Modeling Interactive 3D and Multimedia Animation with an Embedded Language

Conal Elliott
Microsoft Research

http://www.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.

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 an alternative to "presentation-oriented" programming, namely "modeling", 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 almost 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, in order to model animations over a broad range of types. 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. (See Elliott and Hudak [1997] for 2D examples.)

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 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. 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, we are now fortunate enough to have various 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 starts with a few examples of modeled animations. Section 3 introduces the notions of presentation and modeling for non-animated 3D graphics, and looks at some concrete benefits. Section 4 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 5 considers the pragmatics of creating a new domain specific language (DSL), and motivates the DSEL approach. Section 6 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 Examples

In this section we present a handful of modeled animations, in order to make later discussion more concrete.

2.1 Static models

To start, we import a simple 3D model of a sphere from “X file” format.

```
sphere :: GeometryB
sphere :: GeometryB
sphere = importX "sphere1.x"
```

The type `GeometryB` represents 3D geometry animations. (This “animation” happens to be a “static”, i.e., not time-varying, one.) Similarly, we import a teapot model. However, the teapot is in an awkward orientation, so we adjust it after importing, rotating around the X axis by an angle of $-\pi/2$:

```
teapot :: GeometryB
teapot =
 rotate3 xVector3 (-pi/2) **%
 importX "tpot2.x"
```

Note that in Haskell, function application binds more tightly than all infix operators. Here are the types of the modeling vocabulary we used:

```
xVector3 :: Vector3B
rotate3  :: Vector3B -> RealB
         -> Transform3B
(**%)    :: Transform3B -> GeometryB
         -> GeometryB
```

The constant `xVector3` is the unit vector pointing in the positive X direction. `rotate3` takes an axis vector and a number and yields a 3D transform. The operator `**%` applies a 3D transform.

2.2 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.

```
redSpinningPot =
  rotate3 yVector3 time **%
  withColorG red teapot
```

The new features are “time”, the unit Y vector and application of a color to a geometric model:

```
time :: RealB
yVector3 :: Vector3B
withColorG :: ColorB -> GeometryG
           -> GeometryG
```

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`, being `**%` are combinators that map functions of time to functions of time.

2.3 Generalizing

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

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

spinPot potColor potAngle =
  rotate3 yVector3 potAngle **%
  withColorG potColor teapot
```

We will make use of the `potSpin` function in a series of three interactive 2D animations.

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

When an 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`. First, we will give their types and an informal description of their purpose.

```
potSpin1, potSpin2 :: RealB -> User
                   -> GeometryB

withSpin :: (RealB -> User -> GeometryB)
         -> User -> ImageB
```

The two `potSpin` functions take as arguments an animated number, which will be related to the rotation angle passed to `spinPot`, and a user from which to get input. In the simplest case, just ignore the user, use red for the pot color, and pass 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.

```
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 combined with the instruction text image. The function renderGeometry takes geometry and camera (animated as always), and yields a 2D animation:

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

2.4 A more interesting pot spinner

Before looking into the definition of `grow`, we will see the second pot-spinning geometry producer, 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 coordinates.
- The "angle" argument generated by `grow` and passed by `withSpin` is integrated, and so is interpreted as the rate of change of the angle.

The definition:

```
potSpin2 potAngleSpeed u =
  spinPot potColor potAngle ‘unionG‘
light
   where
    light = rotate3 yVector3 (pi/4)   **%
            translate3 (vector3Spherical
                            2 time 0) **%
            uscale3 0.1               **%
            withColorG white (
             sphere ‘unionG‘ pointLightG)
    potColor =
       colorHSL (sin time * 180) 0.5 0.5
    potAngle = integral potAngleSpeed u
```

Note the expression "`sin time * 180`" used in defining the teapot's color. The meaning of `sin` and "`*`" are not the usual ones, operating on numbers, but rather counterparts "lifted" to consume and produce number-valued animations (of type `RealB`). Even the numeric literal `180` is taken to mean an unchanging number-valued animation (having type `RealB`). Haskell's overloading ability, based on type classes is responsible for this great syntactic convenience. Several dozen functions have been lifted in this way, so that, for instance, `sin` and "`*`" not only have the usual types

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

but also

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

2.5 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 of the value generated by `bSign`, defined below, which produces an animated number that 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) u
```

(The reason that even `integral` takes a user argument is that integration is done numerically, and must somehow know how hard to work on the approximation.)

The `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 =
  condB (leftButton u)
        (constantB left)
     (condB (rightButton u)
            (constantB right)
        (constantB none))
```

Some explanation: the use of a lower-case type name (“a”) above means that `selectLeftRight` is polymorphic, applying to any type of argument. The function `condB` is a behavior-level conditional, taking an animated boolean and two animated values, and choosing between the two continuously. The Fran primmitive `constantB` turns a regular “static” value into a constant animated value (as required here by `condB`). 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, using the event handling operator “**`-=>`**” and the event merging operator “**`.|.`**”. As a result, the constructed event occurs with value `True` whenever go occurs and with value False whenever `False` occurs. (Note: the event operators are described in Elliott and Hudak [1997], 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 functions of a `User` rather than a start time.)

2.6 Adding instructions

Finally, to produce 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 =
  moveXY 0 (-1) (
   withColor yellow (
    simpleTextImage messageB ))
 where
   messageB =
     selectLeftRight
       "Use mouse buttons to \
            \control pot’s spin"
       "left" "right" u
```

Many more examples of functional animation may be found in Elliott and Hudak [1997], Elliott [1997], and Daniels [1997]. See also the user’s manual (Peterson and Ling [1997]), which contains precise types and informal meanings of the embedded animation modeling vocabulary and still more examples.

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.

3 Presentation vs. modeling for 3D geometry

The practice of 3D graphics programming has made tremendous progress over the past three decades. Originally, 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 of course underlying hardware). With a rendering library, such as GL by Silicon Graphics, 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 naturally construct models, because their end users think in terms of models, and typically have neither the expertise nor interest in programming model 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 do not allow interference among components. 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 implement, optimize, and port their programs for all required platforms. In other cases, as in the case of current state-of-the-art commercial video games, the resources and expertise are available and well 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.

4 Modeling vs. presentation for animation

The conventional approach to constructing richly interactive animated content 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 types 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. (In fact, for today's PC capabilities, sound and 2D imagery are more important than 3D geometry.) 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 (Pesce [1995]) 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. 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 exactly 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 the 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.

5 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, 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 differ 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 us 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, 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 (Peyton Jones and Santos [1996]), 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 sufficiently suitable existing host language. We now take a closer look at the question of what features constitute suitability.

6 Choosing a host language for modeled animation

We have found a variety of host language features to be helpful for animation modeling, while others were harmful. The helpful features include the following, some of these features are obvious from a programming language perspective, but are in fact missing or very weakly present in popular model formats 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 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 really 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 an 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, such as "`untilB`", "`==>`" 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. We even overload constants, e.g., "`pi`" and "`1`", to denote animations (though not very animated ones). We also introduce 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 non-infix 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 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.

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. Common language features that are statement-oriented, and which thus do *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.
- Conditional statements.
- *Sequential iteration*. Really just a compact way to specify possibly infinite sequencing and conditional execution.

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* (Hudak et al [1992b], Hudak and Fasel [1992a]) has the largest following, has an international standard (Haskell 1.4) and is undergoing considerable development. For these reasons, we have chosen Haskell for our own implementation of the ideals of modeled animation. Other languages can be used as well, with varying tradeoffs. For example, Java is more popular than Haskell, and while predominately statement-oriented, it does support garbage collection.

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 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.

Aside from issues of familiarity, there will always be an important role for imperative computation in the construction of *complete* applications, which is best described using statement-oriented programming languages. One then 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.

7 Related work

The idea of an "domain-specific embedded language" is, we believe, the central message in Landin's seminal "700" paper:

> 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. (Landin [1966]).

Arya [1994] used a lazy functional language to model non-interactive 2D animation as lazy lists of pictures, constructed using list combinators. 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 (Elliott et al [1994], Schechter et al [1994]). It also used the idea of lifting function 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 the absence of higher-order functions. The latter may be simulated with objects, but without a notational equivalent to lambda expressions.

Obliq-3D is another 3D animation system embedded in a more general purpose programming language (Najork and Brown [1995]). 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 (Strauss [1993]).

Direct Animation is a library developed at Microsoft to support interactive animation (Microsoft [1997]). It is designed to be used from mainstream imperative languages such as Java, and mixes the functional and imperative approaches. Fran and Direct Animation both grew out of an earlier design called *ActiveVRML* (Elliott [1996]), which was an "integrated" DSL.

There are also several languages designed around a *synchronous data-flow* notion of computation, including Signal (Gautier et al [1987]) and Lustre (Caspi et al [1987]), 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. By its very nature 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 a language similar to Signal, rooted again in the notion of a sequence, and owing much of its nature to Lucid (Wadge and Ashcroft [1985]).

8 Conclusions

Traditionally the programming of interactive 3D and multimedia animations has been a complex and tedious task. We have argued that one source of difficulty is that the languages used are suited to describe how to present animations, and 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 impractically 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 a kind of mainstreaming of the content, as in violent video games. Fortunately, however, even these games are often implemented using the modeling approach, and 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 (Peyton Jones and Santos [1996]), 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.

9 Acknowledgements

My thoughts on "domain-specific embedded language" have been greatly influenced by Paul Hudak. 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. 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. Byron Cook gave many helpful comments on an earlier draft to improve readability.

10 Availability

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

References

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

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

Anthony Daniels [1997], "Fran in Action!", in preparation, http://www.cs.nott.ac.uk/~acd/action.ps

Conal Elliott [February 1996], "A Brief Introduction to ActiveVRML", Technical Report MSR-TR-96-05, Microsoft Research, ftp://ftp.research.microsoft.com/pub/tr/tr-96-05.ps

Conal Elliott [1997], "Composing Reactive Animations", To appear in *Dr. Dobb's Journal*, http://www.research.microsoft.com/~conal/fran/tutorial.htm .

Conal Elliott, Greg Schechter, Ricky Yeung and Salim Abi-Ezzi [July 1994], "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://www.research.microsoft.com/~conal/tbag/papers/siggraph94.ps

Conal Elliott and Paul Hudak [June 1997], "Functional Reactive Animation", in *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming*, http://www.research.microsoft.com/~conal/papers/icfp97.ps

Thierry Gautier, Paul Le Guernic, and Loic Besnard [1987], "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.

Paul Hudak and Joseph H. Fasel [May 1992a], "A Gentle Introduction to Haskell". *SIGPLAN Notices*, 27(5). See http://haskell.org/tutorial/index.html for latest version.

Paul Hudak and Simon L. Peyton Jones and Philip Wadler (editors) [March 1992b], "Report on the Programming Language Haskell, A Non-strict Purely Functional Language (Version 1.2)", *SIGPLAN Notices*. See http://haskell.org/report/index.html for latest version.

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

Microsoft [1997], DirectAnimation, in the Microsoft DirectX web page, http://www.microsoft.com/directx**.**

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

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

John Peterson and Gary Shu Ling [1997], "Fran User's Manual", http://www.haskell.org/fran/fran.html

Simon Peyton Jones and Andre Santos [1996], "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://www.dcs.gla.ac.uk/fp/authors/Simon_Peyton_Jones/comp-by-trans.ps.gz

Greg Schechter, Conal Elliott, Ricky Yeung and Salim Abi-Ezzi [1994], "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://www.research.microsoft.com/~conal/papers/eoog94.ps

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

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