From functional animation to sprite-based display

Conal Elliott

Microsoft Research
`http://research.microsoft.com/~conal`

Abstract. Functional animation encourages a highly modular programming style, by supplying a set of arbitrarily composable functions for building up animations. In contrast, libraries for sprite-based display impose rigid structure, in order to allow acceleration by hardware and low level software. This paper presents a method to bridge the gap between functional specification and stateful, sprite-based presentation of animation. The method's correctness is proved informally by derivation from a simple non-effective specification, exploiting algebraic properties of the animation data types that are made explicit in the functional approach. We have implemented this method in the *Fran* system, which is freely available.

1 Introduction

The functional approach to animation offers considerable flexibility and modularity [1, 7]. Animations are first-class values—elements of a data type consisting of a set of constants and combining operators. The data type allows great flexibility in composing these basic building blocks into either directly useful or attractive animations, or new building blocks, parameterized as desired. Moreover, animation is a polymorphic notion, applying to 2D images, 3D geometry, and consituent types like colors, points, vectors, numbers, booleans, etc. Consequently, there is not just one animation type, but a collection of types and type constructors. In a well-designed set of data types, the type system imposes just enough discipline to rule out nonsensical compositions (such as rotating by the angle "purple"), without inhibiting the author's creativity. In this way, the data types are designed to serve the needs of the author (and readers) of an animation.

Lower level graphics presentation libraries are designed not for convenience of a program's author or readers, but rather for efficient execution on anticipated hardware. Programs written directly on top of these libraries must adapt to relatively inflexible representations and tend to be relatively non-modular.

To illustrate the convenience of functional animation, consider the animation in Figure 1.[1] The function `repSpinner` repeatedly transforms an animation `im`,

[1] `ftp://ftp.research.microsoft.com/pub/tr/tr-98-28/animations.htm` contains running versions of this example and a few others. Those versions were recorded at 20 fps (frames per second), but run in Fran at 60 fps.

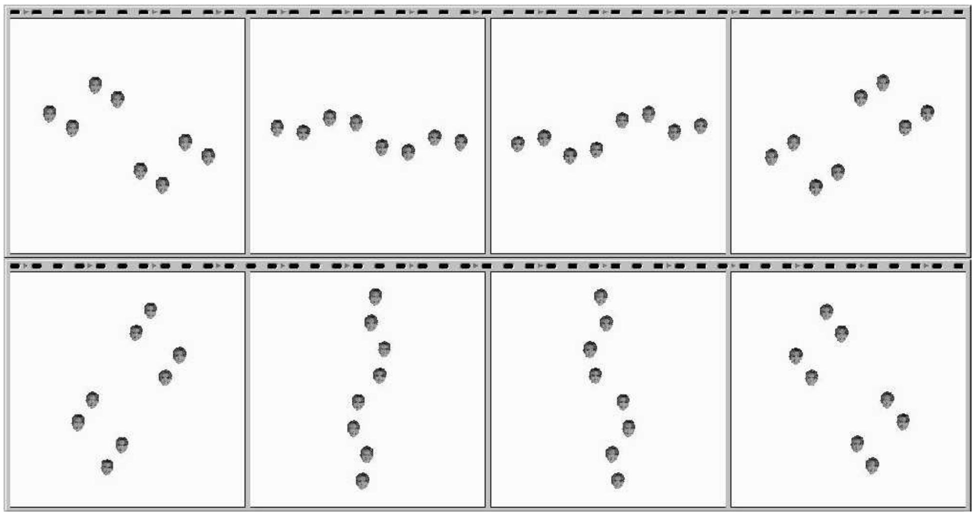
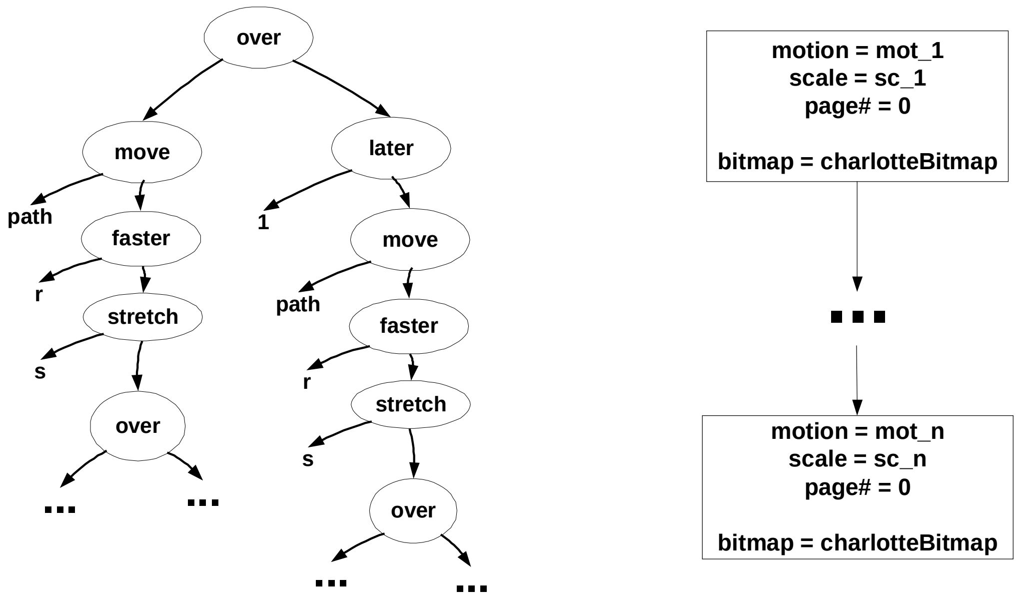


Fig. 1. `repSpinner (-1) 0.5 charlotte 3`

stretching `im` by `r`, speeding it up by `s`, and putting two copies into circular orbit. (`iterate f x` produces the infinite list `[x, f x, f (f x), ...]`, and `l!!n` extracts the `n`-th element of the list `l`. Their use together here results in applying `spinner` to `im`, `n` times.)

```
repSpinner :: RealB -> RealB -> ImageB -> Int -> ImageB
repSpinner r s im n = iterate spinner im !! n
 where
   spinner im = orbit `over` later 1 orbit
    where
      orbit = move path (faster r (stretch s im))
      path  = vector2Polar 0.5 (pi*time)
```

In this example, time- and space-transforming functions (`faster` and `stretch`) are applied to `over`lays of animations that contain more time- and space-transformed animations (when $n > 1$), as shown in Figure 2(a). In contrast, sprite-based subsystems impose rigid structuring on an animation. It must be a sequence of implicitly overlaid "sprites", where each sprite is the result of applying possibly time-varying motion and scaling to a "flip-book" of bitmaps, as in Figure 2(b). (In our example, the flip-book has only one page.) It is tedious and error-prone for a programmer or artist to work with such restrictions, when the animated objects of interest have natural hierarchical structure, as in `repSpinner`. The required sprite motion and scaling vectors (`mot_`i and `sc_`i) are quite complex, because each one is affected by a number of space- and time-transforms above them in the original hierarchy.



(a) Image hierarchy

(b) Sprite list

Fig. 2. Image hierarchies vs. sprite lists

This paper presents an algorithm to convert functional animations into a "sprite normal form" amenable to a fast sprite-based presentation library. As such, it bridges the gap between the high level flexibility desired for specification of animations and the low level regularity imposed by the presentation library. This algorithm is used in the implementation of *Fran* ("Functional Reactive Animation") [7, 3, 4], which is written in Haskell [9, 8] and runs in conjunction with a fast "temporal sprite engine," implemented in C++ for use with Fran and running in a separate thread. The sprite engine, constructed for use with Fran and described in this paper, manages a list of sprites. While some sprites are rendered on the fly, the sprite engine also supports fast flip-book animation. Typically, Fran updates information maintained by the sprite engine ten times per second, while the sprite engine interpolates motion, scaling, and indexing, and updates the screen, all at sixty times per second.

In order to transform flexibly specified animations into the form required by the sprite engine, we exploit several algebraic properties. In addition to time- and space-transformation, Fran's animation algebra supports cropping, conditionals and reactivity, each requiring special care for conversion to sprite normal form.

Fran grew out of an earlier design called ActiveVRML [2], which was a ML-like language extended with reactive behaviors and multimedia. In contrast, Fran

```
emptyImage  :: ImageB                 -- transparent everywhere
solidImage  :: ImageB                 -- solid color image
flipImage   :: HFlipBook -> RealB -> ImageB -- flipbook-based
renderImage :: Renderer -> ImageB     -- text, 2D & 3D geometry
soundImage  :: SoundB -> ImageB            -- embedded sound
over        :: ImageB -> ImageB -> ImageB  -- overlay
withColor   :: ColorB -> ImageB -> ImageB  -- colored image
crop        :: RectB  -> ImageB -> ImageB  -- cropped image

(*%)        :: Transform2B -> ImageB -> ImageB  -- apply 2D transform
untilB      :: ImageB -> Event ImageB -> ImageB     -- reactive
condB       :: BoolB -> ImageB -> ImageB -> ImageB  -- conditional
timeTransform :: ImageB -> TimeB -> ImageB
```

Fig. 3. Abstract interface to the `ImageB` type

is a Haskell library. This "embedding" approach, discussed in detail elsewhere [3], has allowed considerable leverage from the design and implementation of the host language Haskell. Fran is freely available, including the sprite engine, in full source code at

`http://research.microsoft.com/~conal/fran`

2 Animation data types

Fran's animation data types have been described elsewhere (e.g., [7], and for reference [11]). The animation-level types other than animated images, sounds, and 3D geometry come from applying the `Behavior` type constructor to these static types, and lifting operations on the static types to operations on behaviors. Behaviors are functions of continuous time. As a convention, the behavior types have synonyms made by adding a "B" to the static type name, e.g., "`Transform2B`" is a synonym for "`Behavior Transform2`".

The type `ImageB` represents image animations that are spatially and temporally continuous and infinite. The primitives used in the construction of `ImageB` values are shown in Figure 3. (The type `Event a`, for an arbitrary type `a`, represents a stream of time-stamped values of type `a`, and is the basis of reactivity in behaviors.)

The `renderImage` function is used for all of the various kind of synthetic images, based on text, 2D geometry, 3D geometry, etc. The type `Renderer` maps time-varying cropping rectangle, color, scale factor, rotation angle, and time transform to a time-varying bitmap:

```
type Renderer = RectB -> Maybe ColorB -> RealB -> RealB
             -> TimeB -> SurfaceULB
```

```
  (crop rectB           $
   mbWithColor mbColorB $
   stretch scaleB       $
   turn    angleB       $
   (`timeTransform` tt) $
   RenderImage renderer )
==
   surfaceIm (renderer rectB mbColorB scaleB angleB tt)

-- Possibly apply a color
mbWithColor :: Maybe ColorB -> ImageB -> ImageB
mbWithColor Nothing  imB = imB
mbWithColor (Just c) imB = withColor c imB

-- The ImageB contained on a discrete image (for specification).
surfaceIm :: SurfaceULB -> ImageB
```

Fig. 4. Property of a "renderer"

```
-- Bitmap plus upper-left corner location
data SurfaceUL  = SurfaceUL HDDSurface Point2
type SurfaceULB = Behavior SurfaceUL
```

Displaying an animated image also plays the sounds it contains. Spatial transformation has audible effect; the horizontal component of translation becomes left/right balance, and the scale becomes volume adjustment. Cropping silences sounds outside of the crop region.

By definition, a *renderer* not only has the type above, but must also perform cropping, coloring, scaling, rotation, and time transformation, according to its given parameters, as expressed in Figure 4 (The "`$`" operator is an alternative notation for function application. Because it is right-associative and has low syntactic precedence, it is sometimes used to eliminate cascading parentheses. Also, an infix operator (here `` `timeTransform` ``) with one argument but missing the other denotes a function that takes the missing argument (here an `ImageB` value) and fills it in.)

Fran defines convenience functions `move`, `stretch`, and `turn` that make 2D transform behaviors and apply them via "`*%`".

In the early implementations of Fran, `ImageB` was simply defined to be `Behavior Image`, for a type `Image` of static images. This representation made for a very simple implementation, but it has a fundamental problem: the image structure of an `ImageB` value cannot be determined at the behavior level. It must first be sampled with some time t to extract a static image, whose structure can then be examined. To display an animation then, one must repeatedly sample it, and display the resulting static images. Consequently, the display computation

```
data ImageB
 = EmptyImage                           -- transparent everywhere
 | SolidImage                           -- solid color
 | FlipImage  HFlipBook RealB           -- page # behavior
 | RenderImage Renderer                 -- text, 2D & 3D geometry
 | SoundI     SoundB                    -- embedded sound
 | Over       ImageB   ImageB           -- overlay
 | TransformI Transform2B ImageB        -- 2D transformed
 | WithColorI ColorB ImageB             -- colored
 | CropI      RectB ImageB              -- cropped
 | CondI      BoolB ImageB ImageB       -- conditional
 | UntilI     ImageB (Event ImageB)     -- reactivity
 | TimeTransI ImageB  TimeB             -- time transformed
```

Fig. 5. Data type representing 2D image animations

cannot build and maintain data structures and system resources for parts of an animation. In particular, it cannot allocate one sprite for each `flipImage` and `renderImage`, set them moving, and then update the motion paths incrementally. That is, the implementation cannot take advantage of a sprite engine.

It appears then that the modularity imposed by lifting requires an underlying presentation library to work in "immediate mode" (draw this now), rather than "retained mode" (build a model and repeatedly edit and redisplay it), to borrow terms from 3D graphics programming. In designing Fran, however, we targeted the upcoming generation of graphics accelerator cards, which we believe to be increasingly oriented toward retained mode.

For reasons given above, the `ImageB` is not represented in terms of a static `Image` type, but rather as a recursively defined data type, as shown in Figure 5. The functions in Figure 3 are defined as simple optimizations of the `ImageB` constructors from Figure 5. For instance, transforming the empty or solid image has no effect:

```
xf *% EmptyImage = EmptyImage
xf *% SolidImage = SolidImage
xf *% im         = TransformI xf im
```

Although a programmer would not be likely to transform empty or solid images explicitly, such compositions arise at runtime due to modular programming style, as well as some of the reactivity optimizations discussed in [6]. The constructors `CondI`, `UntilI`, and `TimeTransI` are used to define the overloaded functions `condB`, `untilB`, and `timeTransform` that apply to sound, 3D geometry, and all behaviors, as well as to image animations. The "`*%`" operator is overloaded to apply to several types, as well.

3 A temporal sprite engine

The primary purpose of the sprite engine is to scale, move, and overlay a sequence of images, and to do so at a very high and regular rate. Because animations may involve arbitrarily complex behaviors, and because garbage collection and lazy evaluation cause unpredictable delays, we could not meet this goal if the window refresh were implemented in Haskell or even invoked by a Haskell program. For these reasons, we designed and implemented a sprite engine in C++ that does no memory allocation and runs in its own thread.

3.1 Sprites and sprite trees

The sprite engine maintains an ordered collection of sprites, represented via a C++ class hierarchy. The classes representing individual sprites are as follows.

- `FlipSprite` has a bitmap that is selected from a "flip book" object. Each flip book contains a pointer to a bitmap stored in video memory, and information describing a rectangular array of images contained somewhere within the bitmap. This array is taken to be a linear sequence of consecutive "pages". Flip books and the bitmap images to which they refer are immutable and may be shared among any number of flip sprites. The current page number, spatial transform, and cropping rectangle are all stored in the unshared sprites rather than the shared flip book.
- `RenderedSprite` has its own drawing surface and a method for replacing the surface with a new one. This sprite is used for all "rendered" images, such as text, 2D geometry, and camera-viewed 3D geometry.
- `SolidSprite` is a uniformly colored sprite, cropped but not spatially transformed.
- `SoundSprite` is an embedded sound, and is neither cropped nor transformed. It contains a pointer to (typically immutable) shareable sound data, plus its own unshared attributes for volume, frequency, and left/right balance.

A displayed animation could be represented by a list of individual sprites, in back-to-front order, so that upper (later) sprites are painted over lower (earlier) sprites. This representation is the simplest for display, but is awkward for editing. When an event occurs in an animation, an animated object may be introduced, removed, or replaced. Since the appearance of such an object may contain any number of sprites, the event may introduce, remove, or replace a whole contiguous subsequence of sprites. The sprite engine makes it easy and fast to edit the sprite list at a conceptual level by using a list of *sprite trees* rather than a list of individual sprites. The internal nodes of these trees correspond exactly to the points of structural mutability of the animation being displayed. (For Fran's use, the points of mutability are generated from the reactivity construct "`untilB`".) Correspondingly, there is a `SpriteTree` subclass `SpriteGroup` that contains a list of sprite trees and supports a method to replace the list, in its entirety, with another one.

For conditional animations, such as generated by Fran's "`condB`", there is a `SpriteTree` subclass `CondSpriteTree`, which contains an externally mutable boolean and two sprite tree lists.

The sprite engine expects its sprite trees to be updated less often that they are redisplayed. For example, Fran tries to sample behaviors at roughly ten times per second, but the sprite engine redisplays at 60 times per second. Between updates, the sprite engine performs linear interpolation of all sprite attributes, which are are actually represented as linear functions rather than constant values. For smoothness, updates must then be given for times in the future, so that a new linear path may be set from the current time and value to the given one. These linear behaviors form a compromise between arbitrary behaviors, which have unpredictable sampling requirements, and mere constant values, which fail to provide smooth motion. Other possibile compromises include quadratic and cubic curves, for which fast incremental sampling algorithms may be used.

The main loop of the sprite engine is then to traverse its sprite tree list recursively and in depth-first, back-to-front order. Each image sprite in turn is told to paint itself, and in doing so samples its attributes' linear functions and then does a very fast video memory "blit" (copy).

3.2 Interpretation of sprite trees

The goal of `ImageB` display is first to "spritify", i.e., convert an abstract `ImageB` value to an initial list of sprite trees, and then update the trees iteratively. Concurrently, the sprite engine traverses and displays the sprite trees. Although the sprite trees are really implemented in C++, for the purpose of exposition, we will express them here in as the Haskell type definitions in Figure 6.

In order to define "correct" conversion, we need an interpretation of sprite tree lists. We specify this interpretation by mapping sprite tree lists to `ImageB` values. Note that this mapping is hypothetical, serving to (a) specify what the sprite engine does, and (b) justify our implementation in Section 4 of the *reverse* mapping, i.e., from `ImageB` values to `[SpriteTree]`. The interpretation functions are given in Figure 7. The function `treesIm` is the main one, saying that a list of sprite trees represents an overlay of the images represented by the member trees, in reverse order. The first three clauses of the `treeIm` function interpret individual sprites. Note the rigidity of order of applied operations imposed by the sprite constructors, as contrasted with the flexibility afforded by the `ImageB` type (Figure 3). It is exactly because of this difference that, while interpretation of sprite tree lists is straightforward, generation is not.

The `UntilT` case says that a reactive sprite tree represents a reactive image animation. This animation is initially the one represented by the initial sprite tree list. When the event occurs, yielding a new sprite tree list, the animation switches to the one represented by these new trees. (The event `e ==> treesIm` occurs whenever `e` does. At each occurrence, the function `treesIm` is applied to the the event data from `e`.) The conditional case is similar.

```
data SpriteTree =
   SoundSprite RealB           -- volume adjust
               RealB           -- left/right pan
               RealB           -- pitch adjust
               Bool            -- whether to auto-repeat
               SoundBuffer
 | RenderedSprite Vector2B     -- translation vector
                  SurfaceULB
 | SolidSprite RectB           -- cropping region
               ColorB          -- solid color
 | FlipSprite RectB            -- cropping region
              Vector2B         -- translation vector
              RealB            -- scale factor
              HFlipBook
              RealB            -- page number
 | UntilT [SpriteTree] (Event [SpriteTree])
 | CondT   BoolB [SpriteTree] [SpriteTree]
```

Fig. 6. Sprite trees as Haskell types

4 From abstract animations to sprite trees

The interpretation of sprite tree lists as image animations given above helps to specify the reverse process, which we must implement in order to use the sprite engine to display animations: The `spritify` algorithm takes sprite trees to overlay, a cropping rectangle, optional color, a space- and a time-transformation, all applied to the given `ImageB` while spritifying. (Recall `mbWithColor` from Section 2.)

```
spritify :: [SpriteTree] -> RectB -> Maybe ColorB -> Transform2B
         -> TimeB -> ImageB -> [SpriteTree]
treesIm (spritify above rectB mbColorB xfB tt imB) ==
  treesIm above `over`
  (crop rectB          $
   mbWithColor mbColorB $
   (xfB *%)            $
   (`timeTransform` tt) $
   imB)
```

The algorithm works by recursive traversal of `ImageB` values, accumulating operations that are found out of order. To get the algorithm started, Fran has just an `ImageB`, `imB`, and a cropping rectangle, `windowRectB`, based on the display window's size (which may vary with time), and so invokes `spritify` as `spritify [] windowRectB Nothing identity2 time imB`, where `identity2` is the identity 2D transform, and `time` serves as the identity time transform.

Our claim that the algorithm given below satisfies this specification may be proved by induction on the `ImageB` representation. Rather than give the

```
treesIm :: [SpriteTree] -> ImageB
treesIm []            = emptyImage
treesIm (bot : above) = treesIm above `over` treeIm bot

treeIm (RenderedSprite motionB surfaceB) =
  move motionB (surfaceIm surfaceB)

treeIm (SolidSprite rectB colorB) =
  crop rectB (withColor colorB solidImage)

treeIm (FlipSprite rectB motionB scaleB book pageB) =
  crop rectB     $
  move motionB   $
  stretch scaleB $
  flipImage book pageB

treeIm (trees `UntilT` e) = treesIm trees `untilB` (e ==> treesIm)

treeIm (CondT c trees trees') = condB c (treesIm trees) (treesIm trees')
```

Fig. 7. Interpreting sprite trees

algorithm first and then the proof, however, we instead derive the algorithm from the specification so that it is correct by construction. We will rely on many simple algebraic properties of our data types.

Below we derive the `ImageB`-to-`[SpriteTree]` conversion for a small representative set of `ImageB` constructors. See [5] for a full treatment.

4.1 Solid images

Solid images are unaffected by transformations, and are of a default color, as expressed by the following properties:

```
xfB *% SolidImage == SolidImage

SolidImage `timeTransform` tt == SolidImage

withColor defaultColor SolidImage == SolidImage
```

These facts simplify the specification of `spritify`:

```
treesIm (spritify above rectB mbColorB xfB tt SolidImage) ==
  treesIm above `over`
  crop rectB (withColor (chooseColorB mbColorB) SolidImage)
```

where

```
chooseColorB :: Maybe ColorB -> ColorB
chooseColorB Nothing  = defaultColor
chooseColorB (Just c) = c
```

Now considering the interpretation of `SolidSprite` and the definition of `treesIm` given in Figure 7, the specification simplifies further:

```
treesIm (spritify above rectB mbColorB xfB tt SolidImage) ==
  treesIm (SolidSprite rectB (chooseColorB mbColorB) : above)
```

This final simplification then directly justifies the `SolidImage` case in the `spritify` algorithm.

```
spritify above rectB mbColorB xfB tt SolidImage =
  SolidSprite rectB (chooseColorB mbColorB) : above
```

4.2 Rendered images

Because of their spatially continuous nature, `ImageB` values may be thought of as being constructed bottom-up, whereas their implementation via `spritify` is top-down. For instance given "`stretch 10 circle`", we would not want to render a circle to a (discrete) bitmap and then stretch the bitmap, because the resulting quality would be poor. Similarly, for "`stretch 0.1 circle`", it would be wasteful to render and then shrink. Instead, scaling transforms are postponed and incorporated into the rendering, which can then be done at an appropriate resolution. Similarly, rotation is an expensive operation on bitmaps, but can generally be moved into rendering.

For this reason, we want to factor the transform behavior into translation, scale, and rotation. Fran's `factorTransform2` function does just this:

```
xfB == translate motionB `compose2`
       uscale    scaleB  `compose2`
       rotate    angleB
 where
   (motionB, scaleB, angleB) = factorTransform2 xfB
```

Moreover, the meaning of transform composition is function composition.

Factoring the spatial transform and turning transform composition into function composition, our specification becomes the following.

```
treesIm (spritify above rectB mbColorB xfB tt
                  (RenderImage renderer)) ==
  treesIm above `over`
  (crop rectB           $
   mbWithColor colorB   $
   move    motionB      $
   stretch scaleB       $
   turn    angleB       $
   (`timeTransform` tt)  $
   RenderImage renderer)
 where
   (motionB, scaleB, angleB) = factorTransform2 xfB
```

Next, since renderers do not perform motion, we must extract the `move` from within applications of `withColor` and `crop`. Coloring commutes with spatial transformation:

```
withColor c (xfb *% imb) == xfb *% (withColor c imb)
```

Cropping is trickier, requiring that the cropping rectangle be inversely transformed:

```
crop rectB (xfb *% imb) == xfb *% crop (inverse2 xfb *% rectB) imb
```

For example, doubling the size of an image and then cropping with a rectangle it is equivalent to cropping with a half-size rectangle and then doubling in size.

The definition of a *renderer* in Section 2 then leads to the following rule for spritifying rendered images. (The cropping and scaling behaviors should really be added to the `RenderImage` constructor and have the sprite engine use them, even though these operations are already performed during rendering. The reason is that the sprite engine can apply them incrementally at a much higher rate, for smoothness.)

```
spritify above rectB mbColorB xfB tt (RenderImage renderer) ==
  RenderedSprite motionB
      (renderer (move (-motionB) rectB)
                mbColorB scaleB angleB tt)
   : above
  where
    (motionB, scaleB, angleB) = factorTransform2 xfB
```

4.3 Overlays

The treatment of overlays follows from the distribution of time transformation, space transformation, coloring, and cropping over the `over` operation, plus the associativity of `over`.

```
spritify above rectB mbColor xfB tt (top `Over` bot) =
  spritify (spritify above rectB mbColor xfB tt top)
           rectB mbColor xfB tt bot
```

4.4 Space transformation

Time transformation distributes over space transformation:

```
(xfB *% imb) `timeTransform` tt ==
(xfB `timeTransform` tt) *% (imb `timeTransform` tt)
```

Space transforms compose:

```
xfB *% (xfB' *% imB) == (xfB `compose2` xfB') *% imB
```

The rule for `TransformI` then follows easily.

```
spritify above rectB mbColor xfB tt (xfB' `TransformI` imb) =
  spritify above rectB mbColor
           (xfB `compose2` (xfB' `timeTransform` tt)) tt imb
```

4.5 Conditional animation

Time- and space-transformation, coloring, and cropping, all distribute over conditionals:

```
    (condB boolB thenB elseB) `timeTransform` tt
 == condB (boolB `timeTransform` tt)
          (thenB `timeTransform` tt)
          (elseB `timeTransform` tt)

    xf *% (condB boolB thenB elseB)
 == condB boolB (xf *% thenB) (xf *% elseB)

    withColor c (condB boolB thenB elseB)
 == condB boolB (withColor c thenB) (withColor c elseB)

    crop rectB (condB boolB thenB elseB)
 == condB boolB (crop rectB thenB) (crop rectB elseB)
```

The rule then follows:

```
spritify above rectB mbColor xfB tt (CondI c imb imb') =
  CondT (c `timeTransform` tt)
        (spritify [] rectB mbColor xfB tt imb)
        (spritify [] rectB mbColor xfB tt imb)
   : above
```

4.6 State and concurrent updating

The spritifying algorithm above is idealized in that it constructs immutable sprite tree lists containing Fran behaviors. In fact, the sprite engine's data structures are mutable, both parametrically (e.g., sprite position and size) and structurally (e.g., number and kind of sprite), and only accomodate linear behaviors. The actual implementation of `spritify` creates the mutable sprite trees with initial values for position, scale, pitch, etc., and then iteratively updates these attributes, while the sprite engine runs concurrently. For simplicity of implementation, every active sprite is managed by its own Haskell thread, using the Concurrent Haskell primitives [10]. Each such thread is fueled by a request channel, with each request saying either to continue or to quit, and puts status messages into a response channel. Each sprite thread iteratively samples the appropriate behaviors (slightly into the future) and invokes an update method on the sprite object. In response, the sprite engine charts a new linear course for each attribute.

For example, Figure 8 is a slightly simplified implementation of the `SolidImage` case. A monochrome sprite is allocated in the sprite engine, chaining to the sprite trees given as "`above`". The time stream `ts` is used to sample the cropping rectangle and color, using `ats` (which is memoized as described in [6]). The resulting sample values are used to interactively update the sprite attributes, as long as the request channel `requestV` says to continue. The iteration is placed in a new Haskell thread via `forkIO`.

```
spritify :: SpriteTree  -> RectB  -> Maybe ColorB
         -> Transform2B -> ImageB -> Maybe TimeB
         -> [Time] -> MVar Bool -> MVar Bool -> IO SpriteTree

spritify above rectB mbColorB xfB tt SolidImage
         ts requestV replyV = do
  sprite <- newMonochromeSprite above
  let update ~(t:ts')
             ~(RectLLUR (Point2XY llx lly) (Point2XY urx ury) : rects')
             ~(ColorRGB r g b : colors') = do
         continue <- takeMVar requestV
         if continue then do
           updateMonochromeSprite sprite t llx lly urx ury r g b
           putMVar replyV True
           update ts' rects' colors'
          else
           putMVar replyV False
  forkIO $ update ts (rectB `ats` ts) (chooseColorB mbColorB `ats` ts)
  return (toSpriteTree sprite)
```

Fig. 8. `spritify` case with concurrent updating

The handling of a reactive image animation, `imb `untilB` e`, is somewhat tricky. The initial animation `imb` is spritified, starting any number of threads. The resulting sprite tree list is wrapped up in a `SpriteGroup` object (corresponding to the `UntilT` constructor above). One more thread runs to watch for event occurrences and causes the update work corresponding to `imb` to continue until the first occurrence of the event `e`, and then to stop. At that point, a new animation is available and is spritified, generating a new sprite tree list, which is then passed to a method on the `SpriteGroup` object that recursively deletes its old list and installs the new one. The number of running threads thus varies in response to events. Because every thread has accompanying overhead for controlling its work, a useful future optimization would be to create many fewer threads.

5 Conclusions

The implementation techniques described in this paper bridge the gap between functional animation and retained-mode display. Functional animation serves the needs of composability, while retained-mode promotes efficient use of hardware resources. A recurring theme in this work is the application of algebraic properties of our animation data types, in order to normalize animations to the relatively restrictive form imposed by retained-mode presentation libraries. An area of future work is to develop rigorous semantic models for these data types.

The models would form the interface between the informal mental models taught to and used by the everyday animation programmer and the correct and efficient implementation of the animation library itself. This semantic orientation has driven our work from the start, but some work remains to make it complete and precise.

References

1. Kavi Arya. A functional animation starter-kit. *Journal of Functional Programming*, 4(1):1–18, January 1994.
2. Conal Elliott. A brief introduction to ActiveVRML. Technical Report MSR-TR-96-05, Microsoft Research, 1996. `http://research.microsoft.com/scripts/-pubDB/pubsasp.asp?RecordID=38`.
3. Conal Elliott. Modeling interactive 3D and multimedia animation with an embedded language. In *The Conference on Domain-Specific Languages*, pages 285–296, Santa Barbara, California, October 1997. USENIX. WWW version at `http://-research.microsoft.com/~conal/papers/dsl97/dsl97.html`.
4. Conal Elliott. Composing reactive animations. *Dr. Dobb's Journal*, July 1998. Extended version with animations at `http://research.microsoft.com/~conal/-fran/{tutorial.htm,tutorialArticle.zip}`.
5. Conal Elliott. From functional animation to sprite-based display (extended version). Technical Report MSR-TR-98-28, Microsoft Research, 1998. `http://-research.microsoft.com/scripts/pubDB/pubsasp.asp?RecordID=190`.
6. Conal Elliott. Functional implementations of continuous modeled animation. In *Proceedings of PLILP/ALP '98*, 1998. `http://research.microsoft.-com/~conal/papers/plilpalp98/short.ps`. Extended version, MSR-TR-98-25, `http://research.microsoft.com/scripts/pubDB/pubsasp.asp?RecordID=164`.
7. Conal Elliott and Paul Hudak. Functional reactive animation. In *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming*, pages 263–273, Amsterdam, The Netherlands, 9–11 June 1997. `http://research.-microsoft.com/~conal/papers/icfp97.ps`.
8. Paul Hudak and Joseph Fasel. A gentle introduction to Haskell. *SIGPLAN Notices*, 27(5), May 1992. See `http://haskell.org/tutorial/index.html` for latest version.
9. Paul Hudak, Simon L. Peyton Jones, and (editors) Philip Wadler. Report on the programming language Haskell, A non-strict purely functional language (Version 1.2). *SIGPLAN Notices*, Mar, 1992. See `http://haskell.org/report/index.html` for latest version.
10. Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *Conference Record of POPL '96: The* 23rd *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 295–308, St. Petersburg Beach, Florida, 21–24 January 1996.
11. John Peterson, Conal Elliott, and Gary Shu Ling. Fran user's manual, Revised February, 1998. `http://research.microsoft.com/~conal/Fran/UsersMan.htm`.