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

(a) Image composition with 3000 fingerprints / 2000 autographs
(b) Intrinsic-preserving graphic assemblage with 300 logos
(c) Video composition under spatial constraint w/ 2000 flowers, followed by post-processing

Figure 1. Explore the new creative horizons opened by DiffBMP. Essentially, DiffBMP makes the position, rotation, scale, color, and opacity of arbitrary bitmap images *differentiable* (*e.g.*, a fingerprint or the *'Seurat'* autograph in a). This capability extends operations previously limited to vector graphics to any bitmap primitive (*e.g.*, b: intrinsic-preserving graphic assemblage of Marilyn Monroe using numerous brand logos). DiffBMP is highly flexible and expandable, so that it can process videos, spatially constrained images, or both simultaneously, as shown in c(3, 4). Moreover, DiffBMP outputs 2D layers in an editable format (`.psd`), enabling diverse post-processing creation, as the blooming effect in c(1-3). The various image/video creations, including c, can be viewed in the supplementary video. GPU VRAM usage and runtimes were measured on an NVIDIA RTX 3090. Note that c's runtime excludes post-processing.

Abstract

We introduce ***DiffBMP****, a scalable and efficient differentiable rendering engine for a collection of bitmap images. Our work addresses a limitation that traditional differentiable renderers are constrained to vector graphics, given that most images in the world are bitmaps. Our core contribution is a highly parallelized rendering pipeline, featuring a custom CUDA implementation for calculating gradients. This system can, for example, optimize the position, rotation, scale, color, and opacity of thousands of bitmap primitives all in under 1 min using a consumer GPU. We employ and validate several techniques to facilitate the optimization: soft rasterization via Gaussian blur, structure-aware initialization, noisy canvas, and specialized losses/heuristics for videos or spatially constrained images. We demonstrate DiffBMP is not just an isolated tool, but a practical one designed to integrate into creative workflows. It supports exporting compositions to a native, layered file format, and the entire framework is publicly accessible via an easy-to-hack Python package.*[1]

1. Introduction

Many problems in the real world can be formulated as *optimization*: $\min_{\Theta} \mathcal{L}(f(\Theta))$, where $\Theta \in \mathbb{R}^p$ is the parameter, f

*Equal contribution. †Corresponding author. [1]pip install pydiffbmp is a system, and $\mathcal{L}$ is the loss function. For large-scale optimization problems (*i.e.*, $p \gg 1$), first-order methods [1] have become the standard, due to their computational efficiency and effectiveness. However, this imposes a critical prerequisite: the gradient $\frac{\partial f}{\partial \Theta}$ must be well-defined. Ideally, it should also be computationally efficient.

In computer vision and graphics, this same principle has given rise to the field of differentiable rendering [18]. To this end, significant research efforts have focused on enabling informative gradients to flow from the rendered output $f(\Theta)$ back to the 2D or 3D scene parameters Θ. This has allowed a vast array of inverse graphics tasks to be efficiently and effectively solved. To obtain spatial gradients from meshes where they are not naturally defined, researchers have either approximated the backward pass [17, 27] or reformulated the forward pass to be differentiation-friendly [24, 41]. Similar solutions have been employed for other representations like voxels [51] and point clouds [42, 53] to achieve end-to-end differentiability. While the fundamental frameworks of modern volume rendering methods like NeRF [33] and 3DGS [19] are inherently differentiable, considerable effort has been invested in modifying the forward pass [10, 34, 46] or creating custom CUDA kernels [19, 34, 52] for the efficient collection and accumulation of gradients.

Despite this progress, a critical gap remains in the 2D domain. Existing widely used frameworks are constrained to vector primitives. For example, DiffVG [23] and its follow-up works [3, 4, 23, 29] masterfully handle vector paths, which are very memory-efficient while flexible. But the vast majority of real-world 2D assets are not vector graphics; they are bitmap images. While the foundational mechanism for differentiably transforming raster images was introduced [14], its adoption for creation has been limited to non-general tasks like pattern composition [39]. Due to the inherent nature of bitmap images—being discrete, high-dimensional arrays of pixel values—they impose a substantial memory and computational burden. Consequently, a general and scalable engine for differentiably optimizing the transforms of thousands of bitmap images has not existed. This has made it impossible to automatically create compelling compositions, such as those in Fig. 1, using first-order optimization.

To open the door for solving a new class of problems using first-order optimization on bitmap images, we introduce DiffBMP: a highly parallelized differentiable rendering engine designed specifically for bitmap primitives. By leveraging custom CUDA kernels for both forward and backward passes, DiffBMP enables arbitrary raster images to become full participants in the gradient-based optimization ecosystem. Our main contributions are as follows:

- **Scalable Differentiable Framework:** A novel differentiable renderer that overcomes the memory and computational burdens of raster-based optimization, scaling to thousands of bitmap primitives.
- **Research Insights & Optimization Aids:** Analysis of gradient sparsity and convergence dynamics in bitmap optimization, addressed via soft rasterization, structure-aware initialization, and noisy canvas techniques.
- **Algorithmic Innovations & Extensions:** Specialized solutions for video modeling and spatially constrained rendering, integrated into artist-friendly workflows through a Python interface and `.psd` exports for post-processing.

2. Related Work

Differentiable Rendering for Vector Graphics. Differentiable rendering first emerged in the 3D domain as a technique to compute the gradient of a rendering output (typically a mesh) with respect to its parameters [2, 6, 17, 22, 24, 27]. The first practical system to differentiably connect the parameters of vector graphics with a 2D raster image was DiffVG [23]. Its high quality, computational efficiency, and user-friendly Python interface enabled various applications, including image vectorization [3, 4, 29, 40, 54, 55] and text-based vector graphics generation [9, 15, 44, 50]. Although non-DiffVG-based method [26] has recently been proposed, but also is confined to vector graphics. One might wonder if the function of DiffBMP could be achieved by first vectorizing complex bitmap primitives (albeit with significant complexity), applying an existing method like DiffVG, and then simply exporting the result back to a bitmap. However, as demonstrated in Sec. 4.1, we experimentally show that DiffVG struggles significantly with bitmap-level complexity or equivalent complex SVG primitives. This difficulty underscores the necessity of the proposed DiffBMP.

Differentiable Rendering for Bitmap Images. Differentiable rendering for bitmap images, while theoretically possible, lags significantly behind the vector graphics domain in computational efficiency and scalability. The foundational method for making raster image geometry differentiable is to apply differentiable operations, such as bilinear interpolation, to the image grid. This was introduced by Spatial Transformer Networks [14], originally for learning spatial invariance for feature maps within neural networks. Building on this mechanism, Reddy et al. [39] proposed a method for creating patterns from bitmap images. However, this approach was implemented without specialized parallelism and was limited to a relatively narrow task: composing repetitive, opaque (opacity=1) patterns. Table 1 compares DiffBMP with existing differentiable rendering methods for both vector graphics and bitmap images.

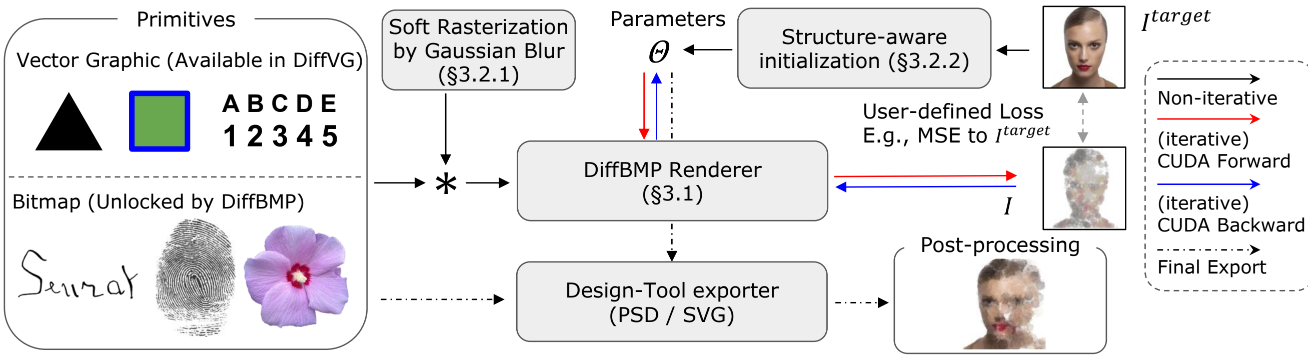


Figure 2. **An illustration of the algorithm flow of DiffBMP.** Our core contribution is the renderer for bitmap images, introduced in Sec. 3.1, which enables scalable and efficient differentiable rendering using a custom CUDA kernel. Additionally, we employ soft rasterization (Sec. 3.2.1) for improved gradient propagation and an initialization technique (Sec. 3.2.2) for better target approximation. After optimization, a native, layered file is generated, which enables diverse post-processing creation.

Table 1. **Comparison with Prior Work.** Prior work can be broadly classified into two main categories. DiffVG [23] is high-performing and broadly applicable, but it is limited to vector primitives. Due to the difficulty of matching DiffVG's performance using bitmap primitives, only a few studies, such as Reddy et al. [39], have attempted implementations for specialized tasks, but their efficiency and performance fall short of DiffVG. To put simply, DiffBMP aims to be the bitmap primitive counterpart to DiffVG.

2D Differentiable rendering method	**Primitive type**	**Applications**	**Opacity Support**	**Parallelism**	**Engine / Interface**
Differentiable Composition by Reddy et al. [39]	Bitmap	Specialized for *patterns* *e.g.*, manipulation, tiling	✗	✗	Pytorch / Python
DiffVG [23] and its extensions [3, 4, 29, 40, 54, 55]	Vector	Very diverse *e.g.*, vectorization, painterly, etc.	✓	✓	C++ / Python
DiffBMP (Ours)	Bitmap	Very diverse. *e.g.*, painterly, assemblage, video, etc.	✓	✓	CUDA / Python

3. The DiffBMP Framework

3.1. Differentiable Forward / Backward

The core of our method is an end-to-end differentiable module that can render a collection of raster primitives with spatial and color transformations. To achieve practical speed/memory performance, we implement a custom CUDA kernel for the forward/backward pass of the renderer, and made available to developers through a Python interface.

3.1.1 Forward Rendering Process Basic

Coordinate Transformation and Primitive Sampling. Here, we present how spatial transformations are performed to bitmap primitives differentiably. Let N be the number of (bitmap) primitives. Each i-th primitive has parameters of position x_i, y_i, scale s_i, rotation θ_i, opacity logit ν_i, and RGB color logit $\boldsymbol{c}_i \in \mathbb{R}^3$. Let (x, y) be a coordinate on the canvas. We first transform (x, y) to normalized primitive coordinates:

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} \cos\theta_i & \sin\theta_i \\ -\sin\theta_i & \cos\theta_i \end{bmatrix} \begin{bmatrix} (x-x_i)/s_i \\ (y-y_i)/s_i \end{bmatrix} \in [-1,1]^2 \quad (1)$$

These normalized coordinates are then mapped to discrete bitmap coordinates (U, V):

$$\begin{bmatrix} U \\ V \end{bmatrix} = \begin{bmatrix} (u+1)/2 \cdot (W_i - 1) \\ (v+1)/2 \cdot (H_i - 1) \end{bmatrix} \in [0, W_i - 1] \times [0, H_i - 1], \quad (2)$$

where (H_i, W_i) is the size of the i-th bitmap primitive. Note that U and V are almost surely non-integer. As in STN [14]'s differentiable image sampling, which is also used in [39], we compute the primitive contribution $M_i(x, y)$ via bilinear interpolation, using four nearest integer grid points (*i.e.*, $\{\lfloor U \rfloor, \lfloor U \rfloor + 1\} \times \{\lfloor V \rfloor, \lfloor V \rfloor + 1\}$), to get the gradient on spatial transformation.

Alpha Compositing. The alpha value for primitive i at pixel (x, y) is:

$$\alpha_i(x, y) = \alpha_{\max} \cdot \sigma(\nu_i) \cdot M_i(x, y), \quad (3)$$

where σ is the sigmoid function and $\alpha_{\max} \in (0, 1]$ is a hyperparameter. Using Porter-Duff over compositing [37], the transmittance and final color are:

$$T_k(x, y) = \prod_{j=0}^{k-1} (1 - \alpha_j(x, y)) \tag{4}$$

$$I(x, y) = \sum_{k=0}^{N-1} T_k(x, y)\alpha_k(x, y)\sigma(\boldsymbol{c}_k) \in [0, 1]^3 \tag{5}$$

Parallelization. To compute Eqs. (1-5) practically, we adopt a tile-and-bin CUDA pipeline, following tile-based differentiable splatting practices as in [52], adapted here for 2D bitmap primitives. On the CPU, we first partition the image plane into $T \times T$ tiles (default T=32), and each primitive is assigned to every tile whose bounding box on the image plane overlaps the tile region with a small padding margin. On the GPU, we launch one CUDA thread block per tile with $T \times T$ threads, for achieving complete pixel-level parallelism. Threads in a block cooperatively stage per-primitive parameters in shared memory and composite the tile-local list in front-to-back order for computing Eqs. (4) and (5).

Optional constraint on Θ. To provide finer user control, we optionally apply a constraint on Θ. Especially, we preserve each primitive's original color $\boldsymbol{c}_i^{\text{org}}$ by blending it with a learnable color $\boldsymbol{c}_i^{\text{var}}$, enabling a unique application. To this end, we fix $\boldsymbol{c}_i^{\text{org}}$ by a ratio defined by the hyperparameter $\mu_{\text{blend}} \in [0, 1]$ within the set of $\boldsymbol{c}_i$'s, and only update the remaining $(1 - \mu_{\text{blend}})$ as follows:

$$\boldsymbol{c}_i = \mu_{\text{blend}}\boldsymbol{c}_i^{\text{org}} + (1 - \mu_{\text{blend}})\boldsymbol{c}_i^{\text{var}} \tag{6}$$

For example, $\mu_{\text{blend}} = 1$ was used in Fig. 0b to use the original colors in the output; otherwise, $\mu_{\text{blend}} = 0$ is default.

3.1.2 Backward Pass in Half Precision

Gradient calculation. The gradients with respect to position, scale, and rotation are computed via the chain rule through the coordinate transformation:

$$\frac{\partial I(x,y)}{\partial x_i} = \frac{\partial I(x,y)}{\partial M_i(x,y)} \left(\frac{\partial M_i(x,y)}{\partial u} \frac{\partial u}{\partial x_i} + \frac{\partial M_i(x,y)}{\partial v} \frac{\partial v}{\partial x_i} \right) \tag{7}$$

For simplicity, we only show $\frac{\partial I(x,y)}{\partial u}$ in Eq. (7); gradients with respect to $\{y_i, s_i, \theta_i\}$ are similar with x_i case. We can exactly calculate (*i.e.*, do not need any approximations) $\frac{\partial I(x,y)}{\partial M_i(x,y)}$ using Eqs. (3, 4, 5). We can also easily obtain $\frac{\partial M_i(x,y)}{\partial u}$ using Eq. (2) and regarded bilinear interpolation [14]. At last, we get $\frac{\partial u}{\partial x_i}$, $\frac{\partial u}{\partial y_i}$, $\frac{\partial u}{\partial s_i}$ and $\frac{\partial u}{\partial \theta_i}$ from Eq. (1). Gradients with respect to color and alpha are more straightforward than the geometric parameters, so we provide explanations in the supplementary material.

Half-precision for efficiency. For precision and performance, texture fetches and per-pixel temporaries use FP16 (packed as `__half2` where profitable), while accumulation of the displayed color/transmittance is evaluated in FP16 arithmetic tailored to our packed data path. This choice follows established mixed-precision practice that reduces bandwidth and VRAM usage [32]. In the backward pass, per-parameter gradients are accumulated *per pixel* directly into packed `__half2` buffers via `__half2 atomicAdd` over two-channel gradient groups, followed by a lightweight post pass that unpacks to legacy FP16 arrays. This design omits block-local reductions and aligns with our accuracy–throughput goal.

3.1.3 Dedicated CUDA Kernel for Export

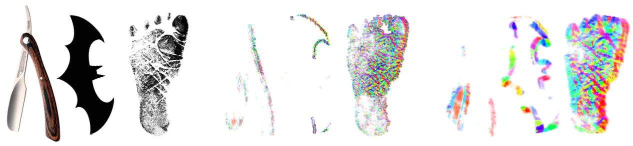
Our optimization kernels (Secs. 3.1.1 and 3.1.2) use atomic operations, which are unsuitable for generating layered PSD files. We therefore implement a dedicated export kernel that uses primitive-level parallelism to efficiently render isolated, editable layers without backward pass overhead. This architecture provides a key advantage: optimization can run at a low resolution while the final, high-quality PSD is exported at a much higher resolution (*e.g.*, $2\times$, $4\times$). Full implementation details are provided in the supplementary material.

3.2. Techniques for Improved Optimization

DiffBMP provides significant flexibility throughout the algorithm, including its hyperparameters, to reflect the diverse intentions of a creator. Nevertheless, just as meaningful artistic expression relies on foundational skill to precisely realize a vision, this creative flexibility is only powerful if it can achieve high fidelity to the target image. We therefore introduce three techniques to achieve this: (Sec. 3.2.1) gradient enriching via soft rasterization, (Sec. 3.2.2) a method for achieving good initialization based on the target image and (Sec. 3.2.3) blending uniform noise canvas to encourage primitives to fill all image regions up.

3.2.1 Blurring Primitives for Enriching Gradients

In Sec. 3.1.2, we established a well-defined gradient path. However, its quantity and extent could be improved for better optimization. In Fig. 3b, the vanilla implementation (*i.e.*, without blurring) shows that $\frac{\partial |I(\cdot,\cdot) - I^{\text{target}}(\cdot,\cdot)|^2}{\partial (x_i, y_i)}$ occurs in relatively fewer pixels. This sparsity arises because the gradient is generated by the non-uniformity of the four operands of bilinear interpolation, which typically have similar values everywhere except at the object boundaries. To enrich (or align) such sparse gradients, soft rasterization (*i.e.*, smoothing the edge of the primitives) is a well-known and prevailing solution, introduced in many prior



(a) Primitives (b) w/o blur (c) w/ blur (§3.2.1)

Figure 3. Blur enriches the spatial gradients of primitives. The figure displays the per-pixel gradients with respect to x_i and y_i for three different primitives (a), extracted during the experiments of Tab. 3, both (b) with and (c) without applying blur (Sec. 3.2.1). Color indicates the direction of the gradient, and opacity represents its magnitude on a log scale (from 10^{-6} to 10^{-3}). Applying blur results in richer and more coherent gradients. This, in turn, leads to the better optimization results shown in Tab. 3.

works [8, 24, 36, 39, 41]. Following these approaches, we simply apply Gaussian blur to each primitives before optimization. As seen in Fig. 3c, this resulted in enriched and aligned gradients with no significant computational cost, consequently leading to better optimization results, which will be shown in Sec. 4.2.

3.2.2 Structure-aware Initialization

Effective initialization is critical for gradient-based optimization. We design a structure-aware initialization strategy that adapts primitive placement and scale to local image complexity. We compute local variance (of I) across RGB channels using a 7×7 sliding window, which identifies regions requiring fine detail (high variance) versus smooth areas (low variance). Initial (x_i, y_i) are sampled among lattice points with probability $\propto 0.1 + 0.9\text{NLV}(x, y)$, where $\text{NLV} : \text{canvas} \to [0, 1]$ is the normalized local variance. s_i was initialized with $s_i = s_{\max} - (s_{\max} - s_{\min}) \cdot \text{NLV}(x_i, y_i)$. This ensures dense and fine coverage in complex regions while maintaining sparse and course coverage in flat areas. We set $c_i \sim \mathcal{N}(I(x_i, y_i), \sigma_c^2)$ for warm start, while we fixed $\nu_i = -2.0$ ($\approx$12% opacity) to ensure gradient flow through all layers. θ_i is sampled uniformly in $[0, 2\pi)$.

3.2.3 Canvas with Uniform Noise to Impose Primitives

A problem arises when a target region shares the same color as the canvas background, as this can prevent primitives from properly splatting onto such regions. We provide an optional mechanism to enforce the placement of primitives in these areas (especially for Sec. 3.3.2): *setting the canvas background* $\boldsymbol{b}(x, y) \sim \mathcal{U}[0, 1]^3$. This optional technique modifies the forward function (Eq. (5)) as follows:

$$I_{\text{FG+BG}} = I_{\text{FG}} + T_N \odot \boldsymbol{b}. \tag{8}$$

This idea is modified from [48]'s work for triangle mesh rendering [21]. Unlike [48], which samples $\boldsymbol{b}$ five times to average the gradients, we sample $\boldsymbol{b}$ only once per iteration.

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

Figure 4. **Heuristics for Dynamic DiffBMP.** Sequential optimization (*i.e.*, initializing Θ^f with $\Theta^{f-1\star}$) can lead to local minima, as in b, where a large primitive is stuck on the face. c shows our remedy: reducing the opacity of such overly dominant primitives, resulting in PSNR gain.

3.3. Heuristics and Losses for Dynamic and Spatially Constrained Rendering

While DiffBMP defaults to ordinary images on a rectangular canvas as the target, it should be capable of handling videos and spatially constrained images for wider applicability. This section details the heuristics and losses designed to achieve this.

3.3.1 Dynamic DiffBMP for Videos

DiffBMP can be extended to model dynamic content. A scalable and straightforward way to achieve both *frame-wise fidelity* and *anti-flicker* is to optimize sequentially, as in [28] for 3DGS. [28] initializes the current frame's parameters Θ^f from the *converged* parameters of the previous frame $\Theta^{f-1\star}$. However, we observed that some highly visible primitives (with large s_i and α_i, and ordered at front) hinder the sequential optimization and *get stuck* at the previous frame, perhaps because they block gradients to other primitives. As shown in Fig. 4, simply removing them improved frame-wise fidelity.

[28] optimizes background primitives only in the first frame and uses the same primitives in the following frames. However, since DiffBMP does not explicitly distinguish the background, we heuristically (using frame-difference or flow-consistency mask) identify *unchanged regions* and reuse their parameters without re-optimization, which improved anti-flicker. See Sec. 4.2 for the results.

3.3.2 Rendering with Spatial Constraint

Our model supports rendering only the foreground, enabling downstream applications such as appearance editing.

Table 2. **Per-iteration performance across implementations and resolutions.** Each cell reports a triple "forward / backward / VRAM" in the order of forward runtime, backward runtime, and total VRAM usage per iteration. Rows list the compared implementations (*e.g.*, PyTorch baseline and our custom CUDA kernels in FP32/FP16), and columns specify the target image resolution and tile size.

Image res / Tile size		512×512 / 16×16	1024×1024 / 16×16	1024×1024 / 32×32	2048×2048 / 32×32
PyTorch	RTX 3090	1360ms / 2337ms / 6.4 GB	5514ms / 9811ms / 12.0 GB	1393ms / 2477ms / 5.0 GB	5405ms / 9483ms / 9.0 GB
	L40S	1342ms / 2413ms / 6.6 GB	4570ms / 8151ms / 12.2 GB	1423ms / 2507ms / 5.2 GB	4914ms / 8707ms / 9.2 GB
CUDA-32bit	RTX 3090	3.9ms / 11.6ms / 1.0 GB	7.2ms / 9.7ms / 2.1 GB	7.6ms / 9.3ms / 2.0 GB	16.1ms / 10.0ms / 6.1 GB
	L40S	4.7ms / 2.9ms / 1.2 GB	10.5ms / 2.8ms / 2.3 GB	8.5ms / 2.9ms / 2.2 GB	23.3ms / 3.8ms / 6.3 GB
CUDA-16bit	RTX 3090	2.3ms / 6.2ms / 1.1 GB	4.2ms / 5.8ms / 1.6 GB	4.3ms / 5.5ms / 1.6 GB	9.0ms / 6.4ms / 3.8 GB
	L40S	2.0ms / 2.1 ms / 1.2 GB	5.4ms / 2.3ms / 1.8 GB	4.5ms / 2.3ms / 1.8 GB	12.8ms / 3.0ms / 4.0 GB

So our goal is: $\forall i, I_\alpha^{\text{target}}(x,y) = 0 \Rightarrow M_i(x,y) = 0$, where I_α^{target} is the target image's alpha. We optimize Θ by applying the following loss to reduce the opacity of background primitives:

$$\mathcal{L} = \|(I_\alpha^{\text{target}} > 0) \odot (I - I^{\text{target}})\|_2^2 + \lambda_\alpha \|I_\alpha - I_\alpha^{\text{target}}\|_2^2, \quad (9)$$

where $I_\alpha = 1 - T_N$ from Eq. (4). Instead of pruning the transparent primitives as in [30], we re-initialize these primitives for further applications. See the results in Fig. 7.

4. Evaluations and Applications

We conduct a series of experiments to validate the effectiveness and versatility of our proposed method.

4.1. Evaluation

We implemented our highly optimized renderer in CUDA and wrapped with Python interface for usability. To quantify computational benefits of our CUDA implementation (our main contribution), we also implemented a naïve PyTorch baseline [35]. We report per-iteration forward/backward runtime and peak VRAM on the same GPUs (RTX 3090, L40S) with identical inputs and hyperparameters. Since all remaining modules (partitioning, loss computation, optimizer, and other Python-side code) share the same PyTorch implementation across all three variants, we restrict the breakdown to the renderer's forward/backward passes. We compare three implementations (PyTorch, CUDA-32bit, and CUDA-16bit) evaluated at 512^2, $1K^2$, and $2K^2$ resolutions with tile sizes 16 or 32. Table 2 summarizes these measurements. This setup isolates implementation effects from workload variance and shows that our custom CUDA parallelization, especially CUDA-16bit, is critical for reducing runtime and memory footprint, enabling scalable DiffBMP at higher resolutions and primitive counts.

DiffVG's struggle with complex SVG limits its extension to bitmaps. DiffBMP addresses key limitations of existing vector-graphic differentiable renderers like DiffVG [23]. While DiffVG remains effective for standard SVG-based vector graphics (as shown in the first row of Fig. 5), its reliance on an analytic calculation leads to low fidelity (poor

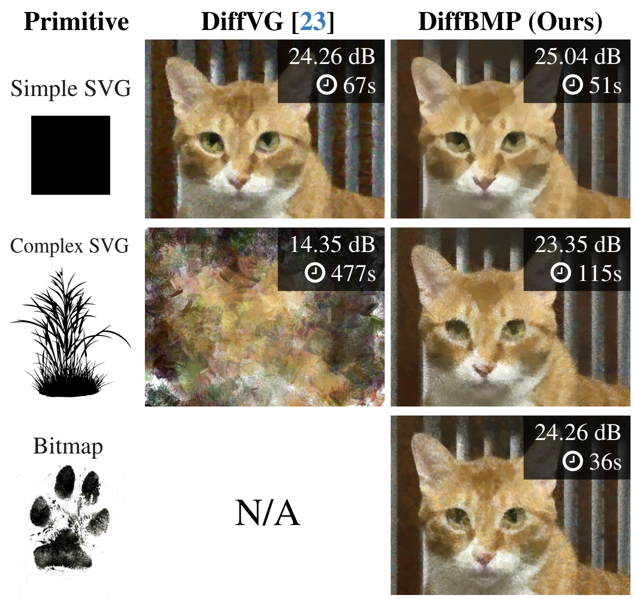


Figure 5. Our DiffBMP works well for complex SVG primitives (second row) and bitmap primitives (third row), whereas existing DiffVG [23] fails or is not available, even though it works well for simple SVG primitives (first row). Note that SVG exportation is included in runtimes for row 1-2, while bitmap exports only PNG, so the third row takes less time (36s).

PSNR) and drastically increased runtime when processing complex SVG curves, as shown in the second row of Fig. 5. This suggests that DiffVG may struggle with many bitmap images even if image vectorization methods are applied. In contrast, DiffBMP does not suffer from it by dealing them as raster, which maintain consistent performance regardless of primitive complexity. Above all, the critical distinction is that, as illustrated in the third row, DiffBMP is the first model capable of performing differentiable rendering using arbitrary raster 2D primitives.

4.2. Ablation study

Soft Rasterization and Initialization. We ablate the two optimization helpers introduced in Sec. 3.2—soft ras-

Table 3. **Ablation of our optimization helpers: soft rasterization (Sec. 3.2.1) and structure-aware initialization (Sec. 3.2.2).** We report PSNR on three targets with indicating whether each component is enabled. Soft rasterization enriches gradients and yields consistent gains, while structure-aware initialization further boosts fidelity; enabling both gives the best PSNR across all scenarios.

SoftRas Sec. 3.2.1	SA-Init. Sec. 3.2.2	Scenario 1 (512×512)	Scenario 2 (512×512)	Scenario 3 (1024×1024)
✗	✗	24.4	20.6	25.9
✓	✗	24.7	21.5	26.5
✗	✓	25.5	21.0	27.1
✓	✓	25.7	21.7	27.4

(a) Target Image (b) w/ white canvas (c) w/ random canvas

Figure 6. Uniform noise (Sec. 3.2.3) makes canvas fully covered by primitives, as shown in (c). (b) Conversely, using the canvas as white removes the need for white primitives to be positioned.

terization (Sec. 3.2.1) and structure-aware initialization (Sec. 3.2.2)—as in the unified protocol of Sec. 4.1. As summarized in Tab. 3, applying Gaussian blur alone yields consistent PSNR gains and stabilizes optimization by providing rich, non-vanishing gradients near primitive boundaries; enabling structure-aware initialization alone accelerates convergence and improves final fidelity, particularly at higher primitive counts. Using both (the last row in Tab. 3) produces the best results across all targets, with sharper edges, fewer spurious fragments, and reduced seed-to-seed variance, whereas disabling both often leads to destabilization of the fit and missing fine details. These helpers operate only before the optimization stage and do not alter the objective; their runtime overhead is almost the same as in Tab. 2.

Canvas with Uniform Noise to Impose Primitives. As explained in Sec. 3.2.3, primitives tend to prioritize covering canvas areas with colors distinct from a designated blending color. This causes some part of the canvas to remain uncovered by primitives, as shown in Fig. 6b. Optimizing with a uniformly noisy canvas resolved this, as in Fig. 6c.

Dynamic DiffBMP. We compare configurations by toggling three lightweight heuristics from Sec. 3.3.1 and report per-frame fidelity (PSNR, SSIM) and anti-flicker (tOF, tLP;

Table 4. **Dynamic DiffBMP ablations.** We observed that improvements in flicker suppression can trade off with per-frame fidelity; using all three of our heuristics in Sec. 3.3.1 targets a balanced solution, having competitive fidelity and the best anti-flicker.

Init. $\Theta^{f-1\star}$	Rem. Stk.	Freez. Unch.	Frame-wise Fidelity		Anti-flicker	
			PSNR↑	SSIM↑	tLP↓	tOF↓
✗	✗	✗	$24.19_{\pm 2.36}$	$0.616_{\pm 0.106}$	$7.41_{\pm 8.89}$	$2.23_{\pm 1.10}$
✓	✗	✗	$24.26_{\pm 1.73}$	$0.629_{\pm 0.073}$	$5.39_{\pm 4.64}$	$1.88_{\pm 1.13}$
✓	✓	✗	$24.66_{\pm 2.03}$	$0.647_{\pm 0.088}$	$4.98_{\pm 4.93}$	$1.89_{\pm 1.11}$
✓	✗	✓	$24.23_{\pm 2.13}$	$0.617_{\pm 0.099}$	$3.50_{\pm 2.52}$	$1.91_{\pm 1.32}$
✓	✓	✓	$24.38_{\pm 2.23}$	$0.630_{\pm 0.069}$	$3.49_{\pm 2.26}$	$1.84_{\pm 1.19}$

(a) Ours (default) (b) (a) + opacity loss (c) (b)+re-init.

Figure 7. Comparison of rendered output I and primitives with spatial constraint under different settings: (a) Ours (default), (b) Ours with opacity loss, and (c) Ours with both opacity loss and re-initialization. Applying both opacity loss and re-initialization produces the best results.

[5]). Metrics are computed per frame, averaged within each video, then reported as mean±std over 17 videos. As summarized in Tab. 4, initializing from $\Theta^{f-1\star}$ already reduces flicker and modestly improves fidelity. On top of this initialization, removing stuck primitives yields the *highest fidelity*, while freezing primitives in unchanged region delivers a *large flicker reduction*. Combining all three achieves the best temporal stability overall while maintaining competitive fidelity.

Spatially Constrained. In Fig. 7, we present a qualitative comparison of the methods discussed in Sec. 3.3.2. The target object is rendered with full opacity by applying opacity loss, which is crucial in post-processing. However, when using this loss, some primitives remain on the boundary of the object with low opacity. To address this issue, we re-initialize such low-opacity primitives by reusing them for further optimization.

4.3. Creative Workflow Examples

A core objective of DiffBMP is to serve not just as an optimization algorithm, but as a practical tool that integrates into existing creative pipelines. The ability to export compositions into native, layered PSD files is central to this goal. While previous sections focus on quantitative metrics, this section demonstrates the qualitative benefits of our artist-friendly exports through several workflow examples, showcasing how DiffBMP can act as a collaborative tool.

Intrinsic-Preserving Graphic Assemblage. Figure 0b, which we term an 'Intrinsic-Preserving Graphic Assemblage', expands upon traditional photo mosaics [7]. Conventional mosaics often alter the color tint of source images to match a target palette and are typically restricted to a rigid, grid-based layout. In contrast, DiffBMP preserves the original color of each primitive (*e.g.*, various brand logos [20]) and arranges them in a grid-free manner, successfully forming a recognizable final shape. This result suggests that DiffBMP can serve as a powerful tool for pioneering new forms of computational art.

Creation from text. DiffBMP is capable of optimizing an arbitrary loss function on the output raster. Since CLIP [38] is a neural network trained to learn the similarity between an image and text, combining it with DiffBMP enables us to optimize the parameters Θ to increase the similarity with a given text prompt. Following the approach of CLIPDraw [9], which combines DiffVG and CLIP, we duplicate the output of DiffBMP and perform data augmentation through perspective transformation, cropping, and resizing. We minimize the negative of the cosine similarity between the augmented DiffBMP output images and the input text, adding the cosine similarity with some negative prompts. See Fig. 8 for the results.

Figure 8. DiffBMP can also be combined with CLIP for creation. Each caption shows the text prompt and the primitive used.

5. Limitations and Future Work

DiffBMP requires a GPU. While DiffVG [23] can be executed on a CPU, DiffBMP fundamentally requires a GPU, and our renderer was specifically implemented in CUDA based on this necessity. DiffBMP operates using bitmap primitives, which are data structures rather than purely mathematical representations. Since bitmap primitives require significant memory usage, GPU operation is essential for DiffBMP.

DiffBMP is sensitive to hyperparameters. The generality of DiffBMP increases its sensitivity to hyperparameter selection and initialization, which can make some tasks susceptible to local minima. Developing methods to automatically determine optimal hyperparameters based on the user's specific task, primitives, and target remains a significant area for future work.

Extending to Autoregressive / RL Drawing. Graphical composition and drawing problems were previously addressed without a differentiable renderer, relying on rule-based [12], reinforcement learning (RL) [11, 13, 16, 31, 43], or neural network-based methods [25, 45, 47, 49]. Since most of these methods do not utilize gradient-based *optimization*, their performance has inherent limitations. We believe DiffBMP provides a critical foundation for applying first-order optimization to these types of autoregressive and RL drawing problems in future work.

6. Conclusion

In this paper, we presented DiffBMP, the first general-purpose differentiable rendering engine designed to utilize arbitrary 2D bitmap images as primitives. DiffBMP demonstrates a powerful ability to scalably and efficiently approximate target designs under various constraints. Crucially, this core capability not only enables the creation of highly artistic compositions but also ensures seamless integration into the professional designer's workflow. Furthermore, by packaging DiffBMP with a user-friendly interface, we anticipate that its public release will significantly broaden the creative horizons of computational art and design.

Acknowledgements

This work was supported in part by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT) [NO.RS-2021-II211343, Artificial Intelligence Graduate School Program (Seoul National University)], the National Research Foundation of Korea(NRF) grants funded by the Korea government(MSIT) (Nos. RS-2025-02263628, RS-2022-NR067592) and the BK21 FOUR program of the Education and Research Program for Future ICT Pioneers, Seoul National University.

References

[1] Stephen Boyd and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004. 2

[2] Wenzheng Chen, Huan Ling, Jun Gao, Edward Smith, Jaakko Lehtinen, Alec Jacobson, and Sanja Fidler. Learning to predict 3d objects with an interpolation-based differentiable renderer. *NeurIPS*, 2019. 2

[3] Ye Chen, Bingbing Ni, Xuanhong Chen, and Zhangli Hu. Editable image geometric abstraction via neural primitive assembly. *ICCV*, 2023. 2, 3

[4] Ye Chen, Bingbing Ni, Jinfan Liu, Xiaoyang Huang, and Xuanhong Chen. Towards high-fidelity artistic image vectorization via texture-encapsulated shape parameterization. *CVPR*, 2024. 2, 3

[5] Mengyu Chu, You Xie, Jonas Mayer, Laura Leal-Taixé, and Nils Thuerey. Learning temporal coherence via self-supervision for GAN-based video generation. *ACM Trans. Graph. (TOG)*, 39(4):75:1–75:15, 2020. arXiv:1811.09393. 7

[6] Martin de La Gorce, David J Fleet, and Nikos Paragios. Model-based 3d hand pose estimation from monocular video. *IEEE transactions on pattern analysis and machine intelligence*, 33(9):1793–1805, 2011. 2

[7] Adam Finkelstein and Marisa Range. Image mosaics. In *International conference on raster imaging and digital typography*, pages 11–22. Springer, 1998. 8

[8] Michael Fischer and Tobias Ritschel. Plateau-reduced differentiable path tracing. *CVPR*, 2023. 5

[9] Kevin Frans, Lisa Soros, and Olaf Witkowski. CLIPDraw: Exploring text-to-drawing synthesis through language-image encoders. *NeurIPS*, 2022. 2, 8

[10] Sara Fridovich-Keil, Alex Yu, Matthew Tancik, Qinhong Chen, Benjamin Recht, and Angjoo Kanazawa. Plenoxels: Radiance fields without neural networks. *CVPR*, 2022. 2

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

[12] 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

[13] Xin Huang and Minglun Gong. Attention-guided deep reinforcement learning for realistic neural painting. *IEEE Access*, 2025. 8

[14] Max Jaderberg, Karen Simonyan, Andrew Zisserman, et al. Spatial transformer networks. *NeurIPS*, 2015. 2, 3, 4

[15] Ajay Jain, Amber Xie, and Pieter Abbeel. VectorFusion: Text-to-svg by abstracting pixel-based diffusion models. *CVPR*, 2023. 2

[16] 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

[17] Hiroharu Kato, Yoshitaka Ushiku, and Tatsuya Harada. Neural 3d mesh renderer. *CVPR*, 2018. 2

[18] Hiroharu Kato, Deniz Beker, Mihai Morariu, Takahiro Ando, Toru Matsuoka, Wadim Kehl, and Adrien Gaidon. Differentiable rendering: A survey. *arXiv preprint arXiv:2006.12057*, 2020. 2

[19] Bernhard Kerbl, Georgios Kopanas, Thomas Leimkühler, and George Drettakis. 3d gaussian splatting for real-time radiance field rendering. *ACM Trans. Graph.*, 42(4):139–1, 2023. 2

[20] Koustubh Khandekar. Popular brand Logos - Image Dataset. Kaggle, 2021. Accessed: 6 November 2025. 8

[21] Samuli Laine, Janne Hellsten, Tero Karras, Yeongho Seol, Jaakko Lehtinen, and Timo Aila. Modular primitives for high-performance differentiable rendering. *ACM Transactions on Graphics (ToG)*, 39(6), 2020. 5

[22] Tzu-Mao Li, Miika Aittala, Frédo Durand, and Jaakko Lehtinen. Differentiable monte carlo ray tracing through edge sampling. *ACM Transactions on Graphics (TOG)*, 37(6), 2018. 2

[23] Tzu-Mao Li, Michal Lukáč, Michaël Gharbi, and Jonathan Ragan-Kelley. Differentiable vector graphics rasterization for editing and learning. *ACM Transactions on Graphics (TOG)*, 39(6), 2020. 2, 3, 6, 8

[24] Shichen Liu, Tianye Li, Weikai Chen, and Hao Li. Soft rasterizer: A differentiable renderer for image-based 3d reasoning. *ICCV*, 2019. 2, 5

[25] 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

[26] Xi Liu, Chaoyi Zhou, Nanxuan Zhao, and Siyu Huang. B\'ezier splatting for fast and differentiable vector graphics. *arXiv preprint arXiv:2503.16424*, 2025. 2

[27] Matthew M Loper and Michael J Black. OpenDR: An approximate differentiable renderer. *ECCV*, 2014. 2

[28] Jonathon Luiten, Georgios Kopanas, Bastian Leibe, and Deva Ramanan. Dynamic 3d gaussians: Tracking by persistent dynamic view synthesis. *3DV*, 2024. 5

[29] Xu Ma, Yuqian Zhou, Xingqian Xu, Bin Sun, Valerii Filev, Nikita Orlov, Yun Fu, and Humphrey Shi. Towards layer-wise image vectorization. *CVPR*, 2022. 2, 3

[30] 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. 6

[31] 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

[32] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, et al. Mixed precision training. *arXiv preprint arXiv:1710.03740*, 2017. 4

[33] Ben Mildenhall, Pratul P Srinivasan, Matthew Tancik, Jonathan T Barron, Ravi Ramamoorthi, and Ren Ng. NeRF: Representing scenes as neural radiance fields for view synthesis. *Communications of the ACM*, 65(1):99–106, 2021. 2

[34] Thomas Müller, Alex Evans, Christoph Schied, and Alexander Keller. Instant neural graphics primitives with a multiresolution hash encoding. *ACM transactions on graphics (TOG)*, 41(4), 2022. 2

[35] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. PyTorch: An imperative style, high-performance deep learning library. *NeurIPS*, 2019. 6

[36] Felix Petersen, Amit H Bermano, Oliver Deussen, and Daniel Cohen-Or. Pix2Vex: Image-to-geometry reconstruction using a smooth differentiable renderer. *arXiv preprint arXiv:1903.11149*, 2019. 5

[37] 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. 4

[38] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. Learning transferable visual models from natural language supervision. *ICML*, 2021. 8

[39] Pradyumna Reddy, Paul Guerrero, Matt Fisher, Wilmot Li, and Niloy J Mitra. Discovering pattern structure using differentiable compositing. *ACM Transactions on Graphics (TOG)*, 39(6), 2020. 2, 3, 5

[40] Pradyumna Reddy, Michael Gharbi, Michal Lukac, and Niloy J Mitra. Im2Vec: Synthesizing vector graphics without vector supervision. *CVPR*, 2021. 2, 3

[41] Helge Rhodin, Nadia Robertini, Christian Richardt, Hans-Peter Seidel, and Christian Theobalt. A versatile scene model with differentiable visibility applied to generative pose estimation. *ICCV*, 2015. 2, 5

[42] Riccardo Roveri, A Cengiz Öztireli, Ioana Pandele, and Markus Gross. PointProNets: Consolidation of point clouds with convolutional neural networks. In *Computer Graphics Forum*, pages 87–99. Wiley Online Library, 2018. 2

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

[44] Yiren Song, Xuning Shao, Kang Chen, Weidong Zhang, Zhongliang Jing, and Minzhe Li. CLIPVG: Text-guided image manipulation using differentiable vector graphics. *AAAI*, 2023. 2

[45] 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. 8

[46] Cheng Sun, Min Sun, and Hwann-Tzong Chen. Direct voxel grid optimization: Super-fast convergence for radiance fields reconstruction. *CVPR*, 2022. 2

[47] 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

[48] Yingtao Tian and David Ha. Modern evolution strategies for creativity: Fitting concrete images and abstract concepts. In *Artificial Intelligence in Music, Sound, Art and Design*, pages 275–291, Cham, 2022. Springer International Publishing. 5

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

[50] Ximing Xing, Haitao Zhou, Chuang Wang, Jing Zhang, Dong Xu, and Qian Yu. SVGDreamer: Text guided svg generation with diffusion model. *CVPR*, 2024. 2

[51] Xinchen Yan, Jimei Yang, Ersin Yumer, Yijie Guo, and Honglak Lee. Perspective transformer nets: Learning single-view 3d object reconstruction without 3d supervision. *NeurIPS*, 2016. 2

[52] 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, 4

[53] Wang Yifan, Felice Serena, Shihao Wu, Cengiz Öztireli, and Olga Sorkine-Hornung. Differentiable surface splatting for point-based geometry processing. *ACM Transactions On Graphics (TOG)*, 38(6), 2019. 2

[54] Kaibo Zhao, Liang Bao, Yufei Li, Xu Su, Ke Zhang, and Xiaotian Qiao. Less is more: Efficient image vectorization with adaptive parameterization. *CVPR*, 2025. 2, 3

[55] Hengyu Zhou, Hui Zhang, and Bin Wang. Segmentation-guided layer-wise image vectorization with gradient fills. *ECCV*, 2024. 2, 3