Fran in action!

Anthony C. Daniels
Languages and Programming Group
Department of Computer Science, University of Nottingham
acd@cs.nott.ac.uk

July 14, 1997

Abstract

Fran is a Haskell library for creating real-time interactive animations. This paper demonstrates how the system can be used to create a realistic animation and indicates how this can be extended to form large, complex animations. We emphasize the flexibility, composability and ease of construction afforded by the system from this pragmatic perspective. No knowledge of Fran is required to read this paper. We hope that functional programmers will be able to use it as a tutorial.

1 Introduction

Fran [4] stands for Functional Reactive Animation. It is being developed principally by Conal Elliott at Microsoft Research and in conjunction with researchers at Glasgow, Nottingham and Yale Universities. Its aim is to ease the task of building interactive animations. The system is intended to be very general, but various specific application areas appear particularly well suited, for example, multimedia, teaching, simulation, communication and WWW pages. Furthermore, we intend that Fran will exploit the current and future hardware developments for graphics on PCs.

Some of the key ideas behind Fran [4, 3] were developed in previous systems, in particular TBAG [2, 5]. Fran is the latest prototype implementation of this work. So far, the benefits of choosing a lazy functional language for the implementation have been considerable, especially for prototyping different representations of the central concepts. Strong typing, type classes, polymorphism, higher order functions and laziness have all been useful. The major drawback is low efficiency, but we have also experienced difficulties analyzing the behavior of complicated reactive elements (often recursively defined) within the system.

1.1 The aim of this paper

For keen readers this paper can be used as an introductory tutorial; Fran is freely available on the WWW at:

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

The complete source code for these animations can be obtained from the authors home-page:

```
http://www.cs.nott.ac.uk/~acd/
```

Alternatively it can be read more quickly to get a feel for the system and how it may be used. Elliott and Hudak [4] give detailed explanations of the main concepts; their paper is essential reading for those with an interest in the theoretical foundations on which Fran is based. This paper comments on the applicability of functional languages to this domain, motivated by an example animation. Readers without experience of functional programming, or of Haskell in particular, are referred to introductory texts as prerequisites [1, 7].

1.2 Our example

In this paper we develop an animation of an oarsman performing the rowing action. Fran is currently restricted to 2D graphics, limiting the realism of the final animation. However, our animation demonstrates the fundamentals of good rowing technique; it might, for example, serve as a rowing tutor or as a component of a larger animation. We build up the animation componentwise (i.e., bottom up) since we find this a natural way to approach the problem and Fran allows us to combine separate components easily.

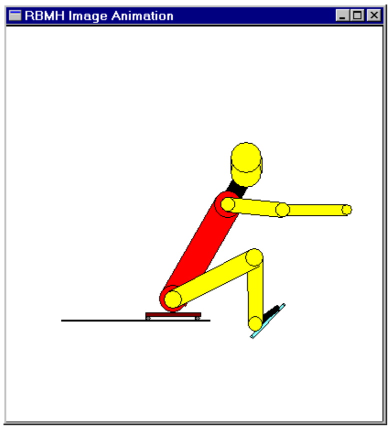


Figure 1: The rower

2 An oarsman

To create any animation we first need to construct images.[1] In Section 2.1 we define static images using primitives in Fran. These are really animated images that are constant, meaning that their position and form does not vary with time. It is possible to modify these definitions to introduce motion; this is the subject of Section 2.2. Section 2.3 desribes how the animation can be modified to make the model more realistic. We find it convenient to describe the legs using a constraint in Section 2.4. Finally we mention the remaining components of the animation, without going into all the details, in Section 2.5 and summarize in Section 2.6.

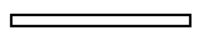
2.1 Constructing images

Our starting point is the sliding seat, which gives us a base to put our oarsman upon. We can crudely represent the seat with a small rectangle. The function `rectangleLW` is a rectangle of the length (horizontal component)

[1]Note that we distinguish between the construction of images from smaller components and the process of drawing or rendering those images on a display. Naturally, the implementation of Fran includes code to deal with the latter, but it is the former that is emphasized in the users view of the system [3].

and width (vertical component) given by its arguments[2]:

```
rectangleLW 0.3 0.02
```



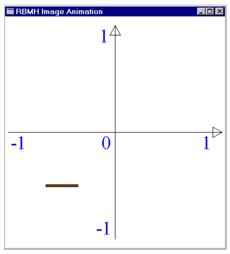
The default color is white; the `withColor` function takes a color and an image and produces a new version of the image in the given color. Thus we can re-define our seat to be brown as follows:

```
withColor brown (rectangleLW 0.3 0.02)
```

By default, image primitives are displayed in the center (origin) of the window which has logical coordinates from -1 to 1 on both axes. We need to move our seat down and to the left. To do this we apply a transformation. The operator `*%` takes a transformation as its first argument and applies it to an image, given as the second argument. In this case our transformation is a translation. The diagram below shows this image; we have added the axes and numbers to mark the logical co-ordinates:

[2] To display such a definition using Fran, write a module that imports both `Fran` and `Disp` and then assign the definition to a top level value, for example, `seat = rectangleLW 0.3 0.02`. Then enter `disp (const seat)` at the prompt.

```
translate2 (vector2XY (-0.5) (-0.5)) *%
  withColor brown (rectangle 0.3 0.02)
```



The constructor `vector2XY` builds a two dimensional vector from its arguments which are Cartesian x and y coordinates. `translate2` then creates a translation transformation in two dimensions from the given vector.

The seat runs on wheels on a slide. To create a wheel, we begin with a blue circle and apply a scaling transformation so that it is the required size; in this case a scale factor of 0.014 (found by experiment) works well because the wheels are quite small:

```
aWheel = uscale2 0.014 *% (withColor blue circle)
```

The `uscale2` function shown here constructs a uniform scaling transformation in two dimensions. This image of a wheel is used for both the front and rear wheels. We apply appropriate translations to position the wheels. These components could be combined in one definition. However, we will give names to some of the components to make the definition more readable and introduce top level definitions for the width and length dimensions because they are used elsewhere:

```
seatW, seatL :: RealB       -- (see next section)
seatW = 0.02                -- Width
seatL = 0.3                 -- Length

seatPosition = vector2XY (-0.5) (-0.5)

seat = translate2 seatPosition *%
       (seatImage `over`
```

```
        rearWheel `over`
        frontWheel)
  where
    seatImage = withColor brown
                  (rectangleLW seatL seatW)
    aWheel = uscale2 0.014 *%
               (withColor (gray 0.5) circle)
    rearWheel  = translate2 rearPos *% aWheel
    rearPos = vector2XY (0.014 - seatL/2.0) (-seatW)
    frontWheel = translate2 frontPos *% aWheel
    frontPos = vector2XY (seatL/2.0 - 0.014) (-seatW)
```

The `over` function simply overlays one image on top of another; hence we now have a seat with two small wheels at each end:

The use of Haskell's infix notation with `over` results in a natural reading. The slide, which the seat moves back and forth upon, is a gray rectangle of appropriate position and dimensions.[3] The whole image so far can be defined as:

```
seat `over` slide
```

When more parts of the rower are defined, we can add them to the animation by adding the new part using `over` to the above definition. The order of the components determines the order the images are overlaid; earlier images are always on top of later ones. The manner in which images are composed from the primitives is reminiscent of Henderson's Functional Geometry[6]. The same benefits Henderson identified therefore apply to constructing static images in Fran; the resulting code is clear, concise and easily modifiable.

[3] Complete definitions are given in the source code.

2.2 Animating

To create an animation the seat must move back and forth along the slide. Although the images described above are static, they are actually animations in which the positions of the components are constant. In Fran, we use behaviors for temporal modeling. A behavior is a value that varies with time; in the functional paradigm you can think of behaviors (conceptually) as functions from time to values.[4] The type of these values may differ in different contexts. For example, `seat` defined above is an image behavior. It is constructed from primitive image behaviors and transformations that, in turn, are built from vector and scalar behaviors. We can create behaviors of any type, for example a real-valued behavior[5]. In fact, Fran uses overloading to lift all numeric constants so that they are (constant) real-valued behaviors. Haskell's overloading is also used to create lifted versions of many operators, such as `cos`, `+` and `*`. However, the type restrictions sometimes prevent us from overloading operators, for example, we can not overload `+` to add a vector to a point because the argument types must be the same.

`Point2` is the type of a co-ordinate in two dimensions and `Point2B` is the behavior version (or lifted type). It is therefore simple to create a moving seat by defining its position using a non-constant `Point2` behavior. We factor out the initial position and the distance that the seat moves from this position as separate top-level values:

```
-- The central position of the seat, as above.
seatPosition0 = point2XY (-0.5) (-0.5)

-- The distance the seat moves from its central position
-- on the slide.
seatDistance = 0.23 :: RealB

-- Our new seat position behavior (replaces previous one).
seatPosition = seatPosition0 .+^
               vector2XY (seatDistance * (cos time)) 0.0
```

[4] For a number of reasons, particularly efficiency considerations, the implementation does not represent behaviors this way.

[5] Haskell's Double type is given the synonym `RealVal` and the real-valued behavior type is called `RealB`.

Here we used a new operator `.+^` which adds a vector (right argument) to a point (left argument). We also use our first non-constant behavior: `time`. This behavior gives the current time, which changes as the animation progresses. The function `cos` is the usual cosine function lifted to the behavior level. So our seat oscillates about a central position by a vector of magnitude `seatDistance` in the x direction multiplied by the cosine of time. Here is an illustration of the animation:

Ideally we would like to control the rating, or strokes per minute, that our oarsman performs. We can achieve this by defining two simple functions and some appropriate constants:

```
rating = 20.0 :: RealB
strokesPerSec = rating/60.0

-- Given a time, multiplies so that a stroke takes one time
-- unit.
strokeIn1 x = x*strokesPerSec

-- The cosine function with period 1.
cosPeriod1 x = cos (2.0*pi*x)

seatCos = cosPeriod1 . strokeIn1
```

Now if we replace `cos` in the definition of `seatPosition` above with `seatCos`, the seat will go up and down the slide exactly twenty times per minute.

2.3 Modeling

We shall now turn our attention to modeling the motion of a real oarsman. Our rower will be composed of a number of connected limbs that we will define using rounded rectangles (rectangles with circles at each end). We can abstract from the low level description of these limbs by factoring out the common elements into a general `limb` function. Rather than giving the position of the center of a limb and its size, we will use the end points and width as parameters to `limb`. This is because we will calculate the position of the oarsman's joints, which are the end points of limbs. Hence our `limb` function has the following type:

```
limb :: Point2B -> Point2B -> RealB -> ColorB -> ImageB
```

The first two arguments (point behaviors) are the end points, followed by the width and color of the limb. Its definition is simply a rectangle of appropriate dimensions with a rotation and a translation applied so that the ends of the rectangle are at the given end points. Circles (of radius half the width of the rectangle) are overlaid at the end points to round off the appearance of the limbs. We think using functions such as `limb` greatly improves the readability of the source code.

Let's add the torso. We adopt a naming convention in our code using postfix `W` and `L` for the width and length of the limbs respectively. The hip is positioned centrally just above the seat:

```
shoulder = hip .+^ vector2Polar bodyL bodyAngle
hip = seatT .+^
      vector2XY (seatL/2.0) ((seatW + bodyW)/2.0)
bodyW = 0.15 :: RealB
bodyL = 0.6 :: RealB

bodyAngle = pi/2.0 - (pi/6.0)*bodyCos :: RealB
bodyCos = cosPeriod1 . strokeIn1

body = limb shoulder hip bodyW red
```

The `vector2Polar` constructor takes a length and an angle and creates the vector using polar coordinates. Hence our rowers hip is positioned above the seat and his torso extends at an angle given by `bodyAngle` from the hip. We have defined this angle to vary between -$\pi/6$ and $\pi/6$ from the vertical, using `bodyCos` which is the same as `seatCos` defined previously. However, on running this animation, any rowing enthusiast will immediately identify a number of serious flaws in our rowers technique, even though at this stage he only consits of a body on a seat. Essentially, the back should not swing backwards until the seat is near backstops, i.e., its left most position, to maximize the power obtained from the leg drive. To reflect this we will need to increase the accuracy of our model. One approach would be to define a periodic function that directly gives the angle of the back at any given time. We use a simpler solution; the motion proceeds in one direction and then back in the opposite direction during one stroke, or in other words, the motion is monotone over a half period. We can write any such function using a special function composed with `cos`, giving a smooth, continuous periodic function. The required function maps from the origin to (1, 1) monotonically increasing and represents the relative rate of the motion. Here are the definitions, leading to a new `bodyCos` function which replaces the one above:

```
-- Maps times to congruent times in the fundamental
-- interval [0, 1) of the periodic function.
modulo1 x = x - fromIntegerB (floorB x)

periodicFn relativeRate
  = cosPeriod1 . relativeRate . modulo1 . strokeIn1

bodyCos = periodicFn relativeRate time
  where
    relativeRate x = cond (x <* openBackT0)
                          (drive x)
                          (cond (x <* bodyLeanT1)
                                (bodyLean x)
                                (bodyLeant x))
      where
        bodyLeanT1  = 0.55   -- Body now lent over fully.
        openBackT1  = 0.35   -- Now leaning back at finish.
        drive x     = 0.0
        bodyLean x  = 2.5*x - 0.375
        bodyLeant x = 1.0
```

`cond` is the analogue of `if ... then ... else` for behaviors and `<*` is the behavior level `<` operator. Now the body swings over towards the end of the drive as required. We can use the same technique to model the motion of the seat, which moves quickly during the drive (right to left) and slowly during the recovery.[6] Higher-order functions helped us to describe this aspect of the model and we feel the resulting code is clear and modular. In particular, the general principal of using cos and a relative rate to create various periodic functions was abstracted by the `periodicFn` function. This function is quite complicated, but it naturally decomposes into four simple functions which are composed to form the complete definition.

2.4 Constraints

To form the thigh, we must know the position of each end of the limb. We already know where the hip joint is but not the knee. The knee connects the thigh to the lower leg (shin) which is connected to the foot. The foot is strapped onto the foot-plate, so we know that the bottom of the shin (ankle) is fixed. To calculate the knee joint we observe that it is just the point of intersection of two circles, one centered at the hip the other at the ankle with radii given by the thigh length and the shin length, respectively. Our definition of knee uses a function `circleIntersectsCircle` which gives this point. Note that there are two points where overlapping circles intersect; the final argument to `circleIntersectsCircle` is a Boolean value to enable the selection of the point we require:

```
kneeJoint = circleIntersectsCircle hip ankle
                                   thighL shinL True

thighW = 0.1 :: RealB
thighL = 0.5 :: RealB

thigh = limb hip kneeJoint thighW yellow
```

Definitions for the shin are similar, and the foot and foot-plate are simple static images.

2.5 Finishing touches

It is straightforward to add the upper arm and forearm; the position of the shoulder is known and the arm swings according to `armCos`. This is a

[6] The part of the stroke when the rower is going back up the slide towards frontstops, with the blade held clear of the water.

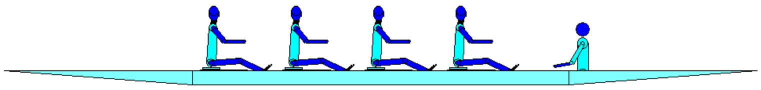


Figure 2: The crew rowing animation

function defined using `periodicFn` and the appropriate relative rate function to describe how the arm swings. The neck and head are also straightforward to define; since the rower always looks ahead we calculate the top of his head using a vertical vector added to the bottom of his head (which is joined to the neck and so on.)

2.6 Summary so far

We have encapsulated the essential model components (limbs and periodic functions) in general functions. Defining our man was then easy; join the limbs in sequence and work out where the other end goes. For animating people in general we could apply inverse kinematics techniques and encode these principles in some general functions abstracting out the necessary parameters. Such generalizations are made possible by the underlying functional language, Haskell, on which Fran is based. Furthermore, we could expect general partial evaluation techniques to optimize these definitions, helping to produce more efficient compiled code.

3 Making the rower work for us

In this section we will give some examples to illustrate re-use and composibility of animations in Fran, by creating various animations based on our rower.

3.1 Crew Rowing

An obvious application is to put our rower in a boat with some other rowers. Our crew rowing animation consists of four rowers positioned horizontally across the screen in a boat with a coxswain.[7] Juxtaposing images is a very common operation so we define a general function that takes a list of images and overlays them horizontally. It assumes the images fill the display (i.e., the region (-1, -1) to (1, 1)) and re-sizes them horizontally so the resulting image also fits in the display. Also, it takes a further argument that specifies the gap to use between adjacent images. In our animation, we actually want the rowers to overlap, so we supply a negative argument:

```
besidesGap is g
  = foldl1 over (map scaleAndMove (zip [0..] is))
  where
    n = length is :: Int
    p = 1/(fromIntegral n) :: Double

    scaleAndMove (r, image)
      = translate2 pos *%
        scale2 (vector2XY (lift0 p) (lift0 p)) *% image
      where
        pos = vector2XY (lift0 (-1.0 + p +
                  2*(fromIntegral r)*p + gap r)) 0
        gap r
          | even n
            = g * (1/2 + fromIntegral (r - (n `div` 2)))
          | otherwise
            = g * fromIntegral (r - ((n-1) `div` 2))
```

Using `besidesGap`, the crew is quite easy to define:

```
crew = besidesGap (replicate 4 rower) (-0.15)
```

The `boat` and `cox` are defined using the primitive for creating static images provided by Fran. Now `boat`, `crew` and `cox` can be combined and we can make them move:

```
boatAndCrew t0 = translate2 pos *%
                 (boat `over` crew `over` cox)
  where
    pos = point2XY (2.0 - (time/10.0)) 0.0
```

[7] The coxswain, or cox, is the steersman, usually seated in the stern of the boat.

With the current implementation this animation has quite a low frame rate, even on a powerful PC, so developing it further is difficult at this stage. However, it is easy to imagine, how it could progress: a river, another crew, a race, an entire regatta!

3.2 Play, stop, fast-forward

To illustrate how interactive features can be used with Fran animations, we will add VCR-like buttons to the display so that the user can stop, play and fast-forward the animation. The rower has limited scope for interactivity, so this example is illustrative rather than particularly inspiring. To implement this feature we use a time transformation and hence we do not provide a rewind button because we can not go back in time. Rewinding could be achieved by reversing the rower's actions while still moving forward in time. However, that would require a different model of controlling the VCR, based on changing the actual animation, rather than a simple manipulation of real-time.

We will now describe the main definition which is called `controlRower`. The first line of `controlRower` describes the three buttons overlaid on our basic rower, with a time transformation applied:

```
controlRower t0
  = (buttons `over` rower)
    `timeTransform`
    integral (playV t0 1.0) t0
  where ...
```

The buttons themselves are simple static images. A time transformation literally maps actual time to a different time frame which is then used for the animation. In practice, when sampling the animation (evaluating one frame) at a time t, the time transformation is first applied to t and then the resulting value is used to sample the behavior. The `t0`'s appearing in the above definition are start times. When an animation is displayed a start time is passed to the definition so that it can use this information. We have not needed to use start times in any of the previous definitions; they are ignored by writing animations that do not take a start time and using `const`.

The tricky part of this definition is the time transformation, which is broken down in the `where` clause. The reactive behavior `playV` is the rate at which time is advancing on the VCR. Thus, in the main definition above we integrate `playV` and this gives the (VCR) time the animation has reached (because integrating a rate gives the a value that is changing at this rate).

Now we will describe the definition of `playV` which is given below. It is a piecewise constant behavior that, over an interval, is one of three values representing the current playing rate: 1 for play mode, 4 for fast-forward or 0 for stop. We change between these three constant values by detecting button press events and recursively calling `playV` with the new playing rate. To implement this we need to make use of Fran's event combinators. An event occurs at a (one) particular time yielding a value. `` `untilB` `` is used to create piecewise behaviors; `` b `untilB` e `` gives the value of the behavior `b` until the event `e` occurs. Then it takes the value that the event yielded. The first line of the definition below can now be read as: firstly lift the constant value `rate0` to a constant behavior. The behavior `playV` will then take this value until the event on the RHS of `untilB` occurs. The occurrence of a button press event is captured by `newRate t0` which is the event that yields the (new) rate corresponding to the control clicked. The actual behavior we require is the behavior which starts off with our new rate and then continues to behave like a VCR. This is of course `playV t v` where `t` is the time the last event occurred and `v` is the new rate - the value which the event yielded. We use the event combinator `+=>` which passes on the time and value to avoid having to explicitly write these parameters.

So `newRate` is the event that occurs when a control is clicked on and yields the corresponding rate. It uses the event combinator `.|.` which gives the earlier of two events, so the `newRate` event occurs when either of the three button press events occur, and corresponds to whichever one occurs first. The individual events representing each button are defined using `-=>` which takes an event (clicking on a button) on the LHS and gives a new event which occurs the same time but yields the value on the RHS; in this case the new rate of 1, 4 or 0:

```
playV t0 rate0 = lift0 rate0 `untilB` newRate t0 +=> playV
  where
    newRate t0 =     pickEvent playIm t0 -=> 1.0
                 .|. pickEvent ffwdIm t0 -=> 4.0
                 .|. pickEvent stopIm t0 -=> 0.0
```

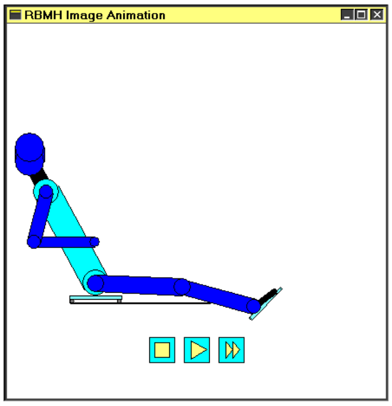


Figure 3: The rower with VCR controls

The events that occur when a button is clicked on are given in terms of a general function called `pickEvent`. This takes an image and a start time and gives the event that occurs when the user clicks on the given image. The event yields the value of the release event which we ignore above by using `-=>`. The exact details of the definition of `pickEvent` can be understood by reading Elliott and Hudak's paper [4]. Essentially, when the mouse button is clicked (the event `lbp t0`) we grab the mouse position (`snapshot` the mouse position). Then we use `suchThat`, which ensures that the main event only occurs when this event occurs (clicking the left button) and a further condition is satisfied. This further condition is that the mouse is positioned over the given image when the button was pressed. A simple picking function, `S.pick2`, is used to determine this:

```
pickEvent iB t0
  = ((\t0 -> lbp t0 `snapshot` (pairB (mouse t0) iB))
       `suchThat`
     mouseOnImage) t0 ==> fst
  where
    mouseOnImage (releaseEvent, (mousePos, i))
      = S.pick2 i mousePos
```

3.3 Sophisticated modeling

The previous animation introduced simple interaction with the user, but interaction can also occur between components of the animation. So, for example, the crew could contain separate rowers who interact with each other. To accomplish this it would be necessary to define how each rower responds to the other rowers. For example, the stroke[8] could row at a given rating and the other members of the crew could follow him, perhaps with slight variability to model the imperfection of a real oarsman's timing. The definitions would become more complicated, but because all components are first-class values they can be built compositionally which helps animators create extendible, maintainable programs. Of course, the sophistication of the model is virtually limitless; one can imagine imperfect rowers who all respond to their senses, for example making adjustments to correct the balance in a three-dimensional model. A serious mistake such as 'catching a crab'[9] could cause major upset to the crew, and justly the rower may be struck from behind by an oar handle in his kidney. Of course such models would require considerable computing power and are beyond the scope of the average user, but they illustrate the potential of the forms of declarative reactivity afforded by the system. Now consider rowers reacting not only to the person in front of them, but also to the person behind (for example, stopping when accidentally hit in the back). Any rower's actions depend on the rower behind, who in turn depends on this rower. In other words, we have a system of mutually recursive definitions. Fran is able to solve these equations by sampling at discrete times[10]. From this perspective, the system is actually executing a specification, by computing an approximate solution to some potentially complicated set of equations.

3.4 Using modules

We used Haskell's module system to structure our animation. Here are the modules used with a brief description:

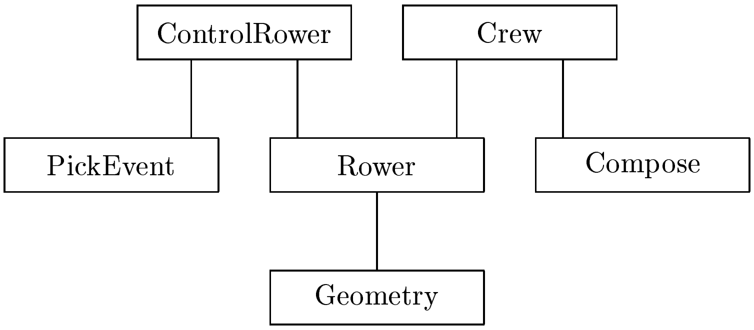
[8] In a crew, the stroke is the rower nearest the cox who sets the rating, or strokes per minute, which is followed by the rest of the crew.

[9] Catching a crab means failing to extract the blade from the water at the finish (the end of the stroke). The boat continues to move but your blade is stuck in the water holding back the boat.

[10] The general applicability of this idea is still under investigation; it has been used to good effect in specific cases, for example, an animation of planetary orbits.

1. Geometry: This includes functions like `circleIntersectCircle`.
2. Rower: The basic rower, as defined in Section 2.
3. Crew: The crew rowing definitions from Section 3.1.
4. Compose: Contains functions like `besidesGap`.
5. ControlRower: The definitions from Section 3.2.
6. PickEvent: The `pickEvent` function.

All modules import the standard Fran library. Further module import dependencies are shown below; each module imports the modules below it when they are joined by a line:



4 Evaluation

We will evaluate three different aspects of this system. The first two are from an animators' perspective; they examine the actual results produced and the difficulty of creating them. We then evaluate the use of Haskell for implementing Fran.

4.1 The end result

Our animation shows the rowing action quite accurately. We were able to model the body movements in two dimensions and, in particular, to adjust the speed of these movements to reflect actual rowing (so our rower has a good 'rhythm'). The performance of this animation was fairly poor on a good PC, reflecting the early stage of development of this prototype system.

The realism would be greatly enhanced by the addition of 3D data types to the system, at the cost of slightly more work defining the positions and shapes. In theory, more realism could also be obtained using texture mapping and other rendering techniques, although in practice this may require specialized graphics hardware to obtain acceptable real-time performance on PCs. Still, our simple rower serves a purpose - demonstrating the basic rowing technique - and can be put to many uses, as described in the previous section.

4.2 The ease of construction

A good measure for how (relatively) easy it is to construct an animation can be obtained by considering the pure modeling approach. Fran uses modeling based semantics which helps to abstract away from presentation tasks [3]. Let us define a pure model as a complete (mathematical) description of the animation that is unambiguous but uses any reasonable notation deemed appropriate. Such models are optimal in terms of requiring the minimum amount of work to define. If you try writing such a definition usually it will not differ significantly from the corresponding Fran program. In our opinion, this is because Fran has a very natural declarative semantics. Because we have not had to do much more work than describing the pure model, we claim that for some interactive animations the Fran system is near optimal in terms of ease of construction. However, it is not difficult to find examples, particularly where complex interaction is involved, when this is not true. This suggests where most of the future research effort on the semantics of the system is likely to be directed.

More specifically, we cite the following reasons for relative easy of construction:

- Behaviors are first-class values of various types. This makes them easy to manipulate and compose.
- Fran has a declarative semantics that is naturally implemented in a language like Haskell. The declarative style is exactly what we want for describing our models of animations on a computer.
- Systems of recursive equations can be entered and assuming no circularity Fran will do the right thing.
- Fran is almost executing a specification.

- Using the Haskell module system, we can separate components. By encapsulating components in modules, we gain the usual advantages of security, re-usability, understandability and separate compilation.

It is interesting to note some technical problems that arise when trying to create animations like this using primitive techniques, such as programming in C without the use of a powerful library like Fran:

- Animating in real time. How do you ensure the rower performs at the exact rating specified by the model? How would our `controlRower` animation be defined?
- Composing animation components. If components use side effects will they interfere, limiting composability?

Essentially, these problems are due to combining the modeling and presentation tasks in the implementation. There are considerable advantages to be gained by separating these two tasks, although how to do this efficiently in practice is still a hot research topic.

To sum up, for this example and many others we have found Fran to be an easy and productive system for creating interactive animations.

4.3 The use of Haskell

4.3.1 Haskell as the host language

Haskell plays a critical role as the host language of the Fran library. Some of the benefits of Fran detailed in the previous section are due to Haskell. Indeed, it is difficult to see how Fran could be embedded into an existing imperative language for reasons given below; instead a completely new language specific to animation would need to be implemented. Using a functional language avoids this considerable overhead. Other functional languages may be suitable host languages, the essential criteria being:

- No (or restricted) side-effects, so that components can be composed and re-used without interfering with each other. However, it may be the case that the ability to create abstract data types, even in an imperative language, is sufficient.
- A declarative style, which is beneficial for modeling.

- Polymorphism, so, for example, behaviors can be represented in a uniform manner. In particular, it is difficult to see how lifting could be achieved without parametric polymorphism to define the lifting functions and overloading to enable convenient notation (e.g., a Num instance for behaviors). Without uniform lifting it may not be possible to encapsulate behaviors as an abstract data type.
- The order of evaluation should be independent from the model, so that the presentation engine can be separated and given full control over execution of the animation.

4.3.2 Haskell as the implementation language

Many features of Haskell have eased the implementation of Fran. Polymorphism is almost essential because we lift many different types to behaviors, and without polymorphism it may not be possible to encapsulate behaviors as mentioned above. Type classes have also been useful for defining operations that apply to some types of behaviors but not all, for example we can only integrate behaviors that are vectorspaces.

Laziness appears extremely useful for the representation of behaviors. Behaviors contain an infinite amount of information but, at any one time, we are only interested in a small part of this. We can use laziness to simplify the construction of behaviors by building the entire infinite structure which describes the behavior over all times in the future. Because evaluation is lazy, the presentation engine drives the evaluation of this infinite structure so that for each sample time only the values necessary for constructing the current frame are computed.

Finally, we have found many of the advantages usually enjoyed when programming in a functional language useful:

- Strong typing.
- Higher-order functions.
- Concise syntax.
- Composability and orthogonality.

So Haskell has been excellent for implementing the system, but, as noted earlier, it is rather slow. However, the system is in an early stage of development and there is a lot of scope for improvement. In particular, we hope that Fran will be able to exploit a commingle of new technologies and that together these will transform the performance.

5 Conclusion and future

We have created a basic animation and built different animations re-using this component. We have evaluated the Fran system for this task.

Some ideas for animations you might like to try are: bungee jumping, GUI components, educational programs (e.g., teaching mathematics and physics such as simple harmonic motion, resonance, Kepler's laws), colliding balls, springs and weights, juggling, planetary motion and pong.

Future system enhancements include 3D graphics, sound and optimizations for improved performance.

References

[1] Richard Bird and Philip Wadler. *An Introduction to Functional Programming.* Prentice-Hall., 1987.

[2] Ricky Yeung Conal Elliott, Greg Schechter and Salim Abi-Ezzi. TBAG: A high level framework for interactive, animated 3D graphics applications. In *SIGGRAPH*, 1994.

[3] Conal Elliott. The essence of active VRML. Technical report, Microsoft Research, 1996.

[4] Conal Elliott and Paul Hudak. Functional reactive animation. In *ICFP*, 1997.

[5] Ricky Yeung Greg Schechter, Conal Elliott and Salim Abi-Ezzi. Functional 3D graphics in C++ - with an object-oriented, multiple dispatching implementation. Technical report, SunSoft, Inc., 1995.

[6] Peter Henderson. Functional geometry. In *ACM Symposium on LISP and Functional Programming*, pages 179–187, 1982.

[7] P. Hudak and J. Fasel. A gentle introduction to Haskell. *SIGPLAN Notices*, 27(5):Section T, 1992.