Editing Motion Graphics Video via Motion Vectorization and Transformation

SHARON ZHANG, Stanford University, USA
JIAJU MA, Stanford University, USA
JIAJUN WU, Stanford University, USA
MANEESH AGRAWALA, Stanford University and Roblox, USA

Motion graphics videos are widely used in Web design, digital advertising, animated logos and film title sequences, to capture a viewer’s attention. But editing such video is challenging because the video provides a low-level sequence of pixels and frames rather than higher-level structure such as the objects in the video with their corresponding motions and occlusions. We present a motion vectorization pipeline for converting motion graphics video into an SVG motion program that provides such structure. The resulting SVG program can be rendered using any SVG renderer (e.g., most Web browsers) and edited using any SVG editor. We also introduce a program transformation API that facilitates editing of a SVG motion program to create variations that adjust the timing, motions and/or appearances of objects. We show how the API can be used to create a variety of effects including retiming object motion to match a music beat, adding motion textures to objects, and collision preserving appearance changes.


Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM. 0730-0301/2023/12-ART229 $15.00 https://doi.org/10.1145/3618316
Additional Key Words and Phrases: vector graphics, motion vectorization, scalable vector graphics, SVG, visual programs

ACM Reference Format:

1 INTRODUCTION

Programs have proven to be useful in many areas of computer graphics. The structure and repetition found naturally in our surroundings, combined with the symbolic reasoning that humans use to describe objects, can make programs particularly effective in representing visual content. For instance, biologists use L-systems to model plant structures [Prusinkiewicz and Lindenmayer 1996]; digital artists use shader graphs to generate materials and textures [Cook 1984]; data analysts use grammar-based APIs to create visualizations [Satyanarayan et al. 2016; Wickham 2016]; and SVG is a widely adopted declarative program format for vector graphics [W3C 2018].

There are several benefits to representing visual content with a program rather than working directly in the output space of pixels and frames. For one, programming languages often provide meaningful abstractions and concepts (i.e., language primitives) that operate at a higher level than pixels and align better with the ways that humans think about the underlying content. With SVG, for example, we can describe an animation as a collection of object primitives moving in time, instead of specifying individual pixel colors over time (Figure 1). Another benefit is that programs provide meaningful control parameters. SVG programs can describe the motions of objects using a sequence of affine transforms and editing the small set of transform parameters can generate a wide range of motions.

In this work we focus on a particular domain of visual content—namely, motion graphics—which are essentially animated graphic designs usually consisting of shapes and typography in choreographed motions. Such motion graphics are ubiquitous in Web design, digital advertising, animated logos, and film title sequences. Yet, creating effective motion graphics requires expertise in crafting eye-catching motions and skill with animation software. Moreover, once they have been rendered as video—the most common format for motion graphics on the Web—they become very difficult to edit. Creating variations of a motion graphics video (e.g., swapping out objects, changing the text, or retiming motions of individual objects to music) is impractical without access to a higher level representation.

We present tools for editing a motion graphics video by first converting it into an SVG motion program. Our motion vectorization pipeline identifies objects, tracks their motions and occlusion relationships across the video, and generates an SVG motion program (Figure 1 top row). Our approach adapts the differentiable image compositing optimization method of Reddy et al. [2020] to our tracking problem. The resulting motion program can be rendered using an SVG renderer (e.g., most Web browsers) and edited using an SVG animation editor. To take further advantage of our representation, we introduce a program transformation API that allows users to programmatically create variations of the SVG motion program. Our approach is to treat the SVG motion program as a scene graph composed of objects and their motions. We demonstrate how our API can be used to create a variety of effects, including retiming object motion to match music beats, adding motion textures (e.g., pulsing, wobbling) to objects and programmatically changing the appearance of objects (Figure 1 middle, bottom rows).

In summary, we make two main contributions:

(1) A motion vectorization pipeline that converts a motion graphics video into an SVG motion program.

(2) A program transformation API for programmatically editing SVG motion programs to create variations.

2 RELATED WORK

Recovering programs from visuals. Because programs are such a useful representation for visual data, graphics and vision researchers have investigated how to automatically infer such programs from raw visual data. This problem has been explored in multiple visual domains, including 3D shape modeling [Deng et al. 2022; Du et al. 2018; Jones et al. 2020, 2021, 2022; Kania et al. 2020; Li et al. 2020, 2022; Ren et al. 2021; Tian et al. 2019; Willis et al. 2021; Wu et al. 2021; Xu et al. 2021; Yu et al. 2022], 2D shape and layout modeling [Ellis et al. 2018; Ganin et al. 2021, 2018; Reddy et al. 2021; Seff et al. 2022; Sharma et al. 2018; Xu et al. 2022], material and texture modeling [Guerrero et al. 2022; Hu et al. 2019, 2022; Tchapmi et al. 2022], extracting human motion primitives from video [Kulal et al. 2021, 2022] and deconstructing visualizations [Harper and Agrawala 2014, 2018; Poco and Heer 2017; Savva et al. 2011]. Deep learning is a popular technique, either to detect primitives which are then combined into programs using an optimization process [Ellis et al. 2018; Guo et al. 2020], to guide a search algorithm [Ellis et al. 2021; Wang et al. 2019] or to predict higher-level functions that make programs more compact and easier to edit [Ellis et al. 2021; Jones et al. 2021]. In our work, we leverage the visual regularity of motion graphics videos to perform per-frame primitive detection without heavyweight neural network machinery; we then turn these per-frame primitives into a temporally-consistent SVG motion program via optimization.

Motion tracking. Multi-object motion tracking for natural video is a well-studied problem [Ciaparrone et al. 2020; Luo et al. 2021]. Many of these systems output coarse-level motion information such as per-frame object bounding boxes; they cannot reconstruct an input video. Moreover, motion graphics videos tend to be relatively textureless and may contain objects that undergo large motions between frames. As a result, feature-based tracking methods such as SIFT [Lowe 2004] and KLT [Lucas and Kanade 1981; Tomasi and Kanade 1991] are less reliable. Recent neural network models for optical flow [Dosovitskiy et al. 2015; Teed and Deng 2020] also take advantage of the high-frequency textured nature of realistic video and are less effective on motion graphics. In our work, we instead use neural optical flow as initialization for additional optimization or motion parameters.

Researchers have also developed motion tracking techniques for cartoon style video [Liu et al. 2013; Škora et al. 2009; Zhang et al. 2012; Zhu et al. 2016]. While these methods are built for flat-colored cartoon sequences, they often produce undesirable correspondences in motion graphics videos containing many repeated objects (e.g.,
letters). Our work is inspired by Bregler et al.’s [2002], who motion capture and retarget the exaggerated deformations of cartoon characters. However, they require manually annotated object contours as input, whereas our goal is to further automate the object detection and motion tracking process and to recover an SVG motion program that we can retarget via a program transformation API.

Layered video decomposition. Our work enables object-level manipulation of motion graphics video, which calls for an object-centric layered decomposition. Prior work in decomposing natural videos uses motion cues to generate layers based on relative depth from the camera [Brostow and Essa 1999; Wang and Adelson 1994] or on coherent camera motion [Fradet et al. 2008]. More recent neural methods decompose video into layers represented as frame sequences [Lu et al. 2020, 2021] or as neural atlases [Kasten et al. 2021; Ye et al. 2022]. Such outputs can support appearance editing but do not enable motion editing. Zhang et al. [2022] generate sprite decompositions of cartoon videos, where each sprite is a sequence of frames and a corresponding sequence of homographies that map between sprite and frame coordinates. Since the appearance of each sprite can change from frame to frame, the corresponding homographies do not fully characterize the sprite motion. They also assume a fixed depth ordering of the layers which results in artifacts when objects change in relative depths. Our pipeline adapts Reddy et al.’s [2020] differentiable compositing method to compute relative depth (and motion parameters) as a function of time, allowing for dynamic object occlusion relationships.

3 BACKGROUND

Characteristics of motion graphics video. Motion graphics videos are commonly composed of a set of foreground objects, including basic shapes (e.g., rectangles, discs, etc.) and typography moving over a static background. The objects may occlude one another as well as split into separate objects, or merge together into a single object. In general, motion graphics videos may use textures and gradients to color both the foreground objects and the background, and foreground objects may move and deform non-rigidly. But we have found that in many contexts where motion graphics are prevalent—e.g., Web design, animated logos, digital advertising, film title sequences—a common stylistic choice is to use mostly solid-colored foreground objects undergoing affine motions over a static background. Sparing use of texture and photographic elements in combination with simpler motions can improve legibility and make it easier to guide the viewer’s gaze through the video. which is crucial in contexts such as advertising. We focus on converting this important class of motion graphics video into SVG programs.

Structure of SVG motion programs. Scalable vector graphics (SVG) is a declarative programming format for vector graphics that is widely implemented in Web browsers across a variety of devices [W3C 2018]. To convert a motion graphics video into an SVG motion program we can represent each foreground object, as an SVG group <g> containing its appearance <image> and a sequence of per-frame motion transforms. SVG natively supports affine transforms for warping elements with separate parameters.

4 MOTION VECTORIZATION

The goal of our motion vectorization pipeline is to recover an SVG motion program from an input motion graphics video. The primary challenge is to identify and track each of the objects in the input video as they appear, move, occlude one another and disappear. We use a four stage pipeline: (1) we segment frames into regions (e.g. potential objects), (2) we generate candidate mappings explaining how objects might move from frame-to-frame, (3) we select the best collection of mappings explaining the frame-to-frame movements of the objects and finally (4) we write an SVG motion program. Our motion vectorization pipeline builds on Reddy et al.’s [2020] differentiable compositing optimization technique. We first describe how we adapt differentiable compositing to our problem setting in Section 4.1; we then present each stage of our pipeline in Sections 4.2 to 4.5.

4.1 Differentiable image compositing

Differentiable image compositing [Reddy et al. 2020] is an optimization technique originally designed to decompose a graphic pattern comprised of discrete elements (which may partially occlude one another) into a layered representation (Figure 2). It takes in a target pattern image T and a set of source element images \( S = \{S_1, ..., S_N\} \) that appear in the pattern and optimizes a similarity transform (translation, rotation, and uniform scale) for each source element. It also computes a depth ordering so that when the transformed elements are rendered in back-to-front order they reproduce the target pattern. That is,

\[
DC(S, T) = \{(S_i, \Theta_{S_i}, \Lambda_{S_i}) | S_i \in S\},
\]

The scale and translate parameters allow separate control over x and y.
where $\Theta_S$ is the transform that places $S_i$ in $T$, and $\Lambda_{S_i}$ is the layer $z$-ordering for $S_i$ in $T$ with respect to the other elements in $\mathcal{S}$ after transforming by their $\Theta_i$’s. We refer to the resulting set of layering placement tuples as $\mathcal{S}^* = \{(S_i, \Theta_{S_i}, \Lambda_{S_i})\}_{i=1}^N$.

With this information, we can define two additional image operators: (1) a compositing operator $C(S^*)$ composites all of the transformed source elements and $\Lambda_{S_i}$ in back-to-front order according to their $\Lambda$’s; (2) a visibility mask operator $M_{\text{vis}}(S_i, I)$ produces a binary mask of the pixels of image $I$ where $S_i$ is visible. Importantly, $M_{\text{vis}}$ always operates in the frame space represented by $I$. For example, $M_{\text{vis}}(S_i, C(S^*))$ is the set of pixels of the transformed $\Theta_S(S_i)$ that are visible in $C(S^*)$. See Figure 2 for examples of both of these operators.

To apply differentiable compositing to the context of tracking objects in motion graphics video, we have extended the optimization to compute an affine transformation $\Theta_S(S_i)$ (translation, rotation, non-uniform scale and skew) rather than a similarity transform. Specifically, we add scaleX, scaleY, skewX, and skewY as independent parameters in the optimization.

4.2 Stage 1: Region extraction

The first stage of our vectorization pipeline is to segment each input frame $F_t$ into regions. Since we focus on motion graphics with mostly solid colored objects, as a default we use color clustering in LAB colorspace and mark the pixels in the cluster of the mode color as background. Alternatively users can specify a background image if the video has a photograph, texture or colored gradient as background. To separate the remaining foreground pixels into regions, as a default we construct an edge map for the frame [Canny 1986] and then apply Zhang et al.’s [2009] trapped-ball segmentation. This gives us a set of regions $\mathcal{R}_t = \{R_1, \ldots, R_N\}$ for each frame $F_t$. If the foreground is textured, users can choose to skip edge detection and apply connected-components segmentation on the foreground pixels to form regions. Finally, we let users manually specify pixel-level region boundaries if necessary, as noted in Section 5.

4.3 Stage 2: Generate candidate mapping types

Given a set of regions for every input frame, our goal is to identify unique foreground objects and track them between frames. We initialize this process at the first frame $F_1$ by treating each region $R_t \in \mathcal{R}_t$ as an object $O_t$ so that $O_t = \mathcal{R}_t$. For each subsequent frame $F_t$, our task is to determine how objects in the previous frame map to regions in the current frame $\mathcal{R}_t$ under affine transformations. Figure 3 shows the eight types of mappings that can occur between objects and regions.

To determine which of these mapping types best matches objects in $F_{t-1}$ with regions in $F_t$, we construct an initial set of the likeliest mapping types in the form of two bipartite graphs: (1) the forward candidate mapping graph $B_{\text{fwd}}$ holds likely mappings taking objects to regions; (2) the backward candidate mapping graph $B_{\text{bwd}}$ holds likely mappings taking regions to objects. We first describe how we build the graphs and then explain how they encode likely mappings.

**Build candidate mapping graphs.** Figures 4 and 5 show how we build $B_{\text{fwd}}$ and $B_{\text{bwd}}$. For $B_{\text{fwd}}$, we first apply differentiable compositing as $DC(O_{t-1}, F_t) = O^*$, treating $O_{t-1}$ as the set of source elements and the current frame $F_t$ including all of its regions $\mathcal{R}_t$ as the target image. Then, for each object $O_t \in \mathcal{O}_{t-1}$, we consider each region $R_t \in \mathcal{R}_t$ and compute a source coverage weight as

$$W^{\text{cov}}_{\text{fwd}}(O_t, R_t) = \frac{|M_{\text{vis}}(O_t, C(O^*)) \cap M_{\text{vis}}(R_t, F_t)|}{|M_{\text{vis}}(O_t, C(O^*))|}.$$ (2)
In practice, we have found that DC is sensitive to the initial placement of source elements. Therefore, we initialize the source placement using shape context [Belongie et al. 2006], optical flow [Teed and Deng 2020] and RANSAC to estimate how each object (or region) moves to $F_t$ (or $F_{t-1}$). Note also that when we use DC, we save the resulting sets of layering placement tuples $O^*$ and $R^*$ for use in later stages of our pipeline.

**Extract candidate mappings.** The forward and backward candidate mapping graphs encode multiple candidate mappings. To extract the individual candidate mappings from either of these graphs, we first consider each connected component of the graph. We treat any such component that is one-to-one, one-to-many, or many-to-one (i.e., the component contains exactly one object or exactly one region) as a candidate mapping. If the component forms a many-to-many graph, we further break it into pieces (see inset) as follows. For each node (object or region) in the component, we form a subgraph that includes all edges the node is part of. Each resulting subgraph is then either a one-to-one, one-to-many, or many-to-one mapping candidate.

As shown in Figure 3, many-to-many mappings are ambiguous because they require object(s) to simultaneously split and merge. In practice, we have found that such split-merges are rare for the kind of motion graphics videos we focus on in this work. Thus, our approach is to force our algorithm to explain many-to-many mappings as a combination of one-to-one, one-to-many, or many-to-one mappings. Figure 7 shows the complete set of mappings we extract from $B_{fwd}$ and $B_{bwd}$ for the example in Figure 4.

### 4.4 Stage 3: Select best collection of mappings

To select a set of mappings that best explain how objects move from frame $F_{t-1}$ to $F_t$ we first score each candidate mapping we obtain in stage 2 using a visibility-based penalty loss. Suppose $H$ is a candidate mapping type extracted from the forward graph, and $O^H, R^H$ are the set of object(s) and region(s) in $H$. We define the visibility loss $L^{\text{vis}}$ as a masked $L_2$-norm of color differences between the composite image $C(O^*)$ of the transformed and layered objects, and the current frame $F_t$. That is,

$$ L^{\text{vis}}(O^H, R^H) = ||(C(O^*) - F_t) \otimes M^\text{all}||_2, \quad (4) $$

where $\otimes$ denotes pixel-wise multiplication and $M^\text{all}$ is a mask

$$ M^\text{all} = \left( \bigcup_{O_i \in O^H_{t-1}} M^{\text{vis}}(O_i, C(O^*)) \right) \cup \left( \bigcup_{R_j \in R^H_t} M^{\text{vis}}(R_j, F_t) \right), \quad (5) $$

consisting of the union of the visible pixels of all of the transformed objects $O_i \in O^H_{t-1}$ (first term) with the union of all of the regions $R_j \in R^H_t$ (second term). This loss is minimized when the pixels of the transformed objects in $H$ match those of corresponding regions in $H$ and there are no mismatched pixels. Similarly, if $H$ is a candidate mapping type from the backward graph, we compute the penalty score as $L^{\text{vis}}(R^H, O^H_{t-1})$, while replacing $O^*$ with $R^*$ and $F_t$ with $F_{t-1}$ in Equations 4 and 5. In particular, the visibility loss

![Diagram](image-url)
Thus we maintain a high-resolution appearance for each object.

We next select a set of conflict-free mappings from our set of candidates that collectively best explain how objects move, appear, or disappear between frames \( F_{t-1} \) and \( F_t \). A pair of candidate mappings are in conflict if they include the same object or region (Figure 7). Starting with the complete set of candidate mappings, we repeatedly select the candidate with the lowest penalty score and remove all conflicting candidates from the set. We stop when the candidate mapping set is empty, or the lowest score of the remaining candidates is greater than a threshold \( \epsilon \). We have found that \( \epsilon = 0.1 \) gives good results across all our examples.

Finally, we propagate object IDs from the previous frame objects \( O_{t-1} \) to current frame regions \( R_t \) based on the selected mappings as shown in Figure 8. Anytime an object disappears we do not propagate its ID to any subsequent regions. Thus, objects which become completely occluded will re-appear with a new ID by default, though this can be easily changed with user input (Section 5). During this process we also keep track of a *canonical image* for each object. When an object first appears, we save its labeled pixels as its canonical image. Every time an object appears unoccluded and covers a larger region of pixels in a subsequent frame, we update that canonical appearance by replacing the entire canonical image. Thus we maintain a high-resolution appearance for each object.

### 4.5 Stage 4: Write an SVG motion program

In the final stage, we refactor the frame-to-frame affine motion transforms for each object into an affine transform mapping the object’s canonical image to each frame. This motion refactoring could be obtained by multiplying the frame-to-frame transforms or their inverses. In practice, we have found that we can further increase motion accuracy by re-running the DC optimization using the canonical images as the source and the corresponding labeled pixels in each frame as the target. Finally, we write out a SVG motion program with a static background image and a set of foreground objects, each represented by a canonical image, a per-frame sequence of affine transforms placing the canonical image in the frame, and a per-frame z-index depth for the object.

### 5 RESULTS: MOTION VECTORIZATION

Figure 1 shows an abstracted example of the SVG motion program our vectorization pipeline recovers from an input motion graphics video. We apply our motion vectorization pipeline on a test set of 38 motion graphics videos sourced from the Web, with many containing occlusions or fast object motion. A few videos include textures, photographic elements or color gradients in the foreground or background. Table 1 (Appendix A) shows the number of tracking errors as well as the count of errors amongst each mapping type for all the videos in our test set. The average \( L_2 \) RGB reconstruction error for sprite-from-sprite on this subset of videos is 0.018, compared to 0.0079 using our method. See supplemental materials A for a more detailed discussion of this comparison.

We also compute the number of tracking errors in each video. We define a tracking error as any time a mapping from objects in frame \( F_{t-1} \) to regions in \( F_t \) is incorrect with respect to a manually annotated set of ground truth mappings. Table 1 (Appendix A) shows the total number of such tracking errors as well as the count of errors amongst each mapping type for all the videos in our test set.

We find that 24 videos in our test set contain no tracking errors at all, even as some of them contain fast motion, occlusions, or both. The remaining 14 videos all contain 15 errors or fewer. Across all the videos, 75% of the tracking errors occur in one-to-one mappings. Such errors are often due to frame motion and occlusions when objects enter or exit the frame (Figure 9 right top). The next most common tracking error type, at 21%, is incorrect one-to-many (no-splits) mappings. Such errors often occur when objects occlude one another and the mapping is misidentified as a one-to-many (split) (Figure 9 right bottom). Two of the three remaining tracking errors occur when many-to-one (no merge) mappings are misidentified as many-to-one (merge) mappings. In these cases the video contains

![Candidate mappings (\( L^\text{vis} \))](image)

**Fig. 7.** We compute penalty scores \( L^\text{vis} \) for each candidate mapping and then select the best conflict-free set of mappings using a greedy approach.

![Penalty scores](image)

**Fig. 8.** Propagating IDs based on mapping type. For one-to-one and one-to-many (no split) mappings, we assign all pixels of the corresponding region(s) the ID of the object. For one-to-many (split) and unmapped region (appear) mappings, we create new IDs and label the pixels of each region with a different ID. For many-to-one (merge) mappings, we create a new ID to assign to the pixels of the region and then relabel all previous instances of the corresponding objects in the mapping to this new ID. For many-to-one (no merge) mappings, we assign the IDs of each object \( O_i \) in the mapping to the corresponding pixels in \( \Theta_{O_i} (O_i) \).

ACM Trans. Graph., Vol. 42, No. 6, Article 229. Publication date: December 2023.
Fig. 9. **Left:** Reconstruction errors ($L_2$ RGB difference) between frames of input motion graphics videos (5k, avokiddo) and the corresponding frames rendered with the SVG motion program generated by our vectorization pipeline. **Right:** Tracking errors due to fast object motions and occlusions. **Right Top:** The kapptivate input video contains characters translating quickly right to right. In frame 13 the ‘a’ is correctly assigned object ID 3, but in frame 14 it is incorrectly assigned a new object ID 8. This occurs because the leftmost ‘p’ in frame 14 is the closest similar looking region to the ‘a’ in frame 13 but the candidate mapping between the ‘a’ and the ‘p’ is rejected as being too low quality. The ‘p’ in frame 13 is also incorrectly mapped to the rightmost ‘p’ in frame 14 for similar reasons, while the leftmost ‘p’ in frame 14 is incorrectly assigned a new object ID 7 since it remains unmatched. Thus this example yields 2 one-to-one mapping errors and 1 unmatched region (appear) error. **Right Bottom:** In the lucy video object 22 is correctly tracked before frame 76 (we visualize it in frame 69 to show the complete unoccluded object). In frame 76 occlusions alter the visibility of the corresponding region so much that a one-to-many (no split) mapping is misidentified as a one-to-many (split) mapping and the additional regions are given brand new IDs 27, 29 and 31.

**Discussion.** The SVG motion programs produced by our vectorization pipeline provide a representation of motion graphics videos that can be rendered using a SVG renderer, including most Web browsers. In addition, the motion programs can be edited using a SVG animation editor. We have built SVG motion program importers for Adobe After Effects [Christiansen 2013] and Blender [Community 2018]. Such editors allow users to manually customize the motion and appearance of the objects using a graphical interface they may already be familiar with (see supplementary video).

6 MOTION PROGRAM TRANSFORMATION

Our program transformation API lets users programmatically express different ways of manipulating an SVG motion program to generate variations of it. Our approach is to treat the SVG motion program as a scene graph that describes the motions of objects over time. Our API adopts a well-known design pattern for working with a scene graph via two types of methods; (1) **state queries** that look up information about the objects and events in the scene, and

Similarly colored overlapping objects that move in unison, so our pipeline merges them into one object. The final tracking error occurs when a newly appearing region is incorrectly mapped to an existing object. The unmatched region (appear) mapping is misidentified as a one-to-one mapping (example in Figure 9 top right). Our test set did not produce errors of the other four mapping types.

**Correcting tracking errors.** Most tracking errors are easily fixed by reassigning object IDs to regions. For instance if a region was assigned object ID 3 but should have been assigned object ID 7, we can manually relabel it. We provide a programmatic interface for such reassignment. An error in a many-to-one (no merge) mapping can require breaking the pixel mask of a region into multiple regions. In this case users can manually specify the pixel boundaries of each region in the frame where the error appears in Stage 1 of our pipeline to enforce the correct region boundaries. We found this correction to only be necessary for two videos (shapeman, confetti) in our test set. In general however, because our pipeline produces relatively few tracking errors they can often be corrected very quickly.

ACM Trans. Graph., Vol. 42, No. 6, Article 229. Publication date: December 2023.
(2) **operators** that modify the appearance or motion of objects. A transformation program typically starts by querying for a set of objects based on their properties (e.g. red colored objects) or the events they participate in (e.g. collisions) and then applies one or more operators to modify the selected objects. This design pattern of querying and then modifying a scene graph is often used in game engines (e.g., Unity [Unity Technologies 2023]) as well as Web APIs (e.g., jQuery [OpenJS Foundation 2023], D3 [Bostock et al. 2011], CSS [Mozilla 2023] and Chickenfoot [Bolin et al. 2005]) that treat the DOM as a scene graph.

We describe the methods of our program transformation API (Sections 6.1 and 6.2) and briefly describe how we can use them to build a variety of higher-level transformation effects (Section 6.3). The supplemental materials B provides additional details about our API as well as multiple code examples. While our proof-of-concept implementation of the API enables all of the examples that follow, it is meant to minimally demonstrate our approach. In practice, it could be extended to include additional state queries and operators as necessary.

### 6.1 Program Transformation API: State Queries

State queries retrieve properties or events for a specific object, over a range of frames:

```python
propQuery(obj, propType, [frmA, frmB]): Returns a property of obj for each frame in [frmA, frmB] based on propType. Property types include: all, color, position, size, velocity, etc.

eventQuery(obj, eventType, [frmA, frmB]): Returns a list of events obj is involved in over the range of frames [frmA, frmB] based on eventType. Event types include: heldFrames, collisionFrames, motionCycleFrames, etc.
```

To handle property queries, our API internally computes the chosen property for the object from our motion program representation. For example, to compute the color property of an object it clusters the pixels of the canonical image in color space and returns the color of the largest cluster for each frame in the frame range. Properties that vary based on the motion (e.g., position, size, velocity) are computed using the objects motion transform and reported in the global coordinates of the video frame. The all property type returns all objects that appear in motion program over the frame range.

To handle event queries, our API internally processes the motion of the object to find frames when the chosen event type occurs. For example, to identify heldFrames we look for successive frames of the object where the motion transform from the canonical image to the frame placement remains fixed and return a list of all such frames. To identify collisionFrames we look for frames where the closest distance between the object boundary and another object boundary is below a threshold (e.g. the objects touch) and at least one of the objects experiences a large change in velocity. The API returns a list of collisions including the other object(s) involved and the points of contact on each object. To identify motionCycleFrames we look for peaks in the autocorrelation of motion parameters (translation, rotation, scale skew) of the object and return a list of the corresponding frames.

### 6.2 Program Transformation API: Operators

Our API provides operators to modify the appearance or motion of a specific object over a range of frames including:

```python
retime(obj, [sFrmA, sFrmB], [tFrmA, tFrmB], easeFn[t]): Linearly remap motion transforms in source frame range [sFrmA, sFrmB] to target frame range [tFrmA, tFrmB]. Then resample the transforms in the target frame range using easing function easeFn[t].

adjGlobalMotion(obj, xformFn[t], [frmA, frmB]): Adjust motion of obj in local coordinate frame (i.e., of canonical image), over the range of frames [frmA, frmB] based on affine transforms generated by linearly sampling xformFn[t] in the range [0, 1]. This method post-multiplies canonical-to-frame transform of obj.

adjLocalMotion(obj, xformFn[t], [frmA, frmB]): Adjust motion of obj in global coordinate frame (i.e., of video frame), over the range of frames [frmA, frmB] based on affine transforms generated by linearly sampling xformFn[t] in the range [0, 1]. This method pre-multiplies the canonical-to-frame transform of obj.

changeAppearance(obj, newAppearance, [frmA, frmB]): Set canonical image of obj to newAppearance for frames in [frmA, frmB].
```

In addition to the operators listed here, our API provides basic operators for creating new objects, deleting objects, copying motions, setting the motion transforms (rather than adjusting them via pre- or post-multiplication), etc.

Figure 10 shows the the general pattern of a motion program transformer, written with our API. An objSelector code block (or function) selects one or more objects for transformation using a propQuery or eventQuery. An objTransformer code block (or function) then applies one or more operators to change the timing, motion or appearance of the selected object(s). For example, to transform all of the red colored objects to blue, the objSelector function would run a propQuery to obtain the color of each object and then select out the red ones. Then the objTransformer code block would use changeAppearance to set the color of the selected objects to blue.

### 6.3 Higher-level object transformer effects

Using our motion program transformation API we have built a variety of objTransformer functions that each produce a different, higher-level effect on the timing, motion or appearance of objects (e.g. anticipation/follow-through, motion textures). Several of these transformers implement motion adjustments commonly found in other animation editing systems [Kazi et al. 2014, 2016; Ma et al. 2022]. Importantly, the functions in our API are designed to compose with one another and facilitate the creation of many variations of a motion graphic, thereby supporting iterative design and exploration. Figure 10 provides code for a few object transformers, and supplemental materials B includes code for all of them. The supplemental website also includes multiple example SVG motion programs transformed by each of the higher-level effects described here that can be executed in a Web browser. The following sections give a brief overview of the types of object transformers.

**Retiming:** These object transformers manipulate an individual object timeline. This includes functions that linearly stretch or shrink the time scale of an object, apply slow in/out easing, retime object motions to reference audio beats, etc. See Figure 10c and 10d for examples.
spatial motion adjustment. These adjustment object transformers manipulate how an individual object moves across the frame. This includes functions that add anticipation/follow-through (Figure 10e) and functions that apply motion textures (e.g. wobbling or pulsing) to an existing motion.

appearance adjustment. The changeAppearance object transformer updates the appearance of a given object by replacing the canonical appearance of an object with a new image. One unintended consequence of an appearance change is that collisions between objects may be affected. For instance, naively changing the dark blue circle in Figure 11 to a smaller-sized coin would not maintain collisions between the smaller coin and the yellow circle. Since collisions are often important events in a video, we also allow for collision-preserving appearance changes. This type of appearance change uses event queries to find collisionFrames and then applies local motion adjustments to best preserve the original collisions at those frames.

7. RESULTS: MOTION PROGRAM TRANSFORMATION

(a) Motion program transformer (skeleton)

(b) Object selector

(c) Object transformer: Linear time stretch/shrink

(d) Object transformer: Retiming object motion to music beats

(e) Object transformer: Anticipation/follow-through

Fig. 10. The general structure of motion program transformer (a) takes an SVG motion program P as input and alternates object selector blocks with object transformer blocks to modify the SVG program. The object selector function objSelector (b) selects one or more objects for transformation. It first runs queryFn (i.e., either propQuery or eventQuery) using the specified queryType (i.e., color, collisionFrames) and then filters the objects to only those that match the specified criteria. The object transformers adjust the timing (c, d) motion (e) or appearance of a set of selected objects selObjs. See the supplemental material B for additional examples of object transformers we have built to achieve a variety of effects.

Spatial motion adjustment. These adjustment object transformers manipulate how an individual object moves across the frame. This includes functions that add anticipation/follow-through (Figure 10e) and functions that apply motion textures (e.g. wobbling or pulsing) to an existing motion.

Appearance adjustment. The changeAppearance object transformer updates the appearance of a given object by replacing the canonical appearance of an object with a new image. One unintended consequence of an appearance change is that collisions between objects may be affected. For instance, naively changing the dark blue circle in Figure 11 to a smaller-sized coin would not maintain collisions between the smaller coin and the yellow circle. Since collisions are often important events in a video, we also allow for collision-preserving appearance changes. This type of appearance change uses event queries to find collisionFrames and then applies local motion adjustments to best preserve the original collisions at those frames.

7 RESULTS: MOTION PROGRAM TRANSFORMATION

By combining objSelector and objTransformer blocks, we can create a variety of motion graphic variations. Figure 1, Figure 11 and Figures 5–6 in supplemental materials A show examples where we have composed multiple objSelector and objTransformer blocks to generate complex variations of retiming, spatial motion adjustment and appearances changes. Executable SVG motion programs and program transformer code for other additional examples with retiming, spatial motion adjustments and appearance adjustments are provided in the supplemental website. We encourage readers to browse the examples to see the breadth of different transformations and variations that can be achieved with our motion program transformation API.

Usability evaluation. To further evaluate the usability of our program transformation API, we asked 10 people (all experienced Python coders, 5 familiar with query-then-operate design pattern) to use the API to programmatically create a variation of an SVG motion program (Figure 12). We first gave each participant a 30 minute tutorial (a combination of oral instruction and a Colab notebook) explaining how to use the API. We then gave them 15 minutes to write their own program transforming an animated digital card into one suitable for a different occasion.

All participants successfully wrote a transformation program containing two or more object queries and transformations. On a 5 point Likert scale (1 = very hard, 5 = very easy) they all rated the query-then-operate pattern as easy or very easy to understand. Two participants who were familiar with the design pattern compared the structure of our API to SQL and other scene-graph based content creation APIs like Maya [Autodesk, INC. 2023a] and MotionBuilder [Autodesk, INC. 2023b]. Multiple participants stated in free-response feedback that the API was “intuitive to understand,” “lightweight and natural,” and “easy to use.”

Many participants liked the expressivity of the API. Nine participants noted that the API was flexible enough to accomplish the edits they wanted to make. One participant liked “how powerful the API is while still being easy to use,” further commenting that “it covered a lot of possible transformations within relatively simple operations.” Another wrote that the programmatic approach of our
"would be especially useful for mass producing animations or graphics video with a static background and solid-colored, lightly Lifting assumptions on input video. Our work focuses on motion there are a few limitations that warrant future work.

8 LIMITATIONS AND FUTURE WORK

Our work enables editing of motion graphics video by first converting the video into an SVG motion program and then using motion program transformers programmatically create variations. However there are a few limitations that warrant future work.

Lifting assumptions on input video. Our work focuses on motion graphics video with a static background and solid-colored, lightly textured or gradient-filled objects undergoing affine motions. Extending our approach to handle natural video containing moving backgrounds with highly textured, photographic foreground objects undergoing deformable motions, may be possible using recent video matting techniques [Kasten et al. 2021; Lu et al. 2021]. Handling non-affine motions within our pipeline would require modifications to the differentiable compositing optimization (Section 4.1) to account for the deformations.

Vectorizing canonical images. Our SVG motion programs represent the appearance of each object using a canonical image. Converting these canonical images into a vector representation (e.g., composed of paths, shapes, gradients, etc.) would bring the benefits of a higher-level abstraction to the appearance of the objects in addition to their motions. Techniques for converting images into vector representations [Orzan et al. 2008; Reddy et al. 2021] is an active area of work that might be adapted to this context.

Higher-level program abstraction based on gestalt principles. Our SVG motion programs represent motion graphics video using abstractions (e.g., objects) and controls (e.g., affine transform parameters) that are more meaningful than pixels and frames of video. One way to provide further meaningful abstraction might be to group objects based on perception and gestalt principles. For example if a motion graphic contains objects (e.g., letters) that move together and are near one another, they might be grouped together to form a higher-level composite object (e.g., a word). Such higher-level grouping could further facilitate program transformation as changes and adjustments could be applied to the composite objects.

GUI for motion editing. Our system enables users to work with a programmatic representation of motion graphics video rather than pixels and frames. However, we have not developed a graphical user interface for editing the resulting SVG motion programs. Indeed, we believe many different GUIs could be built using our motion program representation and our program transformation API. One approach that may be especially fruitful is to extend the bidirectional SVG editing interface of Sketch-n-Sketch [Hempel et al. 2019], so
that direct manipulation changes to the graphics are immediately reflected in the SVG representation and vice versa. Inferring how direct, graphical manipulations should affect an underlying motion program is an important direction for future work.

9 CONCLUSION
While motion graphics videos are prevalent on the Web today, they are difficult to edit because they are simply a collection of pixels and frames. We have presented a motion vectorization pipeline that converts such video into a SVG motion program that represents the video as objects moving over time. We further provide a motion program transformation API that enables programmatic editing of the resulting SVG programs to create variations of the timing, motions, and object appearance. We believe that these tools can allow users to more easily explore motion graphics design options by borrowing from widely-available motion graphics video examples and that they open the door to dynamically adapting the graphics to the preferences of the viewer.

ACKNOWLEDGMENTS
We thank Lvmin Zhang for valuable discussions on sprite-from-sprite. We would also like to thank the reviewers for their feedback. This research is supported by NSF Award #2219864, the Brown Institute for Media Innovation and the Stanford Institute for Human-Centered AI (HAI).

REFERENCES

Table 1. Our test set of 38 motion graphics videos. Six of the videos contain no occlusions and no fast motion. Twelve contain only occlusions and no fast motions. Seven contain only fast motion. Thirteen contain both. Some of the videos contain textures, photographic elements or color gradients in the foreground or background (marked with ‡). The reconstruction $L_2$ error shows the average RGB error for the SVG motion program produced by our vectorization pipeline. The rightmost columns show the total number of tracking errors (all) and the errors by mapping type (Figure 3).

<table>
<thead>
<tr>
<th>Video</th>
<th>Num. frames</th>
<th>Num. objs</th>
<th>Recon. $L_2$ error</th>
<th>Tracking errors</th>
<th>All Mapping type</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>No occlusions and no fast motion</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ball2</td>
<td>500</td>
<td>4</td>
<td>0.0034</td>
<td>0</td>
<td>0,0,0,0,0,0,0,0</td>
</tr>
<tr>
<td>ball3</td>
<td>215</td>
<td>8</td>
<td>0.0024</td>
<td>0</td>
<td>0,0,0,0,0,0,0,0</td>
</tr>
<tr>
<td>eyes</td>
<td>312</td>
<td>14</td>
<td>0.0050</td>
<td>0</td>
<td>0,0,0,0,0,0,0,0</td>
</tr>
<tr>
<td>format</td>
<td>151</td>
<td>6</td>
<td>0.0036</td>
<td>0</td>
<td>0,0,0,0,0,0,0,0</td>
</tr>
<tr>
<td>levers</td>
<td>144</td>
<td>6</td>
<td>0.0063</td>
<td>0</td>
<td>0,0,0,0,0,0,0,0</td>
</tr>
<tr>
<td>support</td>
<td>299</td>
<td>9</td>
<td>0.0024</td>
<td>0</td>
<td>0,0,0,0,0,0,0,0</td>
</tr>
<tr>
<td>Occlusions only</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dog</td>
<td>133</td>
<td>12</td>
<td>0.017</td>
<td>0</td>
<td>0,0,0,0,0,0,0,0</td>
</tr>
<tr>
<td>five</td>
<td>144</td>
<td>5</td>
<td>0.0024</td>
<td>0</td>
<td>0,0,0,0,0,0,0,0</td>
</tr>
<tr>
<td>giftbox1</td>
<td>80</td>
<td>8</td>
<td>0.0078</td>
<td>0</td>
<td>0,0,0,0,0,0,0,0</td>
</tr>
<tr>
<td>giftbox2</td>
<td>80</td>
<td>10</td>
<td>0.012</td>
<td>0</td>
<td>0,0,0,0,0,0,0,0</td>
</tr>
<tr>
<td>hype1</td>
<td>144</td>
<td>4</td>
<td>0.022</td>
<td>0</td>
<td>0,0,0,0,0,0,0,0</td>
</tr>
<tr>
<td>hype2</td>
<td>144</td>
<td>4</td>
<td>0.024</td>
<td>0</td>
<td>0,0,0,0,0,0,0,0</td>
</tr>
<tr>
<td>pingpong</td>
<td>144</td>
<td>21</td>
<td>0.0093</td>
<td>0</td>
<td>0,0,0,0,0,0,0,0</td>
</tr>
<tr>
<td>playDesign</td>
<td>438</td>
<td>13</td>
<td>0.0068</td>
<td>0</td>
<td>0,0,0,0,0,0,0,0</td>
</tr>
<tr>
<td>sundance</td>
<td>336</td>
<td>70</td>
<td>0.0071</td>
<td>0</td>
<td>0,0,0,0,0,0,0,0</td>
</tr>
<tr>
<td>ball5</td>
<td>289</td>
<td>4</td>
<td>0.0072</td>
<td>0</td>
<td>0,0,0,0,0,0,0,0</td>
</tr>
<tr>
<td>sydney (‡)</td>
<td>98</td>
<td>44</td>
<td>0.0394</td>
<td>4</td>
<td>4,0,0,0,0,0,0,0</td>
</tr>
<tr>
<td>morningShow</td>
<td>147</td>
<td>162</td>
<td>0.011</td>
<td>5</td>
<td>5,0,0,0,0,0,0,0</td>
</tr>
<tr>
<td>Fast motion only</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ball4</td>
<td>79</td>
<td>2</td>
<td>0.0026</td>
<td>0</td>
<td>0,0,0,0,0,0,0,0</td>
</tr>
<tr>
<td>book2 (‡)</td>
<td>36</td>
<td>36</td>
<td>0.0095</td>
<td>0</td>
<td>0,0,0,0,0,0,0,0</td>
</tr>
<tr>
<td>transforms</td>
<td>358</td>
<td>27</td>
<td>0.0034</td>
<td>0</td>
<td>0,0,0,0,0,0,0,0</td>
</tr>
<tr>
<td>seesaw (‡)</td>
<td>188</td>
<td>4</td>
<td>0.0017</td>
<td>0</td>
<td>0,0,0,0,0,0,0,0</td>
</tr>
<tr>
<td>wordAWeek</td>
<td>151</td>
<td>12</td>
<td>0.0036</td>
<td>0</td>
<td>0,0,0,0,0,0,0,0</td>
</tr>
<tr>
<td>deconstruct</td>
<td>156</td>
<td>11</td>
<td>0.0010</td>
<td>0</td>
<td>0,0,0,0,0,0,0,0</td>
</tr>
<tr>
<td>both occlusions and fast motion</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ball1 (‡)</td>
<td>394</td>
<td>2</td>
<td>0.0083</td>
<td>0</td>
<td>0,0,0,0,0,0,0,0</td>
</tr>
<tr>
<td>face</td>
<td>156</td>
<td>5</td>
<td>0.0011</td>
<td>0</td>
<td>0,0,0,0,0,0,0,0</td>
</tr>
<tr>
<td>filmRadio</td>
<td>177</td>
<td>60</td>
<td>0.0040</td>
<td>1</td>
<td>1,0,0,0,0,0,0,0</td>
</tr>
<tr>
<td>183</td>
<td>96</td>
<td>32</td>
<td>0.010</td>
<td>2</td>
<td>2,0,0,0,0,0,0,0</td>
</tr>
<tr>
<td>g suite (‡)</td>
<td>481</td>
<td>24</td>
<td>0.017</td>
<td>3</td>
<td>3,0,0,0,0,0,0,0</td>
</tr>
<tr>
<td>book1 (‡)</td>
<td>108</td>
<td>7</td>
<td>0.0036</td>
<td>4</td>
<td>4,0,0,0,0,0,0,0</td>
</tr>
<tr>
<td>kapp illuminate</td>
<td>50</td>
<td>13</td>
<td>0.0063</td>
<td>4</td>
<td>3,0,0,0,0,0,0,1</td>
</tr>
<tr>
<td>avokiddo</td>
<td>130</td>
<td>20</td>
<td>0.0033</td>
<td>6</td>
<td>6,0,0,0,0,0,0,0</td>
</tr>
<tr>
<td>dates (‡)</td>
<td>181</td>
<td>36</td>
<td>0.023</td>
<td>6</td>
<td>6,0,0,0,0,0,0,0</td>
</tr>
<tr>
<td>5k (‡)</td>
<td>119</td>
<td>18</td>
<td>0.033</td>
<td>7</td>
<td>4,3,0,0,0,0,0,0</td>
</tr>
<tr>
<td>shapeman</td>
<td>70</td>
<td>14</td>
<td>0.0048</td>
<td>10</td>
<td>6,3,1,0,0,0,0,0</td>
</tr>
<tr>
<td>confetti</td>
<td>45</td>
<td>143</td>
<td>0.012</td>
<td>13</td>
<td>12,0,1,0,0,0,0,0</td>
</tr>
<tr>
<td>lucy</td>
<td>353</td>
<td>33</td>
<td>0.013</td>
<td>15</td>
<td>4,11,0,0,0,0,0,0</td>
</tr>
</tbody>
</table>

A APPENDIX: MOTION GRAPHICS VIDEO TEST SET

We created a test set of 38 motion graphics video to evaluate our motion vectorization pipeline (Table 1). Tracking foreground objects through occlusions (either between objects or at the edge of the frame as on object entry/exit) and across fast motions (which we define as moments when an object’s bounding box in frame $F_{t-1}$ does not overlap with its bounding box in $F_t$) is especially challenging. Many of the test videos contain such challenging features. A few of the more challenging videos also contain textures, photographic elements or color gradients in the foreground or background (marked with ‡). The two rightmost columns of Table 1 show the total number of tracking errors and a breakdown of these errors by mapping type; each element of the 8-tuple records the number of errors for corresponding mapping type as shown in Figure 3. Thus, the video named lucy contains 4 one-to-one mapping errors (e.g., a region in $F_t$ is assigned an incorrect object ID) and 11 one-to-many (no split) errors (e.g., two or more regions that should be assigned the same object ID were incorrectly assigned different object IDs). See Figure 9 (bottom right) for an examples of this error for the lucy video.