J. Functional Programming 4 (1): 1–18, January 1994 © 1994 Cambridge University Press

A functional animation starter-kit

KAVI ARYA[1]

IBM TJ Watson Research Center, Yorktown Heights, NY 10598, USA

Abstract

A functional approach presents a fresh perspective on the problem of animation. We present an implementation of a functional animation system written in Haskell, and illustrate how it may be used to create simple and colourful animations.

Capsule review

The process of *animation* – generating a series of pictures to form a movie – turns out to be an excellent application of functional programming languages. This paper shows how higher-order functions and lazy evaluation can play a natural and essential role in the composition of individual pictures to form entire scenes, and in the creation of streams of scenes to form complete animations. It should be of interest to researchers in functional programming as well as those interested in animation. For the latter, even if performance is an impediment, the methodology can at least be viewed as an excellent vehicle for animation *prototyping*.

1 Introduction

As the cost of computing power falls still further and the use of graphics becomes widespread, we note the increasing use of computer generated pictures and animation. These trends indicate the need for making animation more accessible and amenable to rapid prototyping. For example, the news graphics team of a television company typically needs to create animated sequences under severe time constraints. This anticipates the need for simple systems which allow users to quickly generate animation sequences to present data or to illustrate ideas.

As graphic engines deliver faster processing speeds, it is the labour of the animator which is becoming the critical factor in the cost of computer animation. Unfortunately, there has not been enough work on the development of languages that allow animators to step back adequately from implementation details to consider the forms and the structures for creating animated sequences. There are two reasons for this: the first is that animation is about visual effect, and this requires a degree of control that is often difficult to achieve solely through script-driven programming. The second is that animators often come from non-computer backgrounds and feel uncomfortable with a programmer-oriented interface. However, there is a large

[1] Current address: Tata Research Development and Design Centre, 1 Mangaldas Road, Pune 411, 001 India.

domain of animation applications that are particularly suited to a script based paradigm. These domains include scientific visualization, animation previewing, presentation graphics, television news graphics – in short, any application which needs to rapidly prototype animation sequences. The basis for our approach to animation is just this script-based system. We build an animation system around a compact set of primitive operations which are used as building blocks to construct more complex operations.

Many of the problems faced by animators are similar to those faced by software engineers working on large systems within a computing environment and using tools that are quite cumbersome. An analogy may be drawn with desk-top publishing systems just over a decade ago. Functional languages, in our experience (Arya, 1986, 1988), are very effective for the rapid prototyping of systems, due particularly to the use of higher-order functions which leads to very compact programs. It is felt that a functional approach helps create a prototype much more quickly than by using an imperative language. In our experience, it is through experience with initial implementations that we understand a problem better. The understanding gained from a functional prototype may be used to develop a production-quality system using a conventional language such as C.

The motivation for this paper is threefold. The first is to present the reader with a guide to our functional animation system. The second is to offer this code as the basis for other applications – since it is often easier to build a working system by modifying an existing program. Lastly, this implementation serves as a pointer on functional style and as an example of a relatively large (functional) program.

We start in Section 2, by discussing the critical role of a non-strict semantics in our work. A higher-order style of programming allows us to build a model of our animation system as a collection of functions. We identify the separate concerns of *editing* movies in Section 3, and of *animating* characters in Section 4. This is brought together in a series of examples in Section 5. Our earliest inspiration for work on functional graphics came from the paper by Peter Henderson (1982) on 'Functional Geometry'. We briefly discuss this in Section 6.

All the functions introduced in this paper exist as part of a (1500 line) functional animation system written in Haskell (1990), running under Unix.[2] The user interacts with the functional system to produce movies which are displayed using an X11 preview program.

2 The role of laziness

The non-strict semantics of our functional environment – with lazy evaluation as its operational manifestation – is the basis of our approach to animation. The strength of this approach lies in the way it allows us to reason about sequences – especially infinite ones. Lazy evaluation may be informally described as an expression evaluation strategy where a sub-expression is only evaluated when its value is required. A consequence is that we can write definitions of infinite data-structures.

[2] A complete listing of this program may be obtained from The Haskell Project, Yale University, Department of Computer Science, Box 2158 Yale Station, New Haven, Ct 06520 USA, or by sending electronic mail to *haskell-request@cs.yale.edu*.

Consider the function *osc* (a function we use frequently) which oscillates a given sequence:

$$\begin{aligned} &osc :: [a] \rightarrow [a] \\ &osc\, s \mid (length\, s) == 0 = [\,] \\ &osc\, s \mid (length\, s) == 1 = head\, s : osc\, s \\ &osc\, s \qquad\qquad\qquad = s +\!\!+ ((tail\,.\,reverse\,.\,tail)\, s) +\!\!+ (osc\, s). \end{aligned}$$

Its use:

$$\begin{aligned} &osc\,[1,2,3] \\ &[1,2,3,2,1,2,3,2,1,2,3,\ldots. \end{aligned}$$

The function *osc* takes a sequence of finite length and returns an 'oscillated' sequence of (potentially) infinite length. If we typed the above expression (*osc* [1, 2, 3]) into the machine, it would go into an 'endless loop' in building the infinite structure. However, by using another function, *take*, we prevent the building of the infinite structure by requesting only a finite section of it:

$$\begin{aligned} &take\, 10\, (osc\,[1,2,3]) \\ &[1,2,3,2,1,2,3,2,1,2]. \end{aligned}$$

Laziness guarantees that only the part of the sequence that is actually used in computing the result is generated. This gives the advantage in animation that we do not need to fix the length of the finished sequence in advance by specifying the length of the component sequences. We reason in terms of component programs generating infinite sequences from which we take the length we need. Laziness frequently leads to substantial improvements in the efficiency of certain patterns of processing over sequences (Wadler, 1985). We have used it extensively in our work to program animation sequences, and also to construct functional processes modelled as lazy stream processing functions (Arya, 1989).

3 Functional movies

Animation consists of manipulating sequences of pictures or movies. A movie consists of a sequence of pictures made up of closed sets of polygons. We use the words *frame* and *picture* interchangeably to refer to a picture in the context of animation. The polygons need to be closed to allow us to model opacity to use pictures to obscure other pictures. This approach necessarily makes lines and open-polygons special cases of a polygon:

$$\begin{aligned} &\textbf{type}\, Movie = [Pic] \\ &\textbf{type}\, Pic \quad\; = [(Colour, [Vec])] \\ &\textbf{type}\, Colour = Int \\ &\textbf{type}\, Vec \quad\; = (Int, Int). \end{aligned}$$

The type 'Vec' corresponds to a vector ('(op_1, op_2)' is the Cartesian product operator which constructs a set of pairs). Thus pictures consist of a sequence of (coloured) polygons each of which consists of a sequence of vertices and an associated colour-value. These colours are actually assigned in the animation previewer where the numbers are mapped onto actual colours.

Our movie-operations are discussed under four headings. Section 3.1 illustrates how movies are created, and Section 3.2 presents the functions which combine movies to give new movies. *Cueing* allows us to stagger the activation of animation sequences or to trigger them by various means – this is discussed in Section 3.3. In Section 3.4 we present the interpolation function, which allows us to modify a picture into another picture over a number of steps. We have not made any mention of operations that geometrically transform movies (move, rotate, etc.); these are regarded as *behaviours*, and are discussed in the next section. The reader is referred to the appendix, and to Arya (1988) for a complete description of the functions available.

3.1 Creating movies

Consider animating a man going through the motion of walking. We start by creating a sequence of *key-frames* for each character – this consists of pictures capturing the key elements of its action (which may already exist as a library sequence):

$$\begin{aligned} man &= osc\,[man1, man2, man3] \\ &= [man1, man2, man3, man2, man1, man2, \\ &\quad man3, man2, man1, \ldots]. \end{aligned}$$

The function *osc* takes a finite sequence of key-frames and returns an infinite sequence of oscillating key-frames that show the man walking on the same spot *ad infinitum*. For the rest of this section we assume the existence of the following definitions:

$$\begin{aligned} man &= osc\,[man1, man2, man3, man4] \\ gull &= osc\,[gull1, gull2] \\ ball &= osc\,[ball1]. \end{aligned}$$

Henceforth, we refer to this sort of primitive movie – consisting of a sequence of cycled key-frames – as a *character*. It is quite reasonable to reason in terms of characters rather than in terms of individual pictures, since this is how a conventional animator reasons about lengths of film. In animation, when we see a stationary picture on the screen we are seeing the same picture at the rate of 25 frames/s. It is this intuitive understanding of animation that we have modelled. We shall keep drawing the attention of the reader to this point since it is quite important. It may be noted that most of our primitive functions work on movies of potentially infinite length.

3.2 Editing movies

Once we have created instances of movies we want to combine them in various ways. Some of the combining forms that let us 'edit' movies are:

$$\begin{aligned} &overlay :: && Movie \rightarrow Movie \rightarrow Movie \\ &put :: && [Vec] \rightarrow Movie \rightarrow Movie \rightarrow Movie \\ &behind :: && Movie \rightarrow Movie \rightarrow Movie. \end{aligned}$$

Each of the functions takes several sequences as arguments and returns a result only as long as the shortest argument-sequence. The function *overlay* takes two movies and returns a result in which the corresponding frames are overlaid such that the 'world-origins' of the two 'foils' coincide.

Consider the following movie:

overlay flying_bird walking_man.

This animation shows the two threads of action, of the walking man and the flying bird, proceeding simultaneously. We overlay the frames of the movie onto a single frame to show the cumulative effect. The man walks rightward while the bird flies from the bottom-left corner of the screen towards the top-right corner. Figure 1 shows the result of collapsing the first 11 frames of this animation into a single frame. We may, instead of using the function *overlay*, choose one of the other combining forms to combine the component pictures in some other fashion. *Put* combines the two movies similarly but displaces the pictures of the first (relative to the second) by amounts in the given sequence of vectors. *Behind* does a shielding operation by using the pictures in the second sequence to mask out overlapping portions of the pictures in the first sequence. This allows us to implement the notion of *depth* in an animation.

Each picture in a movie is contained in an imaginary box bounding its maximum dimensions. Operations such as *above*, *beside* and *over*, let us combine movies in a manner which preserves symmetry with respect to the bounding boxes of the component pictures.

3.3 Cueing

A simple variety of cueing may be arranged by exploiting the time-ordering implied in the sequence of pictures. Given below are brief descriptions of the sorts of functions available:

```
then      :: [a] → [a] → [a]
wait      :: Int → [a] → [a]
appear    :: Int → [a] → [a]
disappear :: Int → [a] → [a].
```

Suppose we want the bird to fly after the man has walked, we use *then* – this is really just the *append* operator (++):

(*take* 30 *walking_man*) '*then*' *flying_bird.*

Note that if we did not use *take* on the first sequence, it would not finish and we would not see the bird. To make the bird wait *n* frames before appearing, we use wait:

overlay walking_man (*wait n flying_bird*)

The function *wait* appends a given number of empty frames to the front of the sequence; functions *appear* and *disappear* cause the character to appear and to

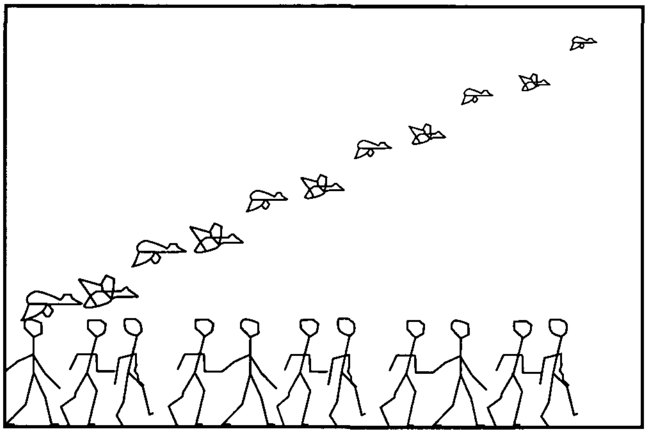
Fig. 1. Take 11 (overlay flying_bird walking_man)

disappear a given number of frames into its action, respectively; *appear n* drops the first *n* frames of the movie and *disappear n* follows the *n*th frame with the infinite *nullseq*. There are a variety of other functions such as *freeze*, *while*, and *until* which implement other kinds of cueing.

3.4 Interpolation

We have a number of flavours of interpolation. One of them is *inbetween* which takes a number and two pictures as argument and returns a sequence of pictures where the animation starts off as the first picture and ends as the second picture over the given number of frames:

$$inbetween :: Int \rightarrow Pic \rightarrow Pic \rightarrow [Pic].$$

This 'inbetweening' algorithm may be as sophisticated as we like – however, we have chosen a simple, linear interpolation between corresponding vertices in the component pictures. 'Corresponding' here implies interpolation by definition-order. Consider a lazy way of animating a palm tree. We have one picture (*palm*1 see Fig. 2) of a palm tree from which we may derive a sequence of key-frames as follows:

```
palm :: Movie
palm
       = osc palmframes
         where
         palmframes
                   = inbetween 3 palm1 (flipx_pic 100 palm1).
```

In this example the third argument to *inbetween* is palm1 'flipped' in the '$x = 100$' axis. The key-frames of the palm are derived from a single picture that has been interpolated over three frames to give a sequence of key-frames. The resulting key-frames may be turned into a primitive palm tree sequence of a palm tree swinging from side to side, *ad infinitum*.

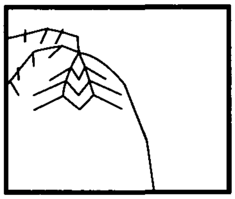

Fig. 2. Palm

4 Behaviour

The last section discussed (editing) functions which allow us to combine and manipulate movies in various ways. Here we describe the *animation* aspect. Again we wish to have a style of combining simpler animations to construct more complex animation. *Behaviour* consists of a sequence of *changes* that may be undergone by a character in a movie:

$$\textbf{type}\, Behaviour = [Pic \rightarrow Pic].$$

Each such *change* is a function of type $(Pic \rightarrow Pic)$. For instance, the behaviour-producing function *mov* returns a behaviour which when *applied* to the key-frames denoting a *character* moves it to the right by ten units over each consecutive frame:

$$\begin{aligned} &mov\ \ :: [Vec] \rightarrow Behaviour \\ &mov4 :: Behaviour \\ &mov4 = mov\, [(10,0),(20,0),(30,0),(40,0)]. \end{aligned}$$

Here *mov* is given a sequence of vectors as argument and returns an instance of *behaviour* (i.e. a sequence of partially applied picture-operations):

$$mov4 = [mov\_pic\,(10,0), mov\_pic\,(20,0), mov\_pic\,(30,0), \ldots].$$

To see the effect of such a behaviour on a given *character* it has to be *applied* using the function *apply*:

$$apply :: Behaviour \rightarrow Movie \rightarrow Movie$$

The function *apply* makes behaviours work – it takes a *behaviour* and a *movie* and returns a movie where the corresponding changes in the behaviour-sequence have been applied to the frames of the movie:

$$apply\ mov4\ walking\_man.$$

The function *apply* simply applies each function in the behaviour sequence to the corresponding pictures in the movie. The sequence of changes in the behaviour are not being accumulatively applied to a *single* picture but separately to the *consecutive* pictures in the movie sequence.

It is laborious to have to construct behaviour from scratch each time we want to animate a character – in Section 4.1 we show how primitive behaviours may be created. These may be combined to give more complicated behaviour – as we see in Section 4.2.

4.1 Creating behaviour

The following may be regarded as our primitive behaviour-creating functions. Each takes a sequence of arguments and returns an instance of behaviour:

```
mov   :: [Vec] → Behaviour
rot   :: [Int] → Behaviour
scale :: [Int] → Behaviour
flipx :: [Vec] → Behaviour
flipy :: [Vec] → Behaviour
flip  :: Behaviour.
```

Given a sequence of vectors, *mov* returns a behaviour that when *applied* to a movie returns a new movie where each picture has been moved (relatively) by the corresponding amount. Should the length of the sequence of vectors be shorter than the movie, the movie is cut-off and replaced by the infinite *nullseq* at the point. The function *rot* takes a sequence corresponding to the angles of rotation and rotates the pictures of a given sequence about their centre. Here, 'centre' refer to the centre of the bounding-box containing the picture. The function *scale*, given a sequence of real numbers, scales the sequence of pictures using the centre as the centre of scaling. Functions *flipx* and *flipy* reflect the picture of a sequence using the appropriate axis passing through the given point. The function *flip* inverts the pictures of each sequence laterally using the vertical axis passing through the centre of the bounding-box.

It is laborious to build behaviours from primitive functions – we resort to the primitive style only when we cannot work at a higher level. By partially invoking the functions (*mov*, *rot*, *scale*, etc.) with default values, we build up a useful collection of commonly used behaviours (each of which has an infinite duration): *left*, *right*, *up*, *down*, *cw*, *ccw*, *bigger*, *smaller*.

For example, consider animating a cloud scene, given movies of just two clouds (*flat_cloud* and *puff_cloud*); as shown in Fig. 3, we get:

```
clouds :: Movie
clouds = overlay
             [apply right puff_cloud,
              apply (right @ right @ smaller) flat_cloud].
```

Here, the two clouds start at the origin and move rightwards, the *flat_cloud* moving at twice the speed of *puff_cloud* and appearing to recede into the distance.

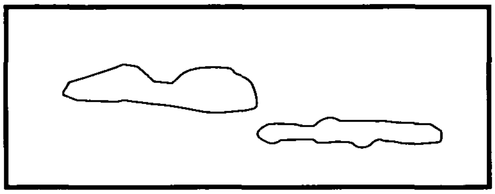

Fig. 3. Clouds

4.2 Combining behaviours

Behaviours may be combined by 'parallel' composition ('@') or by 'sequential' composition (';'):

$$(@) :: Behaviour \rightarrow Behaviour \rightarrow Behaviour$$

$$(;) :: Behaviour \rightarrow Behaviour \rightarrow Behaviour.$$

We use the infix form of the two functions. The expression '*a* @ *b*' returns the sequence of changes where the effect of both behaviours *a* and *b* are seen in each frame of the animation. The expression 'a;b' denotes the sequence of changes *a* followed (accumulatively) by the sequence denoted by *b*. Consider the informal definitions:

$$f, g :: Behaviour$$

$$f = [f_1, f_2, \ldots, f_n]$$

$$g = [g_1, g_2, \ldots, g_n]$$

$$f @ g = [f_1 \cdot g_1, f_2 \cdot g_2, \ldots, f_n \cdot g_n]$$

$$f; g = [f_1, f_2, \ldots, f_n] +\!\!+ [g_1 \cdot f_n, g_2 \cdot f_n, \ldots, g_n \cdot f_n].$$

Note how in '*f*;*g*' the last element of *f* is composed onto all the elements of *g*. The reason for this is to 'have *g* continue where *f* ended'. For example, each of the 'dots' in Figs 4 and 5 exists on a separate frame. The combination 'bounce@slide' produces an additive effect in which the composed effect of the *bounce* and the *slide* is applied to each frame of the *ball* sequence. The combination 'bounce;slide' shows

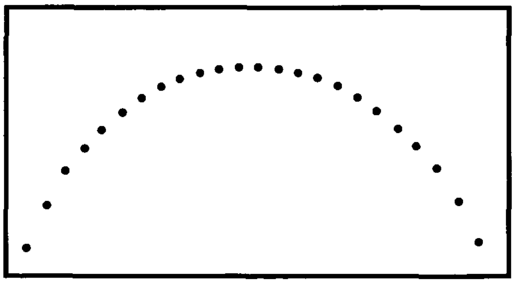

Fig. 4. Apply (bounce@slide) ball

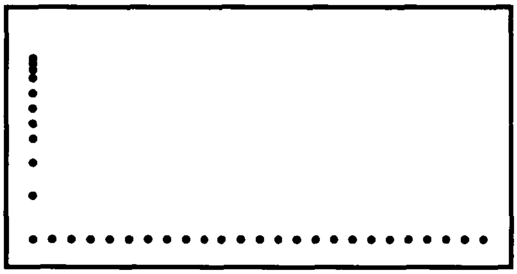
Fig. 5. Apply (bounce;slide) ball

the changes produced by the *bounce* behaviour followed in sequence by the changes produced by the *slide* behaviour. (Note: it is pure coincidence that in this example the ball returns to its original position before being affected by the *slide* behaviour.)

5 Extended example

We illustrate how the material presented earlier may be used to construct animated sequences. In Section 5.1 we animate a seaside scene, and in Section 5.2 we animate a river scene – each illustrates slightly different elements of animation. In Section 5.3 we show how behaviours may be combined to give complex motions, such as those of planets with their moons, in a solar system.

5.1 Seaside

Consider animating a seaside scene with a vending-machine and a walking-man in the foreground. In the background we have some gulls flying, a sun rising out of the sea and a couple of palm-trees swaying in the foreground (see Fig. 6).

We begin by hierarchically decomposing the animation into its constituents and start with the animation of the man and the vending machine. We have four instances of gull, each behaving a bit differently from the others:

apply (*bPar* [*right,bigger*]) *gull*

apply (*bPar* [*right, right, small, bigger*]) (*tail gull*)

apply (*bPar* [*up, up, right, small, bigger*]) *gull*

apply (*bPar* [*up, right, right, right*]) (*tail gull*).

The function '*bPar*' (defined '*fold* (@)') takes a sequence of behaviours and composes them in parallel (using ' @ '), returning a new behaviour as a result. The first gull flies rightward becoming bigger on each frame. The second starts off being a small gull and then flies rightward by a greater degree, becoming bigger on each frame. The expression 'tail gull' denotes a gull whose key-frame sequence is one frame out of synchronization with 'gull'. The third starts off being small and becomes

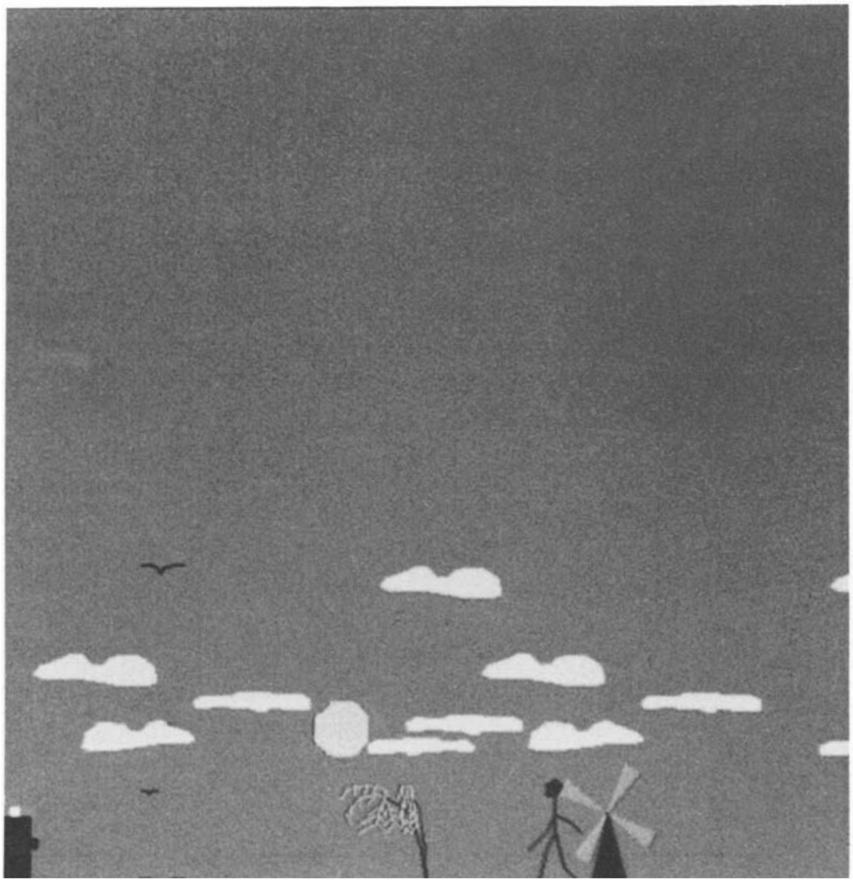

Fig. 6. Seaside

bigger while following a steeper trajectory. We also have an animated 'sun' and a couple of palm trees:

apply (*bPar* [*up*, *cw*, *movto* (*repeat botm*)]) *sun*

apply (*mov* (*repeat botm*)) *palm*

apply (*mov* (*repeat botm*)) (*tail palm*).

The expression '(*mov* (*repeat* v))' denotes a behaviour which displaces a given character by the given amount (v). The expression '*repeat* v' denotes the infinite sequence $[v, v, v, \ldots]$'. The sun is made to rotate and to move upwards from the middle of the lower part of the screen (*botm*). We have already met *palm*. Finally, we need some clouds – we use a slightly more complicated version of *clouds* than that introduced earlier. Here *flat_cloud* and *puff_cloud* have been flipped and scaled to simulate a multitude of clouds:

apply left (*rBESIDE* [*clouds*, *clouds*, *clouds*]).

We place three copies of clouds beside each other to give the effect of a continuous cloud layer moving across the screen. The function '*rBESIDE*' (defined '*fold beside*')

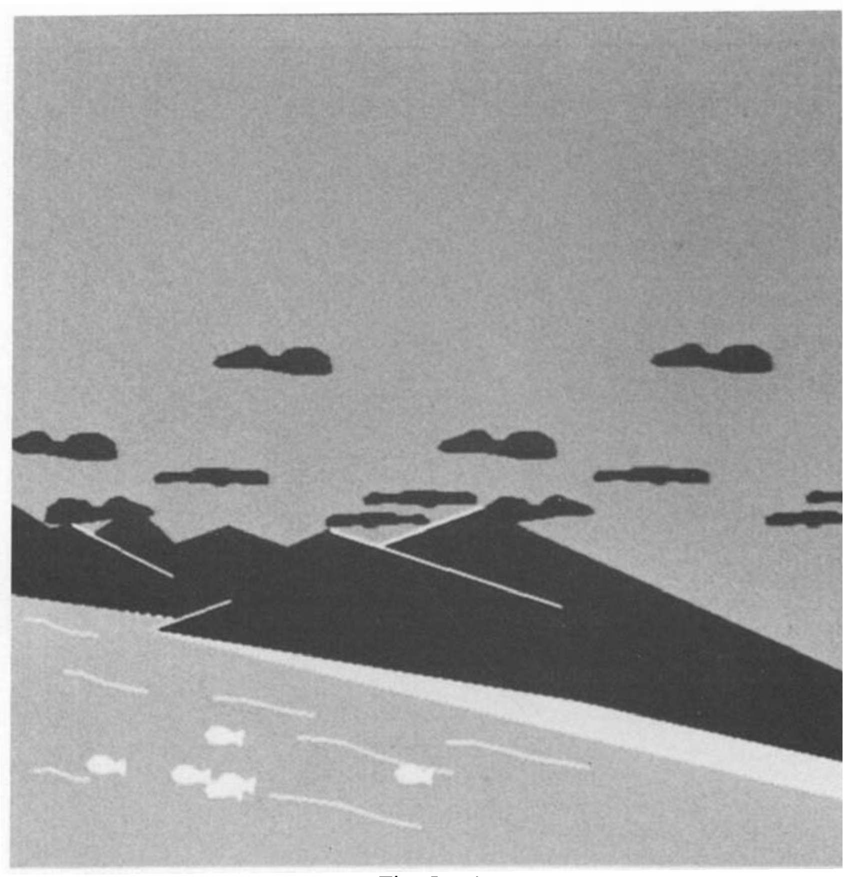
Fig. 7. River

inserts the combinator *beside* between a sequence of movies. The function '*rOVERLAY*' (defined '*fold overlay*') takes a sequence of movies and combines them in the result. The complete program for the animation follows:

```
seaside :: Movie
seaside
  = rOVERLAY
      [man_and_vm,
       apply (bPar [right, bigger]) gull
       apply (bPar [right, right, small, bigger]) (tail gull)
       apply (bPar [up, up, right, small, bigger]) gull
       apply (bPar [up, right, right, right]) (tail gull)
       apply (bPar [up, cw, movto (repeat botm)]) sun
       apply (mov (repeat botm)) palm
       apply (mov (repeat botm)) (tail palm)
       apply left (rBESIDE [clouds, clouds, clouds])
      ]
```

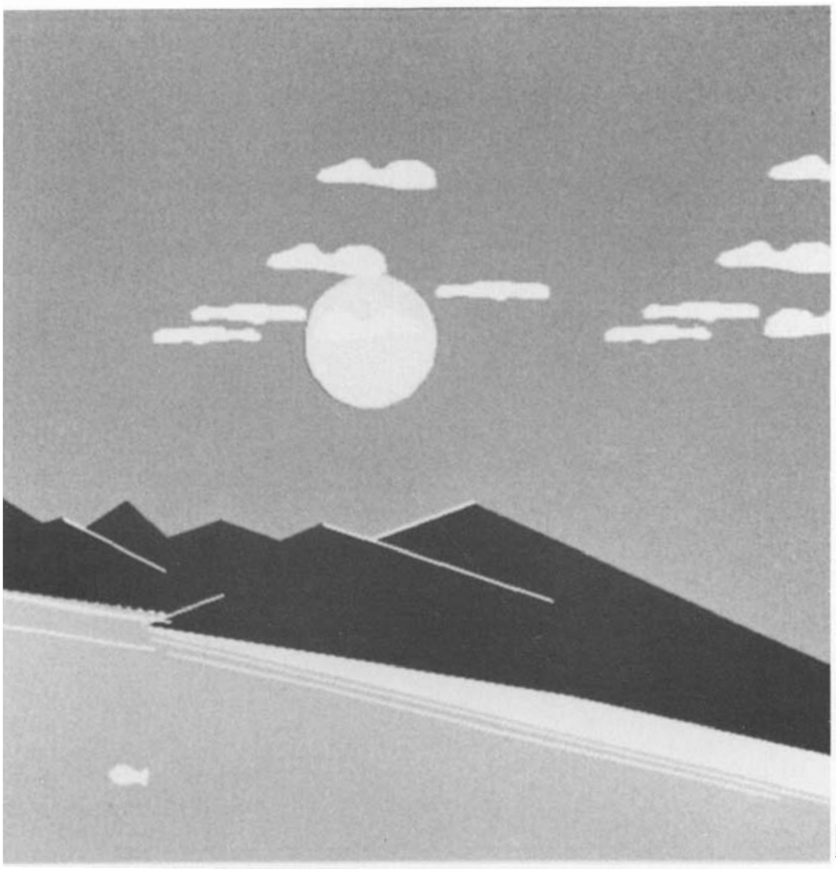
Fig. 8. River (10 frames into animation)

5.2 River

The river scene is similar to the seaside scene (see Figs. 7 and 8). Here a sun rises from behind some hills and as it emerges, it turns from red to yellow, simultaneously turning the clouds from black to white. In the foreground, we have a river with fish and ripples that spread to merge with the shoreline.

As can be seen from the implementation below, the (red) sun rises for 10 frames and then changes colour to yellow and keeps upwards *ad infinitum*. The function '*i*' (*iterate*) takes a single element and returns an infinite sequence of that element, and allows us to use single values with functions that expect infinite sequences.

The expression '((*do 10 up*); (*mov* (*i* (*360, 180*))))' moves the sun to the position (360, 180) in the scene, and ensures that that initial translation is mapped over each of the subsequent frames. The clouds are initially black and then change to white on the tenth frame. The fish are all moving either towards the left or diagonally upwards to the left:

```
river_scene :: movie
river_scene
   = rOVERLAY
```

```
[blue_sky,
 apply (((do 10 up) @ (mov (i (360, 180))))
         ; (up @ (set_color (i yellow))))
         orb,
 apply (bSeq [(take 10 (set_color (i black))) @
                (mov (i (100, 250))),
                up @ (set_color (i white))])
                clouds,
 river,
 hills,
 apply (left @ (mov (i (196, 98)))) fish
 apply (left @ left @ (mov (i (126, 110)))) fish
 apply (up @ left @ (mov (i (235, 114)))) fish
 apply (left @ (mov (i (235, 81)))) fish
 apply (left @ left @ (mov (i (279, 88)))) fish
 apply (left @ (movto (i (500, 110)))) fish
 ripples
].
```

Ripples are interpolations of coloured lines (*ripple*1–7) into the shorelines (*shore*1–2) as follows:

```
ripples ::  movie
ripples = rOVERLAY
              [osc (inbetween 12 ripple1 shore1)
               osc (inbetween 20 ripple2 shore2)
               osc (inbetween 16 ripple3 shore2)
               osc (inbetween 25 ripple4 shore1)
               osc (inbetween 12 ripple5 shore2)
               osc (inbetween 30 ripple6 shore2)
               osc (inbetween 20 ripple7 shore1)
              ].
```

The following are examples of ripples:

```
ripple1 = [(white, [(76, 223), (92, 223), (120, 209), (162, 200)])]
ripple2 = [(white, [(363, 144), (382, 145), (395, 143), (410, 134), (433, 126), (460, 123)])]
```

5.3 Planets

We wish to model a simple solar-system with a sun in the centre and plants following a circular orbit around it (see Fig. 9). Each planet has one moon orbiting it. If we were to specify the behaviours of each entity separately, the moons would turn out to have

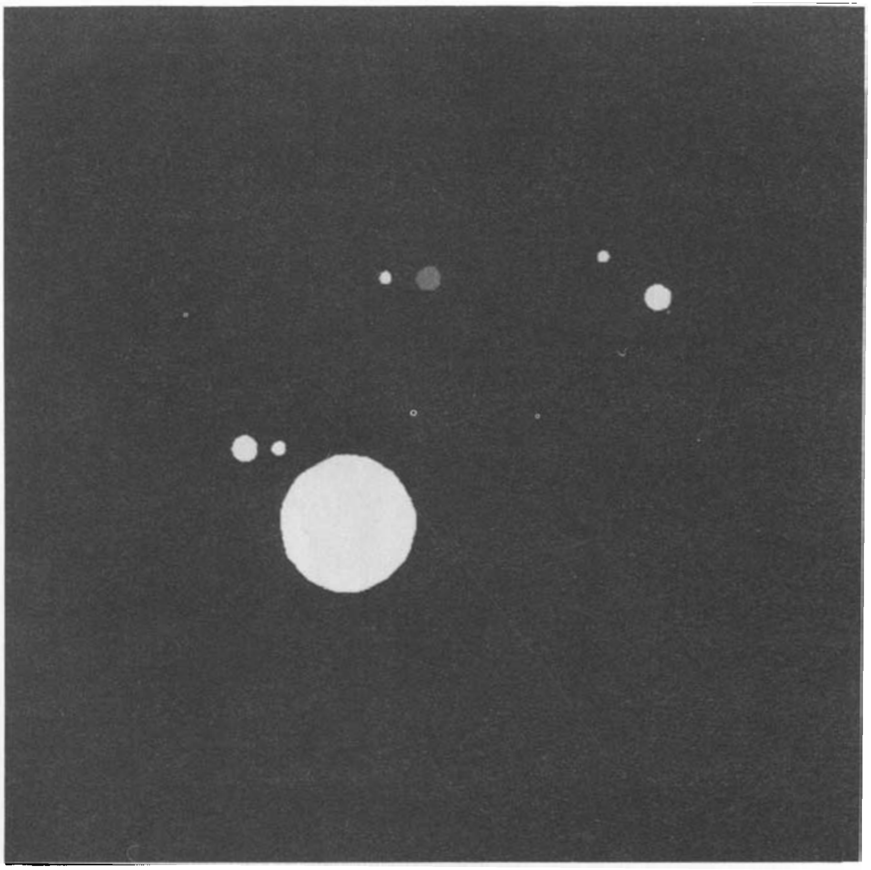

Fig. 9. Planets

extremely complex motions. However, we may break down the motions in the scene to orbital motions. These may be combined quite simply to give more complex motions.

We describe the animation in a top-down manner. We have a background coloured blue, with a yellow sun. In the foreground we have three planet-moon pairs, each returned by the function '*planets*'. The arguments of this function are, in order, the speed of the planet, the speed of the moon, the radius of the planet, the radius of the moon, the colour of the planet and the colour of the moon:

```
planet_scene ::  Movie
planet_scene = rOVERLAY
                    [apply (set_color (i white)) blue_sky,
                     apply ((set_color (i yellow)) @ (movto (i center))) orb,
                     planets (pi.40) (pi/10) 450 80 blue sky_blue,
                     planets (pi/20) (pi/8) 300 50 brown red,
                     planets (pi/10) (pi/4) 150 40 green white
                    ]
```

The function '*planets*' uses the function '*circ_move*' to construct the circular-motion behaviour which is applied to the figure in the animation:

```
planets :: Int -> Int -> Int -> Int -> Int -> Int -> Movie
planet i1 i2 r1 r2 c1 c2
  = rOVERLAY
      [apply f1 earth,
       apply (f1 @ f2) moon
      ]
      where
      f1 = circ_mov r1 i1
      f2 = circ_mov r2 i2
      earth = osc [movto_pic (vplus center (r1, 0)) (circle c1 15)]
      moon = osc [movto_pic (vplus center (r1 + r2, 0)) (circle c2 7)]
```

The definition of '*circ_move*' follows. We use here the curve-generator function '*gen*' to generate an increasing sequence of values:

```
circ_mov :: Int -> Int -> Behaviour
circ_mov r inc = mov (map (vmin' (hd vs)) vs)
                   where vs = [(r * (cos theta), r * (sin theta)) |
                                 theta <- gen 0 inc 0 (2 * pi)]
                         vmin' x y = vmin y x
```

6 Related work

The original motivation for our work on functional graphics is Henderson's (1982) paper on 'Functional Geometry'. In this paper, Henderson developed a functional technique for the specification of recursive pictures such as Escher's 'Square Limit' (or 'Fish Limit'). This work, in the Lispkit functional language, gave the stimulus which led to our own work on functional graphics. A small set of functions allow us to combine and transform pictures in a variety of ways. These pictures consist of sets of lines which may only be displayed relative to a *bounding box* which defines the context. The stretching or shrinking of the bounding-box affects the enclosed picture. Pictures may be combined by using the functions *overlay*, *above* or *beside* – the effects are similar to those of our functions discussed earlier. There are other primitives such as *rot* (rotate ninety degrees) and *flip* (lateral inversion).

The functional geometry functions were designed for the specification of recursive (Escher) pictures for which they are well suited. For the purpose of animation and graphics, the idiosyncracies of these functions are of questionable value. However, this approach led us to consider a different interpretation for the combining forms, which led to our own work on functional animation (Arya 1986).

7 Conclusion

We have presented a functional approach to a variety of key-frame animations. We started by constructing an abstraction of a movie. We then devised a compact set of primitive operations over it. The resulting system may be used to construct denotational descriptions of animation sequences. The user is encouraged to adopt a higher-order approach, and to resolve problems into their primitive components and to combine these to construct more complex components.

As graphic hardware becomes cheaper, the cost of computer animation is bound increasingly to the cost of the skilled manpower involved. By devising a system that is easier to understand, it becomes easier to prototype animation sequences fairly quickly. This has already been seen with systems such as the MacLisp-based ASAS (Reynolds, 1982) which provides a Lisp-based interface to animation software. We find that just as functional programs are easier to read and to understand than their imperative counterparts, so are functional animation scripts. We believe that the rapid-prototyping advantage conferred by functional languages has a definite potential in devising such systems. The lessons learnt from building these prototypes may also be used to construct production-quality systems using conventional languages.

Acknowledgements

The D.Phil. thesis leading to this work was funded by the UK's Science and Engineering Council (SERC) and supervised by Professor Ian Page at the Programming Research Group at Oxford University. Further work was done at IBM TJ Watson Research Center (Yorktown Heights), where I also collaborated with Professor Paul Hudak and Dr R. S. Sundaresh at Yale University. Professor Hudak has been a source of great encouragement from the earliest time. His students, Ami Kleinman and John Tinmouth, have ported the system to the Haskell language – and have systematically documented the implementation in their reports (Tinmouth, 1991; Kleinman, 1990). Ami Kleinman has also extended the 2D system to a 3D version.

References

Arya, K. (1986) A functional approach to animation, *Computer Graphics Forum*, **5** (4): 297–311.

Arya, K. (1988) *The Formal Analysis of a Functional Animation System*, D.Phil. thesis, Oxford University, UK.

Arya, K. (1989) Processes in functional animation. *Proc. ACM Functional Programming Languages and Computer Architecture (FPCA '89) Conference*, Imperial college, London, UK.

Haskell. (1990) Report on the Programming Language Haskell, Version 1.0. YALEU/DCS/RR-777, Yale University, USA.

Henderson, P. (1982) Functional Geometry. *Symposium on Lisp and Functional Programming.*

Kleinman, A. (1990) A three dimensional graphics animation system in Haskell. Computer Science Senior Project, Yale University, 9 May.

Magnetat-Thalman, N. and Thalman, D. (1984) CINEMIRA: a 3D computer animation language based on actor and camera data-types. Technical Report, University of Montreal.

Magnetat-Thalman, N. and Thalman, D. (1985) *Computer Animation: Theory and Practice*. Springer-Verlag.

Tinmouth, J. (1991) A functional animation package in Haskell. Computer Science Senior Project, Yale University, 9 May.

Reynolds, C. W. (1982) Computer animation with scripts and actors. *Proc. Siggraph '82, Volume 16* (*3*), July.

Wadler, P. (1985) Listlessness is better than Laziness. Volume 217 of *Lecture Notes in Computer Science*, Springer-Verlag.