Reactive Animations

• Todays Topics
  – Simple Animations - Review
  – Reactive animations
  – Vocabulary
  – Examples
  – Implementation
    » behaviors
    » events

• Reading from Hudak’s The Haskell School of Expression
  – Read Chapter 15 - A Module of Reactive Animations
  – Read Chapter 17 – Rendering Reactive Animations
Review: Behavior

• A Behavior \( a \) can be thought of abstractly as a function from Time to \( a \).

• In the chapter on functional animation, we animated Shape's, Region's, and Picture's.

• For example:

\[
\begin{align*}
\text{dot} &= (\text{ell} \ 0.2 \ 0.2) \\
\text{ex1} &= \text{paint red} \ (\text{translate} \ (0, \ \text{time} / 2) \ \text{dot}) \\
\text{ex2} &= \text{paint blue} \ (\text{translate} \ (\sin \ \text{time}, \cos \ \text{time}) \ \text{dot})
\end{align*}
\]

Try It
Abstraction

- The power of animations is the ease with which they can be abstracted over, to flexibly create new animations from old.

\[ \text{wander } x \ y \ \text{color} = \text{paint color (translate } (x,y) \text{ dot)} \]

\[ \text{ex3 = wander } (\text{time } /2) \ (\text{sin time}) \ \text{red} \]
Example: The bouncing ball

• Suppose we wanted to animate a ball bouncing horizontally from wall to wall

- The Y position is constant, but the x position varies like:
modula \( x \mod y \) = (period, w)

where \((\text{whole}, \text{fract}) = \text{properFraction} \ x\)

\( n = \text{whole} \mod y \)

\( \text{period} = (\text{whole} \div y) \)

\( w = \text{fromIntegral (toInteger n)} \)

+ fract
Time `mod` N

1 id 2 (N-) 3 negate 4 (-N)
Implementation

\[\text{bounce } t = f \text{ fraction} \]
\[\text{where (period,fraction) = modula } t 2\]
\[f = \text{funs } !! (\text{period `mod` 4})\]
\[\text{funs = [id,(2.0 -),negate,(\x \to x - 2.0)]}\]

\[\text{ex4 = wander (lift1 bounce time) 0 yellow}\]

- Remember this example. Reactive animations will make this much easier to do.
Reactive Animations

- With a reactive animation, things do more than just change and move with time according to some algorithm.

- Reactive programs “react” to user stimuli, and real-time events, even virtual events, such as:
  - key press
  - button press
  - hardware interrupts
  - virtual event - program variable takes on some particular value

- We will try and illustrate this first by example, and then only later explain how it is implemented

- Example:

  color0 = red `switch` (lbp ->> blue)
  moon = (translate (sin time,cos time) dot)
  ex5 = paint color0 moon
A Reactive Vocabulary

• Colors
  - Red, Blue, Yellow, Green, White :: Color
  - red, blue, yellow, green, white :: Behavior Color

• Shapes and Regions
  - Shape :: Shape -> Region
  - shape :: Behavior Shape -> Behavior Region
  - Ellipse, Rectangle :: Float -> Float -> Region
  - ell, rec :: Behavior Float -> Behavior Float -> Behavior Region
  - Translate :: (Float, Float) -> Region -> Region
  - translate :: (Behavior Float, Behavior Float) -> Behavior Region -> Behavior Region
Operator and Event Vocabulary

- **Numeric and Boolean Operators**
  - `(+)`, `(*)` :: `Num a => Behavior a -> Behavior a -> Behavior a`
  - `negate` :: `Num a => Behavior a -> Behavior a`
  - `(*)*, (<*)*, (>=*), (<=*)` :: `Ord a => Behavior a -> Behavior a -> Behavior Bool`
  - `(&&*), (||*)` :: `Behavior Bool -> Behavior Bool -> Behavior Bool`

- **Events**
  - `lbp` :: `Event ()` -- left button press
  - `rbp` :: `Event ()` -- right button press
  - `key` :: `Event Char` -- key press
  - `mm` :: `Event Vertex` -- mouse motion
Combinator Vocabulary

• Event Combinators
  - (->>>) :: Event a -> b -> Event b
  - (=>>>) :: Event a -> (a->b) -> Event b
  - (.|.) :: Event a -> Event a -> Event a
  - withElem :: Event a -> [b] -> Event (a,b)
  - withElem_ :: Event a -> [b] -> Event b

• Behavior and Event Combinators
  - switch :: Behavior a -> Event(Behavior a) -> Behavior a
  - snapshot_ :: Event a -> Behavior b -> Event b
  - step :: a -> Event a -> Behavior a
  - stepAccum :: a -> Event(a -> a) -> Behavior a
Analyse Ex3.

red, blue :: Behavior Color
lbp :: Event ()
(->>>) :: Event a -> b -> Event b
switch :: Behavior a -> Event(Behavior a) -> Behavior a

color0 = red `switch` (lbp ->>> blue)
Either (\.|.) and withElem

color1 = red `switch`
    (lbp `withElem_` cycle [blue,red])

ex6 = paint color1 moon

color2 = red `switch`
    ((lbp ->>> blue) .|. (key ->>> yellow))

ex7 = paint color2 moon
Key and Snapshot

color3 = white `switch` (key =>> \c ->
  case c of 'r' -> red
    'b' -> blue
    'y' -> yellow
    _   -> white  )

ex8 = paint color3 moon

color4 = white `switch` ((key `snapshot` color4) =>>
  \(c,old) ->
    case c of 'r' -> red
      'b' -> blue
      'y' -> yellow
      _   -> constB old)

ex9 = paint color4 moon
Step :: a -> Event a -> Behavior a

size '2' = 0.2  -- size :: Char -> Float
size '3' = 0.4
size '4' = 0.6
size '5' = 0.8
size '6' = 1.0
size '7' = 1.2
size '8' = 1.4
size '9' = 1.6
size _ = 0.1

growCircle :: Char -> Region
growCircle x = Shape(Ellipse (size x) (size x))

ex10 = paint red (Shape(Ellipse 1 1)
        `step` (key =>> growCircle))
stepAccum :: a -> Event(a -> a) -> Behavior a

• stepAccum takes a value and an event of a function. Everytime the event occurs, the function is applied to the old value to get a new value.

\[
\text{power2} :: \text{Event}(\text{Float} \to \text{Float})
\]
\[
\text{power2} = (\text{lbp} \to \to \ x \to x \times 2) \mid \mid (\text{rbp} \to \to \ x \to x \times 0.5)
\]

\[
\text{dynSize} = 1.0 \ `\text{stepAccum}\` \ \text{power2}
\]
\[
\text{ex11} = \text{paint red (ell dynSize dynSize)}
\]
Integral

- The combinator:
  
  \[
  \text{integral} :: \text{Behavior Float} \to \text{Behavior Float}
  \]

  has a lot of interesting uses.

If \( F :: \text{Behavior Float} \) (think function from time to Float) then \( \text{integral } F \ z \) is the area under the curve gotten by plotting \( F \) from 0 to \( z \).
Bouncing Ball revisited

- The bouncing ball has a constant velocity (either to the right, or to the left).
- Its position can be thought of as the integral of its velocity.

\[
\begin{align*}
\text{If velocity is a constant 1} \\
1 & \quad 2 & \quad 3 & \quad 4 & \quad 5 & \quad 6 & \quad 7 & \quad 8 & \quad \ldots
\end{align*}
\]

- At time t, the area under the curve is t, so the x position is t as well. If the ball had constant velocity 2, then the area under the curve is 2 * t, etc.
Bouncing Ball again

```haskell
ex12 = wander x 0 yellow
    where xvel = 1 `stepAccum` (hit >>= negate)
        x = integral xvel
        left = x <=* -2.0 &&* xvel <*0
        right = x >=* 2.0 &&* xvel >*0
        hit = predicate (left ||* right)
```

Mouse Motion

• The variable mm :: Event Vertex
• At every point in time it is an event that returns the mouse position.

```plaintext
mouseDot =
    mm =>> \ (x,y) ->
        translate (constB x, constB y)
        dot

ex13 = paint red (dot `switch` mouseDot)
```
How does this work?

- Events are “real-time” actions that “happen” in the world. How do we mix Events and behaviors in some rational way.
- The Graphics Library supports a basic type that models these actions.

```haskell
type Time = Float

data G.Event
    = Key { char :: Char, isDown :: Bool }
    | Button { pt :: Vertex, isLeft, isDown :: Bool }
    | MouseMove { pt :: Vertex }
    | Resize
    | Closed
  deriving Show

type UserAction = G.Event
```
Type of Behavior

• In simple animations, a Behavior was a function from time. But if we mix in events, then it must be a function from time and a list of events.

• First try:

```haskell
newtype Behavior1 a =
  Behavior1 ([(UserAction,Time)] -> Time -> a)
```

User Actions are time stamped. Thus the value of a behavior \((\text{Behavior1 } f)\) at time \(t\) is, \(f \text{ uas } t\), where \(\text{uas}\) is the list of user actions.

Expensive because \(f\) has to “whittle” down \(\text{uas}\) at every sampling point (time \(t\)), to find the events it is interested in.
Solution

• Sample at monotonically increasing times, and keep the events in time order.

• Analogy: suppose we have two lists xs and ys and we want to test for each element in ys whether it is a member of xs

  - inList :: [Int] -> Int -> Bool
  - result :: [Bool]                                      -- Same length as ys
  - result1 :: map (inList xs) ys

• What’s the cost of this operation?

• This is analagous to sampling a behavior at many times.
If xs and ys are ordered ...

result2 :: [Bool]
result2 = manyInList xs ys

manyInList :: [Int] -> [Int] -> [Bool]
manyInList [] _ = []
manyInList _ [] = []
manyInList (x:xs) (y:ys) =
    if y<x
        then manyInList xs (y:ys)
    else (y==x) : manyInList (x:xs) ys
Behavior: Second try

newtype Behavior2 a =
  Behavior2 ([((UserAction, Time)) ->
  [Time] -> [a])

• See how this has structure similar to the manyInList problem?
  manyInList :: [Int] -> [Int] -> [Bool]
Refinements

newtype Behavior2 a =
    Behavior2 ([(UserAction, Time)] -> [Time] -> [a])

newtype Behavior3 a =
    Behavior3 ([UserAction] -> [Time] -> [a])

newtype Behavior4 a =
    Behavior4 ([Maybe UserAction] -> [Time] -> [a])

• Final Solution

newtype Behavior a
    = Behavior (([Maybe UserAction], [Time]) -> [a])
newtype Event a =
    Event (([Maybe UserAction],[Time]) -> [Maybe a])

- Note there is an isomorphism between the two types
  Event a and Behavior (Maybe a)

- We can think of an event, that at any particular time
  t, either occurs, or it doesn’t.

- Exercise: Write the two functions that make up the
  isomorphism:
  - toEvent :: Event a -> Behavior (Maybe a)
  - toBeh :: Behavior(Maybe a) -> Event a
Intuition

• Intuitively it’s useful to think of a Behavior $m$ as transforming two streams, one of user actions, the other of the corresponding time (the two streams always proceed in lock-step), into a stream of $m$ things.

• User actions include things like
  – left and right button presses
  – key presses
  – mouse movement

• User Actions also include the “clock tick” that is used to time the animation.

[ leftbutton, key ‘x’, clocktick, mousemove(x,y), … ]
[ 0.034, 0.65, 0.98, 1.29, … ]
[ M1, m2, m3, … ]
The Implementation

time :: Behavior Time
time = Behavior (\(_,ts) \rightarrow ts)

\[
([ua1,ua2,ua3, \ldots],[t1,t2,t3, \ldots]) \rightarrow
[t1, t2, t3, \ldots]
\]

constB :: a \rightarrow Behavior a
constB x = Behavior (\_ \rightarrow \text{repeat } x)

\[
([ua1,ua2,ua3, \ldots],[t1,t2,t3, \ldots]) \rightarrow
[x, x, x, \ldots]
\]
Simple Behaviors

red, blue :: Behavior Color
red    = constB Red
blue   = constB Blue

lift0 :: a -> Behavior a
lift0 = constB
Notation

• We often have two versions of a function:

```
xxx :: Behavior a -> (a -> b) -> T b
xxx_ :: Behavior a -> b -> T b
```

• And two versions of some operators:

```
(=>>) :: Event a -> (a->b) -> Event b
(->>) :: Event a -> b -> Event b
```
Lifting ordinary functions

($*) :: \text{Behavior} \ (a \rightarrow b) \rightarrow \text{Behavior} \ a \rightarrow \text{Behavior} \ b

\text{Behavior} \ ff \$* \text{Behavior} \ fb

= \text{Behavior} \ (\lambda \text{uts} \rightarrow \text{zipWith} \ (\$) \ (ff \ \text{uts}) \ (fb \ \text{uts}))

\text{where} \ f \$ x = f x

([t1,t2,t3, \ldots],[f1,f2,f3, \ldots]) \longrightarrow

([t1,t2,t3, \ldots],[x1,x2,x3, \ldots]) \longrightarrow

([t1,t2,t3, \ldots],[f1 \ x1, f2 \ x2, f3 \ x3, \ldots])

lift1 :: \ (a \rightarrow b) \rightarrow \ (\text{Behavior} \ a \rightarrow \text{Behavior} \ b)
lift1 \ f \ b1 = lift0 \ f \$* \ b1

lift2 :: \ (a \rightarrow b \rightarrow c) \rightarrow

\ (\text{Behavior} \ a \rightarrow \text{Behavior} \ b \rightarrow \text{Behavior} \ c)
lift2 \ f \ b1 \ b2 = lift1 \ f \ b1 \$* \ b2
Button Presses

data G.Event
    = Key       { char :: Char, isDown :: Bool }
    | Button    { pt :: Vertex, isLeft, isDown :: Bool }
    | MouseMove { pt :: Vertex }

lbp :: Event ()
lbp = Event (\(uas,\) -> map getlbp uas)
    where getlbp (Just (Button _ _ True True)) = Just ()
          getlbp _ = Nothing

  ([Noting, Just (Button ...), Nothing, Just(Button ...), ...],
   [t1,t2,t3, ...]) -->

    [Nothing, Just(), Nothing, Just(), ...]

Color0 = red `switch` (lbp --> blue)
Key Strokes

key :: Event Char
key = Event (\(uas,_\) -> map getkey uas)
    where getkey (Just (Key ch True)) = Just ch
         getkey _ = Nothing

([leftbut, key ‘z’ True, clock-tick, key ‘a’ True ...],
 [t1, t2, t3, t4, ...])

--->

[Nothing, Just ‘z’, Nothing, Just ‘a’, ...]
Mouse Movement

\[
\begin{align*}
\text{mm} & \::= \text{Event Vertex} \\
\text{mm} &= \text{Event } (\lambda(uas,_) \rightarrow \text{map getmm uas}) \\
&\quad \text{where getmm } (\text{Just } (\text{MouseMove pt})) \\
&\quad \quad = \text{Just } (\text{gPtToPt pt}) \\
&\quad \quad \text{getmm } _\_ = \text{Nothing}
\end{align*}
\]

\[
([\text{Nothing}, \text{Just } (\text{MouseMove } \ldots), \text{Nothing}, \text{Just } (\text{MouseMove } \ldots), \ldots],
\]

\[
[t_1, t_2, t_3, \ldots) \rightarrow
\]

\[
[\text{Nothing}, \text{Just } (x_1, y_1), \text{Nothing}, \text{Just } (x_2, y_2), \ldots]
\]

\[
\text{mouse} :: (\text{Behavior Float}, \text{Behavior Float})
\]

\[
\text{mouse} = (\text{fstB } m, \text{sndB } m)
\]

\[
\text{where } m = (0,0) \ `\text{step}` \ mm
\]

\[
( (uas,ts) \rightarrow [x_1, x_2, \ldots],
\]

\[
(\text{uas,ts} \rightarrow [y_1, y_2, \ldots])
\]

4/7/2014
Behavior and Event Combinators

```haskell
switch :: Behavior a -> Event (Behavior a) -> Behavior a
Behavior fb `switch` Event fe =
    memoB
        (Behavior
            (\uts@(us,ts) -> loop us ts (fe uts) (fb uts)))
where loop (_:us) (_:ts) ~(e:es) (b:bs) =
    b : case e of
        Nothing -> loop us ts es bs
        Just (Behavior fb')
            -> loop us ts es (fb' (us,ts))
```

\([\text{Noting}, \text{Just (Beh \ [x, y, \ldots] \ldots)}, \text{Nothing}, \text{Just(Beh \ [m, n, \ldots]) \ldots}], \[t_1, t_2, t_3, \ldots]\]) \rightarrow \[\text{fb}_1, \text{fb}_2, x, y, m, n \ldots\]
Event Transformer (map?)

\[(=>>) :: Event\ a \to (a\to b) \to Event\ b\]

\[Event\ fe \Rightarrow f = Event\ (\text{\}uts \to \text{map aux (fe uts)})\]
  \[\text{where aux (Just a) = Just (f a)}\]
  \[aux\ Nothing = Nothing\]

\[(-=>>) :: Event\ a \to b \to Event\ b\]

\[e \Rightarrow> v = e \Rightarrow> \_ \to v\]

\[[\text{Noting, Just (Ev x), Nothing, Just(Ev y), …} \to f \to\]
  \[\text{[Nothing, Just(f x), Nothing, Just(f y), …]}\]
**withElem**

withElem :: Event a -> [b] -> Event (a,b)

withElem (Event fe) bs
= Event (\uts -> loop (fe uts) bs)
where loop (Just a : evs) (b:bs)
= Just (a,b) : loop evs bs
loop (Nothing : evs) bs
= Nothing : loop evs bs

withElem_ :: Event a -> [b] -> Event b
withElem_ e bs = e `withElem` bs =>> snd

([Noting, Just x, Nothing, Just y, ...]) ---> [b0,b1,b2,b3, ...] -->
[Nothing, Just(x,b0), Nothing, Just(y,b1), ...]
Either one event or another

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

Event fe1 .|. Event fe2
    = Event (\uts -> zipWith aux (fe1 uts) (fe2 uts))
    where aux Nothing Nothing   = Nothing
            aux (Just x) _        = Just x
            aux _        (Just x) = Just x

([Noting, Just x, Nothing, Just y, ...]) --->
[Nothing, Just a, Just b, Nothing, ...] --->
[Nothing, Just x, Just b, Just y, ...]
Snapshot

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

\[
\text{Event } \text{fe } `\text{snapshot`} \text{ Behavior } \text{fb} = \text{Event } (\\text{uts} \to \text{zipWith aux (fe uts) (fb uts)})
\]

where aux (Just x) y = Just (x,y)

aux Nothing _ = Nothing

\[
\text{snapshot}_\_ :: \text{Event a} \to \text{Behavior b} \to \text{Event b}
\]

\[
\text{snapshot}_\_ \text{ e b } = \text{e `snapshot` b =>> snd}
\]

\[
[\text{Nothing, Just x, Nothing, Just y, } \ldots] \rightarrow\\
[b_1, b_2, b_3, b_4, \ldots] \rightarrow\\
[\text{Nothing, Just(x,b_2), Nothing, Just(y,b_4), } \ldots]
\]
step and stepAccum

step :: a -> Event a -> Behavior a
a `step` e = constB a `switch` e =>> constB

stepAccum :: a -> Event (a->a) -> Behavior a
a `stepAccum` e = b
where b = a `step`
    (e `snapshot` b =>> uncurry ($))

X1 -> [Nothing, Just x2, Nothing, Just x3, ...] --->
     [x1,   x1,   x2,   x2,   x3, ...]

X1 -> [Noting, Just f, Nothing, Just g, ...] --->
     [x1,   x1,   f x1, (f x1), g(f x1), ...]
predicate

\[
predicate :: \text{Behavior Bool} \rightarrow \text{Event ()}
\]

\[
predicate \ (\text{Behavior } fb) = \text{Event } (\lambda \text{uts} \rightarrow \text{map aux (fb uts)})
\]

where aux True = Just ()

aux False = Nothing

[True, True, False, True, False, ...] --->

[Just(), Just(), Nothing, Just(), Nothing, ...]
integral

integral :: Behavior Float -> Behavior Float
integral (Behavior fb)
  = Behavior (\uts@(us,t:ts) ->
    0 : loop t 0 ts (fb uts))
where loop t0 acc (t1:ts) (a:as)
  = let acc' = acc + (t1-t0)*a
    in acc' : loop t1 acc' ts as

([ua0,ua1,ua2,ua3, ...],[t0,t1,t2,t3, ...]) --->
  [0, Area t0-t1, Area t0-t2, Area t0-t3, ...]

F x

time axis

Integral F z

z
Putting it all together

reactimate :: String -> Behavior Graphic -> IO ()
reactimate title franProg
    = runGraphics $
        do w <- openWindowEx title (Just (0,0)) (Just (xWin,yWin))
           drawBufferedGraphic
               (us,ts,addEvents) <- windowUser w
           addEvents
           let drawPic (Just g) =
               do setGraphic w g
               quit <- addEvents
               if quit
                   then return True
                   else return False
           drawPic Nothing = return False
           let Event fe = sample `snapshot_` franProg
           run drawPic (fe (us,ts))
           closeWindow w
where

run f (x:xs) = do
  quit <- f x
  if quit
    then return ()
    else run f xs
run f [] = return ()
The Channel Abstraction

\[(\mathit{us}, \mathit{ts}, \mathit{addEvents}) \Leftarrow \mathit{windowUser}\]

- \(\mathit{us}\), and \(\mathit{ts}\) are infinite streams made with channels.
- A Channel is a special kind of abstraction, in the multiprocessing paradigm.
- If you “pull” on the tail of a channel, and it is null, then you “wait” until something becomes available.
- \(\mathit{addEvents} :: \mathit{IO} ()\) is a action which adds the latest user actions, thus extending the streams \(\mathit{us}\) and \(\mathit{ts}\)
Making a Stream from a Channel

makeStream :: IO ([a], a -> IO ())
makeStream = do
  ch <- newChan
  contents <- getChanContents ch
  return (contents, writeChan ch)
A Reactive window

windowUser :: Window -> IO ([Maybe UserAction], [Time], IO Bool)
windowUser w
    = do (evs, addEv) <- makeStream
        t0 <- timeGetTime
        let addEvents =
            let loop rt = do
                mev <- maybeGetWindowEvent w
                case mev of
                    Nothing -> return False
                    Just e  -> case e of
                        Key ' ' True -> return True
                        Closed -> return True
                        _ -> addEv (rt, Just e) >> loop rt
                in do t <- timeGetTime
                let rt = w32ToTime (t-t0)
                quit <- loop rt
                addEv (rt, Nothing)
                return quit
            in return (map snd evs, map fst evs, addEvents)
The “Paddle Ball” Game

paddleball vel = walls `over` paddle `over` ball vel

walls = let upper = paint blue
            (translate (0,1.7) (rec 4.4 0.05))
       left  = paint blue
            (translate (-2.2,0) (rec 0.05 3.4))
       right = paint blue
            (translate (2.2,0) (rec 0.05 3.4))
in upper `over` left `over` right

paddle = paint red
            (translate (fst mouse, -1.7) (rec 0.5 0.05))

x `between` (a,b) = x >* a &&* x <* b
The “reactive” ball

```
pball vel =
  let xvel    = vel `stepAccum` xbounce ->> negate
  xpos    = integral xvel
  xbounce = when (xpos >* 2 ||* xpos <* -2)
  yvel    = vel `stepAccum` ybounce ->> negate
  ypos    = integral yvel
  ybounce = when (ypos >* 1.5
                  ||* ypos       `between` (-2.0,-1.5) &&*
                  fst mouse  `between`
                  (xpos-0.25,xpos+0.25))
➤ in paint yellow (translate (xpos, ypos) (ell 0.2 0.2))

main = test (paddleball 1)
```