(To appear in IEEE Transactions on Software Engineering)

An Embedded Modeling Language Approach to Interactive 3D and Multimedia Animation[1]

Conal Elliott
Microsoft Research

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

Abstract

While interactive multimedia animation is a very compelling medium, few people are able to express themselves in it. There are too many low-level details that have to do not with the desired content—e.g., shapes, appearance and behavior—but rather how to get a computer to present the content. For instance, behaviors like motion and growth are generally gradual, continuous phenomena. Moreover, many such behaviors go on simultaneously. Computers, on the other hand, cannot directly accommodate either of these basic properties, because they do their work in discrete steps rather than continuously, and they only do one thing at a time. Graphics programmers have to spend much of their effort bridging the gap between what an animation is and how to present it on a computer.

We propose that this situation can be improved by a change of language, and present *Fran*, synthesized by complementing an existing declarative host language, Haskell, with an embedded domain-specific vocabulary for modeled animation. As demonstrated in a collection of examples, the resulting animation descriptions are not only relatively easy to write, but also highly composable.

Index terms: Graphics, animation, multimedia, temporal modeling, domain-specific languages, embedded languages, functional programming, Haskell.

1 Introduction

Any language makes some ideas easy to express and other ideas difficult. As we will argue in this paper, today's mainstream programming languages are ill-suited for expressing multimedia animation (3D, 2D and sound), both in their basic paradigm and their vocabulary. These languages support what we call "presentation-oriented" programming, in which the essential nature of an animation, i.e., *what an animation is*, becomes lost in details of *how to present it*. We consider the question of what kind of language is suitable for capturing just the essence of an animation, and present one such language, *Fran*, synthesized by complementing an existing declarative "host language", Haskell, with an embedded domain-specific vocabulary.

We propose a declarative alternative to presentation-oriented programming, in which a model of the animation is described, leaving presentation as a separate task, to be done automatically. This idea of modeling has been applied fruitfully in the area of non-animated 3D graphics as discussed below, and is now widely, though not universally, accepted. Our contribution is to extend this idea in a uniform style to encompass as well sound and 2D images, and across the time dimension. For brevity, this paper concentrates on 3D animation, but it is really the uniform integration of different types that gives rise to great expressive power.

While imperative programming languages are suited to presentation-oriented programming, the modeling approach requires a different kind of language. Unfortunately, bringing a useful new language into being is quite a daunting task, requiring design of semantics and syntax, implementation of compilers and environment tools, and writing of educational material. However, as Peter Landin taught us more than thirty years ago, we can logically separate a language into (a) a domain-specific vocabulary and (b) a domain-independent way of composing more complex things from simpler ones [20]. In other words, a language is a combination of a "host language" and a "domain-specific embedded language" (DSEL). By reusing the same host language for several different vocabularies, we can amortize the cost of its creation over more uses. In fact, unlike thirty years ago, and thanks in part to Landin's influence, we are now fortunate enough to have some appropriate, established candidate languages from which to choose. In this paper, we examine various features of a candidate host language to see which are helpful and which are not helpful for modeled animation. We find that Haskell is a fairly good fit, requiring only a few compromises.

The rest of this paper is organized as follows. Section 2 informally introduces the domain concepts of interest. Section 3 presents just enough of Haskell to understand Section 4, which contains a few examples of modeled animation. Section 5 introduces the notions of presentation and modeling for non-animated 3D graphics, and looks at some concrete benefits. Section 6 extends the idea and benefits of modeling to a variety of types besides 3D geometry, including sound and 2D images, and across the time dimension. Section 7 considers the pragmatics of creating a new domain specific language (DSL), and motivates the DSEL approach. Section 8 examines the usefulness of host language features in some detail. The remainder of the paper looks at related work and describes some directions for future work on modeled animation.

2 Domain Concepts

A good first step in designing a DSL is to lay out the domain concepts. In our case, these concepts include 3D geometry, 2D images, sound, supporting types like colors, vectors and points, plus the notions that support reactive behavior. In this section, we will look into these domain concepts informally, independently from their concrete realization in a DSL.

2.1 3D Geometry

A 3D geometric model may be thought of a collection of primitive shapes, each having surface properties like color or textured images, and reflectivity.

The primitive shapes may be polyhedra, i.e., made of up connected polygonal faces, or smoothly curved. In fact, polyhedra are often intended as an approximation. Underlying display software and hardware often perform interpolation of lighting across surface polygons, in order to give the impression of a smoothly curved solid. In any case, geometric primitives are often best constructed with interactive modeling tools, and then simply "imported" into a language. We take this importation approach, and so will ignore details of geometric primitives.

Operations that are well suited to linguistic specification include the following:

- Spatial transformation: 3D models may be moved, scaled and rotated, as well as more esoteric transformations like shearing.
- Decoration: Models may be made more attractive, lifelike, etc, by the application of colors, and especially images, to the surface.
- Aggregation: After importing and transforming some models, it is often useful to unite them into a single model. The power of this operation is that the aggregate may then be subjected to further transformation (and aggregation), rather than having to keep track of several models and transform them individually.
- Lighting: Various types of lights (positional, directional, spot) may be created, colored, aggregated with other lights and geometric models, subjected to spatial transformation, etc. Each light affects the appearance of geometric models in the same scene.
- Sound: 3D models may emit sounds. As with light sources, sound sources are subjected to spatial transformation, which will influence the ultimate presentation.

In supporting the operations outlined above, it will be necessary to provide several other supporting types: geometric transforms (translation, scaling, rotation, and composition, and inversion); colors (construction and decomposition in RGB and HSL form, shades of gray, predefined constants), points and vectors (construction and decomposition in rectangular and spherical coordinates, distance, sums, differences, and scaling as sensible), images (for texture-mapping), and sound.

2.2 Images

Languages and modeling representations for 3D typically exclude or trivialize support for (2D) images. (For example, in VRML [32], the texture map for a 3D model may be specified either as a URL that points to a stored image or movie, or as a bitmap. The richness provided for synthetic and hierarchical construction, as well as animation, of 3D is denied to 2D.) In contrast, we believe that images should be given the same kind and level of support as 3D.

Our notion of image has infinite extent and infinite resolution. In contrast, a bitmap is a finite, discrete rectangular array of pixels. Images are thus independent of the size and resolution introduced by viewing, and are not turned into bitmaps until they are actually displayed.

Similarly to 3D geometry, it is very useful to import images from externally created discrete bitmaps, animations, and movies. Again, importation is just a start, and is augmented by a rich set of operations, including the following:

- Rendered 2D geometry, including line segments, polygons, and circles.
- Rendered text using any number of fonts and optional features like bold and italic.
- Rendering of 3D geometry.
- Coloring.
- 2D spatial transformation.
- Overlaying of images.
- Partial transparency.
- Embedded sound.
- Image cropping, according to rectangular regions, or arbitrary paths made up of line- and curve-segments.

As with 3D geometry, several supporting types are needed for the image operations: 2D geometric transforms, colors, 2D points and vectors, text with fonts, cropping rectangles and paths, 3D geometry, and sound.

2.3 Behaviors

So far, we have said nothing about animation. Rather than supporting animation with mechanisms specific to 3D geometry or 2D images, our approach is based on a general notion of *behaviors*, treating geometry and images as special cases. Informally, a behavior is simply a “time-varying value”. For instance, consider an animation of a 3D bouncing ball. The position of the ball varies with time and so is a (3D) point-valued behavior. The ball itself is a 3D geometry-valued behavior. Finally, the animation being viewed is an image-valued behavior.

Since behaviors are values that vary with “time”, we need to be clear about the nature of time. A fundamental decision is discrete vs. continuous, that is, do we think of time as moving forward in a (discrete) sequence of “clock ticks”, or a (continuous) flow? At first thought (especially to a programmer), a discrete model of time may seem natural. After all, the end user will experience a discrete sequence of images. As we will argue in Section 6, however, this reasoning results from a confusion between the ideal *model* and the *presentation* of that model. It is presentation that is discrete, due to the nature of display devices.

Just as spatially continuous image models naturally give rise to spatial resolution-independence and hence scalability, the continuous time model for animation yields *temporal* resolution-independence and scalability. For instance, one can stretch or squeeze a given animation to fit a desired duration. First making the clear distinction between modeling and presentation, and then allowing animation descriptions to be given in modeling terms, has considerable advantages. It fits with own human perception of time and motion as being continuous. It also allows us to exploit our rich heritage of mathematical, scientific, and engineering tools for understanding and describing the basic animation concepts of motion, growth, etc.

2.4 Events

Although motion and other behaviors are for the most part continuous, some can usefully be thought of as undergoing instantaneous change. For instance, whenever a rigid ball bounces, its velocity changes instantaneously. Such sudden changes are often caused by an *event* of some kind (such as collision). For our purposes, the event itself is an abstraction for the *sequence* of its occurrences, rather than a single one. (Note that we are using the term “event” in a somewhat unusual way, but have been unable to find a concise alternative.)

In order to respond to an event, a behavior needs to know the times at which the event occurs, so we might try to conceptualize an event abstractly as a stream of occurrence times. However, reaction often requires an additional piece of information at each occurrence. In the bouncing ball example, the velocity of the ball at the instant of each bounce is required in order to calculate the rebound velocity. If the event were the (set of all) collisions between two balls, the required information might be the velocity of one ball *relative* to the other, together with the ball's masses. Thus, we conceptualize an event as a stream of *occurrences*, each of which is a time/value pair. The information required from an event occurrence can be of any type, but for any given event, the occurrence values will be all of the same type. For collisions, one might use (relative velocity) vector-valued events.

The examples in this paper emphasize behaviors, making very simple use of events. For a much richer treatment of events, see [8]. Briefly, events are created and transformed in the following ways:

- external input;
- replacing or transforming an event's data;
- forming the union of two events;
- filtering out some event occurrences;
- monitoring time-varying conditions; and
- sequential chains of events.

2.5 Users

Because we are primarily interested in *interactive* animation, the user is an important domain concept. We consider a user to be a container for all of the user input that can be made available to an animation. In particular, this input includes mouse and tablet stylus locations, which are 2D point-valued behaviors, and events indicating activity from the pressing or releasing of keyboard keys or buttons on the mouse and stylus.

3 A Haskell mini-primer

We have embedded our DSL in the functional programming language Haskell. See [17] for an introduction to the language, and [19] for full details. In this section, we present just enough Haskell to understand the examples in Section 4.

3.1 Definitions

A Haskell program is made up of a collection of definitions. For instance,

```
msg = "Shall we get started?"

msgSize = length msg
```

The first two lines define the name `msg` to be a particular string, `size` to be the length of (number of characters in) `msg`. Note in the second line that application of a function (`length`) to an argument (`msg`) is written simply by juxtaposing the function and argument, without requiring parentheses around the argument.

A *function* definition is written similarly, but with the from parameter name(s) given on the left hand side, as in the following definition of `greeting`, which uses the infix concatenation operator, "++".

```
greeting name =
  "Hello, " ++ name ++ ".  " ++ msg
```

3.2 Types

Haskell is statically typed, but in most cases, all types can be inferred automatically by the compiler or interpreter. For the three names defined above, the following *declarations* may be either stated by the programmer or left to be inferred:

```
msg      :: String
msgSize  :: Int
greeting :: String -> String
```

The last declaration means "function from `String` to `String`". Note that type names begin with capital letters, while value and function names begin lower-case letters.

3.3 Multiple arguments

Application and definition of functions with more than one argument are written using repeated juxtaposition. For instance, the following defines a function `strangeName` of two arguments, making use of the function `elem` of two arguments. It considers a name to be strange if the name contains a 'z' and is longer than a given length.

```
strangeName name n =
  (elem 'z' name) && (length name > n)
```

The types of `elem` (a Haskell standard library function) and `strangeName` could be

```
strangeName :: String -> Int -> Bool

elem :: Char -> String -> Bool
```

where `Bool` is the type of booleans, i.e., true/false values.

The function type operator, "`->`", is right associative, so this declaration literally says that `elem` takes a character and returns a function that takes a string and returns a boolean. Correspondingly, function application is left associative, so the definition above applies `elem` to 'z', returning a function that is returned to name. The notation may seem odd at first, but is sometimes very useful.

3.4 Polymorphism

It is not really satisfactory for the `elem` function to have such a restricted type. In fact, it applies not just to strings, i.e., sequences of characters, but to sequences of any type of values. The type `String` is just an abbreviation for a list of characters:

```
type String = [Char]
```

The real type of `elem` is the following:

```
elem :: a -> [a] -> Bool
```

In type declarations, the use of a non-capitalized name indicates a *type variable*. This declaration says that the first argument to `elem` has some type *a*, and the second is a list of values, each having the same type *a*. (Note that the substitution for *a* is uniform. The sought element and every member of the list must all have the same type.) In a definition like that of `strangeName` above, the compiler figures out how to specialize the type of `elem` to the context of its use, in this case on characters.

Note that there is not a single "list" type, but rather a list type *constructor*, that maps element types (`a`) to the types of lists of elements (`[a]`). One may define additional type constructors as well – an ability that will be crucial in capturing the general notions of behavior and event, from Section 2. For instance, the notion of binary trees with values at the leaves might be captured by a type constructor `BinTree`, so a tree of strings would have type `BinTree String`. Type constructors may have any number of type arguments.

3.5 Infix operators

Note that the operators "&&" and ">" are used as infix. Such operators are simply function names whose names are composed of non-alphanumeric characters. By default, these names must be used as infix. Alphanumeric names precede their arguments, but may be used in infix form if surrounded by backquotes, as in

```
'z' `elem` name.
```

Names may also have declared syntactic binding strengths (precedences), to reduce the need for parentheses. All infix applications bind less strongly than function application (juxtaposition), so the parentheses is the definition of `strangeName` are unnecessary (because "&&" binds less tightly than "`>`").

3.6 Local definitions

It is often convenient to introduce some definitions just in order to make another definition more efficient or readable. In such a case, one may introduce local definitions, prefixed by "`where`". For instance:

```
thriceLength name = n + m
 where
   n = length name
   m = n + n
```

Here, the scope of the names `n` and `m` include the right hand side the definition of `thriceLength`, as well as the right hand side the definitions of `n` and `m` themselves. That is, definitions in the body of a `where` clause are mutually recursive (as are definitions at top level scope).

4 An introduction to Fran

The ideas presented in this paper are embodied in a system called "Fran" (for "functional reactive animation"). Fran consists of a collection of types, constants and functions, that correspond to the domain concepts outlined in Section 2.

The concepts from Section 2 – 3D geometry, 2D images, geometric transforms, points, vectors, and colors – correspond to Fran types with names like `Geometry`, `Image`, `Transform3`, `Point3`, etc. The notions of behavior and event are represented via the Fran type constructors `Behavior` and `Event`. Because Fran programs work mainly at the behavior level, Fran also defines several convenient synonyms for behavior types:

```
type RealB       = Behavior Double
type GeometryB   = Behavior Geometry
type ImageB      = Behavior Image
type Transform3B = Behavior Transform3
```

In the remainder of this section we introduce Fran through a handful of modeled animation examples. Many more examples of functional animation may be found in [10], [3], [5], [8], and [30]. See also the user's manual [25], which contains precise types and informal

Figure 1. Imported, scaled teapot

meanings of the embedded animation modeling vocabulary and still more examples.

4.1 Static models

To start, we import a simple 3D model of a sphere from "X file" format, by applying the `importX` function to the file name.

```
sphere = importX "sphere.x"
```

Similarly, we import the teapot model shown in Figure 1. However, the teapot is smaller than we would like, so we adjust it after importing, scaling it uniformly by a factor of 1.5:

```
teapot = uscale3 1.5 **%
         importX "teapot.x"
```

The function `uscale3` takes a number and produces a 3D transform. The infix operator `**%` applies a 3D transform to a 3D model to yield a new model. Here are the types of the modeling vocabulary we used. (The third declaration says that "`**%`" takes two arguments.)

```
importX  :: String -> GeometryB
uscale3  :: RealB -> Transform3B
(**%)    :: Transform3B -> GeometryB
         -> GeometryB
```

4.2 Spinning

Although types like `GeometryB` and `Vector3B` are potentially animated, the example so far uses static animations. Next we will color the teapot red and make it spin around the Y axis, as shown in Figure 2. (Scan the frames from left to right, starting with the top row.)

```
redSpinningPot =
  rotate3 zVector3 time **%
  withColorG red teapot
```

The new features are rotation, the unit Z vector, "time" and application of a color to a geometric model:

```
rotate3 :: Vector3B -> RealB
        -> Transform3B
zVector3 :: Vector3B
time :: RealB
withColorG :: ColorB -> GeometryB
           -> GeometryB
```

The function `rotate3` takes an axis vector and a number (both potentially animated) and yields a 3D transform. The use of `time` here deserves special attention. It is a primitive number-valued animation (hence the type `RealB`) representing the flow of time. Note that `time` is not a mutable real value, but a fixed animation. Animations are essentially functions of time, with `time` being the identity function, and operations like `rotate3`, `withColorG`, and `**%` being combinators that map functions of time to functions of time.

4.3 Generalizing

Next, generalize this simple spinning teapot, so that its color and rotation angle are parameters.

```
spinPot :: ColorB -> RealB -> GeometryB

spinPot potColor potAngle =
  rotate3 zVector3 potAngle **%
  withColorG potColor teapot
```

We will make use of the `spinPot` function in two interactive 2D animations.

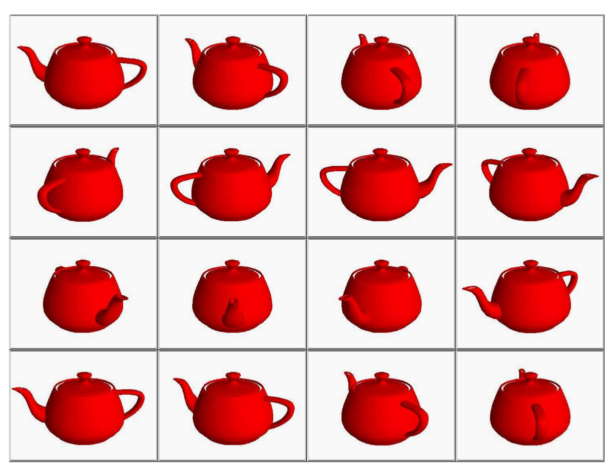

Figure 2. Spinning teapot

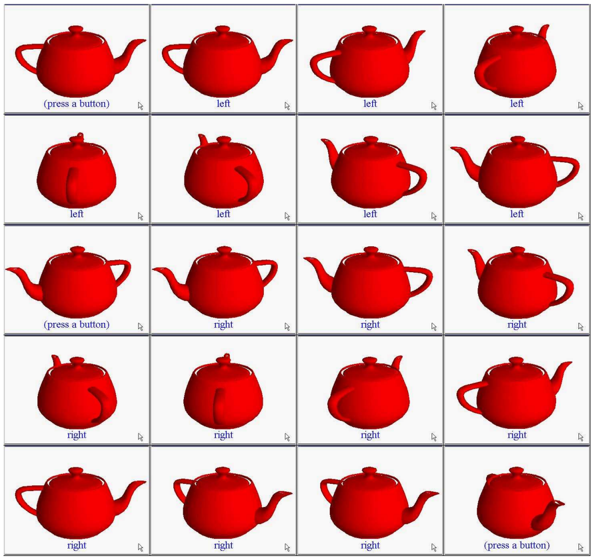

Figure 3. `spin1 = withSpin potSpin1`

```
spin1, spin2 :: User -> ImageB
spin1 = withSpin potSpin1
spin2 = withSpin potSpin2
```

When a 2D animation is interactive, its type is a function from the user supplying input. Hence the type above. Yet to be defined are `withSpin`, `potSpin1`, and `potSpin2`.

The argument to the function `withSpin` will be a "spinner", i.e., something that knows how to turn an animated number and a user into an animated geometric model, using the animated number to determine a time-varying rotation angle. In an object-oriented language, one would formalize the "spinner" type as an abstract class, and then provide some number of concrete subclasses. In Haskell, we can capture the notion of spinner as a function type:

```
type Spinner = RealB -> User-> GeometryB

potSpin1, potSpin2 :: Spinner
```

The `withSpin` function takes a spinner and a user and produces an image animation:

```
withSpin :: Spinner -> User -> ImageB
```

Functions in Haskell are first class values and may be passed into and out of functions. Like objects, they can combine data with code. In fact, a good way to think of Haskell functions is as single-method objects, supported by a lightweight notation. (New functions may be expressed in-line, without naming them, using "lambda" notation.)

As a simple case, `potSpin1` just ignores the user, uses red for the pot color, and passes on the angle argument unchanged:

```
potSpin1 angle u = spinPot red angle
```

The `withSpin` function takes one of these geometry producers and renders it together with some textual instructions. Figure 3 shows the result combined with `potSpin1`.

```
withSpin f u =
  growHowTo u ‘over‘
  renderGeometry (f (grow u) u)
                 defaultCamera
```

The function `grow` will be defined below. Its job is to turn user input into an animated angle, which gets passed to the geometry producer. The produced geometry is rendered with a default camera to produce a 2D animation, which is overlaid by the instruction text image produced by `growHowTo`. The function `renderGeometry` takes geometry and camera (both potentially animated, as always), and yields a 2D animation:

```
renderGeometry :: GeometryB
               -> Transform3B -> ImageB
```

4.4 A more interesting pot spinner

Before looking into the definition of `grow`, consider a second spinner, which adds a few new features:

- A light source is added and visualized as a white sphere that orbits the spinning teapot. For convenience, the translation vector is specified in spherical coordinates.
- The teapot's color is animated, and specified in HSL (hue/saturation/lightness) coordinates.
- The "angle" argument generated by `grow` and passed by `withSpin` is really meant to be the rate of change of the desired angle, and so is integrated (over time). Integration is meaningful because the angle is a function of time (a behavior).

The definition:

```
potSpin2 potAngleSpeed u =
  spinPot potColor potAngle
   ‘unionG‘ light
 where
  light = rotate3 yVector3 (pi/4)    **%
          translate3 (vector3Spherical
                          1 0 time) **%
          uscale3 0.1               **%
          withColorG white (
           sphere ‘unionG‘ pointLightG)
  potColor = colorHSL time 0.5 0.5
  potAngle = integral potAngleSpeed
```

Figure 4 shows the result.

4.5 Reactive growth

Now we turn to `grow`, which converts user input to a time-varying angle (of type `RealB`). It is defined as the integral (over time) of the value generated by `bSign`, defined below, which produces an animated number. That number has value zero when no mouse buttons are pressed, but switches to negative one or positive one while the user is holding down the left or right mouse button. The angle value produced by `grow` is thus growing while the right button is pressed, shrinking while the left is pressed, and constant when neither button is pressed.

```
grow :: User -> RealB
grow u = integral (bSign u)
```

The polymorphic `bSign` function is itself defined in terms of a more general function `selectLeftRight`, which switches between three values, depending on the left and right button states.

```
bSign :: User -> RealB
bSign u = selectLeftRight 0 (-1) 1 u

selectLeftRight :: a -> a -> a -> User
                -> Behavior a

selectLeftRight none left right u =
  ifB (leftButton u)
      (constantB left)
    (ifB (rightButton u)
         (constantB right)
       (constantB none))
```

Some explanation: the function `ifB` is a behavior-level conditional, taking an animated boolean and two animated values, and choosing between the two continuously. The primitive `constantB` turns a regular “static” value into a constant animated value (as required here by `ifB`). The `leftButton` and `rightButton` functions tell whether the mouse buttons are pressed.

It is easy to define these two button state functions in terms of a toggling function that takes an initial value and two events that tell when to switch to true and when to false.

```
leftButton, rightButton ::
  User -> BoolB
leftButton  u = toggle (lbp u) (lbr u)
rightButton u = toggle (rbp u) (rbr u)

toggle :: Event a -> Event b -> BoolB
toggle go stop =
  stepper False (  go   -=> True
             .|. stop -=> False)
```

The functions `lbp`, `lbr`, `rbp`, and `rbr`, yield left and right button press and release events.

```
lbp, rbp, lbr, rbr :: User -> Event ()
```

The `stepper` function takes an initial value *v* and an event *e*, and yields a piecewise-constant behavior that starts out as *v* and switches to the values associated with occurrences of *e*. In the definition of `toggle`, the event is constructed from the `go` and `stop` argument events. The event handling operator “`-=>`” is used to replace the trivial values from the button-press events, and the event merging operator “`.|.`” is used to merge the two boolean-valued events (streams) into a single one. As a result, the constructed event occurs with value `True` whenever `go` occurs and with value `False` whenever `stop` occurs. (Note: the event operators are described in [10], but their semantics have changed since that publication, and now consist of a *sequence* of occurrences, not just a single one. Also, the button press events and mouse motion behavior are

Figure 4. `spin2 = withSpin potSpin2`

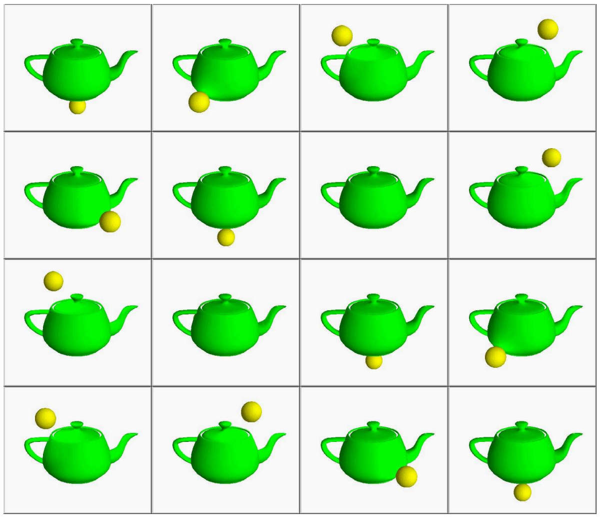

Figure 5. `potAndLight`

functions of a `User` rather than a start time.)

4.6 Adding instructions

Finally, to generate instructions and user feedback, we define `growHowTo`, which produces a rendered string, colored yellow and moved down to be out of the way. The text gives instructions when neither button is pressed, says "left" while the left button is pressed, and "right" while the right button is pressed. Its definition involves 2D versions of vectors, transform formation and application, and coloring, plus the polymorphic function `selectLeftRight`, defined above.

```
growHowTo :: User -> ImageB

growHowTo u =
  translate2 (vector2XY 0 (-1)) *%
  withColor blue (
   stringBIm messageB)
  where
    messageB = selectLeftRight
                 "(press a button)"
                 "left" "right" u
```

4.7 Lights

In addition to visible geometry, we can add lights to a 3D model. In this next example, shown in Figure 5, we combine a sphere, which is visible but does not emit light, and a "point light source", which is invisible but emits light. We color the sphere/light pair yellow, shrink it, and give it motion.

```
movingLight =
  translate3 motion **%
  uscale3 0.1       **%
  withColorG yellow (
    sphereLowRes ‘unionG‘ pointLightG)
 where
  motion = vector3Spherical 1.5
             (pi*time) (2*pi*time)
```

For convenience, we have expressed the motion path in terms of spherical coordinates, saying that the distance from the origin of space (which is also the center of the teapot) is always 1.5 units, the longitude is π times the elapsed time, and the latitude is twice π times the elapsed time. Consequently, light meanders about, maintaining a fixed distance from the center of the teapot, and repeating its behavior every two seconds.

To finish the example, combine the moving light with a green teapot:

```
potAndLight =
  withColorG green teapot ‘unionG‘
  movingLight
```

The `unionG` function, here used as an infix operator, combines two `GeometryB` values to form another one.

```
unionG :: GeometryB -> GeometryB
       -> GeometryB
```

Note the expression "2 * `pi * time`" used in defining the light's motion. The meaning of "*" is not the usual one, operating on static numbers, but rather a counterpart "lifted" to consume and produce number-valued animations (of type `RealB`). Even the numeric literal `2` and the constant `pi` are taken to mean the corresponding unchanging number-valued animations (having type `RealB`). In Fran several dozen functions have been lifted in this way, so that, for instance, "*"and `sin` have not only the usual types

```
(*) :: Double -> Double -> Double
sin :: Double -> Double
```

but also

```
(*) :: RealB -> RealB -> RealB
sin :: RealB -> RealB
```

(The type name "`RealB`" is a synonym for "`Behavior Double`".) This kind of assists greatly in making animations easy to write and read.

4.8 Time transformation

The next example, shown in Figure 6, replaces the single light from Figure 5 with a string of five lights playing "follow the leader". The leader is the moving light from the previous example. All of the others are time-delayed replicas.

```
potAndLights =
  slower 5 (
   withColorG green teapot ‘unionG‘
   delayAnims3 (2/5) (
    replicate 5 movingLight) )
```

We make a list of five copies of the moving light, using the `replicate` function, stagger them in time with `delayAnims3` (defined below), and combine them with a green teapot. Then we slow down the animation to see it more clearly.

The function `delayAnims3`, takes a time delay and a list of 3D animations, and yields a 3D animation. Each successive member of the given animation list is delayed by the given amount after the previous member.

```
delayAnims3 dt anims =
  unionGs (
   zipWith later [0, dt ..] anims)
```

The notation `[0, dt ..]` means the infinite list of numbers 0, *dt*, 2 *dt*, 3 *dt*, etc. The function `zipWith` applies a given two argument function to the successive values from two given lists, stopping when it reaches the end of the shorter list. We use it here to delay the first animation in `anims` by 0 seconds, the second by *dt* seconds, the third by 2 *dt* seconds, etc. The Fran function `unionGs` is like `unionG`, but it combines a list of 3D animations rather than just two.

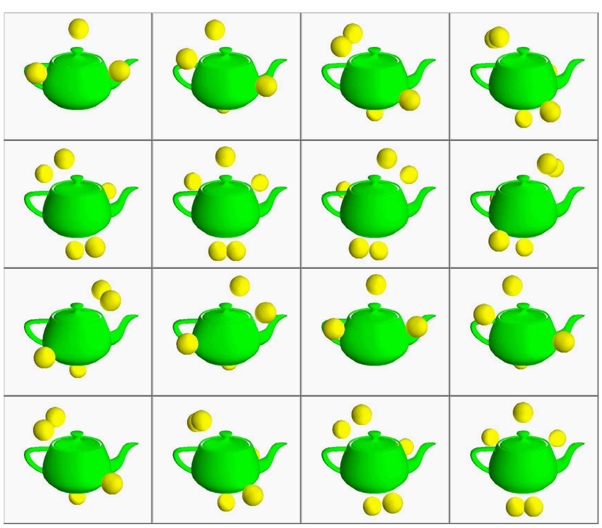

Figure 6. `potAndLights`

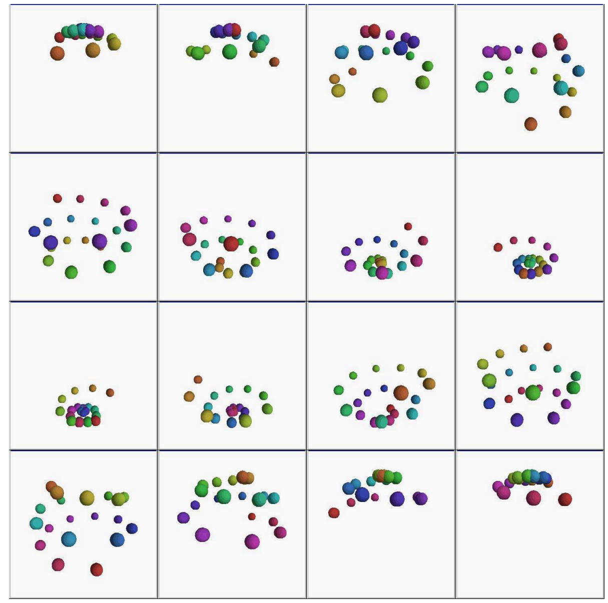

Figure 7. `spiral3D`

Figure 7 shows another use of `delayAnims3`. Here we create a single `ball` having a spiral `motion` that traces the surface of an unseen sphere of radius 1.5, with a longitude angle changing ten times as fast as the latitude angle. From this one moving ball, we make twenty balls, colored with evenly spaced hues, and then stagger them in time with `delayAnims3`.

```
spiral3D = delayAnims3 0.075 balls
   where
     ball = translate3 motion **%
            uscale3 0.1       **%
            sphere
     balls  = [ withColorG (bColor i) ball
              | i <- [1 .. n] ]
     motion = vector3Spherical
                1.5 (10*time) time
     n = 20
     bColor i =
       colorHSL
         (2 * pi * fromInt i / fromInt n)
         0.5 0.5
```

The Fran functions `later` and `slower` are both examples of *time transformation*. They apply to 3D animations, but also to 2D animations, sounds, and behaviors of all types. *Spatial* transformation is widely recognized as a useful tool for making specifications of static 3D models more convenient, modular and compact. Fran applies the idea to time as well.

With the given examples in mind, we step back from our chosen approach to expressing interactive animation, and consider the history, the benefits of "modeling", and of language embedding.

5 Presentation vs. modeling for 3D geometry

The practice of 3D graphics programming has made tremendous progress over the past three decades. In the early days, if you wanted your program to display some graphics you had to work at the level of pixel generation. You had to master scan-line conversion of lines, polygons, and curved surfaces, hidden surface elimination, and lighting and shading models—rather complex tasks. A significant advancement was the distillation of this expertise into rendering libraries (and underlying hardware). With a rendering library, you could express yourself at the level of triangles and transformation matrices. While an advancement, these libraries presented a view of a somewhat complex state machine containing registers such as the current material properties and the current local or global transformation matrices. You had to drive this state machine, push register values onto a stack, change them, instruct the library to display a collection of triangles, and restore the registers at the right time.

The next major advancement was to further factor out common chores of graphics presentation into libraries that presented complex structured *models*, as exemplified in such systems as PHIGS, SGI's Inventor and Performer, VRML, and Microsoft's Direct3D RM (retained mode). The paradigm shift from presentation to modeling for geometry has had several practical benefits:

- *Ease of construction.* Models are generally easier for people to express and read than the corresponding presentation programs. (In the case of experienced programmers, there may be an initial period of *unlearning* presentation-oriented thinking habits, i.e., unconscious tendencies to think in terms of *how* to display some geometry, rather than simply what the geometry *is*.) In fact, model specifications are often not *programs* at all, but simply descriptions, such as "a red chair, doubled in size". Presentation specifications, on the other hand, generally are programs.
- *Authoring.* Content creation systems are naturally based on models, because their end users think in terms of models, and typically have neither expertise nor interest in programming the corresponding presentation.
- *Composability.* Models tend to be more robustly composable than presentation programs, thanks to the absence of side effects, which could otherwise interfere in subtle ways with the meaning of other components. Composability is a crucial factor in the scalability of any programming or modeling system, as well as the key to enabling powerful end-user features like cut-and-paste and drag-and-drop. The keys to robust composability are that (a) composition must construct the same kind of thing as the composed components, so that the result can be composed again, arbitrarily, and (b) composition operations allow only well-controlled interactions among components (e.g., lighting and occlusion). Note that there is an industry that sells a variety of specialized geometric models, but there is not one that sells specialized presentation code snippets.
- *Optimizability*. Model-based systems contain a *presentation sub-system* that contains code to render any model that can be constructed with the system. Because higher level information is available to the presentation sub-system than with presentation programs, there are many more opportunities for optimization. Examples include hierarchical culling, display-sensitive triangle generation from curved surfaces, set-up for various hidden surface removal algorithms when lacking Z-buffered hardware, and vertex data conversion from application representation to device representation. SGI's Performer and Microsoft's Direct3D RM products were largely motivated by these opportunities for optimization. Imagine how hard it would be to do these optimizations if the application explicitly managed each step of geometry presentation. It would be akin to reverse engineering the model out of the imperative presentation code.
- *Economy of scale.* Because the presentation sub-system is used for many different applications, it is worthwhile to invest considerably in optimization and functionality. When an application does its own presentation, such an investment is not as likely to be warranted.
- *Usefulness and longevity.* Models have broader usefulness and a longer lifetime than presentation programs, because models are platform independent. Presentation sub-systems can be separately tuned or totally re-implemented to run on a variety of radically different hardware architectures, from no graphics hardware, to SMP platforms, to SGI-like 3D hardware, and well beyond. Models will

not only be *able* to be presented on these different architectures, but their presentation can exploit the best features of each architecture. Again, economy of scale makes this tuning and re-implementation work worthwhile.

- *Regulation.* The presentation sub-system can perform automatic level-of-detail management, determining the sequence of low-level presentation instructions executed dynamically, based on scene complexity, machine speed and load, etc. In contrast, a presentation-oriented application either hardwires a level-of-detail, and so is appropriate for only a narrow range of machines and circumstances, or must make a considerable investment in doing explicit, specialized regulation.

In spite of the benefits listed above, not everyone has made the shift from presentation to modeling of geometry. The primary source of resistance to this paradigm shift has been that it entails a loss of low level control of execution, and hence efficiency. As mentioned above, handing over low level execution control from the application to the presentation sub-system actually benefits execution efficiency where authors lack the significant resources and expertise required to implement, optimize, and port their programs for all required platforms. In other cases, as in the case of state-of-the-art commercial video games, the resources and expertise are available and worth the considerable investment. An example is Doom, which would have been a failure at the time if implemented on top of a general-purpose presentation library. On the other hand, even Doom and its successors really adopt the modeling paradigm, in that they consist of a rendering engine paired with a modeling representation. In addition to the loss of direct control of efficiency, modeling tends to eliminate some flexibility in the form of presentation-level tricks that do not correspond to any expressible model. In our experience, these tricks tend not to scale well and are not composable, and in cases that do, are achievable through model extensibility.

There have been many other similar paradigm shifts, generally embodied in specialized languages, sometimes with corresponding tools that generate the language. Examples include dialog box languages and editors; grammar languages and parser generators; page layout languages and desktop publishing programs; and high-level programming languages and compilers.

6 Presentation vs. modeling for animation

The conventional approach to constructing richly interactive animated content is much like the old days of graphics rendering, as described briefly above, that is one must write sequential, imperative programs. (Much animation is in fact modeled rather than programmed, because it comes from animation authoring tools, but interaction is severely limited, for instance to hyper-linking.) These programs must explicitly manage common implementation chores that have nothing to do with the content of animation itself, but rather its *presentation* on a digital computer. These implementation chores include:

- sampling in time for simulation and frame generation, even though the animation is conceptually continuous;
- capturing and handling of sequences of motion input "events", even though motion input is conceptually continuous;
- time slicing to update each time-varying animation parameter, even though these parameters conceptually vary in parallel;
- management of network connections and remote messaging for distributed applications such as shared virtual spaces, multi-player games, and collaborative design, even though the various users and objects are conceptually in a single world;

The essence of modeled animation is to carry the presentation/modeling paradigm shift beyond static (non-time-varying) 3D geometry, and thus more broadly reap the kind of benefits described in the previous section. The extensions to static geometric modeling embodied in modeled animation include the following:

- Apply the modeling principle to richly model sound and 2D imagery. These domains are as complex and important in their own right as geometry, and so are supported on equal footing with 3D, rather than as decorations on an essentially 3D representation, such as VRML's scene graphs. Going even further, treat the multitude of other data types that arise from 3D, 2D, and sonic modeling (transforms, points, colors, etc.) on an equal footing as well.
- Go beyond modeling of static geometry, images, etc., to behaviors and interaction—what one might call *temporal modeling*.

- Recognize that for a modeling representation to be sufficiently rich, it must inevitably be a quite expressive language, though not necessarily an imperative programming language. Even static modeling representations like VRML [24] had to incorporate the essential mechanisms of a language, which are composition (through hierarchy and aggregation), naming, and information passing (through attributes), though in ad hoc, limited forms.

By extending modeling from static 3D to other types and to animation, we also extend the modeling benefits listed in the previous section. Most of these benefits translate in straightforward ways, but some possible non-obvious extensions are as follows:

- *Composability.* Temporal models compose into new temporal models, to an arbitrary level of complexity. There are no execution side-effects to cause interference among the composed components, nor are there any evaluation order dependencies. For example, in the examples above, consider the use of the relatively simple `spinPot` to help define the more complex `potSpin2`, and the use of `selectLeftRight` to define `bSign` and `growHowTo`.
- *Optimizability.* Techniques such as culling via spatial bounding volume hierarchies can be applied infrequently to temporal models, rather than at every frame on static models. Such analyzability is especially important for intensive computations like collision detection, which has been shown to be amenable to temporal analysis techniques [15]. Moreover, because animation is modeled explicitly, rather than being the result of side-effects to a mutable static model carried out by imperative code, the engine knows exactly what values and relationships are fixed and which ones vary dynamically. Note that it is precisely this knowledge that has proved vital to the success of programming language compilers. Most programmers gave up the control afforded by writing self-modifying code, and as a result, compilers gained enough information about the run-time behavior of a program to be able to perform significant optimization. As a result, most portions of even performance- and space-sensitive code are now written in languages like C or C++, rather than assembler. Many of the benefits of, and objections to, modeling vs. presentation listed above directly apply to the issue of programming in C vs. assembly language.
- *Usefulness and longevity*. Because model definitions have no artificial sequentiality, temporal models may be executed in parallel, where parallel hardware is available. In contrast, imperative programs are notoriously difficult to parallelize and in practice must be rewritten.
- *Regulation*. The presentation of an interactive animation involves a multitude of sampling rates, including simulation parameter sampling, input sampling, geometry generation, geometry rendering, and image display. These many sampling rates may all be varied automatically, based on the computational and visual complexity of a scene, machine speed and load, etc. Moreover, computation of simulation parameters based on kinematics or dynamics can choose and adaptively vary numerical integration algorithms.

7 Language considerations

So far, we have used the term “modeling language” loosely. In this section, we make a more precise examination of the different possible notions of “language” and some of their pros and cons for practical use.

A language may be thought of as the combination of two complementary aspects. One aspect is domain-generic, and contains fundamental syntactic and semantic notions like definition and use of names for values and types, construction and application of functions or procedures, control flow, data flow, exception handing, and typing rules. The other language aspect is a domain-specific vocabulary, describing, e.g., math operations on floating point numbers, string manipulation, lists and trees, and in our context, geometry, imagery, sound and animation.

Holding these two language aspects in mind, there are two strategies we could adopt in making concrete the idea of an animation modeling language, or any DSL, which we will call “integrated” and “embedded” respectively. In the integrated approach, the DSL combines both language aspects. In the embedded approach, the domain-specific vocabulary is introduced into an existing “host” programming language. While these two strategies may be similar in spirit, the pragmatics of carrying them out differs considerably.

The chief advantage of integration is that one can have a perfectly suited language, semantically and syntactically, while the embedded approach requires toleration of compromises made to accommodate a broad range of domains. In return for this toleration, the embedded DSL approach allows one to use already existing language infrastructure.

To be useful in practice, not just a toy or a research experiment, a complete DSL needs several components, well designed and well executed:

- A language definition. Despite the best intentions of their original designers, successful DSL's (ones with users) tend to grow, eventually incorporating more and more general purpose language features. When this growth is not anticipated, by providing good general purpose abstraction mechanisms a priori, the results can be ugly.
- A language implementation. Depending on the domain, an interpreter may suffice, but for some cases, including real-time animation, compilation is important. A good compiler, such as the Glasgow Haskell Compiler [26], requires years to develop.
- Environment tools. Programmers need debuggers and profilers to get their programs working correctly and efficiently. Also, inherent in domain-specificity, there needs to be a way to package up components of functionality in such a way that it can inter-operate with components implemented in other languages, domain-specific or otherwise.
- Educational material. Users must be provided with tutorials to get them started, and reference manuals to fill in the details.

Given this list, we have ample incentive to try to make the embedded DSL approach work, if we can find a suitable existing host language. We next take a closer look at the question of what features constitute suitability.

8 Choosing a host language for modeled animation

We have found a variety of host language features to be helpful for animation modeling and language embedding in general, while others were harmful. The helpful features include the following, some of which are obvious from a programming language perspective, but are in fact missing or very weakly present in popular model representations for geometry and animation.

- *Expressions*. Models are specified primarily in terms of other models, applying various kinds of transformations, forming aggregates, transforming some more, etc. Expressions, in the programming language sense, are well suited for this *compositional* style of specification, since they nest conveniently and suggest manipulation of *values* (models) rather than *effects* (presentations). One kind of expression that is particularly useful is the conditional, as in C's often ignored: "*cond ? exp1 : exp2*".
- *Definition*. In order to use a model more than once, or to separate the definition of a model from its uses, there needs to be a mechanism for referring to a single model any number of times in different contexts. A simple and general such mechanism is the definition of names for models denoted by expressions, together with the use of names to denote the corresponding models. Such definitions should have controllable scope, such as introduced by "`where`" in the some of the examples above.
- *General parameterization.* Values such as numbers, strings and booleans are not nearly as interesting a set of reusable building blocks as are the *functions* that create these values. Exactly the same is true for values/models such as geometry, images, sounds, transformations, and animations. The most powerfully reusable building blocks tend to be parameterized models, such as `spinPot` and `leftRightSelect` above, and therefore a modeling language needs a mechanism for expressing functions from arguments of arbitrary types to results of arbitrary types.
- *Higher-order programming (first class functions).* Higher-order functions allow succinct expression and encapsulation of useful domain-specific *programming patterns*. Consequently, it is useful to allow for parameterized models to accept other *parameterized* models as arguments and/or produce them as results. As a particular example, response to user interaction events is often expressed in terms of call-backs. In a higher-order language, these call-backs may be specified succinctly, using lambda-abstraction or locally-defined functions. (Strong static typing eliminates the need for unsafe type coercion or run-time checking.) In the examples above, `withSpin` makes critical use of higher-order programming.
- *Strong, static typing*. Models and their components are of a variety of different types, such as geometry, image, sound, 2D and 3D transform, 2D and 3D point and vector, color, number and Boolean, as well as animations over all of these types, and events yielding information of all of these types. A static type system guides authors toward meaningful model descriptions, enabling helpful error messages before execution. Static typing also improves performance by eliminating the need for

run-time type checking, while retaining execution safety. In order not to clutter a model definition, it is helpful is types can be inferred automatically, rather than always being specified explicitly.

- *Parametric polymorphism*. Animation is a polymorphic concept, applying to geometry, images, sound, 2D and 3D points vectors, colors, numbers etc. Similarly, reactive animation makes essential use of the polymorphic notion of an *event*, occurrences of which carry with them not only a time, but also a value of some type. Several of Fran's animation- and event-building operations "`ifB`", "`stepper`", "`==>`" and "`.|.`", apply to an infinite family of types. Note that non-polymorphic languages generally have polymorphic primitives, such as conditionals. To serve as a host language for an embedded DSL, however, the polymorphism must be available for the embedded vocabulary, as in the function `selectLeftRight`, which was applied to numbers and to strings above.

- *Notational flexibility*. It is convenient to give new, domain-specific, meanings to old names. In particular, Fran overloads most of the names of the standard math functions, e.g., "`sin`" and "`+`", to operate at the level of animations. It even overloads constants, e.g., "`pi`" and "1", to denote animations (though not very animated ones), and introduces new infix operators with suitable associativity and binding strength. While "merely" a notational convenience, this notational flexibility is largely responsible for giving the "look and feel" of a tailored domain specific language, and makes the resulting programs much easier to read than they would be if we had to introduce a whole new collection of names.

- *Automatic garbage collection*. An animation typically contains many components that contribute for a short while, or in any case, less than the full duration of the animation. Automatic garbage collection makes for safe and efficient memory use.

- *Laziness*. An interactive animation is a "big" value, often infinitely big, containing repetition and branching. It is important, even crucial, that parts of an animation be available for consumption before the rest of the animation has actually been produced. The idea of laziness is to postpone production of parts of a value until the last possible moment, i.e., when those parts need to be consumed for display. Often parts are completely unused, and so should never actually be computed. For example, in a computer game, many possible branches are not taken and many simulated characters are not seen during the play of a single game. As a simpler example, the animations produced by `bSign` and `growHowTo` can have an infinite number of phase changes, according to user input, but they are available immediately for partial consumption.

  What are usually thought of as primitive control structures, such as conditionals and iteration, are often definable in lazy languages. As a consequence, "domain-specific control structures" are also definable. (Higher-order programming with lambda abstraction makes it possible to define domain-specific variable binding constructs as well.) For instance, one could define animation repetition operators in Fran.

  Laziness also plays a role complementary to garbage collection, for efficient use of memory. Laziness delays consumption of memory until just before an animation component is needed, while garbage collection frees the memory when an animation component is no longer needed.

- *Modules*. Like conventional programs, model specifications can grow to be quite complex, and so should be specifiable in parts by different authors and in different files distributed throughout the Internet. Moreover, it should be possible to compile these modules into an executable form such as Intel binary or Java byte-code, with formal interfaces that state the names and types of values and functions implemented in the module.

For programmers not familiar with modern functional languages, it may be easier at first to think of a given set of higher-order, polymorphic, overloaded, and non-strict functions as being "language" features. Going beyond this view yields a much more powerful understanding, by making it apparent that one can effortlessly extend the "language" with new features having these same properties. For instance, in non-lazy programming languages, the polymorphic lazy conditional statement or expression is wired into the language, and it is impossible to introduce a variation, unless it happens to be expressible as a macro. Even where applicable, macro facilities are often weakly expressive, statically untyped, and/or problematic with respect to scoping.

Imperative programming languages, such as C, C++, Java and Visual Basic, have *statements* in addition to *expressions*, and in fact, emphasize statements over expressions. For example, in these languages, it is possible to introduce a scoped variable in a statement, but not in an expression. Also, `if` works on statements, though C has its ternary "`?:`" expression operator. While expressions are primarily for denoting values, statements are for denoting changes to an internal or external state. State changes certainly occur during *presentation* of a model, but are not appropriate in the model itself, as they interfere with composability, optimizability, and multithreaded, parallel and distributed execution. There are also "semi-functional" languages, such as Lisp and ML, which give richer treatment to expressions without discarding imperative programming.

Common language features that are statement-oriented, and which thus are *not* useful for modeled animation, include the following:

- *Sequencing*. Without statements, there is no role for the usual notion of sequencing, which is executing multiple statements in serial, and relies on *implicit communication* of information from one statement to the next through side effects.
- "*Goto*".
- *Conditional statements*.
- *Sequential iteration*. Really just a compact way to specify possibly infinite sequencing and conditional execution.

In another sense, rather than avoiding state changes, our approach is to impose a discipline on state-changing computations and to treat them as first-class values, in the spirit of "functional imperative programming" [34][27]. Moreover, we generalize from discretely changing to *continuously changing* state.

Given the language requirements and non-requirements above, we now return to the "integrated-vs-embedded" question, keeping in mind that design and implementation of a new programming language and development tools, and creation of required educational material are formidable tasks, not to be undertaken unless genuinely necessary. Fortunately, there are well-suited existing languages, the so-called "statically typed, higher-order, purely functional" languages. Of those languages, *Haskell* [19], [17], [30] has the largest following, has an international standard (Haskell 1.4), and is undergoing considerable development. For these reasons, we have chosen Haskell as Fran's host language. Other languages can be used as well, with varying tradeoffs. For example, Java is more popular than Haskell and supports garbage collection, but is predominately statement-oriented and lacks parametric polymorphism, notational flexibility and laziness.

While neither the current development tools and educational material for Haskell programming, nor the size of the Haskell programming community, is impressive compared to those of mainstream languages, we believe that both are sufficient to act as a seed, with which to generate initial compelling applications. We hope that these initial applications will inspire the curiosity and creativity of a somewhat larger set of programmers, leading to better development tools and written materials, yet more compelling applications, and so on, in a positive feedback cycle.

Having touted the benefits of lazy functional languages for modeling geometry and animation, it is only fair to point out that some domains are better served by descriptions not easily expressed in a functional setting. For example, some domains already have problem solving techniques based around satisfying systems of equations or non-linear constraints. Constraint logic programming allows specification of *relationships* among model components. Given these relationships, the underlying search and constraint satisfaction engines automatically determine how to compute some components in terms of others. We would really like to find a language paradigm that combines functional and constraint logic programming into a single one that has the strengths of both. Such a combination is a matter of ongoing research.

Aside from issues of familiarity, there will always be an important role for imperative computation in the construction of *complete* applications. One could throw such features into a modeling language, or even try to force imperative programming languages to also serve as modeling languages. We prefer the approach of *multi-lingual integration*, which is to support construction of application modules in a variety of languages and then combine the parts, generally in compiled form, with a language neutral tool [11].

9 Implementation

A discussion of the implementation of Fran is beyond the scope of this paper. We refer the interested reader to [10][6][7], and to [28], which describes the implementation of a predecessor of Fran.

As explained informally in Section 2.3, a behavior is a time-varying value. It may thus be represented directly as a function of time.

```
type Behavior a = Time -> a
```

It is easy to define function overloadings that work on behaviors, e.g.,

```
sin :: RealB -> RealB
(sin b) t = sin (b t)

(+) :: RealB -> RealB -> RealB
(b1 + b2) t = b1 t + b2 t
```

An event may be identified with a stream (lazy list) of occurrences, each of which is a time/value pair:

```
type Event a = [(Time,a)]
```

Now consider `stepper`, as introduced in Section 4.5:

```
stepper :: a -> Event a -> Behavior a
```

Recall that `x0 ‘stepper‘ e` starts out having the constant value `x0` and changes to a new constant value with each occurrence of `e`. It can be defined simply as follows.

```
x0 ‘stepper‘ [] = \ t -> x0
x0 ‘stepper‘ ((t1,x1) : occs’) =
  \ t -> if t1 >= t then x0
         else (x1 ‘stepper‘ occs’) t
```

The first clause says that when the event has no more occurrences, the value for every `t` is `x0`. (The Haskell notation “`\ t ->`” introduces an anonymous function with formal parameter `t`. The backslash is read as “lambda”.) The second clause applies in the case of a nonempty list and calls first element `(t1,x1)` and the remaining elements `occs’`. The sampled value is `x0` only until the time `t1` of the first occurrence, and then becomes the value `x1` of that occurrence. The value at `t` will turns out to be `x1` if there are no more occurrences before `t`.

The definitions of `stepper`, while simple and correct, turn out to be impractical. The problem comes from the context of use. A behavior is typically sampled not just once, but rather at a sequence of times. The definition of stepper given above scans through the occurrences from the beginning each time, in order to find the appropriate value. Typically, the sample times are closely spaced and strictly increasing, so this redundant work should be avoidable.

A more practical representation of behaviors to support `stepper`, is as a function from a *stream* of sample times to a stream of values.

```
type Behavior a = [Time] -> [a]
```

The overloadings are only a little more complicated:

```
sin :: RealB -> RealB
(sin b) ts = map sin (b ts)

(+) :: RealB -> RealB -> RealB
(b1 + b2) ts =
  zipWith (+) (b1 ts) (b2 ts)
```

The standard function `map` applies a function to each element of a list and packages up the list of results. The function `zipWith` is like `map` but applies to functions of two arguments.

This new representation allows an efficient implementation of ``x0 `stepper` e``, given below. Use `x0` until the first occurrence `(t1,x1)` of `e`. Then behave like ``x1 `stepper` e’``, where `e’` is the remainder of the event `e`. When there are no more occurrences, the behavior becomes constant.

```
x0 ‘stepper‘ occs@((t1,x1) : occs’) =
  \ ts@(t:ts’) ->
     if t1 >= t then
       x0 : stepper x0 occs ts’
     else stepper x1 occs’ ts

x0 ‘stepper‘ [] =
  \ ts -> map (\ t -> x0) ts
```

There are still practical problems remaining. For instance, it is usually not possible to determine when an event first occurs until it actually does occur. Consequently, the test “`t1 >= t`” in the definition of `stepper` above cannot be made. A simple solution to this problem is to inject sufficient *non-occurrences* into the representation of an event. This approach and many other issues are described in detail in [6]. We are now developing a very different, and fundamentally imperative, implementation, as well as investigating compile-time optimization.

10 Related work

The idea of a “domain-specific embedded language” is, we believe, the central message in Landin’s seminal paper “The Next 700 Programming Languages”.

> Most programming languages are partly a way of expressing things in terms of other things and partly a basic set of given things. The ISWIM (If you See What I Mean) system is a byproduct of an attempt to disentangle these two aspects in some current languages. [...] ISWIM is an attempt at a general purpose system for describing things in terms of other things, that can be problem-oriented by appropriate choice of “primitives.” So it is not a language so much as a family of languages, of which each member is the result of choosing a set of primitives. [20]

Henderson's *functional geometry* [14] was one of the first purely declarative approaches to graphics, although it does not deal with animation or reactivity. Several other researchers have also found declarative languages well-suited for modeling pictures. Examples include [21][35][12].

Haskore [18] is a purely functional approach to constructing, analyzing, and performing computer music, which has much in common with Henderson's functional geometry, even though it is for a completely different medium. The Haskore work also points out useful algebraic properties that such declarative systems possess.

Arya used a lazy functional language to model non-interactive 2D animation as lazy lists of pictures, constructed using list combinators [1]. This work was the original inspiration for our own; we have extended it to interactivity, continuous time, and many other types besides images.

TBAG modeled animations over various types as functions over continuous time [9][28]. It also used the idea of lifting functions on static values into functions on animations, which we adopted for Fran. Unlike Fran, however, reactivity was handled imperatively, through constraint assertion and retraction, performed by an application program. Like Fran, TBAG was an embedded language, but it used C++ as its host language, in an attempt to appeal to a wider audience. The C++ template facility was adequate for parametric polymorphism. The notation was in some ways even more malleable than in Haskell, because C++ overloading is *genuinely* ad hoc. On the other hand, unlike Haskell, C++ only admits a small fixed set of infix operators. The greatest failings of C++ (or Java) as a host language for a modeling language are its lack of an expression-level "let" and convenient higher-order functions. The latter may be simulated with objects, but without a notational equivalent to lambda expressions. (In the case of Java, anonymous inner classes help considerably.)

Obliq-3D is another 3D animation system embedded in a more general purpose programming language [23]. However, its host language is primarily imperative and object-oriented, rather than functional. Accordingly, Obliq-3D's models are initially constructed, and then modified, by means of side-effects. In this way it is reminiscent of Inventor [29].

Direct Animation is a library developed at Microsoft to support interactive animation [22]. It is designed to be used from mainstream imperative languages, and mixes the functional and imperative approaches. Fran and Direct Animation both grew out of an earlier design called *ActiveVRML* [4], which was an "integrated" DSL.

There are also several languages designed around a *synchronous data-flow* notion of computation, including Signal [13] and Lustre [2], which were specifically designed for control of real-time systems. In Signal, the most fundamental idea is that of a *signal*, a time-ordered sequence of values. Unlike Fran, however, time is not a value, but rather is implicit in the ordering of values in a signal. Time is thus discrete rather than continuous, with emphasis on the relative ordering of values in a data-flow-like framework. The designers of Signal have also developed a clock calculus with which one can reason about Signal programs. Lustre is more similar to Fran in style, but is also rooted in the notion of a sequence, and owes much of its nature to Lucid [33].

Hudak [16] gives several examples of DSELs (and coined the term), argues their general merits, and points to the importance of optimization through techniques like partial evaluation.

11 Conclusions

Traditionally the programming of interactive 3D and multimedia animations has been a complex and tedious task. We have argued that one source of this difficulty the use of languages to describe how to present animations. In such descriptions the essential nature of an animation, i.e., what an animation is, becomes lost in details of how to present it. Focusing on the "what" of animation, i.e., modeling, rather than the "how" of its presentation, yields a much simpler and more composable programming style. The modeling approach requires a new language, but this new language can be synthesized by adding a domain-dependent vocabulary to an existing domain-independent host language. We have found Haskell quite well-suited, as demonstrated in a collection of sample animation definitions.

A running theme of this paper has been economy of scale. We recommend making choices that amortize effort required over several uses of the fruits of that effort. The alternatives are poor quality or impracticably high cost. Specifically:

- "Modeling" vs "presentation". Graphics modeling allows reuse of a single graphics presentation engine, and temporal modeling allows reuse of a single temporal presentation engine.

- "Embedded" vs "integrated" language. Languages, if they are to be genuinely useful, require a large investment of effort. An embedded language inherits design, compilers, environment tools, and educational material from its host language.
- Composability. Because modeled, parameterized animations are neatly composable, they may be reused in a variety contexts, instead of being repeatedly reinvented with slight variations for each similar situation.

A notable exception to the necessity of modeling, embedding and composability for high quality interactive animation is in software that can sell in huge quantity, which then exploits an end-user economy of scale. The unfortunate consequence to this exception, however, is mainstreaming of the content, as in violent video games. Fortunately, however, even these games are often implemented using the modeling approach, and thus allow consumers to create new characters and worlds for them.

There are ample opportunities for future work in modeled animation, including the following.

- Multi-lingual integration. We believe that in order for Haskell, or any other non-mainstream language to make a serious contribution in the software industry, it should be cast not as a language for implementing entire *applications*, but rather *software components*. This identity then implies strong support for generating language-independent calling interfaces. As a concrete goal, one should be able to program animation modules in Haskell, compile them into binaries with COM interfaces, and then distribute them. A Java or Visual Basic programmer should then be able to wire together the Haskell-based animation components without knowing in what language they were implemented.
- Domain-specific optimization. In theory, it is possible for a domain-generic compiler to do domain-specific compilation, by using various forms of partial evaluation. We intend to investigate this approach, by using the Glasgow Haskell compiler [26], perhaps with some domain-generic enhancements.
- Notational compromises. As mentioned above, using Haskell required only a few compromises. One has to do with overloading. We cannot, for instance, use "`+`" for the addition of 2D or 3D points and vectors (or even "`.+^`", which now can be used for 2D or 3D, but not both). Similarly, we cannot use "`==`" for the lifted form of equality, applying to two like-typed animations to yield a boolean animation. Extending Haskell to allow "multi-parameter type classes" might eliminate some of these compromises.

12 Acknowledgements

Byron Cook and Dave Wile gave many helpful comments on earlier drafts to improve readability. Paul Hudak influenced my thoughts on "domain-specific embedded languages". Philip Wadler pointed out the connection to Landin's "700" paper. Todd Knoblock and Jim Kajiya helped to explore the basic ideas of modeled animation. Sigbjorn Finne helped with the implementation during a summer research internship and since. Alastair Reid made many implementation improvements. Paul Hudak, Alastair Reid, and John Peterson at Yale provided many helpful discussions about functional animation, how to use Haskell well, and lazy functional programming in general. Gary Shu Ling helped get Fran running under GHC.

13 Availability

Fran runs under Windows 95, 98 and NT 4.0, and is freely available at `http://research.microsoft.com/~conal/Fran/`.

References

[1] K. Arya, "A Functional Animation Starter-Kit", *Journal of Functional Programming*, 4(1):1-18, January 1994.

[2] P. Caspi, N. Halbwachs, D. Pilaud, and J.A. Plaice, "Lustre: A Declarative Language for Programming Synchronous Systems", in *14th ACM Symposium. on Principles of Programming Languages*, January 1987.

[3] A. Daniels, "Fran in Action!", unpublished, `http://www.cs.nott.ac.uk/~acd/action.ps`, 1997.

[4] C. Elliott, "A Brief Introduction to ActiveVRML", Technical Report MSR-TR-96-05, Microsoft Research, `http://research.microsoft.com/scripts/pubDB/pubsasp.asp?RecordID=38`, February 1996.

[5] C. Elliott. “Composing Reactive Animations”, *Dr. Dobb’s Journal*, July 1998. Expanded version with animated GIFs: `http://research.microsoft.com/~conal/fran/tutorial.htm`.

[6] C. Elliott, “Functional Implementations of Continuous Modeled Animation”, *Proceedings of PLILP/ALP ’98*. Expanded version: `http://research.microsoft.com/scripts/pubDB/pubsasp.asp?RecordID=164`.

[7] C. Elliott, “From Functional Animation to Sprite-Based Display”, *Proceedings of PADL ’99*. Expanded version: `http://research.microsoft.com/scripts/pubDB/pubsasp.asp?RecordID=190`.

[8] C. Elliott. “Declarative Event-Oriented Programming”, Technical Report MSR-TR-98-24, Microsoft Research, `http://research.microsoft.com/scripts/pubDB/pubsasp.asp?RecordID=188`, October, 1998.

[9] C. Elliott, G. Schechter, R. Yeung and S. Abi-Ezzi, “TBAG: A High Level Framework for Interactive, Animated 3D Graphics Applications”, in Andrew Glassner, editor, *Proceedings of SIGGRAPH ‘94 (Orlando, Florida)*, pages 421-434. ACM Press, `http://research.microsoft.com/~conal/tbag/papers/siggraph94.ps`

[10] C. Elliott and P. Hudak, “Functional Reactive Animation”, in *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming*, `http://research.microsoft.com/~conal/papers/icfp97.ps`

[11] S. Finne, D. Leijen, E. Meijer, S.L. Peyton Jones, “H/Direct: A Binary Foreign Language Interface for Haskell”, *International Conference on Functional Programming, 1998.* `http://research.microsoft.com/Users/simonpj/Papers/hdirect.ps.gz`.

[12] S. Finne and S.L. Peyton Jones, “Pictures: a Simple Structured Graphics Model”, *Proceedings of the Glasgow Functional Programming Workshop*, July 1996.

[13] T. Gautier, P. Le Guernic, and L. Besnard, “Signal: A Declarative Language for Synchronous Programming of Real-Time Systems”, in Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, edited by G. Goos and J. Hartmanis, pages 257-277. Springer-Verlag, 1987.

[14] P. Henderson, “Functional Geometry”, *ACM Symposium on Lisp and Functional Programming*, pp. 179-187, 1982.

[15] P..M. Hubbard, “Interactive Collision Detection”, *Proceedings of the IEEE Symposium on Research Frontiers in Virtual Reality*, October 25-26, 1993, pp. 24-31.

[16] P. Hudak, “Modular Domain Specific Languages and Tools”, *Fifth International Conference on Software Reuse*, June 1998.

[17] P. Hudak and J. H. Fasel, “A Gentle Introduction to Haskell”, *SIGPLAN Notices*, 27(5), May 1992. See `http://haskell.org/tutorial` for latest version.

[18] P. Hudak, T. Makucevich, S. Gadde and B. Whong, “Haskore Music Notation – An Algebra of Music”, *Journal of Functional Programming*, 6(3), pp. 465-483, May 1996.

[19] P. Hudak, S.L. Peyton Jones and Philip Wadler (editors), “Report on the Programming Language Haskell, A Non-Strict Purely Functional Language (Version 1.2)”, *SIGPLAN Notices*, 27(5), March 1992b. See `http://haskell.org/report` for latest version.

[20] P.J. Landin, “The Next 700 Programming Languages”, Communications of the ACM, 9(3), March 1966, pp. 157-164.

[21] P. Lucas and S. N. Zilles, “Graphics in an Applicative Context”, CS Res. Rep., Number RJ 5542 (56566), IBM Almaden Res. Center, March 1987.

[22] Microsoft, “DirectAnimation SDK”, `http://www.microsoft.com/directx/dxm/help/da/c-frame.htm#default.htm`

[23] M.A. Najork and M.H. Brown, “Obliq-3D: A High-Level, Fast-Turnaround 3D Animation System”, IEEE Transaction on Visualization and Computer Graphics, 1(2), June 1995.

[24] M. Pesce, VRML Browsing and Building Cyberspace: the Definitive Resource for VRML Technology, New Riders Publishing, 1995.

[25] J. Peterson, C. Elliott and G. Shu Ling, “Fran User’s Manual”, `http://www.haskell.org/fran/fran.html`, 1998.

[26] S. Peyton Jones and A. Santos, “Compiling Haskell by Program Transformation: a Report

from the Trenches", ESOP '96: 6th European Symposium on Programming, Linköping Sweden, April 22—24, 1996, Lecture Notes in Computer Science, Vol. 1058, Springer-Verlag Inc. http://research.microsoft.com/Users/-simonpj/Papers/comp-by-trans.ps.gz .

[27] S. Peyton Jones and P. Wadler. "Imperative functional programming", *20'th Symposium on Principles of Programming Languages*, ACM Press, Charlotte, North Carolina, January 1993.

[28] G. Schechter, C. Elliott, R. Yeung and S. Abi-Ezzi, "Functional 3D Graphics in C++ - with an Object-Oriented, Multiple Dispatching Implementation", in *Proceedings of the 1994 Eurographics Object-Oriented Graphics Workshop*. Springer Verlag, http://research.-microsoft.com/~conal/papers/eoog94.ps

[29] P.S. Strauss, "IRIS Inventor, A 3D Graphics Toolkit", in *Proceedings of the OOPSLA '93 Conference on Object-oriented Programming Systems, Languages and Applications*, pp. 192-200,October 1993.

[30] Simon Thompson, *Haskell: The Craft of Functional Programming*. Addison-Wesley. 1996. http://stork.ukc.ac.uk/-computer_science/Haskell_craft/.

[31] S. Thompson, "A Functional Reactive Animation of a Lift using Fran", TR 5-98, Computing Laboratory, University of Kent, May 1998. http://www.cs.ukc.ac.uk/pubs/1998/-583/index.html.

[32] The VRML Consortium Incorporated, VRML97 International Standard, http://www.vrml.-org/Specifications/VRML97/part1/-nodesRef.html.

[33] W.W. Wadge and E.A. Ashcroft, *Lucid, the Dataflow Programming Language*, Academic Press, U.K. 1985

[34] P. Wadler, "The essence of functional programming", *19'th Symposium on Principles of Programming Languages*, ACM, January 1992.

[35] S. N. Zilles, P. Lucas, T. M. Linden, J. Lotspiech, and A. Harbury, "The Escher Document Imaging Model", *ACM Conference on Document Processing Systems*, pp. 159-168, ACM Press, December 1988.

Conal Elliott received a PhD from Carnegie Mellon University in 1990, concentrating on program derivation and higher-order unification. He has been a member of the Graphics group in Microsoft Research since 1994, and before that worked at Sun Microsystems. His research interests include computer graphics, animation, interaction, and lazy functional programming.