**
	fixed	**1425**	**337**	22.10	0.239	**7.2**	115
DashGaussian	PACE	**428**	**101**	21.94	0.271	**4.7**	119
	fixed	1216	288	21.97	**0.240**	5.5	118