Fast Sprite Decomposition from Animated Graphics

Tomoyuki Suzuki, Kotaro Kikuchi, and Kota Yamaguchi

CyberAgent
{suzuki_tomoyuki,kikuchi_kotaro_xa,yamaguchi_kota}@cyberagent.co.jp

Abstract. This paper presents an approach to decomposing animated graphics into sprites, a set of basic elements or layers. Our approach builds on the optimization of sprite parameters to fit the raster video. For efficiency, we assume static textures for sprites to reduce the search space while preventing artifacts using a texture prior model. To further speed up the optimization, we introduce the initialization of the sprite parameters utilizing a pre-trained video object segmentation model and user input of single frame annotations. For our study, we construct the Crello Animation dataset from an online design service and define quantitative metrics to measure the quality of the extracted sprites. Experiments show that our method significantly outperforms baselines for similar decomposition tasks in terms of the quality/efficiency tradeoff.

Keywords: sprite decomposition · animated graphics · optimization

1 Introduction

Designers build animated graphics such as social media posts or advertisements using *sprites*, a basic set of animated objects or layers in video editing. Sprite allows intuitive manipulation of objects in videos thanks to its compact and interpretable data representation. However, once a video is composited into a raster video through a rendering engine, it is almost impossible to manipulate objects in a video instantly. Still, it is common to observe a situation where one wishes to edit certain parts of the raster animated graphic video, for example, when a user attempts to create an original from a reference raster animated graphic video. This is where the decomposition of a raster video into the sprites comes into play.

In this work, we tackle the decomposition of raster animated graphics into sprites for video editing applications. While there has been literature on video decomposition [10,33,34], we argue that the decomposition of animated graphics poses unique challenges. Compared to the decomposition of natural scenes, animated graphics include more objects in a single video and involve many types of elements, such as background images, illustrations, typographic elements, or embellishments. Each element has different dynamics, which are typically defined

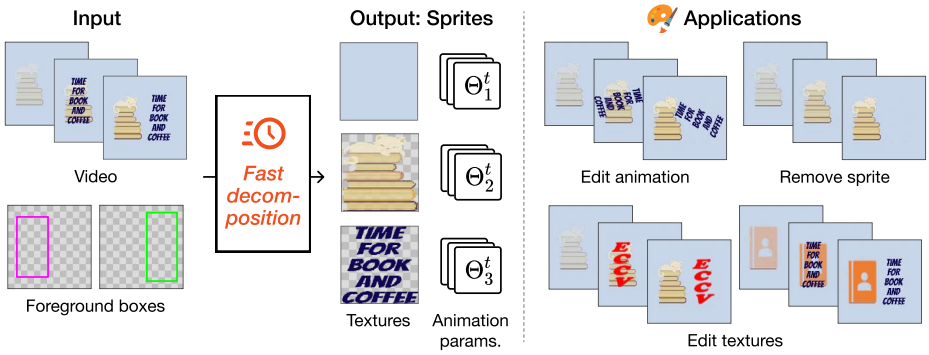


Fig. 1: Sprite decomposition from animated graphics. Given a raster video and auxiliary bounding box annotations, our method decomposes sprites that consist of static textures and animation parameters. The decomposed parameters are easily applicable to various video-editing applications.

as animation effects (*e.g.*, zoom-in or fade-out) in a video authoring tool. While the dynamics or texture of the objects are usually simpler than natural scenes, any artifact resulting from the failure of decomposition is perceptually unacceptable for video editing applications. For ease of manipulation, the animation parameters should be a compact parametric representation (in our case, affine transformation) . Also, considering that the users are designers, it is important to make a decomposition approach fast enough to implement in an interactive editing tool; *e.g.*, it is not acceptable to spend hours processing a raster video in a workflow.

Our sprite decomposition approach is designed to address the aforementioned challenges. Fig. 1 illustrates the overview of our decomposition task. We formulate the sprite decomposition as an optimization problem to fit the sprite parameters to the given raster video. Considering the typical scenario in animated graphics, we introduce a static sprite assumption that all the textures are static and only animation parameters change over time, significantly reducing the parameter space in the optimization process. Under this assumption, we incorporate an image-prior model to prevent undesirable pixel artifacts. For an efficient optimization process, we employ a gradient-based optimizer with an effective initialization procedure that builds on a video object segmentation model from minimal user annotation of object bounding boxes in a single video frame. Combining those simple yet effective approaches, we achieve much faster convergence in the decomposition, which we show in the experiments.

To evaluate the quality and speed of animated graphics decomposition, we build *Crello Animation* dataset that consists of high-quality templates of animated graphic designs, which we collect from an online design service. Also, we define benchmark metrics to evaluate the quality of decomposition tailored for animated graphics. In the experiment, we show our approach considerably outperforms similar decomposition baselines regarding the trade-off between quality

and efficiency. Finally, we present application examples of video editing using decomposed sprites by our approach. We summarize our contributions as follows:

1. We propose a simple and efficient optimization-based method for decomposing sprites from animated graphics.
2. We construct the Crello Animation dataset and benchmark metrics to evaluate the quality of sprite decomposition.
3. We empirically show that our method constitutes a strong baseline in terms of the quality/efficiency trade-off and achieves significantly faster convergence to reach the same decomposition quality.

2 Related work

2.1 Image vectorization and decomposition

Vectorizing or decomposing images is the inverse problem of rasterizing or rendering, *i.e.*, the task of converting an input image into a parametric representation that can be rasterized or rendered as visually identical to the input. Motivations behind this task include editing raster content and obtaining scalable vector representations. Several studies have been made in the computer vision and computer graphics communities for representations such as image layers [2, 22], vector graphics [14, 16], and text attributes [23]. A common approach is to use gradient-based optimization to search for parameters that minimize reconstruction error. While we share the motivation and general approach with the above studies, our work is differentiated by a new data representation, which we refer to as animated graphics, and a method designed specifically for this purpose.

2.2 Video decomposition

There have been many attempts to decompose a raster video into a sequence of layers. Omnimatte [15], Layered Neural Rendering [6], DyStaB [32], Double-DIP [5] and amodal video object segmentation [12, 29] aim to decompose a video into layers (pixel arrays or masks), but they do not aim to parameterize them.

Wang and Adelson [28] proposed a method to represent a layer as a pair of appearance and parameterized animation in addition to decomposing a video into layers, and following attempts [1, 3, 8, 18, 20] have been made. While the basic formulation has not changed since these early works, recent studies adopt machine learning approaches to the decomposition pipeline for better quality. Layered Neural Atlases (LNA) [10] represent primary objects in a video as 2D atlases and their dynamics as moving reference points in the 2D atlases. The coordinate-based multilayer perceptrons (MLPs) [4, 24, 26] represent the 2D atlases and the mapping of reference points. Lee *et al.* [13] extends LNA to edit the appearance of the atlas based on text prompts and compensate for changes in deformation through estimated semantic correspondences. Deformable Sprites

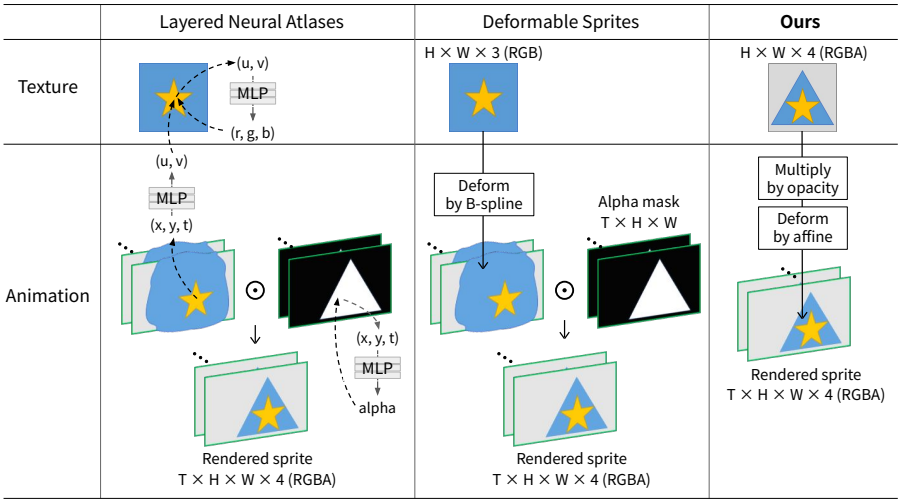


Fig. 2: Comparison of sprite representations of Layered Neural Atlases [10], Deformable Sprites [33], and ours. Our approach limits parameter space to static texture and affine transformation, which enables faster convergence while keeping the necessary representation for animated graphics.

(DS) [33] is similar to LNA but differs in that the 2D atlas (or texture) is simplified to a pixel grid, and deformations are parameterized as B-splines. Sprite-from-Sprite [34] decomposes a cartoon into sprites, where each sprite is represented by a homography warping and spatiotemporal pixel grid including all other information. In this representation, it is difficult to propagate the appearance manipulation temporally, unlike LNA, DS and ours.

We compare our method with LNA [10] and DS [33], which have static textures in their sprite representation like ours. The detailed differences in sprite representation are summarized in Fig. 2. Compared to these methods, our method has a minimal yet sufficient parametrization to cover the typical cases of animated graphics and is particularly beneficial when editing textures. As discussed in [13], LNA requires correction of the mapping between the original and edited textures. The B-spline transform used in DS is flexible for general video decomposition but can result in unwanted deformation for our animated graphics. Both methods use only pixel-level alpha masks, which may not capture the temporal changes in sprite-level opacity often seen in fade-in and fade-out animations. Our simplified representation also leads to faster convergence, is further accelerated by our dedicated initialization and gives a prior for better decomposition quality.

Another line of approach is to decompose and parameterize a video using auto-encoder-based disentangled representation learning [17, 25]. However, these methods assume that a certain amount of training data is available in the target domain, while we only require the target video for optimization.

3 Sprite decomposition

3.1 Data definition

In this work, we define an animated graphic X as a sequence of K sprites, each consisting of a static texture image $\boldsymbol{x}_k$ and its animation parameters Θ_k:

$$X = \big((\boldsymbol{x}_1, \Theta_1), (\boldsymbol{x}_2, \Theta_2), \ldots, (\boldsymbol{x}_K, \Theta_K)\big) = \big((\boldsymbol{x}_k, \Theta_k)\big)_{k=1}^{K}. \tag{1}$$

Here, k is the sprite index, $\boldsymbol{x}_k \in [0,1]^{H \times W \times 4}$ is a RGBA texture with of size $H \times W$. In a more general video decomposition, the texture image is dynamic; *i.e.*, $\boldsymbol{x}_k \in [0,1]^{H \times W \times 4 \times T}$. However, our main applications of animated graphics often do not include a dynamic texture, so we drop the temporal dynamics in the formulation in this work. We define the animation parameters Θ_k by a temporal sequence of tuples of affine warping parameters and a sprite-level opacity parameter: $\Theta_k = (\Theta_k^t)_{t=1}^T = (\boldsymbol{a}_k^t, o_k^t)_{t=1}^T$, where $\boldsymbol{a}_k^t \in \mathbb{R}^6$ is affine matrix parameters and $o_k^t \in [0,1]$ is a sprite-level opacity parameter.

We can render the graphic X into a raster video $Y = (\boldsymbol{y}^t)_{t=1}^T$. Dropping the notation t for simplicity, at a given time, an RGB frame $\boldsymbol{y} \in [0,1]^{H \times W \times 3}$ is obtained by rendering sprites from the back ($k = 1$) to the front ($k = K$):

$$\boldsymbol{b}_1 = \mathcal{D}(\boldsymbol{x}_1; \Theta_1), \tag{2}$$

$$\boldsymbol{b}_k = \mathcal{B}(\mathcal{D}(\boldsymbol{x}_k; \Theta_k), \boldsymbol{b}_{k-1}), \tag{3}$$

$$\boldsymbol{y} = \boldsymbol{b}_K, \tag{4}$$

where $\boldsymbol{b}_k$ is an intermediate backdrop of the rendering and $\mathcal{B}$ is the source-over alpha blending function. The function $\mathcal{D}$ warps the image by affine transform: $\mathcal{D}(\boldsymbol{x}; \Theta) = \mathcal{W}([\boldsymbol{x}_{\mathrm{RGB}}, \boldsymbol{x}_{\mathrm{A}} \odot \boldsymbol{o}]; \boldsymbol{a})$, where $\mathcal{W}$ is an image warping function by a given affine matrix, $[\cdot, \cdot]$ denotes the channel-wise concatenation, $\odot$ denotes the element-wise product, and $\boldsymbol{o} \in [0,1]^{H \times W}$ is a 2D array filled with opacity o.

3.2 Problem formulation

We define the decomposition problem as finding the optimal parameter X^* that can be rendered visually identical to the target raster video Y. Let $\mathcal{L}$ be the function to measure differences between two videos. The decomposition problem can be expressed by:

$$\min_X \mathcal{L}\big(\mathcal{R}(X), Y\big), \tag{5}$$

where $\mathcal{R}(\cdot)$ is the rendering function that applies Eqs. (2) to (4) frame by frame. Our experiments use the mean squared error for $\mathcal{L}$.

4 Approach

While it is possible to apply any optimization approach to the problem in Eq. (5), the problem is practically hard to solve due to the complexity of the search

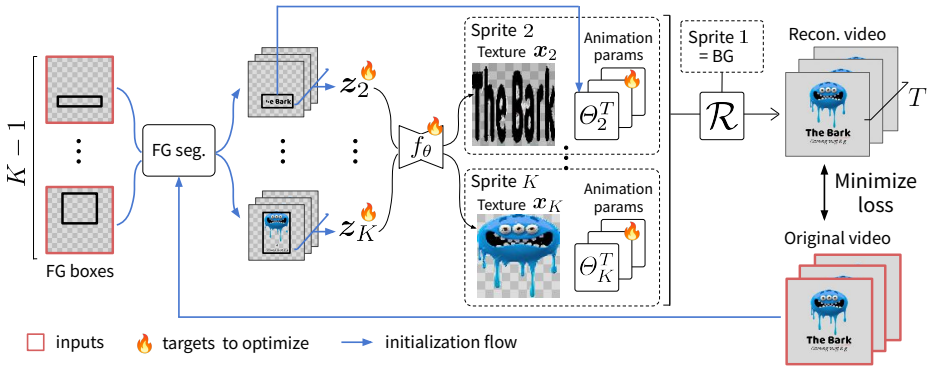


Fig. 3: Our decomposition pipeline. Given a raster video and bounding box annotation for a single frame, we first apply a video object segmentation model to initialize texture and animation parameters. Then, we apply a gradient-based optimizer to find the optimal texture codes, animation parameters, and the texture prior parameters.

space. For example, naively applying a gradient-based optimizer often results in unwanted artifacts with undesired sprite boundaries. We employ the following approaches to achieve the good quality/efficiency tradeoff. 1) We introduce an image prior model [27] and re-formulate texture optimization as a search for model parameters and codes to prevent undesired artifacts. 2) We slightly simplify the problem setup by assuming the user provides additional auxiliary bounding boxes in a single frame, leading to an efficient initialization method by video object segmentation. 3) We use a robust pre-trained video object segmentation model [31] to identify a good initial solution for the optimal sprites.

Given a raster video and auxiliary bounding boxes, we first apply video object segmentation for the foreground sprites and initialize the texture codes and rendering parameters. Then, we apply a standard gradient-based optimization to find the optimal parameters. Our simplified representation itself works as a prior and eliminates the need for regularization losses and optimization scheduling as in existing methods [10, 33].

4.1 Prior-based formulation

We introduce an image prior model [27] to re-formulate our optimization problem (Eq. (5)). An image prior model f_θ represents a mapping of texture code $\boldsymbol{z} \in \mathbb{R}^{H \times W \times 4}$ to a texture image $\boldsymbol{x}$ with model parameters θ: $\boldsymbol{x} = f_\theta(\boldsymbol{z})$. Assuming textures are generated from this prior model, we can transform the texture optimization problem into a search over the code $\boldsymbol{z}$ and the model parameters θ. Let us define $Z = \big((\boldsymbol{z}_k, \Theta_k)\big)_{k=1}^K$. We can rewrite Eq. (5) in the following:

$$\min_{Z,\theta} \mathcal{L}\big(\mathcal{R}'(Z;\theta), Y\big), \tag{6}$$

where $\mathcal{R}'$ is the rendering function from Z and the prior model f_θ.

Following the existing study [33], we adopt U-Net [21] as a prior model architecture and take the output from the code $\boldsymbol{z}$ as the texture. With multiple sprites, we have an input code $\boldsymbol{z}_k$ for each sprite and generate texture $\boldsymbol{x}_k$ for each using the shared model parameter θ. The introduction of the prior model could increase the number of variables in the optimization problem and the time per iteration. However, we empirically find that the convolutional architecture's inductive bias effectively prevents undesirable artifacts, improves the resulting texture quality, and eventually achieves a good quality/efficiency trade-off.

4.2 Auxiliary input

The transformed formulation of Eq. (6) is still a hard problem to find a reasonable solution. In this work, we slightly change the problem setting and assume an additional auxiliary user input that specifies the bounding box of the visible objects in a single video frame. This auxiliary input tells us 1) the number of sprites to decompose and 2) the rough location of the sprite at time step τ, which allows us to initialize variables effectively. Our auxiliary input is practically effortless to obtain in interactive video editing, where users are only asked to annotate bounding boxes in a single video frame.

4.3 Segmentation-based initialization

The goal of the initialization step is to derive good initial values for Z given the auxiliary user input $((\tau_k, \boldsymbol{\beta}_k))_{k=1}^K$, where $\boldsymbol{\beta}_k$ is a bounding box for the sprite k. In this work, we initialize the prior parameters θ by random values [27], and opacity parameters o_k to 1 in all time steps, assuming that all sprites are visible throughout the video frames.

For the initialization of the texture codes $\boldsymbol{z}_k$ and affine parameters $\boldsymbol{a}_k^t$, we employ an off-the-shelf video object segmentation model [31]. Given the auxiliary user input, we apply a tracking model and obtain bounding boxes and segmentation masks for all time steps $t \neq \tau_k$. Using the bounding boxes, we initialize the affine parameters $\boldsymbol{a}_k^t$ to the box locations with no sheer component.

For initializing textures, we first obtain an initial RGB texture image for each foreground sprite by averaging pixels over time within the bounding box regions. Similarly, we obtain the alpha channel by averaging segmentation masks over time. For the background texture ($k = 1$), we average the visible pixel values over time for the RGB at each spatial position and set the alpha to 1. When there are always occluded pixels, we in-paint the region by average pixel values of the visible areas. Once we obtain the initial texture image, we naively treat them as the initial codes $\boldsymbol{z}_k$, which empirically yields good performance. After initialization, we apply a standard gradient-based optimization to solve Eq. (6).

Our initialization is not perfect due to errors in various sources, such as segmentation, spatial misalignment caused by the texture's deformation, or the transparency effect, but it is still sufficiently effective as the initial solution. Also, the inference time for the video object segmentation model is negligible compared to the reduction of optimization time thanks to the good initialization.

4.4 Sprite ordering

Our initialization approach has another limitation: our model does not know the order of sprites. If the rendering order is incorrect, the initial solution may need to swap the order of sprites. Otherwise, the whole process may fall into a local minimum with a wrong order. To address this issue, we search for the rendering order that minimizes the reconstruction error for the first N_{warm} steps of the optimization and then optimize with the rendering order obtained at the N_{warm}-th step.

5 Crello Animation dataset

We construct a new dataset to study animated graphics. Inspired by a dataset of static design templates [30], we scrape animated templates from the online design service[1] that comes with complete sprite information in each animated graphic. Our dataset, named the *Crello Animation*[2] dataset, consists of hundreds of visually appealing animated graphics, mostly designed for social media platforms such as Instagram or TikTok.

In the dataset construction process, we simplify the original templates into the format described in Sec. 3. We first export the static image textures for all sprites at the same size. Then, we compute the animation parameters for each sprite. The original templates come with sprite animations in one of 17 preset types, such as `Zoom in/out`, `Fade in/out`, `Slide in/out`, and `Shake`. If an animation is set on the sprite, we generate per-frame affine matrices and opacities based on their corresponding function. We apply the identity affine matrix and opacity to all frames if no animation is set. After converting all the templates, we excluded templates with animated backgrounds as outliers, duplicated templates, and templates with less than two or more than six layers to avoid too complex sprites. In the end, we obtained 299 samples. We randomly split them into 154 / 145 samples for validation and test splits. More details of the dataset are presented in the supplementary material.

Compared with natural video datasets such as DAVIS [19] used for video layer decomposition, our Crello Animation has unique characteristics: videos contain various types of texture, including natural images, text, or illustrations, and consist of various numbers of sprites, while the natural scene datasets contain at most two or three objects. Our dataset preserves complete composition information without artifacts in the background. This allows us to evaluate the quality of appearance, including occluded areas, while the existing works [10,33] on video layer decomposition have evaluated using Intersection-over-Union of only visible part. Although several studies [7, 9, 12, 29] have proposed datasets with composition information like ours, these are synthetic. Our dataset, sourced from real-world design templates, enables more proper practical evaluations.

[1] `https://create.vista.com`
[2] `https://huggingface.co/datasets/cyberagent/crello-animation`

6 Experiments

We evaluate the performance of our method in decomposing animated graphics and conduct a comparative evaluation with existing video decomposition baselines that output similar layered representations using our Crello Animation.

6.1 Implementation details

We first tuned the hyperparameters using Crello Animation's validation split and then evaluated their performance on the test split with the tuned hyperparameters. We used Adam [11] as the optimizer with a learning rate of 10^{-3}, set N_{warm} to 100 as described in Sec. 4.4 and set the resolution of $\boldsymbol{z}_k$ and textures in our model to 100×100. We conducted all experiments on a workstation with a single NVIDIA Tesla T4 accelerator. We resized the frame size of videos to have a short side of 128 while keeping the aspect ratio.

For the initialization step, we adopted TAM [31] as the video object segmentation model and used the official implementation[3]. TAM segments a target object by first specifying the target with a user prompt at the keyframe. We used box prompts in our experiments. Since in Crello Animation it often happens that most or all of an object's area becomes invisible due to overlapping objects or fading, we calculated the visible area for each sprite, selected the frame with the largest visible area, and generated the rectangle surrounding the visible area in that frame as the prompt. Our approach to extracting bounding boxes follows the idea that a user annotates a frame where the object has a large visible area. We obtained the segmentation of the entire video by applying TAM in the forward and backward time directions from the keyframe. We study the robustness to noise of the user prompt in Sec. 6.5.

6.2 Evaluation metrics

We need to measure the quality of the decomposed $\hat{X}$ compared to ground-truth X. An animated graphic X consists of multiple interdependent variables, so the evaluation metrics need to be carefully designed. We develop multiple evaluation metrics to evaluate the overall quality of the decomposed result.

Frame error. We measure the reconstruction error between the rendered video frames $\hat{Y} = \mathcal{R}(\hat{X})$ and the original video frames $Y = \mathcal{R}(X)$:

$$\mathcal{E}_{\text{frame}}(\hat{X}, X) = e(\mathcal{R}(\hat{X}), \mathcal{R}(X)), \tag{7}$$

where $e(\cdot, \cdot)$ is a function that measures the error between frames, and we use the mean of pixel-level L1 error or LPIPS [35]. Since $\hat{Y}$ is reconstructed using all the information of $\hat{X}$, this metric measures the overall quality of the decomposed $\hat{X}$. However, the frame error alone is insufficient because proximity in the rendered video does not directly reflect the quality of the decomposition result.

[3] `https://github.com/gaomingqi/Track-Anything`

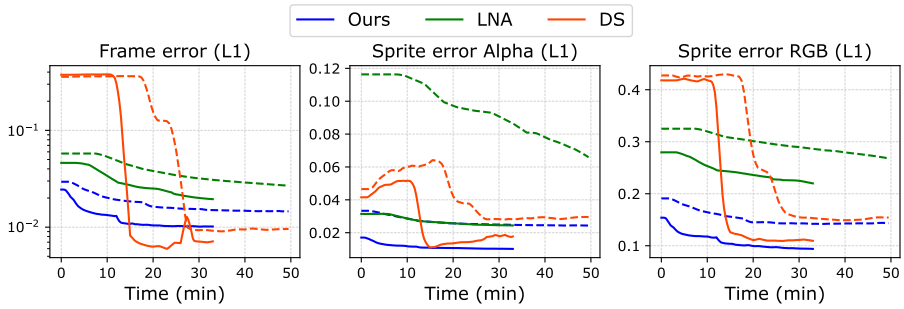


Fig. 4: Comparison of the trade-off between the quality and optimization time on the test split. The solid lines show the average of the samples with four or fewer layers, and the dashed lines show the average of the samples with five and six layers.

Sprite error. The optimal solution for a sprite $(\hat{\boldsymbol{x}}_k, \hat{\Theta}_k)$ is not unique. For example, two sprites can look identical if the texture is shifted by 1 pixel, but the affine transformation adjusts the shift by -1 pixel. Thus, we render each sprite and measure the reconstruction error in pixel space. Also, in this metric, we aim to measure the quality of each sprite independently, we measure the error after searching for the optimal sprite assignment:

$$\mathcal{E}_{\text{sprite}}(\hat{X}, X) = \min_{\sigma \in S_K} \quad \frac{1}{KT} \sum_{k=1}^{K} \sum_{t=1}^{T} e(\mathcal{D}(\hat{\boldsymbol{x}}_{\sigma(k)}; \hat{\Theta}^t_{\sigma(k)}), \mathcal{D}(\boldsymbol{x}_k; \Theta^t_k)), \tag{8}$$

where $\mathcal{S}_K$ represents the set of all possible permutation functions for K elements, with the background always fixed (*i.e.*, $\sigma(1) = 1$). We define the RGB error by the L1 error weighted by the alpha channel:

$$e_{\text{RGB}}(\hat{\boldsymbol{y}}, \boldsymbol{y}) = \phi\big(\|\hat{\boldsymbol{y}}_{\text{RGB}} - \boldsymbol{y}_{\text{RGB}}\|_1 \odot \boldsymbol{y}_{\text{A}}\big), \tag{9}$$

where subscripts represent the channels, respectively, and ϕ is the average operator over the spatial dimensions. For the alpha channel, we use the L1 error.

6.3 Comparison to prior work

We compare our method with LNA [10] and DS [33], which output similar sprite representations to ours as described in Sec. 2. We used the official implementations of both methods, with slight modifications to fit our setting. For LNA, we used the TAM's segmentation masks to calculate the mask bootstrapping loss. For DS, we simplify the transformation as the affine transformation and initialize the parameters in the same way as our method. We conducted hyperparameter tuning on the validation split and evaluated the performance on the test split for all baselines, including ours.

Fig. 4 shows the trade-off curve between optimization time and quality for each method. Our method shows small errors even at the early stage compared to

Table 1: Quantitative comparison on the test split. All values are averages of the samples. The best and the second best result for each metric are highlighted in **bold** and underlined, respectively. * indicates the results after optimization convergence.

Method	# Iter.	Time (min.)	Frame error ↓		Sprite error ↓	
			L1	LPIPS	RGB L1	Alpha L1
LNA	3k	10.4	0.0339	0.2955	0.2654	0.0370
DS	9k	10.6	0.2446	0.3869	0.2965	0.0499
Ours	11k	10.2	0.0163	0.0670	0.1179	0.0193
LNA*	11k	40.7	0.0163	0.1321	0.2332	0.0336
DS*	16k	22.8	**0.0054**	**0.0224**	0.1190	0.0294
Ours*	91k	91.8	0.0101	0.0411	**0.0984**	**0.0179**

other methods. DS receives smaller frame errors than ours as the optimization progresses, but ours is still better in the sprite errors. We suspect this situation was caused by DS's too-high degree of representation, which can reduce the reconstruction error even if it does not decompose a video well. Our static texture assumption and limited animation parameters effectively regularize the optimization process and prevent this local minima.

Tab. 1 shows the quantitative comparison on the test split at approximately the same optimization time (10 minutes) and after convergence. We define the maximum number of iterations for each method as the iteration where the best sprite error is not updated for a quarter of the current iteration on the validation split, and report the errors at the iteration where the loss is minimized as the converged results on the test split (the results on the validation split is in the supplementary material). When the optimization time is 10 minutes, our method shows the best results across all metrics, and moreover, it achieves better sprite error than the converged comparative methods. After convergence, ours achieves the best results in terms of the sprite error. We emphasize that the frame error is an auxiliary metric that can be low even if the decomposition fails.

Also, we show the qualitative comparison in Fig. 5. In LNA, the reconstruction results are generally blurred, and the sprite boundaries are rough. We suspect LNA has a bias to generate smooth masks since it represents masks with an MLP that tends to output smooth value for the input, *i.e.* coordinate. DS achieves more precise boundaries than LNA, but DS sometimes fails to group objects. For example, in the first sample in Fig. 5, the second sprite is partially included in the first sprite. Our method does not allow a single sprite to have a complex animation and successfully decomposes this case. We observe a subtle artifact where the foreground remains in the background. We suspect this is because the foreground alpha of 1 in the ground truth is not exactly optimized to 1. This causes the occluded area to slightly impact reconstruction, leading to minor RGB inaccuracies that reduce reconstruction loss. We might be able to rely on post-processing or manual editing since those artifacts often stand out in easily fixable homogeneous regions. We provide more qualitative results in the supplementary material.

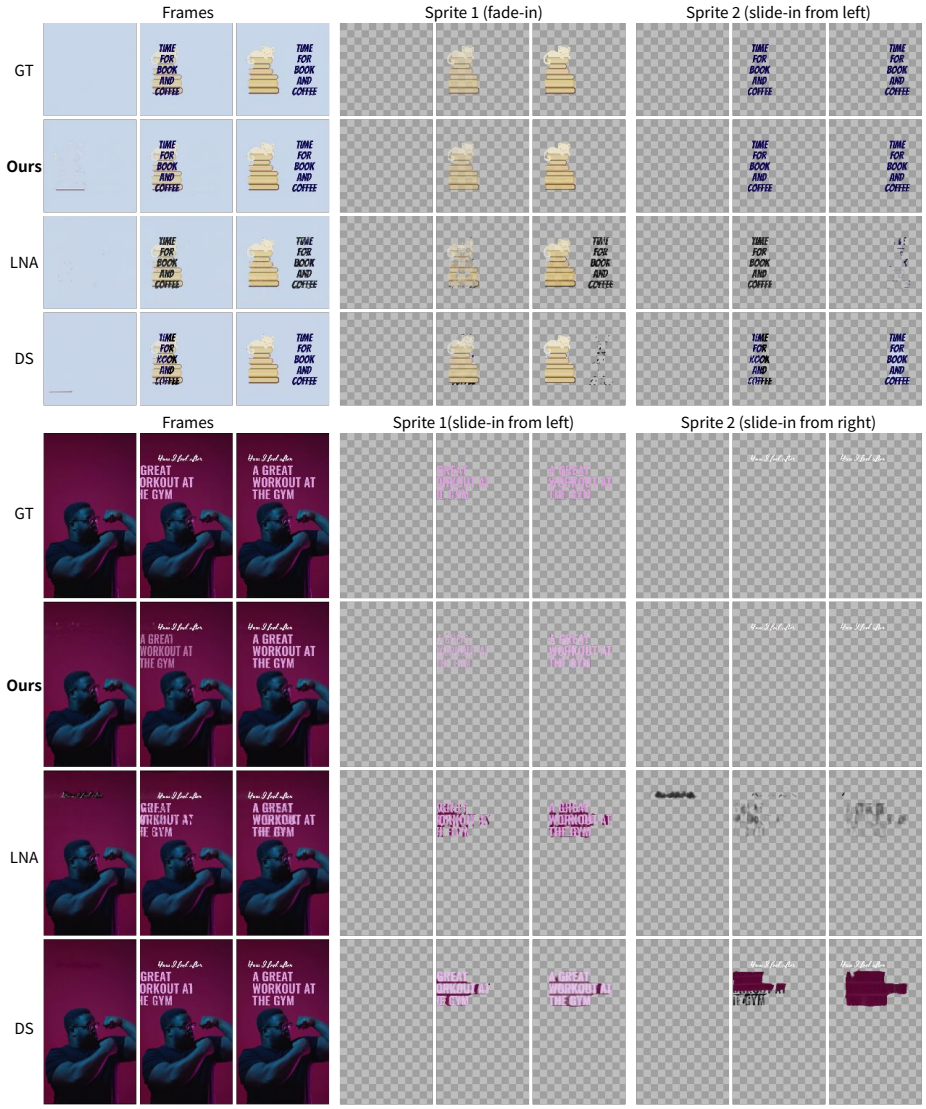


Fig. 5: Qualitative comparison between LNA [10], DS [33], and our method. We put the description of the animation above each sprite.

6.4 Ablation study

We ablate the effect of each component of our method using the validation split. We summarize the results in Tab. 2. We can see that using the texture prior improves the decomposition quality in all metrics from the comparison between

Table 2: Ablation study of our decomposition pipeline. In all settings, the number of total iterations is 20,000. The best and the second best result for each metric are highlighted in **bold** and underlined, respectively.

Texture prior model	Texture init.	Matrix init.	Frame error ↓ L1	Frame error ↓ LPIPS	Sprite error ↓ RGB L1	Sprite error ↓ Alpha L1
✓	✓	✓	**0.0207**	**0.0793**	**0.1344**	**0.0237**
	✓	✓	0.0324	0.1473	0.1995	0.0283
✓	✓		0.0336	0.2090	0.2651	0.0324
✓			0.0318	0.2258	0.3064	0.3384

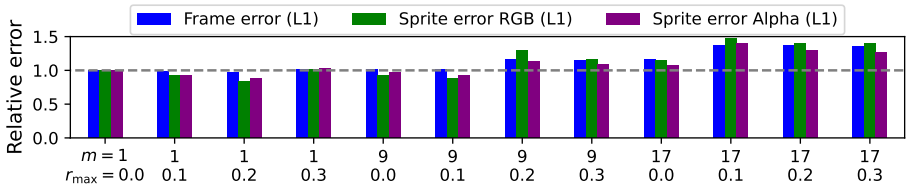


Fig. 6: Results of evaluating the robustness to prompt's noise. The vertical axis shows the relative error normalized by the one when no noise is added (the leftmost bar). The gray dashed line indicates where the relative error becomes 1.

the first and second rows. This conforms to the previous report in generative tasks [27]. Comparing rows 3 and 4, texture initialization significantly improves the sprite alpha L1. This suggests that with no appropriate texture initialization, optimization tends to fall into local minima where the reconstruction error is small, but the sprite is inappropriately decomposed. Initializing both affine matrices and textures achieves the best results in all metrics.

6.5 Robustness to prompt noise

Although we simulate box prompts by users in our experiments, bounding box annotation usually contains noise in a real-world application. We verify the robustness of our method to the annotation noise. We consider two types of noise: noise in the keyframe selection and noise in the box's position and size. For the former, we select the frame with the m-th largest visible area (described in Sec. 6.1) for each foreground sprite, varying the m. For the latter, we add noise to a box directly: $p' = p + s \times r$, where p is top, bottom, left, or right coordinate of the box, s is the height if p is the top or bottom and the width otherwise, and r is a random variable sampled from $[-r_{\max}, r_{\max}]$. We vary the $r_{\max}$ and evaluate the decomposition quality. We show the results in Fig. 6. The results confirm that our method does not suffer from critical performance degradation even in the presence of substantial noise ($m = 9$ and $r_{\max} = 0.3$), indicating that users do not need to be nervous about the accuracy of bounding box annotation.

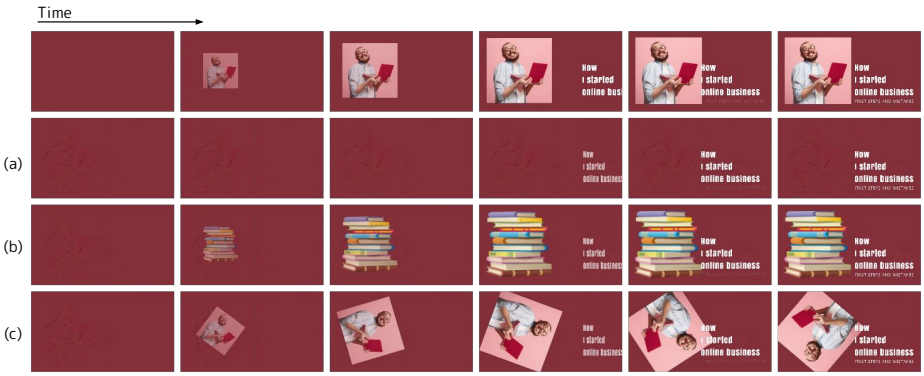


Fig. 7: Application examples of our decomposition results. We decomposed a video in the test split (the first row) using our method and applied three types of editing: (a) sprite removal, (b) texture replacement, and (c) animation (rotation) insertion.

7 Application

We demonstrate a video editing application using our decomposition approach, shown in Fig. 7. Here, we first decomposed videos and applied three types of editing: (a) sprite removal, (b) texture replacement, and (c) animation insertion. In the texture replacement, we replaced the texture of a sprite with a different one while keeping the animation. In the animation insertion, we added rotation to the original animation while keeping the texture. We can observe that the new occlusion and re-appearance caused by the editing are appropriately reflected, and the original animations are correctly transferred to the new textures. Thanks to our accurate sprite decomposition, these examples do not have major artifacts, which is preferable for video editing.

8 Conclusion

We addressed sprite decomposition from animated graphics. Our optimization-based approach introduces several strategies to make efficient decomposition for the animated graphics, and the evaluation in our newly created Crello Animation shows that our method successfully outperforms existing methods in the trade-off between the quality of the decomposition and the convergence time.

In the future, it would be interesting to relax our static sprite assumption and represent deformation and opacity as a function of time rather than per-frame (*e.g.*, animation representation via keyframes and their interpolation, which is common in video editing software). This would allow for applications such as increasing the temporal resolution and may improve the performance of the decomposition, functioning as an additional prior. Also, we are interested in parameterizing videos with more types of animation than motion and opacity, such as blur change and lighting effects, to support creative video workflow.

References

1. Agarwal, S., Wills, J., Belongie, S.: What went where. In: CVPR (2003)
2. Aksoy, Y., Aydın, T.O., Smolić, A., Pollefeys, M.: Unmixing-based soft color segmentation for image manipulation. TOG (2017)
3. Brostow, G.J., Essa, I.A.: Motion based decompositing of video. In: ICCV (1999)
4. Chen, Y., Liu, S., Wang, X.: Learning continuous image representation with local implicit image function. In: CVPR (2021)
5. Gandelsman, Y., Shocher, A., Irani, M.: "double-dip": unsupervised image decomposition via coupled deep-image-priors. In: CVPR (2019)
6. Geng, C., Peng, S., Xu, Z., Bao, H., Zhou, X.: Learning neural volumetric representations of dynamic humans in minutes. In: CVPR (2023)
7. Johnson, J., Hariharan, B., Van Der Maaten, L., Fei-Fei, L., Lawrence Zitnick, C., Girshick, R.: Clevr: A diagnostic dataset for compositional language and elementary visual reasoning. In: CVPR (2017)
8. Jojic, N., Frey, B.J.: Learning flexible sprites in video layers. In: CVPR (2001)
9. Kabra, R., Burgess, C., Matthey, L., Kaufman, R.L., Greff, K., Reynolds, M., Lerchner, A.: Multi-object datasets. `https://github.com/google-deepmind/multi_object_datasets` (2019)
10. Kasten, Y., Ofri, D., Wang, O., Dekel, T.: Layered neural atlases for consistent video editing. TOG (2021)
11. Kingma, D., Ba, J.: Adam: A method for stochastic optimization. In: ICLR (2015)
12. Lamdouar, H., Xie, W., Zisserman, A.: Segmenting invisible moving objects. In: BMVC (2021)
13. Lee, Y.C., Jang, J.Z.G., Chen, Y.T., Qiu, E., Huang, J.B.: Shape-aware text-driven layered video editing. In: CVPR (2023)
14. Li, T.M., Lukáč, M., Michaël, G., Ragan-Kelley, J.: Differentiable vector graphics rasterization for editing and learning. TOG (2020)
15. Lu, E., Cole, F., Dekel, T., Zisserman, A., Freeman, W.T., Rubinstein, M.: Omnimatte: Associating objects and their effects in video. In: CVPR (2021)
16. Ma, X., Zhou, Y., Xu, X., Sun, B., Filev, V., Orlov, N., Fu, Y., Shi, H.: Towards layer-wise image vectorization. In: CVPR (2022)
17. Monnier, T., Vincent, E., Ponce, J., Aubry, M.: Unsupervised layered image decomposition into object prototypes. In: ICCV (2021)
18. Pawan Kumar, M., Torr, P.H., Zisserman, A.: Learning layered motion segmentations of video. IJCV (2008)
19. Perazzi, F., Pont-Tuset, J., McWilliams, B., Van Gool, L., Gross, M., Sorkine-Hornung, A.: A benchmark dataset and evaluation methodology for video object segmentation. In: CVPR (2016)
20. Rav-Acha, A., Kohli, P., Rother, C., Fitzgibbon, A.: Unwrap mosaics: A new representation for video editing. TOG (2008)
21. Ronneberger, O., Fischer, P., Brox, T.: U-net: Convolutional networks for biomedical image segmentation. In: MICCAI (2015)
22. Sbai, O., Couprie, C., Aubry, M.: Unsupervised image decomposition in vector layers. In: ICIP (2020)
23. Shimoda, W., Haraguchi, D., Uchida, S., Yamaguchi, K.: De-rendering stylized texts. In: ICCV (2021)
24. Sitzmann, V., Martel, J., Bergman, A., Lindell, D., Wetzstein, G.: Implicit neural representations with periodic activation functions. In: NeurIPS (2020)

25. Smirnov, D., Gharbi, M., Fisher, M., Guizilini, V., Efros, A., Solomon, J.M.: Marionette: Self-supervised sprite learning. In: NeurIPS (2021)
26. Tancik, M., Srinivasan, P., Mildenhall, B., Fridovich-Keil, S., Raghavan, N., Singhal, U., Ramamoorthi, R., Barron, J., Ng, R.: Fourier features let networks learn high frequency functions in low dimensional domains. In: NeurIPS (2020)
27. Ulyanov, D., Vedaldi, A., Lempitsky, V.: Deep image prior. In: CVPR (2018)
28. Wang, J.Y., Adelson, E.H.: Representing moving images with layers. TIP (1994)
29. Xie, J., Xie, W., Zisserman, A.: Segmenting moving objects via an object-centric layered representation. In: NeurIPS (2022)
30. Yamaguchi, K.: Canvasvae: Learning to generate vector graphic documents. In: CVPR (2021)
31. Yang, J., Gao, M., Li, Z., Gao, S., Wang, F., Zheng, F.: Track anything: Segment anything meets videos. arXiv preprint arXiv:2304.11968 (2023)
32. Yang, Y., Lai, B., Soatto, S.: Dystab: Unsupervised object segmentation via dynamic-static bootstrapping. In: CVPR (2021)
33. Ye, V., Li, Z., Tucker, R., Kanazawa, A., Snavely, N.: Deformable sprites for unsupervised video decomposition. In: CVPR (2022)
34. Zhang, L., Wong, T.T., Liu, Y.: Sprite-from-sprite: Cartoon animation decomposition with self-supervised sprite estimation. TOG (2022)
35. Zhang, R., Isola, P., Efros, A.A., Shechtman, E., Wang, O.: The unreasonable effectiveness of deep features as a perceptual metric. In: CVPR (2018)