Editing Motion Graphics Video via Motion Vectorization and Transformation

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

ACM Reference Format:
Sharon Zhang, Jiaju Ma, Jiajun Wu, Daniel Ritchie, and Maneesh Agrawala. 2023. Editing Motion Graphics Video via Motion Vectorization and Transformation. *ACM Trans. Graph.* 42, 6 (December 2023), 6 pages. https://doi.org/10.1145/3618316

This document provides more details on results of the motion vectorization pipeline and program transformation API. Section 1 provides more comparison between our method and the sprite-from-sprite method [Zhang et al. 2022] (Section 5 of the main paper) and more examples of vectorization errors. Section 2 includes further implementation details on object transformers (Section 6.3 of the main paper), as well as additional examples of program transformers (Section 7 of the main paper) using these object transformers.

1 RESULTS: MOTION VECTORIZATION

Comparison to Zhang et al. [2022]. We compare our results to sprite-from-sprite [Zhang et al. 2022], which takes a video as input and decomposes it into N sprites. There are several key differences between sprite-from-sprite and our method. Most importantly, the appearance of each sprite in the sprite-from-sprite representation is time-dependent, meaning the per-frame homographies may not fully characterize the sprite motion. On the other hand, our representations contains a single appearance for each object, so the object motion throughout the video is fully explained by our motion parameters. Consequently, the sprite-from-sprite representation may reconstruct the input video more accurately, but editing the video still has to be done at the per-frame level rather than the object level. Moreover, sprite-from-sprite often reconstructs the input video more accurately at the expense of a meaningful sprite decomposition (see Figure 1). Another difference in the sprite-from-sprite method is that it assumes fixed depth ordering. This can impact the quality of the sprite appearances. Figure 2 shows the results of a video with alternating depth ordering after applying our method and sprite-from-sprite.

Out of the 38 test videos, sprite-from-sprite decomposed 30 of them and ran out of memory for the remaining 8 videos. Overall, the average reconstruction error of sprite-from-sprite [Zhang et al. 2022] is 0.018 on the 30 successfully decomposed videos, compared to our average reconstruction error of 0.0079 on the same subset of videos. A number of sprite-from-sprite decompositions also resulted in trivial sprites, *i.e.*every pixel was assigned to a single sprite. We use a Nvidia Titan RTX as opposed to a Nvidia RTX 3080, which was used in the original paper. The Github repository for sprite-from-sprite notes that this may affect the sprite decomposition results.

Non-affine motions and severe occlusions. As mentioned in the main paper, our motion vectorization pipeline assumes an affine motion model. When videos do not follow this assumption well, our pipeline is still able to generate an SVG motion program, but the resulting representation may contain extra objects and the reconstruction may not be as accurate. Objects which never appear fully unoccluded may also appear to go against the affine motion assumption, as we never have a true canonical appearance from the video content. Our pipeline is still able to produce an SVG motion program, but the motion program may not reconstruct the input video as well. Figure 3 illustrates these two cases.

2 MOTION PROGRAM TRANSFORMATION: EXAMPLES

The object transformers in our API transform the timing, spatial motion and appearances of objects. Figure 4 shows code for several object transformers, and Figures 5–7 illustrate applications of different object transformers to create complex variations of an input motion graphic. The following sections describe the implementation of each object transformer in more detail.

Retiming

We have developed several object transformers that change the timing of individual object(s) using the `retime` operator.

Linear time stretch/shrink. To linearly stretch (or shrink) the timing of an object by a factor of k over a source frames `[sFrmA, sFrmB]`, we specify a target frame range `[tFrmA, tFrmB]` such that its duration is k times the duration of the source frame range and we use the identity easing function $f(t) = t$. Code shown in Figure 4c.

Slow in/out easing. To add slow in/out easing to the timing of an object over source frames `[sFrmA, sFrmB]` we specify a target

Authors' addresses: Sharon Zhang, Stanford University, USA, szhang25@stanford.edu; Jiaju Ma, Stanford University, USA, jiajuma@stanford.edu; Jiajun Wu, Stanford University, USA, jiajunwu@cs.stanford.edu; Daniel Ritchie, Brown University, USA, daniel_ritchie@brown.edu; Maneesh Agrawala, Stanford University and Roblox, USA, maneesh@cs.stanford.edu.

0730-0301/2023/12-ART $15.00
https://doi.org/10.1145/3618316

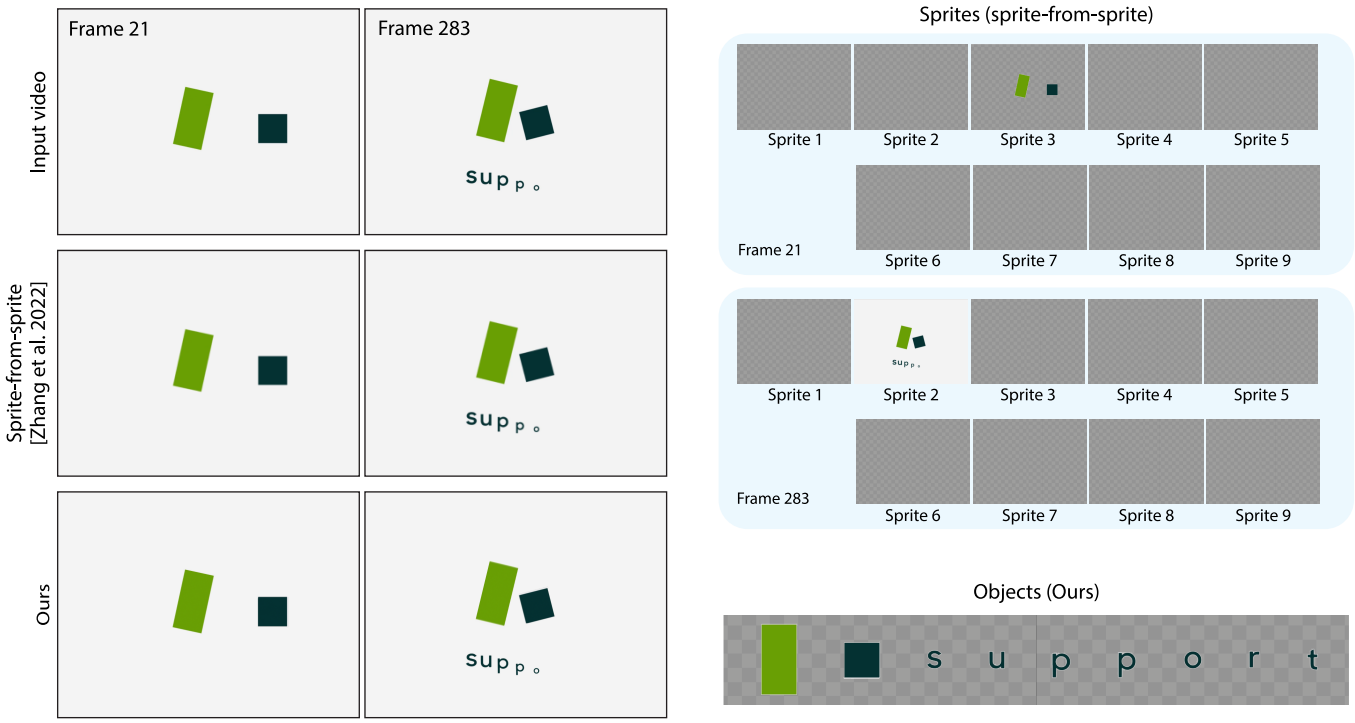


Fig. 1. **Comparison between our method and sprite-from-sprite [Zhang et al. 2022].** While the reconstruction quality of sprite-from-sprite is comparable to ours, the sprite decomposition may not be meaningful. In this video (*logo8*), our method correctly extracts nine objects. Decomposing the same video into nine foreground sprites with sprite-from-sprite results in multiple objects within one sprite, and also inconsistent object assignments across sprites (the two objects in Sprite 3 at frame 21 appear in Sprite 2 at frame 283).

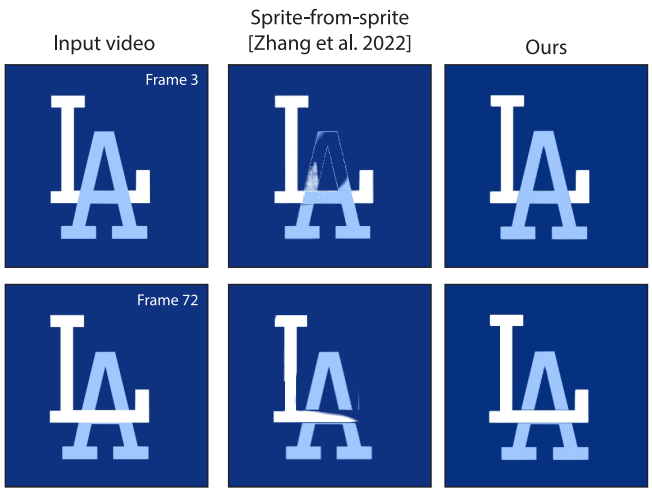


Fig. 2. Sprite-from-sprite [Zhang et al. 2022] decomposes a video into a fixed number of sprites, with depth ordering fixed throughout the video. In this video *LA*, the letters 'L' and 'A' alternate in front and behind positions. Our method is able to model this variation in depth but the sprite-from-sprite reconstruction suffers from flickering artifacts.

frame range of the same duration as the source and use a nonlinear easing function (e.g., $f(t) = t^4$ to generate slow in timing)[1].

Animating on 2s, 3s, Ns. Traditional animators sometimes hold frames of moving objects to produce to stylize the motion or create a stop-motion look. We introduce this effect to the motion of an object, by using a step function as the easing where the size and position of each step is based on how long the each frame should be held and the duration of the target frame range.

Removing held frames for each object. We can also remove held frames from motions of an object to smooth its motion. To do so, we first run an event query to find all the `heldFrames` for an object and linearly shrink each segment of held frames to a single frame. To restore the timing we linearly stretch each reduced frame and next frame back to the duration of the original held frame segment.

Retiming object motion to music beats. Given a piece of music, this object transformer extracts the beats in units of frames using libROSA [McFee et al. 2015] and then breaks the timeline of an object into segments. The segments can either match the length of time between successive beats, or align with events such as `motionCycleFrames` or `collisionFrames`. Next we apply the `retime` operator so that the object motion segments match the beat length and use a non-linear easing function that accentuates the

[1]See `easings.net` for a collection of commonly used easing functions.

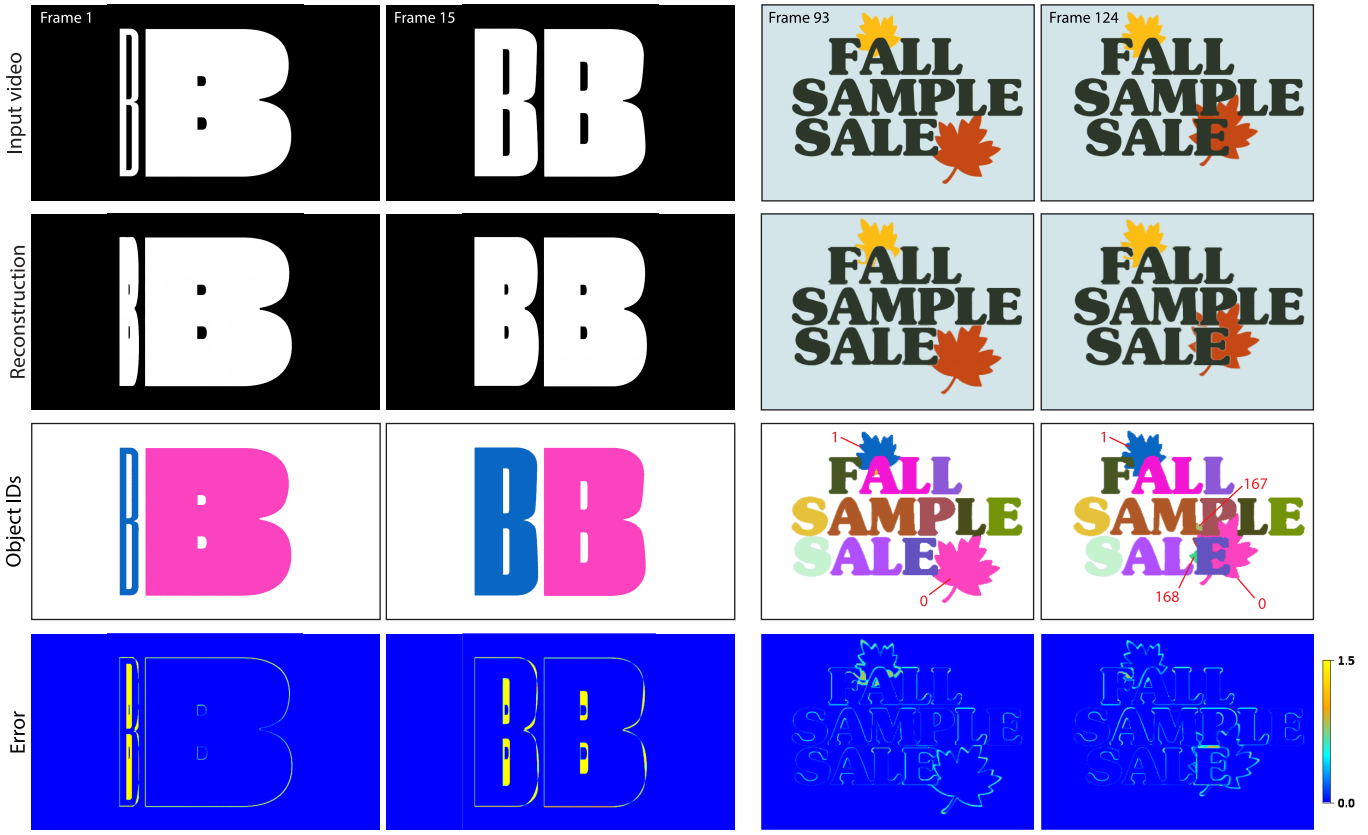


Fig. 3. **Left: Non-affine deformations.** The two 'B's in this video do not deform affinely, so our SVG motion program does not reconstruct it accurately. **Right: Severe occlusions.** The leaves never appear fully unoccluded in this video. This causes the SVG motion program to represent one of them with multiple object IDs (0, 167 and 168) and also results in reconstruction errors along the occlusion boundaries (see misalignments around 'FALL').

motion in and out of each beat point, in the manner of Davis and Agrawala [2018]. Code shown in Figure 4d.

Spatial Motion Adjustment

We can use the `adjLocalMotion` and `adjGlobalMotion` operators to adjust the spatial trajectories of objects.

Motion texture. In the context of motion graphics we define *motion textures* as local spatial perturbations to the motion of an object. We can apply such motion textures to an object by first defining a transform function `xformFn` that specifies a perturbation to be applied in the local coordinate frame (i.e. the canonical image frame) of the object and then passing it into the `adjLocalMotion` operator. For example suppose the input video has and object translating from left to across the screen. We can provide a transform function that translates the object along its local y-axis according to sine function, to produce an oscillating motion up and down as the object moves from left to right in the frame. We can also create multiple copies of an object and add slightly different local perturbations to each one to create a form of Kazi et al.'s kinetic texture [2014].

Anticipation/follow-through. We can apply Wang et al.'s [2006] cartoon animation filter on the motion of an object in the global coordinate frame to add anticipation and follow-through to the motion. As defined by Wang et al., the cartoon animation filter takes a time varying signal $x(t)$, convolves it with an inverted Laplacian of a Gaussian (LoG) filter and adds the convolved result back to the original signal. In our setting, we treat the motion transforms of the object as the signal and define a transform function `xformFn` that performs the convolution. We then pass this transform function to the `adjGlobalMotion` operator to add the convolved result back to the original motion. Code shown in Figure 4e.

Appearance Adjustment

We can use the `changeAppearance` method to replace the canonical image of and object with a completely new visual appearance. However if the new appearance differs significantly in shape from the original it may not preserve collisions with other objects. We have developed a motion program transformer that can can adjust the motions of the colliding objects after such appearance changes to preserve collisions in certain cases.

Collision preserving appearance change. If the new appearance lies within the contour of the original appearance it cannot introduce any new collisions with other objects in the scene. But even with

Table 1. A comparison of L_2 RGB reconstruction errors of sprite-from-sprite [Zhang et al. 2022] against our method. Videos with textures, photographic elements or color gradients in the foreground or background are marked with ‡. Sprite-from-sprite was unable to decompose the eight videos with more than 32 objects due to out-of-memory errors (indicated by —).

Video	Reconstruction L_2 error	
	Sprite-from-sprite [2022]	Ours
No occlusions and no fast motion		
ball2	0.00019	0.0034
ball3	0.0	0.0024
eyes	0.017	0.0050
format	0.29	0.0036
levers	0.0067	0.0063
support	0.00012	0.0024
Occlusions only		
dog	0.0058	0.017
five	0.0035	0.0024
giftbox1	0.0043	0.0078
giftbox2	0.0038	0.012
hype1	0.0011	0.022
hype2	2.7e-5	0.024
pingpong	0.0095	0.0093
playDesign	1.2e-5	0.0068
sundance	—	0.0071
ball5	0.013	0.0072
sydney (‡)	—	0.0394
morningShow	—	0.011
Fast motion only		
ball4	2.8e-5	0.0026
book2 (‡)	—	0.0095
transforms	0.0	0.0034
seesaw (‡)	0.0095	0.0017
wordAWeek	0.054	0.0036
deconstruct	4.3e-5	0.0010
beautiful	0.013	0.0037
Both occlusions and fast motion		
ball1 (‡)	0.0059	0.0083
face	0.0022	0.0011
filmRadio	—	0.0040
183	3.7e-5	0.010
gsuite (‡)	0.024	0.017
book1 (‡)	0.046	0.0036
kapptivate	5.6e-6	0.0063
avokiddo	0.032	0.0033
dates (‡)	—	0.023
5k (‡)	4.1e-5	0.033
shapeman	0.0	0.0048
confetti	—	0.012
lucy	—	0.013

this restriction, the new appearance may not fill the contours of the original object, and thereby miss collisions that appeared in the original video. However, we can preserve such collisions by locally adjusting the motion of the new object as follows. Our event query method for `collisionFrames` provides the collision point in the local coordinate frame of the original object. We find the closest point on the contour of the new appearance to the collision point and add a local translation to the object, using `adjLocalMotion`, so that this closest point matches the collision point. In practice, we spread the local adjustment so that it occurs gradually over the set of frames from the most recent previous collision of the object. We also allow the motion adjustment to occur on the other object involved in the collision or some combination of both objects. Note that we do not check if the local adjustments will move an object outside the contour of the original object and potentially introduce new collisions. But in practice we have found that new collisions are rare. We also note that this approach only considers pairwise collisions and cannot handle more than one simultaneous collision.

REFERENCES

Abe Davis and Maneesh Agrawala. 2018. Visual Rhythm and Beat. *ACM Transactions on Graphics (TOG)* 37, 4, Article 122 (jul 2018), 11 pages. https://doi.org/10.1145/3197517.3201371

Rubaiat Habib Kazi, Fanny Chevalier, Tovi Grossman, Shengdong Zhao, and George Fitzmaurice. 2014. Draco: Bringing Life to Illustrations with Kinetic Textures. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Toronto, Ontario, Canada) *(CHI '14)*. Association for Computing Machinery, New York, NY, USA, 351–360. https://doi.org/10.1145/2556288.2556987

Brian McFee, Colin Raffel, Dawen Liang, Daniel Ellis, Matt McVicar, Eric Battenberg, and Oriol Nieto. 2015. librosa: Audio and Music Signal Analysis in Python. In *Proceedings of the 14th Python in Science Conference*. 18–24. https://doi.org/10.25080/majora-7b98e3ed-003

Jue Wang, Steven M. Drucker, Maneesh Agrawala, and Michael F. Cohen. 2006. The Cartoon Animation Filter. *ACM Transactions on Graphics* 25, 3 (2006). https://doi.org/10.1145/1141911.1142010

Lvmin Zhang, Tien-Tsin Wong, and Yuxin Liu. 2022. Sprite-from-Sprite: Cartoon Animation Decomposition with Self-Supervised Sprite Estimation. *ACM Trans. Graph.* 41, 6, Article 192 (nov 2022), 12 pages. https://doi.org/10.1145/3550454.3555439

(a) Motion program transformer (skeleton)

```
// Program Transformer structure.
MPTransformer(P, *args, [frmA, frmB]):
  // OBJ SELECTOR: Select objects in P via queries using any criteria
  // specified in the args.
  ...

  // OBJ TRANSFORMER: Apply an object operator to selected objects.
  ...
```

(b) Object selector

```
// Returns a list of object data which match some criteria.
function objSelector(P, queryFn, queryType, criteria, [frmA, frmB]):
  selObjs = {}
  selObjsInfo = {}
  for each obj in selObjs:
    x = queryFn(obj, args.queryType, [frmA, frmB])
    if x matches criteria:
      selObjs.insert(obj)
      selObjsInfo.insert(x)

  return selObjs, selObjsInfo
```

(c) Object transformer: Linear time stretch/shrink

```
// Linear time scale by factor of k in frame range [frmA, frmB].
function linearRetimeObjTransformer(selObjs, k, [frmA, frmB]):
  for each obj in selObjs:
    sourceDur = frmB - frmA + 1
    targetDur = k * sourceDur
    // Retime from source range [frmA, frmB] to target frame range
    // [frmA, frm + targetDur].
    retime(obj, [frmA, frmB], [frmA, frmA + targetDur], f(t)=t)
```

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

```
// Retime to music beats (assume video has more segments than beats).
function retimeToBeatsObjTransformer(selObjs, music, eventType, [frmA, frmB]):
  // Get music beat points using libROSA in units of frames.
  beatPts = getMusicBeatPts(music)

  for each obj in selObjs:
  // Form video segments for each beat segment between beat points based on
  // eventType. If eventType is null default to beatPts as segment points.
  if eventType == null:
    segPts = beatPts
  else:
    segPts = eventQuery(obj, eventType, [frmA, frmB])

  for index i in segPts:
    // beatPts is in units of frames and includes a beat point at 0.
    retime(obj, [segPts[i], segPts[i + 1]],
           [beatPts[i], beatPts[i + 1]], f(t)=t^4)
```

(e) Object transformer: Anticipation/follow-through

```
// Add anticipation/follow through via Cartoon Animation Filter.
function anticipateFollowThruObjTransformer(selObjs, [frmA, frmB], A, sigma):
  for each obj in selObjs:
    // Define the cartoon animation filter based on Wang et al.
    function cartoonAnimationFilter(t, obj, [frmA, frmB], A, sigma):
      // Copy and pad segment of xForms to be set up for convolution later.
      tmpXForms = copy(obj.xForms[frmA, frmB])
      pad(tmpXForms, 0.5 * sigma)
      // -LoG is the inverse of the Laplacian of Gaussian function.
      newXForms = A * convolve(tmpXForms, -LoG(sigma))
      return newXForms[t]

  adjGlobalMotion(obj, cartoonAnimationFilter, [frmA, frmB]
```

Fig. 4. 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`.

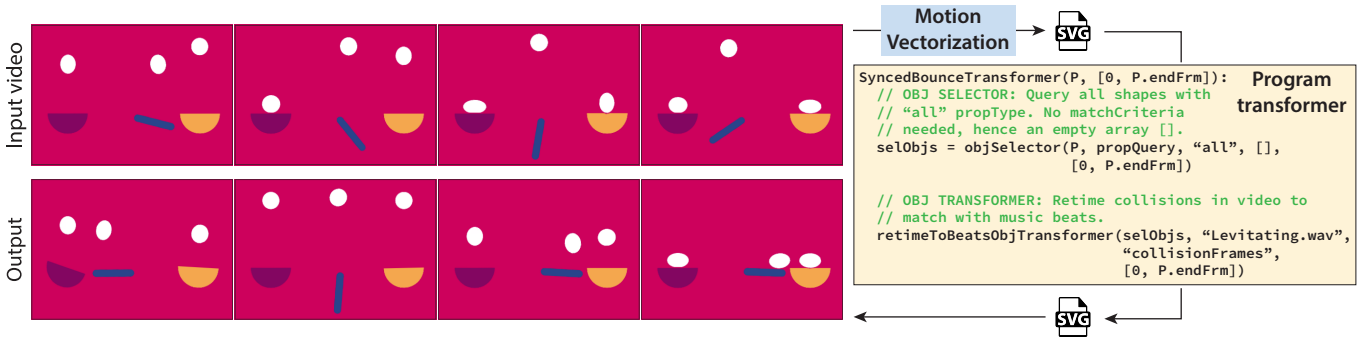


Fig. 5. Retiming collisions to music beat. In the input video the white balls bounce on the platforms underneath at different frequencies. This program transformer retimes the bounces (i.e. collisions) to match the musical beat of a song using the `retimeToBeatsObjTransformer` function (Figure 4d). In the output video all the balls high the platforms at the same time.

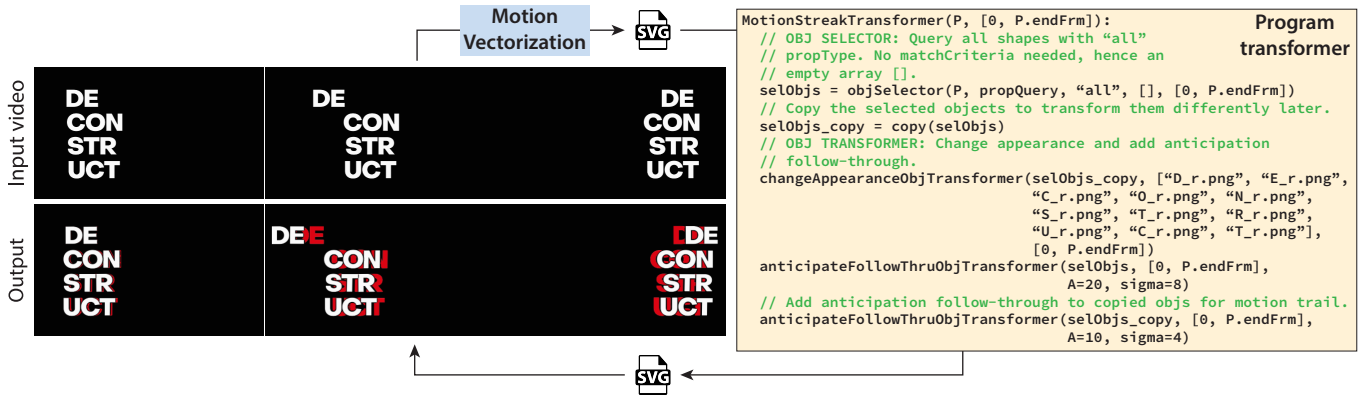


Fig. 6. Adding anticipation/follow-through and motion streak effect. This input video contains white text characters moving over a black background. The program transformer first copies the text objects. It then adds anticipation and follow-through to the original text using the `anticipateFollowThruObjTransformer` (Figure 4e) with a relatively wide $\sigma = 8$. To create the motion streaking effect it it recolors the copied text red using the `changeAppearanceObjTransformer`, and finally adds anticipation and follow-through to the copy using a narrower $\sigma = 4$.

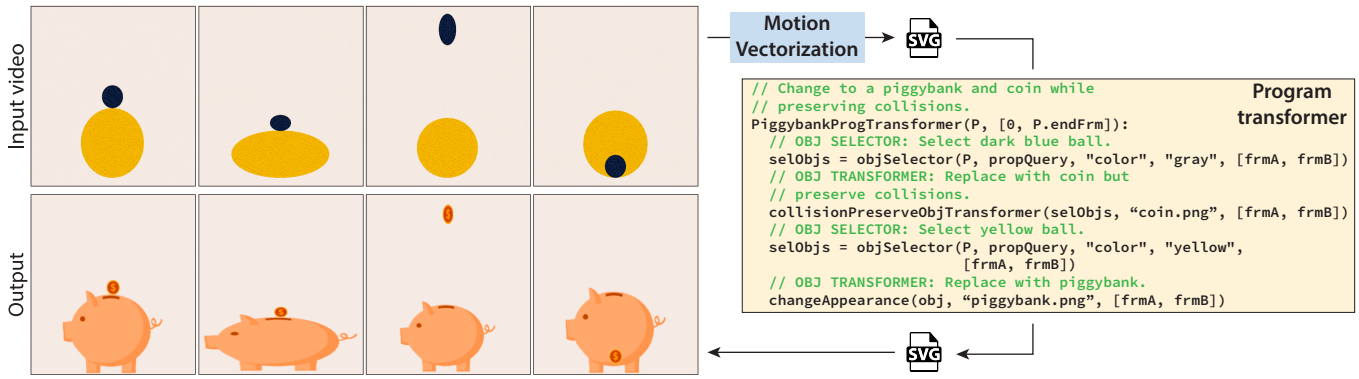


Fig. 7. Changing appearance while preserving collisions. This input video contains two balls that interact with one another with the dark blue ball bouncing around outside and inside the yellow ball. The program transformer changes the blue ball into a coin that is smaller than the blue ball. It then uses the `collisionPreserveObjTransformer` to adjust the motion of the smaller coin so that the collision points are maintained with the yellow ball. Finally it changes the appearance of the yellow ball to a piggy bank with the body of the bank the same size as the yellow ball.

Supplemental Material B:
Program Transformation API

SIGGRAPH Asia 2023 Technical Paper Submission #207

Motion Program Transformers

In general a *motion program transformer* takes a SVG motion program **`P`** and some additional **`args`** as input and transforms it in place into a new SVG motion program **`P'`** . The basic structure of a program transformer is an alternating sequence of *objSelector* blocks followed by *objTransformer* blocks.

```
MPTransformer(P, args, [frmA, frmB]):
    // OBJ SELECTOR block
    // Select objects in P via queries using criteria specified in args.
    ...

    // OBJ TRANSFORMER block
    // Apply an object operator to selected objects.
    ...
```

Object Selectors

An **`objSelector`** is a function which selects a set of objects in the motion program based on some **`criteria`**. It makes use of the **`propQuery`** and **`eventQuery`** methods in the API to determine which objects to select. The function has the following form:

```
// Returns a list of object data which match some criteria.
function objSelector(P, queryFn, queryType, criteria, [frmA, frmB]):
    selObjs = {}
    selObjsInfo = {}
    for each obj in selObjs:
        x = queryFn(obj, queryType, [frmA, frmB])
        if x matches criteria:
            selObjs.insert(obj)
            selObjsInfo.insert(x)

    return selObjs, selObjsInfo
```

Here, the **queryFn** is either a **propQuery** or **eventQuery**, and we select objects whose **queryType** (e.g., **"color"** for **propQuery** or **"collisionFrames"** for **eventQuery**) matches the specified **criteria** (e.g., **"red"** for **"color"** or **isNonEmpty** for **"collisionFrames"**) within the frame range **[frmA, frmB]**. Using an object selector, we can retrieve objects and the relevant query information based on specific properties or events.

Object Transformers

Once we have selected a subset of objects in **P** with an object selector, we apply modification to those objects using an *ObjTransformer* block. We can define specific object transformers for individual modifications using the operators in our API (pink keywords).

Retiming

Linear time stretch/shrink

```
// Linear time stretch/shrink by factor k in src frame range [frmA, frmB]
function linearRetimeObjTransformer(selObjs, k, [frmA, frmB]):
    for each obj in selObjs:
        sourceDur = frmB - frmA + 1
        targetDur = k * sourceDur
        // Retime from source range [frmA, frmB] to target frame
        // range [frmA, frm + targetDur]. Identity easing function
        // f(t) = t ensure linear retiming.
        retime(obj, [frmA, frmB], [frmA, frmA + targetDur], f(t)=t)
```

Slow in/out easing

```
// Slow in/out using a given easingFn
function slowInOutObjTransformer(selObjs, easingFn, [frmA, frmB])
   for each obj in selObjs:
      // A possible easing function is f(t)=t^4 for a slow in effect, and
      // f(t)=1-(1-t)^4 for a slow out effect.
      // We can use any easing fn as long as f(0)=0 and f(1)=1.
      retime(obj, [args.frmA, args.frmB], [args.frmA, args.frmB], easingFn)
```

Animating on 2s, 3s, Ns

```
// Animating on 2s, 3s, and Ns.
function animOnNsObjTransformer(selObjs, N, [frmA, frmB]):
  for each obj in selObjs:
    dur = frmB - frmA + 1
    numSteps = dur / N
    stepWidth = 1 / numSteps
    // E.g., if we have 6 frames and animating on 2s, then
    // numSteps = 3, stepWidth = 0.33,
```

```
    // stepFn is then the function f(t)=0 if 0<=t<0.33,
    // f(t)=0.5 if 0.33<=t<0.66, f(t)=1 if 0.66<=t<=1.0
    stepFn = step(dur, numSteps, stepWidth)
    retime(obj, [frmA, frmB], [frmA, frmB], stepFn)
```

Removing held frames for each object

```
// Remove frames in which motion was held for each object.
function removeHeldFramesObjTransformer(selObjs, selObjsInfo, [frmA, frmB])
  for each (obj, heldFrmSegs) in selObjs, selObjsInfo:
    heldFrmSegs = eventQuery(obj, "heldFrames", [frmA, frmB])
    for each seg in heldFrmSegs:
       // Remove held frames first by remapping the held frames to 1 frame.
       heldDur = seg.endFrm - seg.startFrm + 1
       retime(obj, [seg.startFrm, seg.endFrm],
                   [seg.startFrm, seg.startFrm], f(t)=t)

       // Then linearly stretch to the original duration.
       retime(obj, [seg.startFrm, seg.endFrm],
                   [seg.startFrm, seg.startFrm + heldDur], f(t)=t)
```

Retiming object motion to music beats

```
// Retime to music beats (assume that the video has more segments than
// specified beats).
function retimeToBeatsObjTransformer(selObjs, music, eventType,
                                     [frmA, frmB]):
    // Get music beat points using libROSA in units of frames
    beatPts = getMusicBeatPts(music)

    for each obj in selObjs:
    // Form segments of video that will correspond to each beat segment
    // (segment between successive beat points) based on eventType. For
    // example the  eventType "motionCycleFrames" returns the peaks of an
    // autocorrelation on the motion and is used to sync cyclic motion
    // periods to the audio beat. If eventType = null default to using the
    // beatPts as the video segment points.
    if eventType == null:
       segPts = beatPts
    else:
       segPts = eventQuery(obj, eventType, [frmA, frmB])

    for index i in segPts:
```

```
      // beatPts is in units of frames and includes a beat point at 0.
      retime(obj, [segPts[i], segPts[i + 1]],
                  [beatPts[i], beatPts[i + 1]], f(t)=t^4)
```

Spatial Motion Adjustment

Motion texture

```
// Add local “textural” motion to object
function motionTexObjTransformer(selObjs, xFormFn, xFormFnArgs,
                                 [frmA,frmB]):
  for each obj in selObjs:
    // E.g., if we want to apply a small up and down oscillation
    // we can set xFormFn to f(t, [y, amp]) = y[t] + amp * sin(t)
    adjLocalMotion(obj, xFormFn(xFormFnArgs), [frmA, frmB])
```

Anticipation/follow-through

```
// Add anticipation/follow through via Cartoon Animation Filter
function anticipateFollowThruObjTransformer(selObjs, [frmA, frmB], A,
sigma):
  for each obj in selObjs:
    // define the cartoon animation filter based on the work by Wang et al.
    function cartoonAnimationFilter(t, obj, [frmA, frmB], A, sigma):
        // copy segment of xForms to be set up for convolution later
        tmpXForms = copy(obj.xForms[frmA, frmB])
        // pad the tmpXForms for the Laplacian of Gaussian (LoG)
        pad(tmpXForms, 0.5 * sigma)
        // convolve(arr, signal) performs a 1d convolution.
        // -LoG is the inverse of the Laplacian of Gaussian function
        newXForms = A * convolve(tmpXForms, -LoG(sigma))
        return newXForms[t]

    adjGlobalMotion(obj, cartoonAnimationFilter, [frmA, frmB])
```

Appearance Adjustments

Basic change of object appearance

```
// Change object appearance, where newAppearances is an array of filenames
// for each selected object.
```

```
function changeAppearanceObjTransformer(selObjs, newAppearances, [frmA,
frmB])
  for each (obj, newAppearance) in (selObjs, newAppearances):
    changeAppearance(obj, newAppearance, [frmA, frmB])
```

Collision preserving appearance change

```
// Collision preserving appearance change.
function collisionPreserveObjTransformer(selObjs, newAppearances,
                                         [frmA, frmB]):
  for each obj in selObjs:
    collisions = eventQuery(obj, "collisionFrames", [frmA, frmB])
    // We compute a local transformation that will adjust the new object
    // appearance to the collision point.
    for collision in collisions:
      // findNearestPt(newAppearance, point) returns the closest point on
      // the boundary of newAppearance to a specified point.
      nearestPt = findNearestPt(newAppearances[obj], collision.thisObjPt)
      // moveFromAtoB(t, A, B) produce a local transformation that moves
      // an object from a point in A to a point in B at time t.
      adjLocalMotion(obj, moveFromAtoB(t, thisNearestPt, thisObjPt),
                     [frmA, frmB])
      // Set new object appearance.
      changeAppearance(obj, newAppearance, [frmA, frmB])
```

Highlight Results: Motion Program Transformers

sydney: We can accomplish tasks like text translation using our program transformations. In this example, we use appearance changes to translate the text into Chinese. We also update the text to reflect a new travel destination (Spain) and airport (Beijing). Retiming the text at the bottom draws attention to that area in the latter portion of the video.

```
MPTransformer(P, args):
    // Change the background.
```

```
    changeAppearance(P, args.background)

    // OBJECT SELECTOR: Select all objects in the top third of the video.
    // Sort them by y-position at the very end. We will only change
    // two objects.
    function in_top_third(vals):
        if y > m_frame_height / 3:
            return False
        return True
    topObjs = objSelector(
        P, propQuery, "positionY", in_top_third, [0, P.endFrm])
    function yComparator(obj):
        return propQuery(obj, "positionY", [P.endFrm - 1, P.endFrm])[0]
    bottomObjs = sorted(bottomObjs, key=yComparator)
    changeObjs = [topObjs[0], topObjs[-1]]

    // OBJECT TRANSFORMER: Change to new language and shift them so that
    // they end up in the bounds of the screen.
    changeAppearanceObjTransformer(changeObjs, arg.topText, [0, P.endFrm])
    function xShift(t, [x, amt]):
        return x[t] + amt
    adjGlobalMotion(
        changeObjs[0], xShift,
        [changeObjs[0].startFrm, changeObjs[0].endFrm], args.shiftDiscover)
    adjGlobalMotion(
        changeObjs[1], xShift,
        [changeObjs[1].startFrm, changeObjs[1].endFrm], args.shiftSydney)

    // Remove all the other objects.
    removeObjs(topObjs[1:-1])

    // OBJECT SELECTOR: Select all objects in the bottom third of
    // the video.
    function in_bottom_third(y):
        if y < 2 * P.height / 3:
            return False
        return True
    bottomObjs = objSelector(
        P, "positionY", in_bottom_third, [0, P.endFrm])

    // Sort all objects by x position at the very end. We will change one
    // object to the first translated phrase and another object to the
    // second translated phase.
```

```
    function xComparator(obj):
        return propQuery(obj, "positionX", [P.endFrm - 1, P.endFrm])[0]
    bottomObjs = sorted(bottomObjs, key=xComparator)
    copyObjs = copy(bottomObjs[0])
    changeObjs = [bottomObjs[0], copyObjs[0]]

    // OBJECT TRANSFORMER: Change to new language and shift them inwards.
    changeAppearanceObjTransformer(
        changeObjs, args.bottomText, [0, P.endFrm])
    function setX(t, [x, amt]):
        return amt
    adjLocalMotion(
        changeObjs[0], setX,
        [changeObjs[0].startFrm, changeObjs[0].endFrm],
        args.bottomTextShift)
    adjLocalMotion(
        changeObjs[1], setX,
        [changeObjs[1].startFrm, changeObjs[1].endFrm],
        -args.bottomTextShift)

    // Delay the second phrase by a little bit.
    linearRetimeObjTransformer(changeObjs[1], 5, [0, 1])

    // Remove all the other objects.
    removeObjs(bottomObjs[1:])
```

lucy: The following program transformer translates a social media ad into French. Since we only use one word “enchanté” in French for the English phrase “nice to meet you,” we adjust the “enchanté” to float up to the middle of the video. Using an **`adjGlobalMotion`** object transformer on the two circles which are held down briefly by the original English text, we can adjust the circles so that they still get held down by the word “enchanté.”

```
MPTransformer(P, args):
    // Change to french text.
    new_appearances = [
```

```
        'data/misc30/french/enchante.png',
        'data/misc30/french/jesuis.png',
        'data/misc30/french/lucy.png',
    ]
    wordObjs = objSelector(P, propQuery, "color", "black", [0, P.endFrm])
    changeAppearanceObjTransformer(wordObjs[-3:], new_appearances)
    removeObjs(wordObjs[:-3])

    // Positional adjustments.
    function setX(t, [x, amt]):
        return amt / 2
    adjGlobalMotion(
        wordObjs[-3], setX, [obj.startFrm, obj.endFrm], amt=P.width / 2)

    // Get the two objects in contact with it.
    collisions = eventQuery(wordObjs[-3], "collisionFrames", [100, 200])
    function containsID(id, id_list):
        return id in id_list
    id_list = {}
    for collision in collisions:
        id_list.add([obj.id for obj in collision])
    contactObjs = objSelector(
        P, propQuery, "id", containsID, [0, P.endFrm], id_list=id_list)
    function adjust_by(t, [y, amt]):
        return amt
    adjustLocalObjTransformer(
        [wordObjs[-3], contactObjs[0], contactObjs[1]],
        adjust_by, [wordObjs[-3].startFrm, P.endFrm], amt=-75)
    // Readjust the two contact objects motions so that they float from
    // their new starting position to their original ending position.
    function global_interp(t, [y, start, end, length]):
        if t < length:
            return end * t / length + start * (1 - t / length)
        else:
            return end
    for obj in contactObjs:
        y_pos_start = propQuery(
            obj, "positionY", [0, wordObjs[-3].endFrm])[-1]
        y_pos_end = minimum(
            propQuery(obj, "positionY", [obj.startFrm, obj.endFrm]))
        adjGlobalMotion(
            obj, global_interp, [wordObj[-3].endFrm, obj.endFrm],
            start=y_pos_start, end=y_pos_end, length=P.frame_rate)
```

confetti: We change the appearance of all the objects and background in the confetti video to create a spring motion graphic. In order to make the falling petals more realistic, we use a **`linearRetimeObjTransformer`** to slow them down and apply a **`petal_fall()`** motion texture function that makes them sway back and forth. To match the pixel art look of the images, we also animate the falling objects on 3s. Finally, we can adjust the depth of the petals to surround the text rather than fall over it.

```
MPTransformer(P, args):
    // Change background image.
    setBgImg(P, args.background)

    // OBJECT SELECTOR: Select all confetti.
    confettiObjs = objSelector(P, propQuery, "color", "not black", [0,
P.endFrm])
    // OBJECT TRANSFORMER: Remove held frames.
    removeHeldFramesObjTransformer(confettiObjs, [0, P.endFrm])

    // Rotate and resize all confetti by random amounts and fall back and
forth.
    function rotate(t, [theta]):
        deg = 120 * (randFloat() - 0.5)
        return deg
    adjustGlobalObjTransformer(confettiObjs, rotate [0, P.endFrm])

    function random_size(t, [sx, sy]):
        amt = randFloat() * 0.5 + 0.5
        return [amt, amt]
    adjustGlobalObjTransformer(confettiObjs, random_size, [0, P.endFrm])

    function petal_fall(t, [x, offset]):
        return offset * sin(t)
    motionTexObjTransformer(confettiObjs, petal_fall, [0, P.endFrm],
offset=60)

    // OBJECT SELECTOR: Select letter 'C'.
    cObjs = objSelector(P, propQuery, "id", "shape_09", [0, P.endFrm])
    // OBJECT TRANSFORMER: Change to 'SPRING' text.
```

```
    changeAppearanceObjTransformer(cObjs, args.season)

    // Remove remaining letters.
    letterObjs = obj_selector(P, propQuery, "color", "black", [0,
P.endFrm])
    removeObjs(letterObjs)

    // OBJECT TRANSFORMER: Make all confetti slower and animate on 3s.
    linearRetimeObjTransformer(confettiObjs, 6, [0, P.endFrm])
    animOnNsObjTransformer(confettiObjs, 3, [0, P.endFrm])

    // Replace all objects with flowers.
    // The random_choice(arr, size) function selects a random subset of
size with replacement
    // from a list of choices.
    new_appearances = random_choice(args.confetti_imgs, len(confettiObjs))
    changeAppearanceObjTransformer(confettiObjs, new_appearances)

    // Randomly adjust depth layering.
    function random_raise_z(t, [z]):
        return random_choice([-10, 10])
    adjustGlobalObjTransformer(confettiObjs, random_raise_z, [0, P.endFrm])
```


giftbox1: We can achieve complex variations of an input SVG by composing sequences of object selector blocks and object transformer blocks. For example, the following program transformer takes a giftbox video and outputs a box with bubbles. We can write out object transformer blocks using the previously defined object transformer functions, or in expanded for loops (see giftBox.html in the supplemental material).

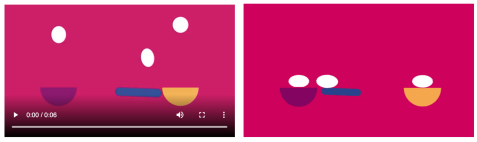
```
BubbleProgTransformer(P, args):
    // OBJECT SELECTOR: Query for the red ball.
    selObjs = objSelector(P, propQuery, "color", "red", [frmA, frmB])

    // OBJECT TRANSFORMER: Change the appearance to bubble.
    newAppearances = ["bubble.png", ..., "bubble.png"]
    changeAppearanceObjTransformer(selObjs, newAppearances, [frmA, frmB])
```

```
// Create lots of bubbles.
newObjs = {}
for each obj in selObjs:
    repeat N times:
      // copy() duplicates an object in the SVG motion program.
      newObjs.insert(copy(obj))

// OBJECT SELECTOR: Find all bubbles.
selObjs = objSelector(P, propQuery, "appearance", "bubble.png",
                      [frmA, frmB])

// Modify motions of new bubbles.
// OBJECT TRANSFORMER: Make bubbles wobble.
function wobbleFn(x_pos, [t, amp, freq]):
    return x_pos[t] + amp * sin(freq * t)
motionTexObjTransformer(selObjs, wobbleFn, args.wobbleFnArgs,
                            [frmA, frmB])

// OBJECT TRANSFORMER: randomly scale bubbles to different sizes.
for each obj in newObjs:
    // randomly scale objs to different sizes
    function scaleFn(t, [s, scaleFactor]):
        return s[t] * randomFloat()
    adjLocalMotion(obj, scaleFn, [obj.startFrm, obj.endFrm])

    // Apply motion adjustment to vary bubble paths.
    function translateXFn(t, x):
        return x[t] + t^4 + randomFloat()
    adjGlobalMotion(obj, translateXFn, [obj.startFrm, obj.endFrm])

    function translateYFn(t, y):
        return y[t] + randomFloat()
    adjGlobalMotion(obj, translateYFn, [obj.startFrm, obj.endFrm])

// OBJECT TRANSFORMER: Retime  the entire timeline randomly so the
// bubbles move at different speeds.
linearRetimeObjTransformer(selObjs, randomFloat(0.5, 1), [frmA, frmB])

// OBJECT TRANSFORMER: Release bubbles at different times.
// freezeFrame(obj, frm, dur) holds a frame for a duration.
for each obj in newObjs:
    duration = randomInt(24, 96)
```

```
        freezeFrame(obj, 0, duration)
```

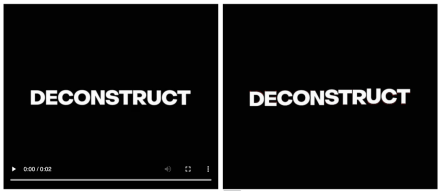
face: The following program transformer modifies elements and motions of an input SVG to create an output SVG motion program which renders variations of a lunar new year video. Here, we make use of the **`motionTexObjTransformer`** block to vary the motions in addition to having appearance changes (see shapes38.html in the supplemental material).

```
NewYearsProgTransformer(P, args):
    // Set background color.
    setAppearance("bg", args.background)

    // OBJECT SELECTOR: Query for the red semicircle.
    selObjs = objSelector(P, propQuery, "color", "red", [frmA, frmB])
    // OBJECT TRANSFORMER: Change the appearance to the year.
    changeAppearanceObjTransformer(selObjs, args.year, [frmA, frmB])
    // Repeat obj selection and obj transformation for banner and animal.
    selObjs = objSelector(P, propQuery, "color", "yellow", [frmA, frmB])
    changeAppearanceObjTransformer(selObjs, args.banner, [frmA, frmB])
    selObjs = objSelector(P, propQuery, "color", "white", [frmA, frmB])
    changeAppearanceObjTransformer(selObjs, args.zodiac, [frmA, frmB])

    // OBJECT SELECTOR: Query for the gray curve.
    selObjs = objSelector(P, propQuery, "color", "gray", [frmA, frmB])
    // OBJECT TRANSFORMER: Change appearance to characters.
    changeAppearanceObjTransformer(selObjs, args.characters, [frmA, frmB])
    // OBJECT TRANSFORMER: Apply an oscillating scale.
    function pulse(t, [sx, sy]):
        return [sx + 0.5 * np.sin(t / 10), sy + 0.5 * np.sin(t / 10)]
    motionTexObjTransformer(selObjs, pulse, args.pulseArgs, [frmA,frmB])

    // OBJECT SELECTOR: Query for the blue circle.
    selObjs = objSelector(P, propQuery, "color", "blue", [frmA, frmB])
    // OBJECT TRANSFORMER: Remove the object.
    removeObj(selObjs, [frmA, frmB])
```



levers: The following program transformer retimes the shapes bouncing and rotating at different rates in the input SVG to all match the same music beats. Here, we make use of the **`retimeToBeatsObjTransformer`** to match each individual shape's collision cycles with music beat intervals (see multipleBouncingBalls2.html in the supplemental material).

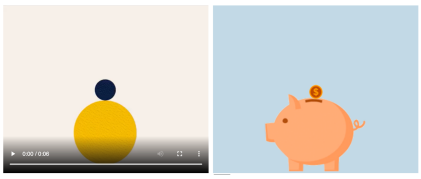
```
BouncingBallsRetimedToMusicProgTransformer(P, args):
    // OBJECT SELECTOR: Query for all shapes with the "all" propType.
    // No matchCriteria is needed, hence an empty array [].
    selObjs = objSelector(P, propQuery, "all", [], [frmA, frmB])

    // OBJECT TRANSFORMER: Retime periods of collisions in the video
    // to match with the music beats by specifying the event type
    // as "collisionFrames".
    retimeToBeatsObjTransformer(selObjs, args.musicFile,
                                "collisionFrames", [frmA, frmB])
```



deconstruct: The following program transformer creates a motion trail effect by duplicating the letters in the input SVG and applying the cartoon animation filter with different parameters. Here, we make use of the **`anticipateFollowThruObjTransformer`** to add different degrees of anticipation and follow through to the two sets of letters. The transformed animations are then overlaid on top of each other (see text2.html in the supplemental material).

```
MotionTrailProgTransformer(P, args):
  // OBJECT SELECTOR: Query for all shapes with the "all" propType.
  // No matchCriteria is needed, hence an empty array [].
  selObjs = objSelector(P, propQuery, "all", [], [frmA, frmB])

  // Copy the selected objects to transform them differently.
  selObjs_copy = copy(selObjs)

  // OBJECT TRANSFORMER: Apply appearance change and add anticipation
  // and follow-through.
  // First add anticipation and follow-through to selObjs with
```

```
  // amplitude=20 and sigma=8.
  anticipateFollowThruObjTransformer(selObjs, [frmA, frmB], 20, 8)

  // Then change the appearances of the copied selObjs to red.
  changeAppearanceObjTransformer(selObjs_copy, args.redLetters,
                                 [frmA, frmB])

  // Finally, add anticipation and follow-through to the copied selObjs
  // with amplitude=10 and sigma=4 for an overall motion trail effect.
  anticipateFollowThruObjTransformer(selObjs_copy, [frmA, frmB], 10, 4)
```



ball1: The following program transformer modifies the appearances of elements of an input SVG to create an output SVG motion program of a coin dropping into a piggy bank. Here, we make use of the **`changeAppearanceObjTransformer`** block to change the appearance of the yellow ball to a piggy bank and the **`collisionPreserveObjTransformer`** block to replace the appearance of the dark blue ball in a collision-preserving manner (see ball3.html in the supplemental material).

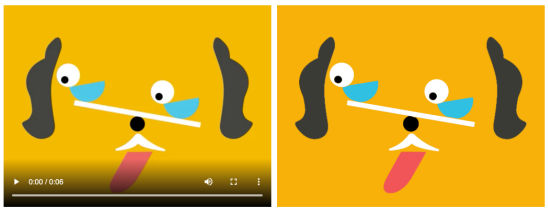
```
PiggybankProgTransformer(P, args):
  // OBJECT SELECTOR: Select dark blue ball.
  selObjs = objSelector(P, propQuery, "color", "dark-blue", [frmA, frmB])
  // OBJECT TRANSFORMER: Replace appearance and preserve collisions.
  collisionPreserveObjTransformer(selObjs, "coin.png", [frmA, frmB])

  // OBJECT SELECTOR: Select yellow ball.
  selObjs = objSelector(P, propQuery, "color", "yellow", [frmA, frmB])
  // OBJECT TRANSFORMER: Replace its appearance with piggybank.
  changeAppearanceObjTransformer(obj, "piggybank.png", [frmA, frmB])
```

morningShow: The following program transformer modifies the appearances of elements of an input SVG to create an output SVG motion program of a subway sign. Here, we make use of the **`objSelector`** function and the **`changeAppearanceObjTransformer`** block to select desired

shapes and change their appearances to appropriate letters and icons (see ball8.html in the supplemental material).

```
SubwaySignProgTransformer(P, args):
  // Set the background image.
  changeAppearance("bg", args.background)

  // Keep track of unchanged objects because we will replace them later.
  unchangedObjs = objSelector(P, propQuery, "position", [[0, 1], [0, 2/9]],
                              [frmA, frmB])

  // Replace the second row with Canal St stations.
  selObjs = objSelector(P, propQuery, "position", [[0, 1], [1/9, 2/9]],
                        [frmA, frmB])
  changeAppearanceObjTransformer(selObjs, args.canalStImgs, [frmA, frmB])
  unchangedObjs.remove(selObjs)

  // Replace the fourth row with NJ transit and 123 trains.
  selObjs = objSelector(P, propQuery, "position", [[0, 1], [1/3 , 4/9]],
                        [frmA, frmB])
  changeAppearanceObjTransformer(selObjs[:3], args.oneTwoThree,
                                 [frmA, frmB])
  changeAppearanceObjTransformer(selObjs[3:], args.njTransit,
                                 [frmA, frmB])
  unchangedObjs.remove(selObjs)

  // Replace the eighth row with Penn Station and ACE trains.
  selObjs = objSelector(P, propQuery, "position", [[0, 1], [7/9, 8/9]],
                        [frmA, frmB])
  changeAppearanceObjTransformer(selObjs[:5], args.ace, [frmA, frmB])
  changeAppearanceObjTransformer(selObjs[5:], args.pennStation,
                                 [frmA, frmB])
  unchangedObjs.remove(selObjs)

  // Replace the black circle with a pink circle.
  blackCircle = propQuery(obj, "color", "black", [frmA, frmB])
  changeAppearanceObjTransformer(blackCircle, "pinkCircle.png",
                                 [frmA, frmB])
  unchangedObjs.remove(blackCircle)

  // Select some random objects to replace with green or yellow taxis.
  // randInt(a, b) chooses a random integer in [a, b].
  numTaxis = randInt(0, len(unchangedObjs))
  taxiImgs = chooseRandom(numTaxis, args.taxiImgs, replace=True)
```

```
  taxis = chooseRandom(numTaxis, unchangedObjs, replace=False)
  changeAppearanceObjTransformer(taxis, taxiImgs, [frmA, frmB])
  unchangedObjs.remove(taxis)

  // Select a random object to replace with pedestrian signs.
  signs = chooseRandom(1, unchangedObjs, replace=True)
  changeAppearanceObjTransformer(signs, args.walkImg, [0, frmB/4])
  changeAppearanceObjTransformer(signs, args.stopImg, [frmB/4,  frmB/2])
  changeAppearanceObjTransformer(signs, args.walkImg, [ frmB/2, 3 *
frmB/4])
  changeAppearanceObjTransformer(signs, args.stopImg, [3 *  frmB/4, frmB])

  // Replace the remaining objects with random colored circles.
  pplCircles = chooseRandom(len(unchangedObjs), args.pplCircles,
replace=True)
  changeAppearanceObjTransformer(unchangedObjs, pplCircles, [frmA,  frmB])
```



dog: The following program transformer first changes the appearance of elements in the input SVG to resemble different animals and then retimes the cyclic seesaw motions to match the music beats. Here, we make use of the **`changeAppearanceObjTransformer`** to change the appearances of shapes and **`retimeToBeatsOjTransformer`** to match each shape's cyclic motions with music beat intervals (see dogSeesaw3.html in the supplemental material).

```
AnimalSeesawRetimedToBeatsProgTransformer(P, args):
  // OBJECT SELECTOR: Query for all shapes with the "all" propType.
  // No matchCriteria is needed, hence an empty array [].
  selObjs = objSelector(P, propQuery, "all", [], [frmA, frmB])

  // Make two copies of the objects to change their appearances later.
  catObjs = copy(selObjs)
  eleObjs = copy(selObjs)

  // OBJECT SELECTOR: Query for static shapes in catObjs
  // and elephantObjs by checking if a shape has 0 velocity.
  catStaticObjs = objSelector(P, propQuery, "velocity", [0], [frmA, frmB])
```

```
eleStaticObjs = objSelector(P, propQuery, "velocity", [0], [frmA, frmB])

// OBJECT TRANSFORMER: Change the appearances of static objects
// to make them look like a cat and an elephant respectively.
changeAppearanceObjTransformer(catStaticObjs, ["catEarL.png",
                            "catEarR.png", "catMouthWhiskers.png"]
                            [frmA, frmB])
changeAppearanceObjTransformer(eleStaticObjs, ["eleEarL.png",
                            "eleEarR.png", "eleTrunk.png"]
                            [frmA, frmB])

// Retime the cat and elephant objects to go after the original objects
for each obj in catObjs:
  retime(obj, [0, frmB], [frmB, frmB * 2], f(t)=t)

for each obj in eleObjs:
  retime(obj, [0, frmB], [frmB * 2, frmB * 3], f(t)=t)

// OBJECT TRANSFORMER: Retime cyclic motions in the entire video to
// match with the music beats by specifying the event type as
// "motionCycleFrames".
retimeToBeatsObjTransformer(selObjs, "BeatOfTheIsland.wav",
                            "motionCycleFrames", [frmA, frmB *3])
```