Declarative Event-Oriented Programming

Conal Elliott

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

ABSTRACT

Events play an important role in the construction of most software that involves interaction or simulation. Typically, programmers make use of a fixed set of low level events supplied by a window system, possibly augmented with timers and UI components. Event handling generally involves some interpretation of these event occurrences, followed by external actions or modifications to program state.

It is possible to extend the event paradigm by using an *algebra of events* to synthesize new kinds of events tailored specifically for a domain or application. In turn, these new events may be used to synthesize yet others, and so on, to an arbitrarily sophisticated degree. This programming paradigm, which we call *event-oriented programming*, aids in the factoring of programs into understandable and reusable pieces.

We propose a *declarative* approach to event-oriented programming, based on a powerfully expressive event language with a lightweight notation. We illustrate this new approach through the design of an interactive curve editor.

1. INTRODUCTION

The notion of *event* is central in the construction of most software that involves interaction or simulation. Such software is typically organized around a centralized event queue and the event loop that removes and acts on events. For example, under Microsoft Windows®, the operating system posts messages describing occurrences of user interaction events, such as mouse clicks, key presses, and window resizing. A Windows program repeatedly removes a message from the queue, and passes it to the window procedure, which examines its type and invokes appropriate application code. It is also often useful to add higher-level event types. Sometimes, as in the case of menus, buttons and dialog boxes, these new types are packaged up into reusable widget libraries. In other cases, as in collisions during a game, the higher-level events are about an application's content rather than its user interface.

While the notion of an event is natural in many applications, the support provided by modern window systems has some serious weaknesses that interfere with program construction and maintenance. Some of these weaknesses stem from the fact that events are often just symbols with no intrinsic meaning.

Consider the event of the user pressing the left mouse button. In Windows, this event is called `WM_LBUTTONDOWN`, and each occurrence contains a snapshot of the mouse position and the state of the control and shift keys. If an application should respond to all left button presses that its window receives, the programmer will write some window procedure code that looks like the following:

```
switch (msg) {
  ...
  case WM_LBUTTONDOWN:
    keyState = wParam;
    xPos = LOWORD(lParam);
    yPos = HIWORD(lParam);
    ...  // response goes here
    break;
  ...
}
```

Suppose that an event of interest is one of the following:

- the user pressing an arrow key;
- the left button being pressed while the mouse is over an object of interest;
- the same example, but with the event data being an index of the selected object;
- the right and left buttons being pressed within 50 milliseconds of each other; or
- the collision of two objects, with the event data being the instantaneous relative velocity.

The nature of events as intrinsically meaningless identifiers prevents any of these examples from being encapsulated as an event. Of course, an application designer can still conceive of these events and implement detection and response to them. In the implementation, however, the represented events remain trivially simple, while the response code and supporting data structures and state variables become increasingly complex.

GUI frameworks like Microsoft's MFC® and Visual Basic® help somewhat by breaking up event handling into separate methods, each with its own, tailored interface. However, the set of possible event response methods is still limited to the same generic set. Moreover, all consequences of any given event are handled in a single handler method.

An object-oriented representation of events, as in the Java 1.1 AWT event model, addresses some of these problems. Event response programming is even less monolithic than in MFC or Visual Basic, since a program may conveniently separate different responses to an event into different event "listeners" that may be registered and unregistered with an event, dynamically [13]. When an event occurs, a corresponding method is automatically invoked on each registered listener. The use of inner classes, a feature added in Java 1.1, is a considerable notational help. It is also possible to define arbitrary new kinds of events. However, doing so is tedious, because the framework's `AWTEvent-Multicaster` class (which manages listener lists) can only supply a fixed set of overloadings for the listener `add` and `re-move` methods. New kinds of events may easily have signatures that do not match any of the given overloadings. In such a case, the programmer of the new kind of event must also implement all of the list management needed to support multiple listeners.

This paper introduces an alternative paradigm we call *declarative event-oriented programming (DEOP)* that addresses the shortcomings mentioned above. DEOP provides an algebra of event combinators with a simple semantic model and embedded in a functional host language. The semantics of events is purely compositional, relying in no way on names.

These ideas in this paper have been implemented in Fran ("Functional reactive animation"), a library for use with the functional programming language Haskell [17]. Previous papers have presented Fran's basic building blocks for reactive animation [12], emphasized behaviors rather than events [9], application to 3D graphics [8], language embedding [8], and implementation [10][11]. The main new contribution of this paper is the explicit focus on declarative event-oriented programming. We attempt to convey this new paradigm for programming interaction applications, illustrate its use by means of a running example, and contrast it with the dominant but ill-structured approach, which is based on imperative callback procedures.

2. DECLARATIVE EVENT-ORIENTED PROGRAMMING

The essential idea in this paper is to enrich the popular notion of events into a powerfully expressive algebra that includes not only primitive events, but also operators for building up more complex events from simpler ones. Benefits of this approach include the following.

- *Modularity/reuse*. Programmers can factor tasks involving interactivity and reactivity into easily understood pieces, and develop libraries of reusable interaction components. Event handling is factored into a set of independent, incremental enhancements, encapsulated within the events that make up a program.
- *Lightweight notation.* Programs describe high level events as *what they are*, not as sequences of steps to decode event data, test conditions, maintain global data structures, post events, etc. The notational style is algebraic, composing operators in succinct nested expressions, as in commonplace calculations on numbers.
- *Flexible naming.* Events are completely independent from their naming, which fully exploits the naming mechanisms supported by a programming language, including lexical scoping, inclusion in objects and data structures, selective exportation from modules, and linking. Accidental name collisions are caught by the compiler or linker. As importantly, an event may be anonymous, being described merely in the construction of a more complex event. (The same property for numerical computation is a key advantage of high-level programming languages over assembly language.)
- *Safety.* Definition and use of events are type-checked at compile time. In contrast, posting and decoding a Windows event requires unsafe type casting to and from the generic `lParam` and `wParam`.

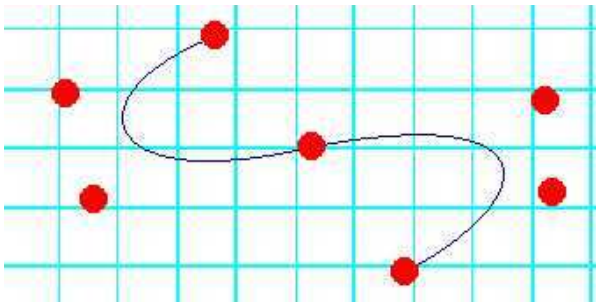

Figure 1. Poly-Bezier curve

In contrast with the conventional approach, the events described in this paper have intrinsic meaning, based on a simple but quite general model. *The meaning of an event is a (possibly infinite) sequence of occurrences, each of which is a time/value pair.* [1] Another notion, useful in conjunction with events, is the *behavior*, whose meaning is simply a function of continuous time.

The event-forming operators discussed in this paper fall into the following categories:

- transforming an event's data;
- forming the union of two events; and
- filtering out some event occurrences.

There are other operators as well, not illustrated in this paper:

- monitoring time-varying conditions; and
- sequential chains of events.

3. THE EXAMPLE

In the next several sections, we will build up an interactive "poly-Bezier" curve editor. Bezier curves are popular in computer graphics and manufacturing because they are well behaved and visually pleasant. Each curve is defined by four *control points*. The first and fourth lie at the curve's endpoints, while the middle two typically lie off of the curve. These inner control points allow the user to tug at the curvature. A poly-Bezier curve is the union of a sequence of simple Bezier pieces, in which the first control

[1] For this reason, "event source" may be a more appropriate name. Also, note that this model is a departure from that given in [12], which modeled an event as a single occurrence. The stream model allows much more powerful encapsulation.

point of each piece after the first one coincides with fourth control point of the previous piece, as in Figure 1.

4. RENDERING CONTROL POINTS

The first step in our development is an editor for a single control point. We will represent a control point as a pair containing a trajectory and a status flag indicating whether it may be grabbed.

```
type CPoint = (Point2B, BoolB)
```

As a convention, the Fran type names "`Point2B`", "`BoolB`", etc are synonyms for "`Behavior Point2`", "`Behavior Bool`", etc, meaning time-varying 2D points, Booleans, etc.

We will want a control point's appearance to indicate whether it can be grabbed. Because this is the first example code, however, we will first consider how to render just a point, without this indication. We "render" a control point, i.e., turn it into an image animation (`ImageB`), simply as a small shape, moved according to the given trajectory. The shape is a star when "excited" (grabbable) and a circle otherwise.[2]

```
renderCPoint :: CPoint -> ImageB
renderCPoint (pos, excited) =
  moveTo pos (
    stretch pointSize (
      ifB excited star circle))

pointSize :: RealB
pointSize = 0.07
```

Functions like `moveTo`, `stretch`, and `renderCPoint` operate on time-varying values (behaviors). One may think of these operations as building up a "temporal display list" to be traversed iteratively by Fran's run-time system during presentation.

Lesson

Motion and animation are not the byproducts of program execution, as in traditional imperative programming, but first-class values, expressed directly.

5. A SIMPLE CONTROL POINT EDITOR

Next comes the control point editor, which takes a static (not time-varying) start point and user input stream (represented in Fran by a value of type `User`), and synthesizes an interactive control point. The user can grab the control point with the left mouse button when the mouse cursor is close enough to it (within two radii).[3]

```
editCPoint :: User -> S.Point2 -> CPoint
editCPoint u p0 = (pos, closeEnough)
 where
   pos, lastRelease :: Point2B
   pos = ifB grabbing (mouse u) lastRelease
   lastRelease =
     stepper p0 (release `snapshot_` pos)

   closeEnough, grabbing :: BoolB
   closeEnough =
     distance2 pos (mouse u) <* grabDistance
   grabbing = stepper False (grab    -=> True
                        .|.  release -=> False)

   grab, release :: Event ()
   grab    = lbp u `whenE` closeEnough
   release = lbr u `whenE` grabbing

grabDistance :: RealB
grabDistance = 2 * pointSize
```

Here are brief, informal descriptions of each definition within the "`where`" clause, from top to bottom. Technical details follow.

- The position is defined as a behavior-level (time-varying) conditional: the mouse position when the user is grabbing, and the position of the last release otherwise.
- The last release position is the start point at first, and then becomes a snapshot of the control point position whenever released.
- The control point position is considered "close enough" to grab when it is within `grabDistance` (two radii) of the mouse's position. (The "`<*`" operator is the behavior-level version of the familiar less-than operator. For most functions and operators on numbers, the usual name is overloaded to mean the behavior-level version as well. In other cases, an asterisk or "`B`" is added to the end of the usual name.)
- The boolean grabbing behavior is piecewise constant (a "step function"), initially false. It becomes true whenever the user grabs the control point, and false when the user releases it.
- The grab event is considered to occur when the user presses the left button when "close enough".
- Similarly, the release event is considered to occur when the user releases the left button while grabbing.

---

[2] *Haskellisms:* The first line of this example is a type declaration for `renderCPoint`, saying that it is a function from `CPoint` values to animations. Although in almost all cases, the Haskell compiler or interpreter can infer types automatically, we give types explicitly to clarify the examples. In place of formal parameters, one may give patterns to be matched. In the second line, pattern matching pulls apart the single, pair-valued argument to `renderCPoint`, extracting the components `pos` and `excited`. Application of a function `f` to arguments `x`, `y`, ... is written simply "`f x y ...`". For example, the body of the definition of `renderPoint` is the application of the Fran `moveTo` function to two arguments, with the first being `pos` and the second being the application of the function `stretch` to `pointSize` and `circle`.

[3] *Haskellisms:* The first line in this example says that `editCPoint` is a function that takes a user value and a static point, and yields a control point. (The "`->`" type operator is right-associative, so literally, `editCPoint` takes one argument and produces a function that takes another argument.) Function names made up of alphanumeric characters may be used as infix operators by surrounding them with backquotes, as in `` `whenE` `` below. Names made up of symbol characters, such as "`.|.`" below, are treated as infix by default. The trivial type "`()`" is similar to C's `void` type. Its one value contains no information.

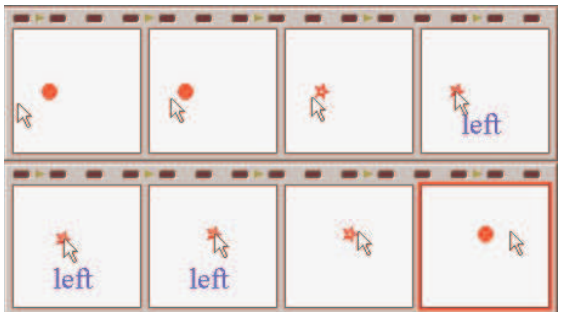

Figure 2. editCPoint

We can try out our simple control point editor by writing a tester function. This function combines `editCPoint` and `renderCPoint`, uses the static 2D origin as the starting position, and places the result over a white background. Note that an "interactive animation" is a function that maps user-input streams to animations.

```
editCPointTest :: User -> ImageB
editCPointTest u =
 renderCPoint (editCPoint u S.origin2) `over`
 whiteIm

whiteIm :: ImageB
whiteIm = withColor white solidImage
```

Figure 2 shows the result, with user input echoed at the bottom of the window.[4]

The definition of `editCPoint` above illustrates declarative event-oriented programming. Some of the events are named (`grab` and `release`), while others are expressed anonymously in the construction of the behaviors `grabbing` and `lastRelease`. Although one may read these definitions idiomatically as in the informal explanation above, their meanings come precisely from the composition of independently meaningful operations. Consider first the definition of `grab`. The function `lbp` maps a user value into an event each of whose occurrences indicates that the given user's left mouse button is being pressed. (Recall that an event is a *sequence* of occurrences, each of which is a time/value pair.[5]) There is no useful explicit data, so it has the following type declaration.

```
lbp :: User -> Event ()
```

The function "`whenE`" (here used in infix) acts to filter out some of the occurrences of an event. It allows through only those occurring when a boolean behavior is true. It has the following type declaration.[6]

```
whenE :: Event a -> BoolB -> Event a
```

The definition of `grabbing` uses two more event-building operators: "`-=>`" and "`.|.`". The "`-=>`" operator has higher syntactic precedence (binding more tightly), and has type

```
(-=>) :: Event a -> b -> Event b
```

It replaces the value in every occurrence of an event with a given value. Here, the value from `grab`, which is always the trivial value (of type "`()`"), is replaced by `True`, and the value from `release`, also trivial, is replaced by `False`, producing two new boolean events.

The merge operator "`.|.`" has type

```
(.|.) :: Event a -> Event a -> Event a
```

Given events `e1` and `e2`, the new event `e1.|.e2` has as its occurrences the union of occurrences of `e1` and `e2`. (For simultaneous occurrences, order in `e1` and `e2` is preserved and those of `e1` appear before those of `e2` in the result.) In our example, the merged event occurs with value `True` whenever the `grab` event occurs and with value `False` whenever the `release` event occurs. The `grabbing` boolean behavior is then defined by applying the `stepper` function, which has the following type.

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

The behavior it creates starts out with the value given by the first argument and changes with each occurrence of the event argument. In this case, the `grabbing` behavior starts `False` and switches to `True` when the control point is grabbed and `False` when the control point is released.

Finally, the definition of `lastRelease` illustrates the use of events to make behavior snapshots. The event function used is

```
snapshot_ :: Event a -> Behavior b -> Event b
```

An event "`e `snapshot_` b`" occurs whenever `e` occurs, but its values are the values of `b` at the occurrence times. In our example above, the control point's position is snapshotted on each release, yielding a point-valued event, which is used to make the piecewise-constant `lastRelease` behavior.

Lessons

This first example of declarative event-oriented programming illustrates a few lessons.

- *Separate* the model from the presentation (here a `Point2B` and an `ImageB`, respectively).
- *Remember* with `snapshot_` and `stepper`.
- *Enrich and merge* events with "`-=>`" and "`.|.`".
- *Specialize* with `whenE`.

6. A FIRST CURVE EDITOR

It is now a simple matter to put together a curve editor, representing curves as lists of control points. First define the appearance of a curve: rendered control points over a blue poly-Bezier curve.

---

[4] Figures like this one show snapshots of an animation. Read the top row from left to right, then the second row, and so on. To save space, many intermediate frames have been removed. See `http://research.microsoft.com/~conal/ppdp00` for animated versions of all of the figures in this paper.

[5] Typically, early occurrences need to be accessible before later ones can be known, so the sequences must be represented lazily.

[6] *Haskellism*: In type declarations, non-capitalized names, like `a` here, are *type variables*, indicating polymorphism, i.e., the ability to work with all types. For instance, `whenE` applies to any kind of event and yields an event of the same type.

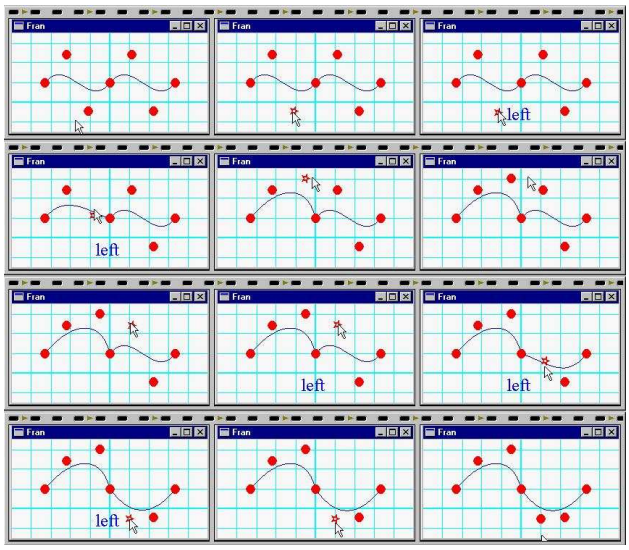

Figure 3. editor

We use the Fran function `overs` to overlay the list of rendered control points, and its binary version `over`, to combine with the Bezier image.[7]

```
renderCurve :: [CPoint] -> ImageB
renderCurve cpoints =
  overs (map renderCPoint cpoints)   `over`
  withColor blue (
   polyBezier (map fst cpoints))
```

Next, define a curve editor. (The `editCPoint` function is being partially applied, yielding a function that is mapped over the initial points.)

```
editCurve :: [S.Point2] -> User -> [CPoint]
editCurve initPoints u =
  map (editCPoint u) initPoints
```

Finally, put the pieces together: given an initial list of static points, make an interactive curve, render the result, and place it over a graph paper background.

```
editor :: [S.Point2] -> User -> ImageB
editor initPoints u =
  renderCurve (editCurve initPoints u) `over`
  graphPaper
```

Figure 3 shows a sample use of `editor`.

Lesson

Look for smaller constituent problems (e.g., point editing), especially those repeated in the larger problem. Solve the subproblems, test your solutions, and then compose them.

[7] *Haskellisms:* For any type `a`, the type `[a]` contains all lists whose members are all of type `a`. Thus `renderCurve` is a function from control point *lists* to image animations. The `map` function takes a function and a list of values and yields a new list made up of the given function applied to each member of the given list. The `fst` function extracts the first member of a pair, in this case each control point's position.

7. A COMPARISON

Consider how one might implement the curve editor in the event loop style, say under Windows. Preserving most of the conceptual structure of the version above, it might work as follows.

- Declare a global array of control point state data structures, initialized when a curve file is loaded. Minimally, each state could consist of a point and a boolean, corresponding to `lastRelease` and `grabbing` respectively.
- On `WM_LBUTTONDOWN`: For each control point, if its position is close enough to the mouse position, set the grabbing flag to true.
- On `WM_LBUTTONUP`: For each control point, if its grabbing flag is true, set its position to the current mouse position and set its grabbing flag to false.
- On `WM_TIMER`: For each control point, if its grabbing flag is set, set its position. to the mouse's.
- On `WM_MOUSEMOVE`: If any control point has its grabbing flag set, mark the window as needing to be repainted, which will generate a `WM_PAINT` event. (A simpler, but less efficient, alternative is to to generate `WM_PAINT` events in response to a timer or in "idle processing".)
- On `WM_PAINT`: First draw the graph paper background and the poly-Bezier curve. Then iterate through the control points. For each one, draw it at a position that is equal to the mouse's position if control point's grabbing flag is set, and equal to the `lastRelease` point otherwise.

Notice that this event loop version would be much less modular and concise than the version given in Sections 5 and 6. The complexities of the point editor are exposed to the curve editor rather than neatly encapsulated. Consider what would happen if more features were added, such as sensitivity of the curve segments. The reactivity to user input of all kinds of elements would be mixed together in the single monolithic event loop.

We can also compare to an object-oriented design. In Java, such a design could represent the control point states as objects in a new `CPoint` class derived from the appropriate event listener classes, and adding a `display` method. After these control point objects are created, they would be inserted into the appropriate listener lists. The reactions sketched just above to `WM_LBUTTONDOWN`, etc would be moved into the `CPoint` event handler methods.

Like declarative programming, object-oriented design encourages explicit modeling of the conceptual entities in a task, moving complexity out of monolithic bodies of code and into well-insulated pieces that may then be composed. However, object-oriented programs typically embrace the imperative programming style within their methods, and in doing so compromise the principle of modeling. Moreover, object-oriented languages impose much more notational overhead. In this light, the style proposed in this paper may be seen as a stateless and extremely fine-grained form of object-orientation, with an unusually lightweight notation. (See [29] for a description of a fine-grained object-oriented implementation of a predecessor of Fran.)

8. CONTROL POINT EDITING WITH *UNDO*

We now add the ability to undo an unlimited number of editing operations. Undo can be implemented by maintaining a stack of information with which to reset the state to what it was just before the change. Edits generate pushes and undos generate pops.

Instead of building stack maintenance into our editor, we will implement a polymorphic, unbounded *stack manager*. This stack manager is a sort of *server*, accepting push and pop requests, and providing replies to "legitimate" pop requests, i.e., those made when the stack is non-empty. The stack itself is hidden in the stack manager's implementation.[8]

```
stacker :: Event a -> Event () -> Event a
stacker push tryPop =
  legitPop `snapshot_` headB stack
 where
   legitPop :: Event ()
   legitPop = tryPop `whenE` notB (nullB stack)
   -- changeStack :: Event ([a] -> [a])
   changeStack = legitPop -=> tail
              .|. push     ==> (:)
   -- stack :: Behavior [a]
   stack = stepAccum [] changeStack
```

The definition works by maintaining a list-valued behavior called `stack`, which is used and defined as follows.

- The `stacker` function returns an event whose occurrences contain a snapshot of the top of the stack at each legitimate pop.
- A legitimate pop is an attempt when the stack is not empty.
- The `changeStack` event's values are *functions* from stacks to stacks. A legitimate pop request leads to popping (via `tail`), and a push with value *x* leads to pushing *x* (by partially applying the Haskell cons function "`:`" to *x*).
- The stack starts out empty, and changes whenever `changeStack` occurs. The function associated with an occurrence of `changeStack` is applied to the previous value of the stack, giving a cumulative effect.

The "`==>`" operator is similar to "`-=>`", but applies a given function to each occurrence value of its event argument. It has the following type.

```
(==>) :: Event a -> (a -> b) -> Event b
```

The function `stepAccum` builds a piecewise-constant behaviors by cumulatively applying occurrences of a function-valued event, and has is the following (polymorphic) type:

```
stepAccum :: a -> Event (a -> a) -> Behavior a
```

Given the stacker function, it is now almost trivial to add undo to the control point editor. The only changes (shown in bold) are to consider undo events to be releases, to push on every `grab` event, and try to undo when the user presses control-Z (as detected by the Fran function `charPress`).

```
editCPointUndo :: User -> S.Point2 -> CPoint
editCPointUndo u p0 = (pos, closeEnough)
 where
   pos, lastRelease :: Point2B
   pos = ifB grabbing (mouse u) lastRelease
   lastRelease =
     stepper p0 (release `snapshot_` pos .|.
                 undo)

   closeEnough, grabbing :: BoolB
   closeEnough =
     distance2 pos (mouse u) <* grabDistance
   grabbing =
     stepper False (grab    -=> True
                .|. release -=> False)

   grab, release :: Event ()
   grab    = lbp u `whenE` closeEnough
   release = lbr u `whenE` grabbing

   grabPos, undo :: Event S.Point2
   grabPos = grab `snapshot_` pos
   undo = stacker grabPos (charPress '\^Z' u)
```

Undoing works correctly in this new version. See Figure 4.

Lessons

- The interface to `stacker` is that of a *service*, rather than a data structure. To form request channels, pass one or more event argument in to the service function. To get results out, return one or more events. Encapsulate internal state by means of locally defined behaviors and events.
- By generalizing a specific requirement (undo) to a more general service (a polymorphic stacker), we avoiding complicating the point editor and we made a very reusable tool. The curve editor in next section benefits from this decision.
- For incremental changes, use function-valued events and `stepAccum`, and leave the accumulation to Fran.

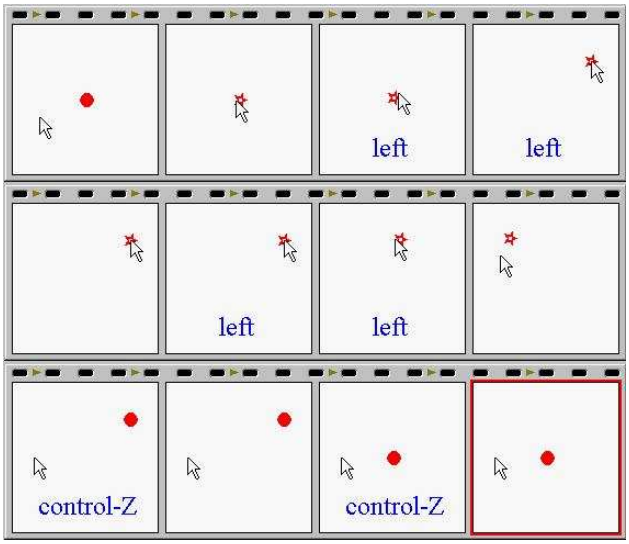

Figure 4. editCPointUndo

[8] *Haskellism:* Lines beginning with "`--`" are comments. Due to a restriction in Haskell's type system, the polymorphic types of `changeStack` and `stack` cannot be given explicitly, so we insert them as comments.

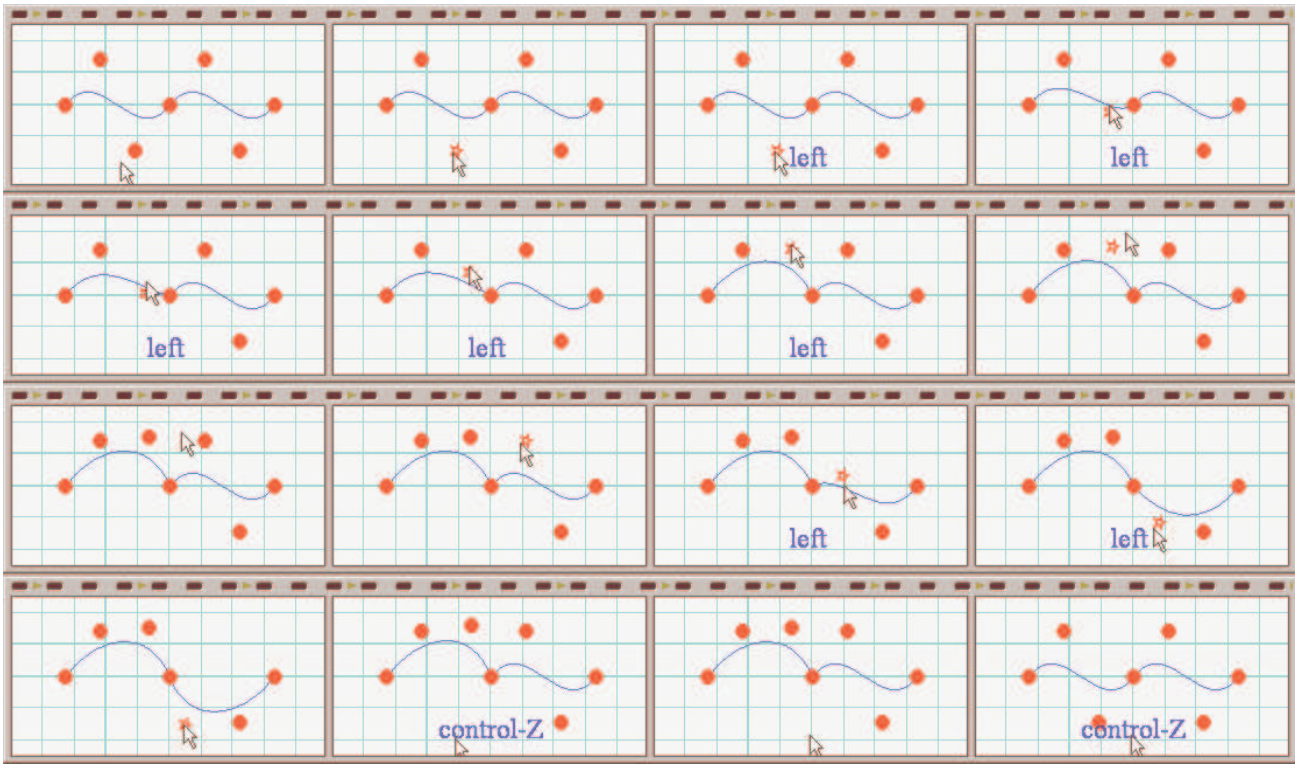

Figure 5. editor with undo

9. CURVE EDITING WITH *UNDO*

Unfortunately, the control point editor in the previous section is not appropriate for making a curve editor with undo. The problem is that each control point has its own undo stack. When the user presses control-Z, all moved control points will back up.

We can fix this problem by moving the undo stacking out of the point editor, where it is replicated, into the curve editor. The point editors can no longer determine the `undo` event by themselves, so they will instead be told as an argument. In addition, since the curve editor will be doing the stacking, the point editor must return the formerly private `grab` event. No other changes are required.

```
editCPointUndo :: User -> Event S.Point2
               -> S.Point2
               -> (CPoint, Event S.Point2)
editCPointUndo u undo p0 =
 ((pos, closeEnough), grabPos)
 where
   pos = ifB grabbing (mouse u) lastRelease
   lastRelease =
     stepper p0 (release `snapshot_` pos .|.
                 undo)

   closeEnough =
     distance2 pos (mouse u) <* grabDistance
   grabbing = stepper False (grab    -=> True
                        .|. release -=> False)

   grab    = lbp u `whenE` closeEnough
   release = lbr u `whenE` grabbing
   grabPos = grab `snapshot_` pos
```

The new curve editor below stacks the position of a point being moved, together with which point. The individual control point grab events are tagged each with its index and then combined with `anyE`, which is the "`.|.`" operator applied to lists of events. The resulting `curveGrab` event is used for stacking, and the resulting `undo` event is then filtered for each control point, using the `suchThat` event operator. Only undos with the appropriate index are passed through, and the indices are dropped.[9]

```
type UndoRecord = (Int, S.Point2)

editCurveUndo :: [S.Point2] -> User -> [CPoint]
editCurveUndo initPoints u = cpoints
 where
   -- Tag and merge the CPoint grab events
   curveGrab :: Event UndoRecord
   curveGrab =
     anyE (zipWith tag indices pointGrabs)
    where
      -- pair i with e's occurrence data
      tag i e = e ==> (i `pair`)

   indices = [1 .. length initPoints]
```

[9] *Haskellisms:* The function `zipWith` is a variant of `map` for functions of two arguments. The operator "." means function composition. An infix operator, such as "`==`" and "`` `pair` `` below, may be surrounded by parentheses and given an argument on the left or right, yielding a function that takes the missing argument and applies the operator.

```
-- The undo event: stack curve grabs and
-- try to restore on control-Z's
undo :: Event UndoRecord
undo = stacker curveGrab (charPress '\^Z' u)

-- Edit an indexed CPoint.
editCP :: Int -> S.Point2
       -> (CPoint, Event S.Point2)
editCP i p0 = editCPointUndo u undoThis p0
 where
   -- Undo if a point tagged i comes off
   -- the undo stack.  Drop tag.
   undoThis =
     undo `suchThat` ((== i) . fst) ==> snd

-- Apply editCP to corresponding indices
-- and initial points, and split (unzip)
-- the resulting cpoints and grabs into two
-- lists.
(cpoints, pointGrabs) =
  unzip (zipWith editCP indices initPoints)
```

This new curve editor correctly supports undoing. See Figure 5.

Lessons

- As with `stacker`, the use of events as arguments and return values in `editCPoint` sets up communication channels between concurrent activities, while local definitions serve to insulate each from irrelevant details in the other's inner workings. In fact, exactly this mechanism is used to communicate user interaction to an animation. A `User` value is essentially an event over values of type `UserAction`.
- Event tagging and event filtering are dual techniques, used to merge and separate sets of events. They allow one program component (`editCurveUndo` here) to act as a "post office", collecting and routing messages.

10. SAVING CURVES

We next add the ability for the user to save an edited curve, by pressing the 's' key, or by closing the editor window. The editor will assure the user that the curve is being saved, by displaying a spinning message, thanks to the following function, which takes a message and an event that indicates when to save, and yields an animated image. [10]

```
spinMessage :: String -> Event a -> ImageB
spinMessage message saveE =
  stretch   saveSize  $
  turn      saveAngle $
  withColor (colorHSL saveAngle 0.5 0.5) $
  stringIm message
 where
   saveDur   = 1.5    -- artificial duration
   sinceSave =  switcher saveDur (
                 timeSinceE saveE)
   -- Fraction remaining (one down to zero)
   saveLeft  = (saveDur - sinceSave)/saveDur
               `max` 0
   saveSize  = 2.5 * saveLeft
   saveAngle = 2 * pi * saveLeft
```

[10] *Haskellism:* An alternative notation for function application is the right-associative, infix operator "$". Because it has very low syntactic precedence, it is sometimes used to reduce the need for parentheses.

The amount of time since the last save event is defined to be `saveDur` initially, and changes each time a save occurs to the (time-varying) length of time since that occurrence. The `timeSinceE` function is not built into Fran, but may be defined easily, as follows.

```
timeSinceE :: Event a -> Event TimeB
timeSinceE e = e `snapshot_` time ==> since
 where
   since :: Time -> TimeB
   since t0 = time - constantB t0
```

At each occurrence of a given event `e`, the time is snapshotted to determine the occurrence time, which is passed to the `since` function, which converts it from a static number into a constant time-valued behavior (with `constantB`), and then subtracts the result from the running time.

By using "`snapshot_`", "`==>`" and a few other basic event operators, we can create highly reusable building blocks like `timeSinceE`, or very task-specific events like `sinceSave`

The `switcher` function is like `stepper`, but takes an initial behavior and a behavior-valued event yielding behaviors to switch to. In fact, `stepper` is defined simply in terms of `switcher`, using `constantB` to turn static values into constant behaviors, as follows.

```
switcher :: Behavior a -> Event (Behavior a)
         -> Behavior a

stepper x0 e = switcher (constantB x0) (
                  e ==> constantB)
```

To see `spinMessage` work, the following definition use the message "goodbye" and restart whenever a key is pressed, as shown in Figure 6.

```
spinMessageTest u =
  spinMessage "goodbye" (keyPressAny u)
```

The curve editor becomes more complicated in order to accommodate saving curves. For one thing, we use a more general variant of `displayU`:

```
displayUIO ::
  (User -> (ImageB, Event (IO ()))) -> IO ()
```

The argument to `displayUIO` constructs not only an image animation, but also an action-valued event. On each occurrence of the event, the corresponding action is executed. In this case,

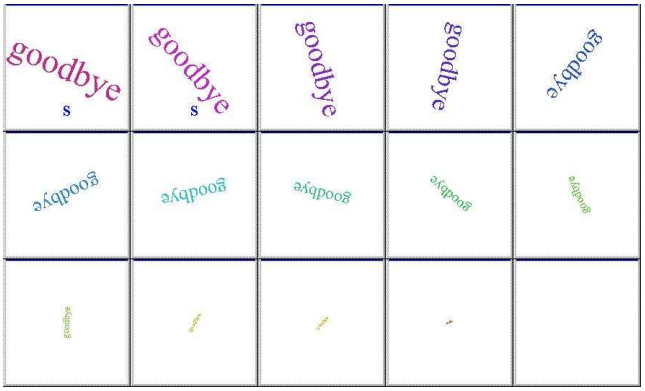

Figure 6. spinMessageTest

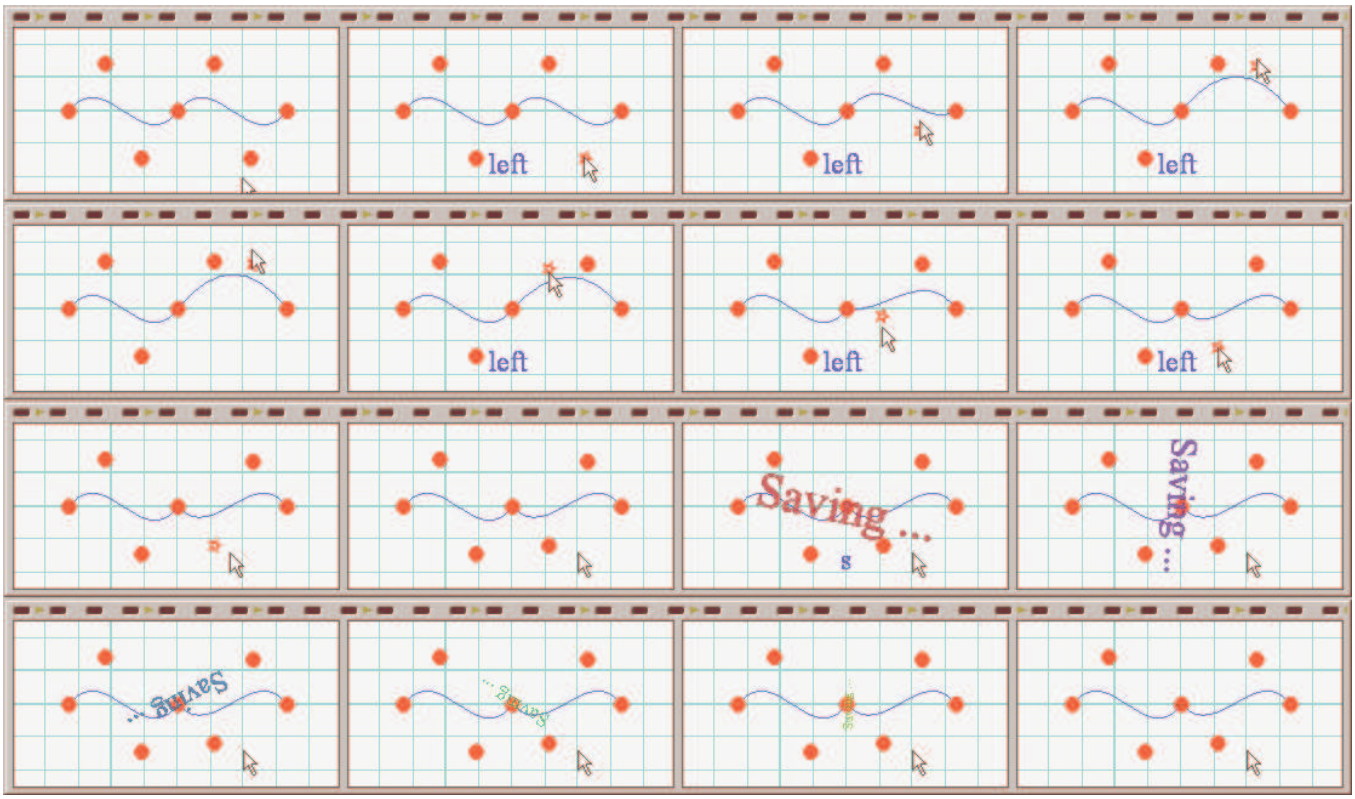

Figure 7. Editor with curve saving

the action saves a curve snapshot in a file. Here is the new editor definition, followed by some brief explanation. Figure 7 shows the execution.

```
editor2 :: String -> IO ()
editor2 fileName = do
  initPoints <- load fileName
  displayUIOMon (editRenderSave initPoints)
 where
   editRenderSave initPoints u =
     ( spinMessage "Saving ..." saveNow
       `over` renderCurve xPoints
       `over` graphPaper
     , doSave )
    where
      xPoints = editCurve initPoints u
      ptsB    = bListToListB (map fst xPoints)
      saveNow = charPress 's' u .|. quit u
      doSave  = saveNow `snapshot_` ptsB
                        ==> save fileName
```

Some explanation:

- The animation part of `editRenderSave`, as used by `displayUIO`, is the spinning message animation overlaying the rendering of the curve being edited. The animated message "Saving ..." is shown when `saveNow` occurs, which is defined to be whenever the user presses the 's' key or closes the window.
- The action-valued event `doSave` also depends on `saveNow`. At each occurrence, the list of curve points is snapshotted, and passed as the second argument to the `save` function, which gets the original file name as its first argument. Snapshotting the control points is a bit tricky. Since `renderPoints` creates a list of pairs, we extract the first half of each pair (via "`map fst`"), and convert the resulting list of behaviors into a behavior over lists.

11. RELATIVE MOTION

The previous versions had a bug: control points recenter on the mouse when grabbing. Consequently, two control points can easily become permanently stuck together:

The problem is that the control point's position is defined to be equal to the mouse's while the control point is being grabbed:

```
  pos = ifB grabbing (mouse u) lastRelease
  lastRelease = stepper p0 (
                  release `snapshot_` pos)
```

Instead, we would like `pos` to be given the same movement as the mouse *relative to the locations at the time of grabbing*. The following revised definition achieves this relative motion.

```
editCPoint :: User -> S.Point2 -> CPoint
editCPoint u p0 = (pos, closeEnough)
 where
   pos = switcher (constantB p0) (
           grab `snapshot` mouse u ==> relMotion
     .|. release                   ==> constantB)
    where
      relMotion (p, mp) =
        constantB p .+^
        (mouse u .-. constantB mp)

      grab     = lbp u `whenE`     closeEnough
                       `snapshot_` pos
```

```
    release  = lbr u `whenE`     grabbing
                     `snapshot_` pos
    grabbing = stepper False (
                       grab    -=> True
                   .|. release -=> False)
closeEnough =
  distance2 pos (mouse u) <* grabDistance
```

For convenience, we have changed the `grab` and `release` events to contain the snapshotted point's position. The new definition of `pos` additionally snapshot the mouse's position. It then feeds the resulting pair to `relMotion`, which adds the control point snapshot to the vector difference between the mouse position and the snapshot of the mouse position. Thus the control point gets the same relative motion as the mouse. The other position-changing event is the control point's `release`, which gets converted from a static point to a constant behavior.

Aside: by convention, names end in "_" indicate a function that forgets something, and have a nonforgetful version. For example, the nonforgetful `snapshot` function has the following declaration.

```
snapshot :: Event a -> Behavior b
         -> Event (a, b)
```

The forgetful version used in the previous editor versions is defined in terms of the nonforgetful version, as follows, where `snd` extracts the second member of a pair.

```
snapshot_ :: Event a -> Behavior b -> Event b
e `snapshot_` b = e `snapshot` b ==> snd
```

Event operators like `snapshot`, `whenE` and "`==>`" are all left-associative and of equal precedence so that they may be cascaded without parentheses, as in this definition and the most recent definition of `pos` within `editCPoint`.

12. RELATED WORK

Esterel [3][2] is a language for synchronous concurrent programming, especially useful in real-time programming. It adopts a synchronous model of concurrency, rather than the more popular asynchronous models, as in CSP [16]. Berry and Gonthier [3] point out that "deterministic concurrency is the key to the modular development of reactive programs and ... is supported by synchronous languages such as Esterel." Communication is based on a instantaneous broadcast model. In contrast, asynchronous models lead to competition for communication, by removing queued events, which leads to nondeterminism. Fran shares these basic properties of instantaneous, synchronous, broadcast communication and the consequent determinism. Unlike Esterel, however, Fran has an additional *continuous* model of time (which is not the focus of this paper). Another difference is that Esterel has an extremely efficient implementation, based on compilation of concurrent programs into deterministic sequential automata. Perhaps the most obvious difference is that Esterel is an extension of imperative programming, while Fran adopts a functional style.

Like Esterel, Lustre [15] and Signal [14] are based on discrete, synchronous, concurrent programming models, and have efficient implementations based on compilation into sequential automata. Unlike Esterel, they are declarative, and so Fran is more closely related to them. Lustre has counterparts to several of the primitive and derived constructs in Fran, including `when`, and variants of `stepper`, and self-snapshotting behaviors (Lustre's "`pre`" operator). The feel is different from Fran, due to the absence of continuous behaviors. Lucid Synchrone [6] is even more closely related. It is quite similar to Lustre, but is embedded in the host language Objective CAML.

CML (Concurrent ML) formalized synchronous operations as first-class, purely functional, values called "events" [27]. Fran's event operators "`.|.`" and "`==>`" correspond to CML's `choose` and `wrap` functions. Also like Fran, CML embeds its event vocabulary in an existing programming language. There are substantial differences, however, between the meaning given to "events" in these two approaches. In CML, events are ultimately used to perform an `action`, such as reading input from or writing output to a file or another process. In contrast, our events are used purely for the values they generate. These values often turn out to be behaviors, although they can also be new events, tuples, functions, etc. In addition, CML is built on a CSP-like nondeterministic rendezvous model for communication, rather than instantaneous broadcast. Similarly, Concurrent Haskell [22] adds a small number of primitives to Haskell, and then uses them to build several higher-level concurrency abstractions. Unlike the Esterel family of languages (and Fran), CML and Concurrent Haskell add concurrency nondeterministically.

Squeak [5] was a small language in the CSP tradition, designed to simplify development of user interfaces. Like the Esterel family of languages, Squeak compiles into an efficient sequential state machine. NewSqueak [25] extended the ideas in Squeak to a full programming language, having a C-like syntax, and anonymous functions. Pike then built a complete window system in fewer than 300 lines of NewSqueak [25].

Scholz described a "monadic calculus of synchronous imperative streams" that has a stream type, somewhat similar to Fran's behaviors, but based on a discrete model of time [30]. Its imperative nature allows one to perform side effects at any level of a behavior, while Fran imposes a discipline, allowing side effects only through action-valued events.

Fran has been used as the foundation of vision processing [1], robot control [21][20], and a mostly-functional GUI toolkit [28].

13. FUTURE WORK

The example presented in this paper is of modest complexity. Examples that are more ambitious will probably suggest improvements in the vocabulary of events, as well as shed more light on design methodology.

Efficient implementation of Fran continues to be a challenging problem. The embedding of Fran in Haskell gives rise to considerable flexibility and expressiveness, as well as a reasonably simple implementation, but has been more difficult to optimize than we expected. In contrast, the efficiency of the Esterel languages is quite impressive. See [10] for a discussion of several approaches to implementing Fran. We have also experimented with a promising fundamentally imperative implementation, described very briefly in [23] and [28]. Debugging and performance analysis are also serious problems in any very high level paradigm, because of the large gap between the programmer's model and the program's execution.

Although Fran is embedded in Haskell, which is a very expressive programming language, we do not propose Fran/Haskell as an

application programming language. We believe that pragmatics require a more modest approach, namely using Fran to generate Haskell-based software *components* [24]. This language hybrid approach allows, for instance, the curve editor to rely on the facilities of a mainstream programming language to provide a modern graphical user interface, file I/O, etc.

14. CONCLUSIONS

Declarative event-oriented programming makes it convenient to encapsulate significant portions of an interactive application into a set of high level, reusable building blocks. These events appear to capture important aspects of an application's design directly, and so may be useful in reducing the cost of creation and maintenance of modern interactive software.

As software becomes increasingly powerful, its internal complexity tends to grow. As Edsger Dijkstra pointed out in his Turing Award lecture, the ideas we can express, and even think, are greatly influenced by the languages we use. In order to achieve our software-building ambitions, we therefore need languages that support abstraction and factoring of algorithms to form "intellectually managable programs".

> One hopes that tomorrow's programming languages will differ greatly from what we are used to now: to a much greater extent than hitherto they should invite us to reflect in the structure of what we write down all abstractions needed to cope conceptually with the complexity of what we are designing. [7]

REFERENCES

[1] Reid, A., J. Peterson, G. Hager, P. Hudak: Prototyping Real-Time Vision Systems: An Experiment in DSL Design, *ICSE '99*. http://haskell.org/frob/icse99/-visionpaper.ps

[2] Berry, G., *The Esterel v5 Language Primer*. ftp://ftp.-esterel.org/esterel/pub/papers/primer.pdf

[3] Berry, G. and G. Gonthier, The Synchronous Programming Language ESTEREL: Design, Semantics, Implementation, *Science of Computer Programming*, vol. 19, no.2 (1992) 83-152. ftp://ftp-sop.inria.fr/meije/-esterel/examples/hdlc.ps.gz

[4] Bird, R. and P. Wadler, *Introduction to Functional Programming*, Prentice-Hall, International Series in Computer Science, 1987.

[5] Cardelli, L. and R. Pike, Squeak: A Language for Communicating with Mice, *Computer Graphics (SIGGRAPH '85 Proceedings)*, Vol. 19, pp. 199-204, July 1985.

[6] Caspi, P. and M. Pouzet, Lucid Synchrone, version 1.01 Tutorial and reference manual. http://ftp.lip6.-fr/lip6/softs/lucid-synchrone/ls-1.01/manual.ps.gz

[7] Dijkstra, E. W., The Humble Programmer (Turning Award lecture), *Communications of the ACM*, 15(10), October 1972.

[8] Elliott, C., An Embedded Modeling Language Approach to Interactive 3D and Multimedia Animation, *IEEE Transactions on Software Engineering*, 25(3), May/June 1999, pp 291-308. http://research.microsoft.com-/~conal/papers/tse-modeled-animation

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

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

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

[12] Elliott, C. 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

[13] Flanagan, D., *Java Examples in a Nutshell: A Companion Volume to Java in a Nutshell*. Published by O'Reilly & Associates, 1997.

[14] Gautier, T., P. le Guernic and L. Besnard, SIGNAL: A Declarative Language for Synchronous Programming of Real-Time Systems, *Functional Programming Languages and Computer Architecture*, pp. 257-277, Springer-Verlag, 1987.

[15] Halbwachs, N., P. Caspi, P. Raymond and D. Pilaud, The Synchronous Data Flow Programming Language LUSTRE, *Proceedings of the IEEE*, 79(9), pp. 1305-1320, September 1991.

[16] Hoare, C. A. R., Communicating Sequential Processes, *Communications of the ACM*, 21(8), pp. 666-677, August 1978.

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

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

[19] Hughes, J., Why Functional Programming Matters. *The Computer Journal*, 32(2), pp. 98-107, April 1989. http://www.cs.chalmers.se/~rjmh/-Papers/whyfp.ps.

[20] Peterson, J., G. D. Hager, and P, Hudak, A Language for Declarative Robotic Programming, *ICRA '99*. http://haskell.cs.yale.edu/frob/icra99/-icra99.ps

[21] Peterson, J., P. Hudak, and C. Elliott, Lambda in Motion: Controlling Robots With Haskell. Proceedings of PADL '99. http://haskell.org/frob/padl99/padl99.ps

[22] Peyton Jones, S., A. Gordon, and S. Finne, Concurrent Haskell, *23rd ACM Symposium on Principles of Programming Languages*, pp. 295-308, 21-24 January 1996, http://research.microsoft.com/~simonpj/-papers/concurrent-haskell.ps.gz.

[23] Peyton Jones, S., S. Marlow, and C. Elliott, Stretching the storage manager: weak pointers and stable names in Haskell, IFL'99. http://research.microsoft.com/-Users/simonpj/papers/weak.ps.gz

[24] Peyton Jones, S., E. Meijer, and D. Leijen, Scripting COM Components in Haskell, *Software Reuse 1998*, http://-www.dcs.gla.ac.uk/~simonpj/com.ps.gz.

[25] Pike, R., Newsqueak: A Language for Communicating with Mice, CSTR143, Bell Labs, March 1989, http://cm.bell-labs.com/cm/cs/cstr/-143.ps.gz.

[26] Pike, R., A Concurrent Window System, *Computing Systems*, 2(2), pp. 133-153, spring 1989.

[27] Reppy, J. H., CML: A Higher-order Concurrent Language. *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 293-305, 1991.

[28] Sage, M., FranTk - A declarative GUI language for Haskell. To appear at ICFP '00.

[29] Schechter, G., 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

[30] Scholz, E., Imperative Streams—A Monadic Combinator Library for Synchronous Programming, ICFP '98 http://www.inf.fu-berlin.de/~scholz/Icfp-Published-Final.ps