Supplementary Material of DiffBMP: Differentiable Rendering with Bitmap Primitives

Seongmin Hong[1,*], Junghun James Kim[2,*], Daehyeop Kim[3], Insoo Chung[3], Se Young Chun[1,2,3,†]

[1]INMC, [2]IPAI, [3]Dept. of ECE, Seoul National University, Republic of Korea

{smhongok, jonghean12, 2012abcd, insoo_chung, sychun}@snu.ac.kr

diffbmp.com

S1. Method Detail

S1.1. Bilinear interpolation

Here, we provide more detail on bilinear interpolation [7], where the spatial gradients occur. Let $\lfloor U \rfloor = u_0$, $\lfloor V \rfloor = v_0$, and define the fractional parts $w_u = U - u_0$, $w_v = V - v_0$. The bilinear weights form:

$$\mathbf{w} = \begin{bmatrix} (1-w_u)(1-w_v) \\ w_u(1-w_v) \\ (1-w_u)w_v \\ w_u w_v \end{bmatrix} \tag{S1}$$

The primitive values at the four neighboring pixels are:

$$\mathbf{p} = \begin{bmatrix} P_i[v_0, u_0] \\ P_i[v_0, u_0+1] \\ P_i[v_0+1, u_0] \\ P_i[v_0+1, u_0+1] \end{bmatrix} \tag{S2}$$

Those four points are depicted as green dots in Fig. S1. The interpolated value is then:

$$M_i(x,y) = \mathbf{w}^T \mathbf{p} = \sum_{j=0}^{3} w_j p_j \tag{S3}$$

S1.2. Backward Pass

Here, we state how the color gradients and alpha gradients are calculated. Entire forward-backward computation diagrams are shown in Fig. S2.

Color Gradients. The gradient with respect to color logits follows the chain rule:

$$\frac{\partial L}{\partial c_{k,j}} = \frac{\partial L}{\partial I_j} \cdot T_k \alpha_k \cdot \sigma(c_{k,j})(1 - \sigma(c_{k,j})) \tag{S4}$$

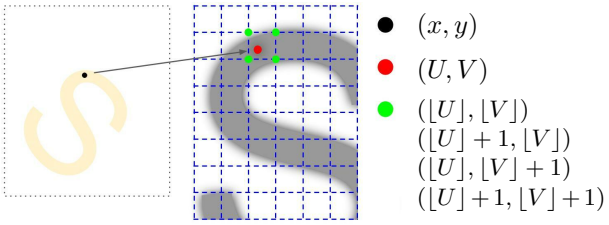


Figure S1. Coordinate transformation and primitive sampling. Color and opacity from i-th primitive at (x, y) in canvas (black dot) is sampled from (U, V) in the primitive's coordinate (red dot). Since U and V are not integers, bilinear interpolation from four nearest lattice points (green dots).

Alpha Gradients. For the Porter-Duff over operator [15], the alpha gradient incorporates both direct color contribution and transmittance effects:

$$\frac{\partial L}{\partial \alpha_k} = \sum_{j \in r,g,b} \frac{\partial L}{\partial I_j} \left(T_k \sigma(c_{k,j}) - \frac{S_j}{1-\alpha_k} \right) + \frac{\partial L}{\partial A} T_k B \tag{S5}$$

where the suffix sum $S_j = \sum_{m>k} \sigma(c_{m,j}) \alpha_m T_m$ and back-product $B = \prod_{m>k}(1-\alpha_m)$ account for the interdependence of layered primitives.

Half2 Precision. In the backward pass, for all gradient calculation, we use `__half2` which packs two FP16, `__half`, in vector format. Per-parameter gradients are accumulated *per pixel* directly into packed `__half2` buffers via `__half2 atomicAdd` for the pairs (m_x, m_y), (α, scale), (θ, c_r), and (c_g, c_b), followed by a lightweight post pass that unpacks to legacy FP16 arrays. In our measurements, `__half atomicAdd` matched kernel time but degraded numerical accuracy, whereas `__half atomicCAS` recovered accuracy at prohibitive cost. By contrast, packed `__half2` based atomics preserved accuracy, improved throughput, and reduced memory traffic. Accordingly, we adopt this

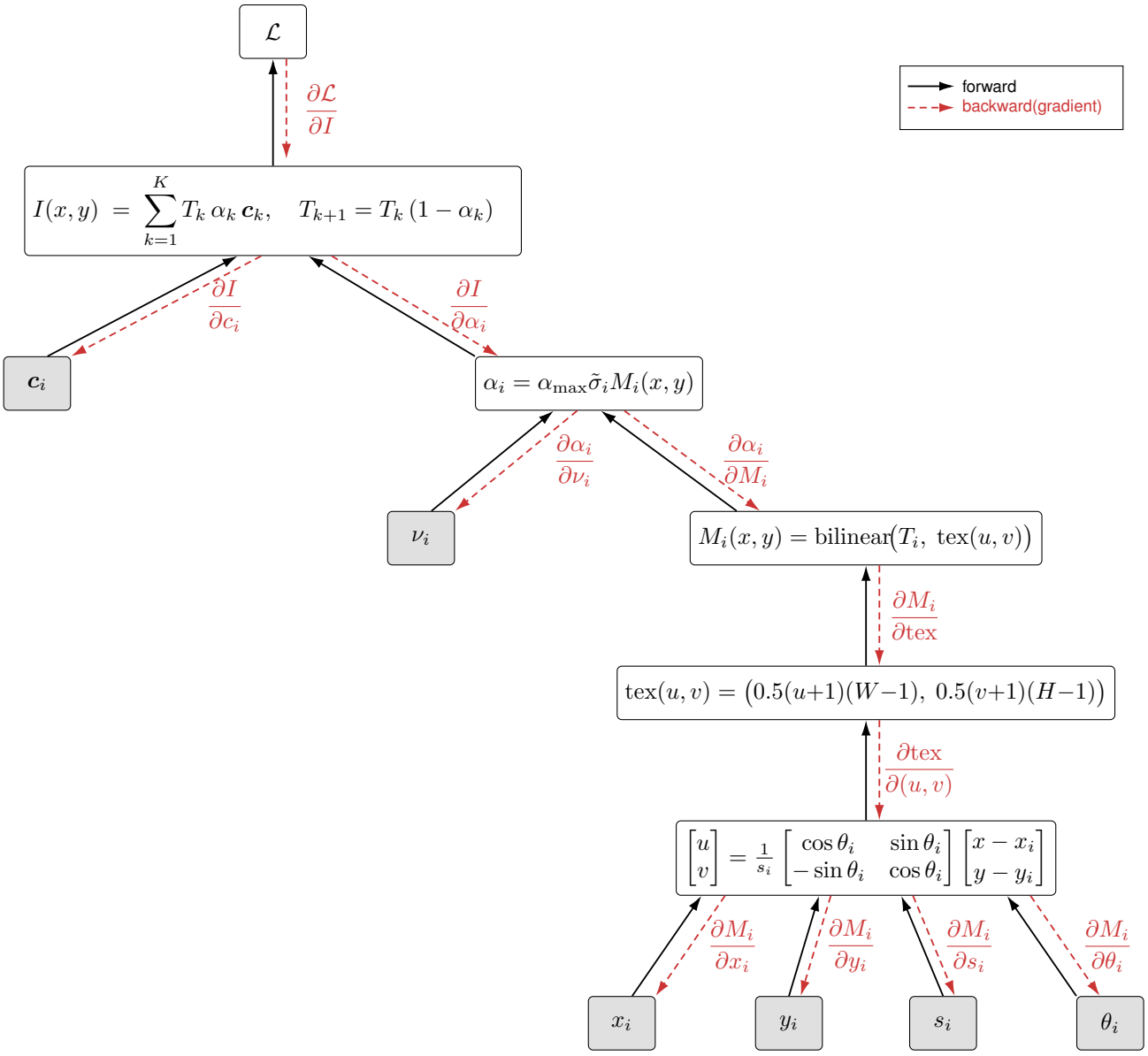


Figure S2. **DiffBMP computation tree (forward/backward).** Forward uses premultiplied-alpha Over in front-to-back order; per-primitive opacity is a sigmoid-scaled value modulated by a template mask sampled at rotated/scaled coordinates. Learnable leaves (gray) are $c_i, \nu_i, x_i, y_i, s_i, \theta_i$. Red dashed arrows indicate gradient flow. See Sec. 3.1.1 for equations.

path and dispense with block-local reductions, aligning with our accuracy–throughput goal.

S1.3. CUDA Implementation Details

In $T = 32$ settings, we partition the canvas into 32×32 tiles on the CPU and assign each primitive to all tiles its bounding box overlaps. Each CUDA thread block processes one tile with 32×32 threads (one thread per pixel), achieving full pixel-level parallelism. We adopt a tile-and-bin CUDA pipeline, following tile-based differentiable splatting practices as in [21], adapted here for 2D bitmap primitives.

S1.3.1 Forward Pass

Primitive templates are loaded into shared memory per block. FP16 halves bandwidth and enables tensor core acceleration via `__hfma()` intrinsics.

S1.3.2 Backward Pass

The backward kernel uses the same grid structure (32×32 threads per tile-block). Each thread rebuilds its cached primitive list and computes gradients via the chain rule.

Algorithm 1 Forward Pass Kernel (FP16)

Require: N primitives $\{(x_i, y_i, s_i, \theta_i, \nu_i, \boldsymbol{c}_i)\}_{i=1}^{N}$, tile-primitive mapping
Ensure: Output image I, alpha channel I_α
Grid: $(\lceil W_{\text{canvas}}/32 \rceil, \lceil H_{\text{canvas}}/32 \rceil)$ blocks, each with 32×32 threads
Each thread handles pixel (x, y) in canvas coordinates
for each primitive i assigned to this tile **do**
    Transform $(x, y) \to (u, v)$ in primitive space via Eq. (1)
    Compute $M_i(x, y)$ via bilinear interpolation at (u, v)
    if $M_i(x, y) < \epsilon$ **then** skip
    end if
    Compute $\alpha_k = \alpha_{max} \cdot \sigma(\nu_i) \cdot M_i(x, y)$, $\boldsymbol{c}_k = \sigma(\boldsymbol{c}_i)$
    Cache $\alpha_k, \boldsymbol{c}_k$ in global memory (FP16)
end for
Alpha composite: $T \leftarrow 1$, $C \leftarrow (0, 0, 0)$
for each cached primitive k **do**
    Store T_k for backward; $C \leftarrow C + T\alpha_k \boldsymbol{c}_k$; $T \leftarrow T(1-\alpha_k)$
end for
Write $I[x, y] \leftarrow C$, $I_\alpha[x, y] \leftarrow 1 - T$

For the Over operator, gradients depend on suffix sums as shown in Sec. 3.1.2. Geometric gradients $(x_i, y_i, s_i, \theta_i)$ flow through bilinear sampling of $M_i(x, y)$ [7] as in Sec. 3.1.2, where $\frac{\partial M_i(x,y)}{\partial u}$ and $\frac{\partial M_i(x,y)}{\partial v}$ come from bilinear interpolation. We accumulate gradients using packed `__half2` atomic operations for parameter pairs (x_i, y_i), (ν_i, s_i), $(\theta_i, c_{r,i})$, and $(c_{g,i}, c_{b,i})$, which halves atomic contention while maintaining accuracy compared to unpacked FP16 atomics.

S1.3.3 PSD Export

For per-primitive layer generation, we avoid atomics by rendering each primitive to its own cropped buffer. The export uses higher resolution (*e.g.*, $2\times$ or $4\times$) and proceeds in two stages: First, we compute the bounding box bbox_i for each primitive at the export scale ρ in parallel across N threads. Second, we launch a 3D CUDA grid with dimensions (tiles in x, tiles in y, primitives), where each primitive i is rendered in parallel to its own layer L_i by computing $M_i(x, y)$, α, and $\boldsymbol{c}$ as in the forward pass, and writing to local coordinates within bbox_i. This primitive-level parallelism eliminates atomic operations, while memory scales with bounding box areas rather than full canvas, enabling 4K+ exports. Layers are editable in Photoshop/After Effects.

S1.4. Heuristics and Losses for Dynamic and Spatially Constrained Rendering

S1.4.1 Dynamic DiffBMP for Videos

DiffBMP can be easily extended to rendering sequential frames by warm starting from previous frames (initialize Θ^f from $\Theta^{f-1\star}$), as in [12] for 3DGS. We add two lightweight controls here, targeting two specific problems: (i) over-dominant "stuck" primitives in changing regions and (ii) drift in static regions that causes flicker.

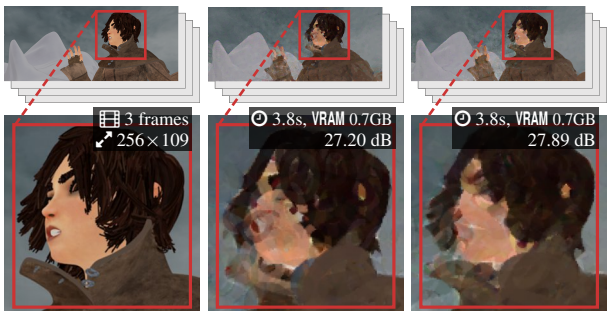
Primitives "Stuck" in regions of change. Fig. S3 illustrates the "stuck" failure and why it occurs. After warm start, new foreground content in I^f may appear where the previous frame I^{f-1} contained background. Without rigidity constraints [12], the optimizer takes the steepest path by *recoloring* background primitives instead of *relocating* the correct foreground ones. This is a situation we do not want. Large, opaque, front-ordered primitives that sit over high inter-frame change absorb the gradients the other primitives. They just get recolored, suppressing high-frequency detail and leaving the finer primitives behind them suboptimal.

Primitives modified in static region. Updating a single primitive affects all pixels under its footprint—including regions with no inter-frame change. These unintended edits in static areas perturb the loss and trigger compensatory updates in neighboring primitives, creating a cascade that propagates across the frame (see Fig. S4b). To arrest this drift, we compute at each step a difference mask $D := \mathbb{1}(I^{f-1}_{\text{target}} \neq I^{f}_{\text{target}})$ and *freeze* every primitive whose bounding box does not intersect D; only primitives overlapping D are allowed to update. This localizes parameter changes to actually changing content, prevents the cascade and visible flicker.

Algorithm 2 formalizes the procedure: (1) compute per-primitive bounding boxes $\mathcal{B}_i$ and freeze flags from the inter-frame difference mask D; (2) partition the canvas into an $n_h \times n_w$ spatial grid to localize decisions; (3) within each region, rank non-frozen primitives by a visibility-weighted score and decay the opacity logit of the top-K candidates by a factor $\eta \in (0, 1)$. A primitive is considered stuck if it satisfies all three criteria simultaneously: *large scale* ($s_i \geq \tau_{\text{scale}} \cdot W$), *high opacity* ($\alpha_i \geq \tau_\alpha$, where $\alpha_i = \alpha_{\max}\sigma(\nu_i)$), and *front z-order* (depth rank above the ζ percentile within its region). Here, (n_h, n_w) controls spatial granularity, K caps interventions per region, ζ targets front-most strokes, τ_{scale} and τ_α gate eligibility, and η controls decay strength. The procedure is lightweight, data-agnostic, and adds negligible overhead.

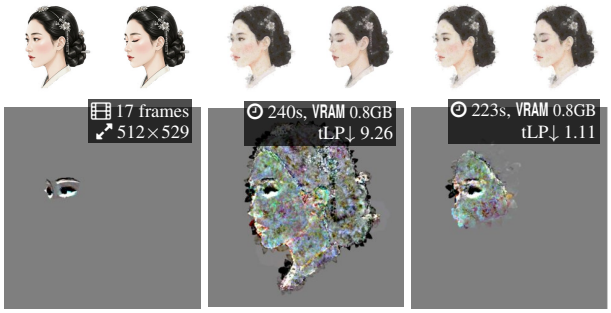
S1.4.2 Rendering with Spatial Constraint

Re-initialization Mechanism. When enabled, Primitives with $\sigma(\nu_i) <$ `prune_threshold` (typically 0.3) are re-initialized. This occurs every `prune_iterations` (50), excluding an initial warmup period and final iterations (optional) to prevent destabilization. Instead of pruning the transparent primitives as in [13], pruned primitives are re-initialized randomly, following our Structure-aware initial-



(a) Target Frame (b) Initialized from $\Theta^{f-1\star}$ (c) (b) + Removing Stuck primitives

Figure S3. **Frame-wise Fidelity Heuristic of Dynamic DiffBMP.** Warm-starting from $\Theta^{f-1\star}$ can leave an over-dominant primitive sitting over a high-change area; rather than relocating, it gets recolored and suppresses local detail, as in b. We apply *Removing Stuck Primitives*: adaptively decaying the opacity of large, opaque, front-ordered strokes so finer primitives behind them take over. This restores facial detail and improves fidelity under the same budget.



(a) Ground Truth (b) Initialized from $\Theta^{f-1\star}$ (c) (b) + Freezing Unchanged Primitives

Figure S4. **Anti-flicker via freezing unchanged regions.** We visualize inter-frame change with an error map $E = 5{\cdot}(I_1 - I_{17}) + 127$, where I_1 is the *first* frame and I_{17} the *last* frame. Mid-gray (≈ 127) indicates no change; brighter/darker values denote larger differences. If all primitives are freely optimized, spurious updates appear even in static regions, producing widespread flicker, as seen in b. Our remedy c freezes primitives, concentrating updates only where the video truly changes.

ization (Sec. 3.2.2).

S2. Experimental Configuration Details

This section provides a comprehensive description of the experimental settings used to generate all figures in the main paper. All configuration files are available in the `configs/` directory of our repository. Parameters not explicitly specified in individual configs default to values defined in `pydiffbmp/util/constants.py`.

Algorithm 2 Dynamic DiffBMP for Videos

Require: Primitives $\{\Theta_i = (x_i, y_i, s_i, \theta_i, \nu_i, \boldsymbol{c}_i)\}_{i=1}^{N}$, spatial grid size (n_h, n_w), inter-frame difference mask $D \in \{0,1\}^{H\times W}$, thresholds $\{\tau_{\text{scale}}, \tau_\alpha, \zeta\}$, per-region budget K, decay factor η
Ensure: Updated opacity logits $\{\nu_i^\star\}_{i=1}^{N}$

Step 1: Compute freeze masks
for $i = 1$ to N **do**
  Compute bounding box $\mathcal{B}_i$ from $(x_i, y_i, s_i, \theta_i)$
  $\text{freeze}_i \leftarrow \mathbb{1}(\mathcal{B}_i \cap \text{support}(D) = \emptyset)$ ▷ freeze if no overlap with change
end for
Step 2: Partition canvas into $n_h \times n_w$ spatial regions
for each region $\mathcal{R}_j$ in $\{1, \ldots, n_h \times n_w\}$ **do**
  $\mathcal{P}_j \leftarrow \{i \mid (x_i, y_i) \in \mathcal{R}_j\}$ ▷ primitives in region j
  Step 3: Score primitives by visibility and stuck criteria
  for each primitive $i \in \mathcal{P}_j$ **do**
    $\alpha_i \leftarrow \alpha_{\max} \cdot \sigma(\nu_i)$
    Evaluate stuck criteria:
      $c_1(i) \leftarrow \mathbb{1}(s_i \geq \tau_{\text{scale}} \cdot W)$ ▷ large scale
      $c_2(i) \leftarrow \mathbb{1}(\alpha_i \geq \tau_\alpha)$ ▷ high opacity
      $c_3(i) \leftarrow \mathbb{1}(\text{depth-rank}_i \geq \zeta \cdot |\mathcal{P}_j|)$ ▷ front z-order
    $\text{is\_stuck}_i \leftarrow (c_1(i) + c_2(i) + c_3(i) = 3) \wedge (\neg \text{freeze}_i)$
    $\text{score}_i \leftarrow s_i \cdot \alpha_i \cdot \text{is\_stuck}_i$
  end for
  Step 4: Select and decay top-K stuck primitives
  $\mathcal{S}_j \leftarrow \text{top-}K\{\text{score}_i \mid i \in \mathcal{P}_j\}$ ▷ by descending score
  for each $i \in \mathcal{S}_j$ **do**
    $\nu_i^\star \leftarrow \eta \cdot \nu_i$ ▷ reduce opacity
  end for
end for
return $\{\nu_i^\star\}_{i=1}^{N}$

S2.1. Common Hyperparameters

Table S1 summarizes hyperparameters and settings that are consistently used across all or most experiments. These form the foundation of our optimization pipeline.

S2.2. Figure-Specific Configurations

Table S2 details the key parameters that vary across different figures in the main paper. Each row corresponds to a specific subfigure or experiment. For more detail and specs, please check the code provided in the supplementary material.

S2.3. Detailed Descriptions of Main Figures

Figure 1: Teaser Examples. The teaser presents various applications of DiffBMP. **Seurat Painting Composition (1a)** uses 2000 fingerprints and 2000 autographs on separate image regions at 1024px resolution. The fingerprint primitives use `radial_transparency=true` to create smooth fading effects. **The Marilyn Monroe assemblage (1b)** employs 300 brand logos [9] with `c_blend=1.0` to pre-

Table S1. **Common hyperparameters used across experiments.** These settings are applied to all experiments unless explicitly overridden in figure-specific configurations. Values marked with † come from `constants.py` when not specified in config files.

Category	Parameter	Value / Description
Initialization	`initializer`	`structure_aware` (default for most; `random` for Fig. 8)
	`v_init_bias`	-4.0 (yields $\sigma(-4) \approx 1.8\%$ initial opacity)
	`std_c_init`†	0.02 (color initialization noise std)
	`variance_window_size`†	7 (local variance computation window)
	`variance_base_prob`†	0.1 (base sampling probability for low-variance areas)
	`max_prims_per_pixel`†	100 (200 for spatially constrained cases)
Optimization	`num_iterations`	100–500 (task-dependent; see Tab. S2)
	`learning_rate.default`	0.1 (base LR; scaled by gains below)
	`lr_gain_x`†	10.0
	`lr_gain_y`†	10.0
	`lr_gain_r`†	10.0 (scale parameter)
	`lr_gain_v`†	1.5 (5.0 for spatially constrained; see Tab. S2)
	`lr_gain_theta`†	1.0
	`lr_gain_c`†	1.0
	`do_decay`	`true` (exponential LR decay)
Rendering	`do_gaussian_blur`	`true` (soft rasterization, Sec. 3.2.1)
	`blur_sigma`†	1.0
	`alpha_upper_bound`	1.0 (0.7 for spatially constrained)
Loss	`loss_config.type`	`mse` or `combined` (see Tab. S2)
	`bg_color`†	`white` (default); `random` for no-bg cases
Postprocessing	`psd_scale_factor`	2.0 or 4.0 (export resolution multiplier)
	`compute_psnr`	`true` (for quantitative evaluation)

serve the original logo colors, using a combined loss with grayscale L1 (weight 1.0) and MSE (weight 0.2) to maintain luminance structure. **The flower video composition (1c)** consists of two parts. First, background flower primitives are optimized using a circle mask. Second, for the foreground video of the girl, an initial frame is optimized with a spatially constrained image (face region). The subsequent frames use warm-start initialization from the previous frame ($\Theta^{f-1\star}$), combined with our dynamic heuristics: removing stuck primitives and freezing unchanged regions, as detailed in Sec. 3.3.1 of the main paper.

Figure 2: An illustration of the algorithm flow of DiffBMP. The target image I^{target} is spatially constrained using four semantic masks (hair, skin, neck, cloth) from the CelebAMask-HQ dataset [10]. The image I depicts the intermediate output after 10 iterations, which continues for a total of 100 iterations.

Figure 4: Heuristics for Dynamic DiffBMP. The example uses an 8-frame clip from *The Gold Rush* (Charlie Chaplin). Warm-starting the current frame from $\Theta^{f-1\star}$ can trap the optimizer in a local minimum—(b) shows an over-dominant primitive stuck across the face that washes out detail. In (c) we apply *Removing Stuck Primitives*: adaptively decaying the opacity of large, front-ordered, high-opacity strokes so finer primitives take over, restoring facial detail and improving per-frame fidelity.

Figure 5: DiffVG vs ours. These experiments demonstrate DiffBMP's superior performance and versatility, particularly with complex bitmap or vector primitives where DiffVG struggles. All experiments use 2000 primitives at 512px resolution and are optimized for 100 iterations.

Figure 6: Noisy Canvas Ablation. This ablation demonstrates the effect of canvas background on primitive coverage (Sec. 3.2.3). Configuration (b) uses `bg_color=white`, while (c) uses `bg_color=random`, encouraging primitives to fill all regions by blending with uniform noise per iteration.

Figure 7: Alpha Loss and Re Initialization Ablation. These experiments ablate the alpha loss and re-initialization mechanisms for spatially constrained rendering(Sec. 3.3.2). Configuration (a) uses MSE loss only (but it follows the spatially constrained initialization), (b) adds the opacity loss component (weight 0.3) without re-initialization, and (c) enables both. We re-initialize primitives with opacity below 0.3 every 50 iterations, with a warmup period of 199 iterations. All use `gain_v=5.0` to accelerate opacity optimization.

Table S2. **Figure-specific experimental configurations.** This table shows parameters that differ across experiments, including primitive details, number of primitives (N), image resolution, scale range, iterations, and special settings.

Figure	Primitive(s)	N	Resolution	Scale Range	Iter.	Special Settings
Figure 1: Teaser examples						
1(a)-fingerprint	fingerprint.jpg	2000	1024	[2, 8]	300	`radial_transparency=true`, `c_blend=0.0`
1(a)-autograph	autograph_seurat.png	2000	1024	[2, 10]	300	`c_blend=0.0`
1(b)	logos/*.png (300 logos)	300	512	[4, 20]	100	`c_blend=1.0`, grayscale L1 + MSE loss
1(c)-flowers	5 flower types	1000	512	[2, 31]	300	`exist_bg=false`, mask, pruning
1(c)-girl	2 flower types	1000	512	[2, 20]	300+100	`exist_bg=false`, initial+sequential (17 frames), Algorithm 2, freeze unchanged
Figure 2: An illustration of the algorithm flow of DiffBMP						
2	4 flower types	500	512	[2, 20]	10	`exist_bg=false`, 4 masks, re-initialization
Figure 4: Heuristics for Dynamic DiffBMP						
4(b)	cane.png, hat.png	1000	256	[8, 48]	100	sequential (8 frames), $\Theta^{f-1\star}$ init, MSE+perceptual loss
4(c)	cane.png, hat.png	1000	256	[8, 48]	100	sequential (8 frames), $\Theta^{f-1\star}$ init, Algorithm 2, MSE+perceptual loss
Figure 5: DiffVG vs ours						
5(a)	square.svg	2000	512	[2, 10]	100	`use_fp16=false`
5(b)	grass.svg	2000	512	[2, 10]	100	`use_fp16=false`
5(c)	paw.jpg	2000	512	[2, 10]	100	`use_fp16=false`
Figure 6: Noisy Canvas Ablation						
6(b)	Lisc_lipy.jpg	1000	312	[2, 50]	100	`bg_color=white`
6(c)	Lisc_lipy.jpg	1000	312	[2, 50]	100	`bg_color=random`
Figure 7: Alpha Loss and Re Initialization Ablation						
7(a)	paw_complicated.png	3000	512	[2, 64]	300	`exist_bg=false`, `lr_gain_v=5.0`, MSE only w/o mask
7(b)	paw_complicated.png	3000	512	[2, 64]	300	`exist_bg=false`, `lr_gain_v=5.0`, MSE+alpha loss, w/o re-initialization
7(c)	paw_complicated.png	3000	512	[2, 64]	300	`exist_bg=false`, `lr_gain_v=5.0`, MSE+alpha loss, w/ re-initialization
Figure 8: CLIP-guided generation						
8-amazon	flowers/*.png	500	224	[2, 20]	500	CLIP loss, random init, custom LR
8-galaxy	maxwell_eq*.png	1000	224	[19, 20]	500	CLIP loss, random init, custom LR
8-witch	bat.png	500	224	[19, 20]	500	CLIP loss, random init, custom LR

Figure 8: CLIP-Guided Generation. These experiments use text-prompt guidance via CLIP loss instead of target images. The initialization is `random` rather than structure-aware as no target image structure is available. All configurations use ViT-B/32 CLIP model with 16 augmentations, normalized CLIP embeddings. The Amazon rainforest uses primitives with small scale [2, 20], while galaxy and witch use larger primitives [19, 20] to create distinct visual styles. Additionally, we apply the negative prompt "blurry" with a weight of 0.1 to mitigate unwanted artifacts.

Spatial Constraint Implementation. Experiments with `exist_bg=false` (Figures 1c, 2, 6, 7) optimize foreground-only rendering using Eq. 9 in the main paper. The alpha loss weight is typically 0.3, and `gain_v` is increased to 5.0 to facilitate rapid opacity adjustments. The `max_prims_per_pixel` is increased to 200 for these cases to ensure adequate coverage in complex regions.

Dynamic DiffBMP Implementation. For dynamic DiffBMP (Figures 1c and 4), we optimize sequential frames by warm-starting from the previous frame's optimized parameters ($\Theta^{f-1\star}$). To prevent stuck primitives and flickering, we employ two heuristics: (1) **Removing Stuck Prim-**

itives adaptively reduces opacity of over-dominant primitives that satisfy all three criteria simultaneously (large scale $s_i \geq \tau_{\text{scale}} \cdot W$, high opacity $\alpha_i \geq \tau_\alpha$, front z-order exceeding the ζ percentile). We partition the canvas into an $(n_h{=}4) \times (n_w{=}4)$ spatial grid and select up to $K{=}4$ problematic primitives per region, reducing their ν_i by a factor η. This occurs at specific epochs ([20, 45, 70] for Fig. 1c; [20, 40, 60, 80] for Fig. 4c). (2) **Freezing Unchanged Regions** computes an inter-frame difference mask D and freezes primitives whose bounding boxes do not intersect D, preventing spurious updates in static regions. Hyperparameters: $\tau_{\text{scale}}{=}0.1$, $\tau_\alpha{=}0.7$, $\zeta{=}0.7$, $\eta{=}0.3$ (Fig. 1c) or 0.1 (Fig. 4c).

FP16 Precision. Most experiments use `use_fp16=true` for memory efficiency. The FP16 implementation maintains accuracy through packed `__half2` atomic operations as described in Sec. 3.1.2.

S3. Naive PyTorch Baseline Comparison

To demonstrate the efficiency of our CUDA implementation, we have compared against a naive PyTorch-based renderer that uses standard tensor operations without tile-based culling or specialized kernels, in Sec. 4.1.

S3.1. Naive PyTorch Implementation

The naive PyTorch approach processes all primitives for every pixel without spatial optimization, as shown in Algorithm 3.

Algorithm 3 Naive PyTorch Baseline (Sequential)

Require: N primitives $\{(x_i, y_i, s_i, \theta_i, \nu_i, \boldsymbol{c}_i)\}_{i=1}^{N}$
Ensure: Rendered image I

$X, Y \leftarrow$ `meshgrid()` over canvas ▷ Full (H_{canvas}, W_{canvas}) grids
$I \leftarrow$ `zeros()`; $T \leftarrow$ `ones()`
for $i = 0$ **to** $N-1$ **do** ▷ Sequential CPU loop, no parallelism
  $\Delta x \leftarrow X - x_i$; $\Delta y \leftarrow Y - y_i$ ▷ (H, W) tensors per primitive
  $u, v \leftarrow$ rotate/scale $(\Delta x, \Delta y)$ by θ_i, s_i
  $mask \leftarrow$ `torch.grid_sample`$(M_i, (u, v))$ ▷ Generic interpolation
  $\alpha_i \leftarrow \alpha_{max} \cdot \sigma(\nu_i) \cdot mask$; $\boldsymbol{c}_i \leftarrow \sigma(\boldsymbol{c}_i)$
  $I \leftarrow I + T \cdot \alpha_i \cdot \boldsymbol{c}_i$; $T \leftarrow T \cdot (1 - \alpha_i)$
end for
return I

S3.2. Key Inefficiencies and Performance Analysis

Table S3 summarizes the architectural differences between the naive PyTorch baseline and our optimized CUDA implementation. The naive PyTorch baseline suffers from several critical bottlenecks:

Table S3. Comparison of naive PyTorch vs. our CUDA implementation

Aspect	Naive PyTorch	Our CUDA
Spatial culling	None	Tile-based binning
Primitive loop	Sequential (N)	Parallel per-pixel
Memory per iteration	$O(N \times HW)$	$O(k \times HW), k \ll N$
Precision	FP32 only	FP16 + FP32 mixed
Intermediate tensors	(H, W) per primitive	Small per-pixel cache
Atomic operations	Many (unoptimized)	Packed `__half2`
Gradient accumulation	Global sync	Per-pixel atomic
Speedup	**1× (baseline)**	**30-50×**

Major Bottlenecks. (1) **No spatial culling**—every primitive is evaluated at all $H \times W$ pixels, yielding $O(N \times HW)$ work even when most primitives contribute nothing. For 1000 primitives on 512^2 canvas, this is 262M wasted evaluations. (2) **Massive memory allocation**—each primitive creates full-canvas tensors for $\Delta x, \Delta y, u, v, mask$ (∼5 GB at FP32 for our example). (3) **Sequential loop**—the Python `for` loop prevents primitive-level parallelism and forces repeated kernel launches. (4) **Generic operations**—`torch.grid_sample()` handles general cases rather than exploiting our structure.

Our CUDA kernel achieves 30-50× speedup via tile-based culling (10-100× work reduction), per-pixel parallelism, compact caches storing only $k \ll N$ affecting primitives, and FP16 optimization with tensor cores.

S4. More Results

Qualitative Results for Table 3. Across our experiments, DiffBMP handles a wide variety of primitive libraries and target images within a single optimization framework, yet the optimization behavior is consistent. It first recovers the global layout and dominant color structure of a target and then progressively sharpens local details as optimization proceeds. This behavior is clearly visible in Fig. S7, where we visualize the same configurations used in Table 3 of the main paper. Early iterations already place large primitives that capture silhouettes and large color regions, while later iterations refine edges, textures, and small features as overlapping primitives are repurposed and reweighted. The three rows in Fig. S7 cover different combinations of primitives and targets including photographic images, portraits, and more stylized graphics, which shows that this coarse to fine optimization pattern holds across diverse content without task specific tuning.

The Number of Primitives N. In Fig. S8, we vary the primitive count N from 1000 to 4000 for several pairings of primitive collections and target images. Larger values of N mainly improve sharpness, fine texture, and small details, resulting in PSNR gains.

Scaling with primitive footprint and tile density.. To better understand how the rendering workload scales with the primitive count N, we analyze three complementary descriptors: the normalized primitive footprint $A_i/(HW)$, the primitive tile-hit degree τ_i, and the per-tile primitive density d_t. We use the same primitive families as in the Seurat painting composition of Fig. 1a, namely fingerprint and autograph primitives. To expose a broader primitive size distribution, we widen the allowed scale range to $[2, 50]$ in this analysis instead of using the original Fig. 1a ranges. As in Fig. 1a, radial transparency is enabled for the fingerprint primitives. Fig. S5 summarizes the resulting distributions of $A_i/(HW)$ and τ_i, together with the measured runtime, as N increases. Here, A_i denotes the area of the padded screen-space bounding box of primitive i, normalized by the image area HW. τ_i denotes the number of tiles intersected by that primitive. d_t denotes the number of primitives assigned to tile t during CPU binning.

For both primitive families, increasing N shifts the footprint CDF toward smaller values and slightly lightens the τ_i distribution, indicating that individual primitives become smaller and intersect fewer tiles on average. However, runtime still increases monotonically with N. Fig. S6 further shows that, despite the smaller average footprint of individual primitives, visually active regions accumulate progressively denser tile-local primitive lists as N grows. Since the total primitive–tile interaction count satisfies $\sum_i \tau_i = \sum_t d_t$, the observed scaling depends not only on primitive count, but also on how primitive overlap is spatially distributed across tiles.

S5. More discussion

Initialization with autoregressive/RL methods. Initializing our method with autoregressive/RL-based methods [2, 4–6, 8, 11, 14, 16–18, 20] and then fine-tuning with DiffBMP could potentially yield high-quality images with fewer primitives. However, the primary goal of this work is to provide a general rendering engine usable for diverse objectives. Utilizing these existing methods, which often rely on primitives with pre-defined shapes, falls outside the scope of our current research. Nevertheless, given that we have created an easy-to-hack Python interface, we hope that creators will find it easy to integrate these techniques.

Diff-3D-Raster. Similar to recent work on non-Gaussian splatting in 3D [1, 3, 19], our work could be extended to 3D for artistic expression, even if it does not offer computational advantages for real-time rendering. However, existing studies often use simple analytic primitives like polyhedra, and we anticipate that 3D raster primitives would present a challenge due to their significantly higher computational demands.

References

[1] Haodong Chen, Runnan Chen, Qiang Qu, Zhaoqing Wang, Tongliang Liu, Xiaoming Chen, and Yuk Ying Chung. Beyond gaussians: Fast and high-fidelity 3d splatting with linear kernels, 2024. 8

[2] Yaroslav Ganin, Tejas Kulkarni, Igor Babuschkin, SM Ali Eslami, and Oriol Vinyals. Synthesizing programs for images using reinforced adversarial learning. *ICML*, 2018. 8

[3] Jan Held, Renaud Vandeghen, Abdullah Hamdi, Adrien Deliege, Anthony Cioppa, Silvio Giancola, Andrea Vedaldi, Bernard Ghanem, and Marc Van Droogenbroeck. 3D convex splatting: Radiance field rendering with 3D smooth convexes. *CVPR*, 2025. 8

[4] Aaron Hertzmann. Painterly rendering with curved brush strokes of multiple sizes. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 453–460, 1998. 8

[5] Xin Huang and Minglun Gong. Attention-guided deep reinforcement learning for realistic neural painting. *IEEE Access*, 2025.

[6] Zhewei Huang, Wen Heng, and Shuchang Zhou. Learning to paint with model-based deep reinforcement learning. *ICCV*, 2019. 8

[7] Max Jaderberg, Karen Simonyan, Andrew Zisserman, et al. Spatial transformer networks. *NeurIPS*, 2015. 1, 3

[8] Biao Jia, Chen Fang, Jonathan Brandt, Byungmoon Kim, and Dinesh Manocha. PaintBot: A reinforcement learning approach for natural media painting. *arXiv preprint arXiv:1904.02201*, 2019. 8

[9] Koustubh Khandekar. Popular brand Logos - Image Dataset. Kaggle, 2021. Accessed: 6 November 2025. 4

[10] Cheng-Han Lee, Ziwei Liu, Lingyun Wu, and Ping Luo. Maskgan: Towards diverse and interactive facial image manipulation. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 5549–5558, 2020. 5

[11] Songhua Liu, Tianwei Lin, Dongliang He, Fu Li, Ruifeng Deng, Xin Li, Errui Ding, and Hao Wang. Paint transformer: Feed forward neural painting with stroke prediction. *ICCV*, 2021. 8

[12] Jonathon Luiten, Georgios Kopanas, Bastian Leibe, and Deva Ramanan. Dynamic 3d gaussians: Tracking by persistent dynamic view synthesis. *3DV*, 2024. 3

[13] Didier Stricker Marcel Rogge. Object-centric 2d gaussian splatting: Background removal and occlusion-aware pruning for compact object models. *arXiv preprint arXiv:2501.08174*, 2025. 3

[14] John FJ Mellor, Eunbyung Park, Yaroslav Ganin, Igor Babuschkin, Tejas Kulkarni, Dan Rosenbaum, Andy Ballard, Theophane Weber, Oriol Vinyals, and SM Eslami. Unsupervised doodling and painting with improved spiral. *arXiv preprint arXiv:1910.01007*, 2019. 8

[15] Thomas Porter and Tom Duff. Compositing digital images. In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pages 253–259, 1984. 1

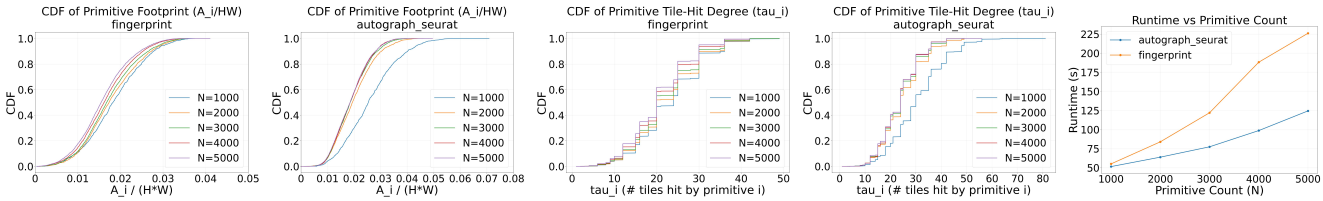


Figure S5. **Additional scaling statistics for the Seurat composition in Fig. 1a.** To expose a broader primitive size distribution, this analysis widens the allowed scale range to $[2, 50]$ while sweeping the primitive count N. Fingerprint primitives retain radial transparency as in Fig. 1a. From left to right, the panels show the CDF of the normalized primitive footprint $A_i/(HW)$ for *fingerprint*, the CDF of $A_i/(HW)$ for *autograph_seurat*, the CDF of the primitive tile-hit degree τ_i for *fingerprint*, the CDF of τ_i for *autograph_seurat*, and the measured runtime versus N. Here, A_i denotes the area of the padded screen-space bounding box of primitive i, normalized by the image area HW. τ_i denotes the number of tiles intersected by that primitive.

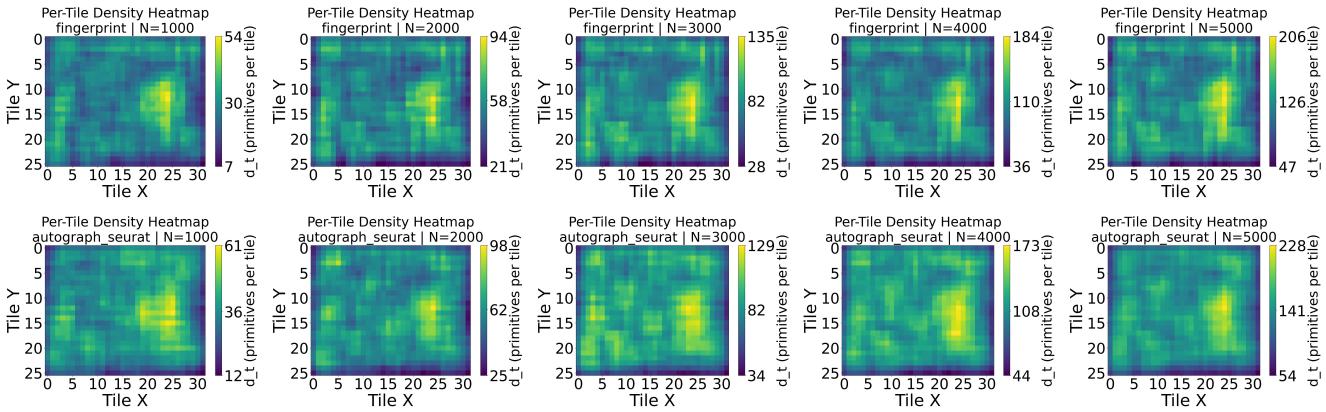


Figure S6. Spatial distribution of per-tile primitive density d_t for the same scaling analysis setting as Fig. S5, with scale range $[2, 50]$ and increasing primitive count N. Each heatmap visualizes the number of primitives assigned to tile t during CPU binning. The top row shows *fingerprint*, and the bottom row shows *autograph_seurat*. In each row, $N \in \{1000, 2000, 3000, 4000, 5000\}$ increases from left to right. Each panel uses its own color scale for readability, so absolute density values should be interpreted using the corresponding colorbar.

[16] Peter Schaldenbrand and Jean Oh. Content masked loss: Human-like brush stroke planning in a reinforcement learning painting agent. *AAAI*, 2021. 8

[17] Yiren Song, Shijie Huang, Chen Yao, Hai Ci, Xiaojun Ye, Jiaming Liu, Yuxuan Zhang, and Mike Zheng Shou. ProcessPainter: Learning to draw from sequence data. *SIGGRAPH Asia*, 2024.

[18] Yizhe Tang, Yue Wang, Teng Hu, Ran Yi, Xin Tan, Lizhuang Ma, Yu-Kun Lai, and Paul L Rosin. AttentionPainter: An efficient and adaptive stroke predictor for scene painting. *arXiv preprint arXiv:2410.16418*, 2024. 8

[19] Nicolas von Lützow and Matthias Nießner. LinPrim: Linear primitives for differentiable volumetric rendering. *arXiv preprint arXiv:2501.16312*, 2025. 8

[20] Qian Wang, Cai Guo, Hong-Ning Dai, and Ping Li. StrokeGAN painter: Learning to paint artworks using stroke-style generative adversarial networks. *Computational Visual Media*, 9(4):787–806, 2023. 8

[21] Vickie Ye, Ruilong Li, Justin Kerr, Matias Turkulainen, Brent Yi, Zhuoyang Pan, Otto Seiskari, Jianbo Ye, Jeffrey Hu, Matthew Tancik, and Angjoo Kanazawa. gsplat: An open-source library for gaussian splatting. *Journal of Machine Learning Research*, 26(34):1–17, 2025. 2

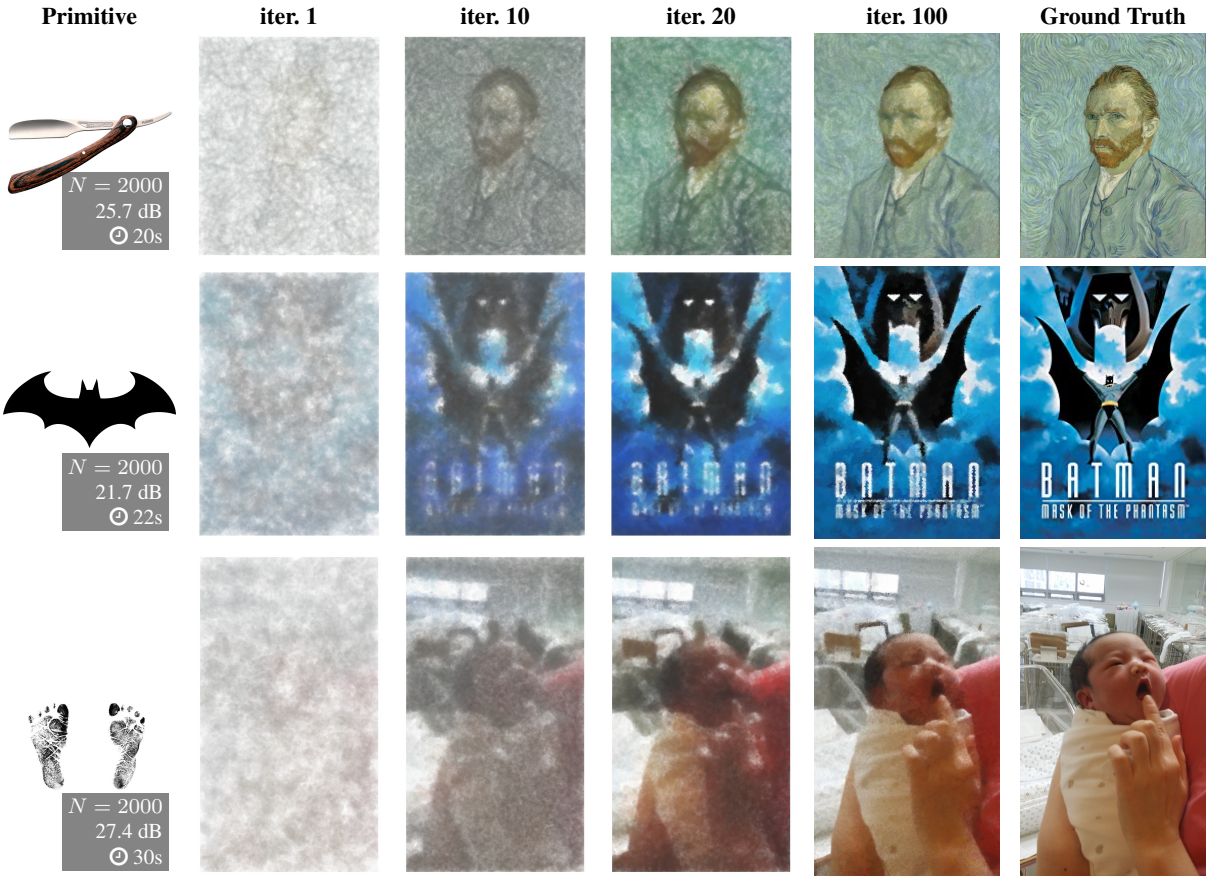


Figure S7. **Optimization progress across iterations for three different examples.** Each row shows the evolution from iteration 1 (left) to iteration 100 (right), illustrating how coarse structure emerges early and fine details are refined in later iterations. All three rows reuse the same primitive sets and target images as the ablation study in Table 3 of the main paper, enabling direct visual comparison with the quantitative results.

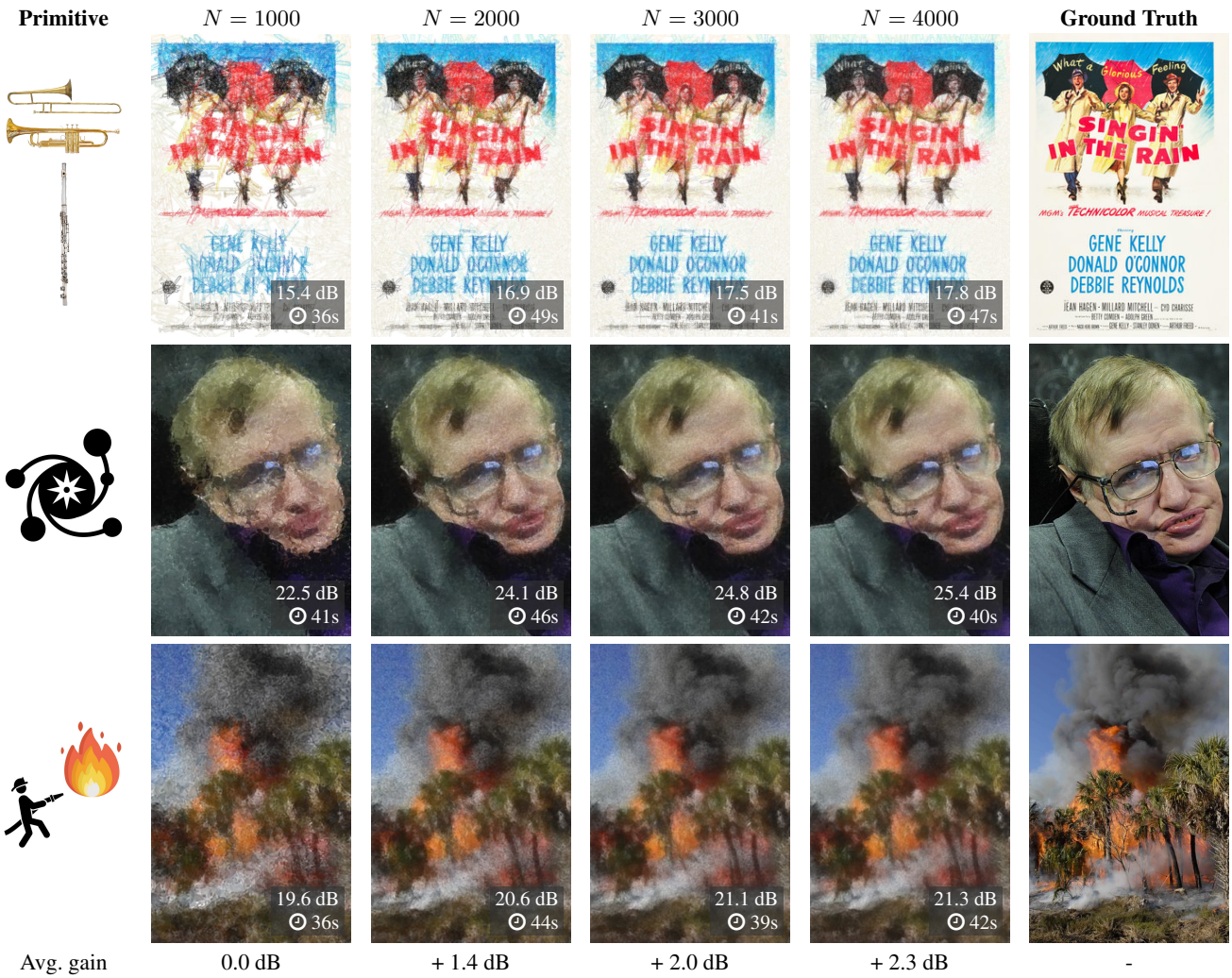


Figure S8. **Effect of primitive count on reconstruction quality across diverse targets and primitive types.** Each row uses a different primitive set (left) to approximate a different target image (right), while columns sweep the number of primitives N from 1000 to 4000. As N increases, our optimizer consistently sharpens details and improves PSNR across movie posters, portraits, and natural scenes, demonstrating robust approximation capability over varied image content and primitive shapes.