Advanced Programming Handout 10

A Module of Simple Animations (SOE Chapter 13)

Motivation

- In the abstract, an *animation* is a continuous, time-varying image.
- In practice, it is a sequence of static images displayed in succession so rapidly that it looks continuous.
- Our goal is to present to the programmer an abstract view of animations that hides the practical details.
- In addition, we will generalize animations to be continuous, time-varying quantities of any value, not just images.

Representing Animations

- As usual, we will use our most powerful tool. functions, to represent animations: type Animation a = Time -> a type Time = Float
- Examples:
 - rubberBall :: Animation Shape rubberBall t = Ellipse (sin t) (cos t)
 - revolvingBall :: Animation Region revolvingBall t = let ball = Shape (Ellipse 0.2 0.2) in Translate (sin t, cos t) ball
 - - tellTime :: Animation String
 tellTime t = "The time is: " ++ show t

An Animator

Given a function...

animate :: String -> Animation Graphic -> IO ()

...we could then execute (display) the previous animations like this:

main1 :: IO ()
main1 = animate "Animated Shape"
(withColor Blue
rubberBall) . shapeToGraphic

Definition of "animate"

animate :: String -> Animation Graphic -> IO ()

animate title anim = runGraphics \$
 do w <- openWindowEx title (Just (0,0)) (Just
 (xWin,yWin))</pre>

drawBufferedGraphic (Just 30) BufferedGraphic (Just 30) t0 <- timeGetTime let loop = do t<- timeGetTime let ft = intToFloat (word32ToInt (t-t0)) / 1000 setGraphic w (ami ft) gtbHindowTick w loop loop

loop

See text for details...

Common Operations

 We can define many operations on animations based on the underlying type. For example, for Pictures: emptyA :: Animation Picture

emptyA t = EmptyPic

- overA :: Animation Picture
- -> Animation Picture -> Animation Picture overA al a2 t = a1 t `Over` a2 t

overManyA :: [Animation Picture] -> Animation Picture overManyA = foldr overA emptyA

- We can do a similar thing for Shapes, etc.
- Also, for numeric animations, we could define functions like addA, multA, and so on. Type Classes!
- But there is a better way...

Behaviors

- Basic definition (replacing Animation): newtype Behavior a = Beh (Time -> a)
- Recall that newtype creates a single-argument datatype with (time and space) efficiency the same as a simple type declaration.
 (So then what is the difference??)

Behaviors

We need to use newtype here because type synonyms are not allowed in type class instance declarations -- only types declared with data or newtype.

Constant Behaviors

 Given a scalar value x, we can lift it to a constant behavior that, at all times t, yields x:

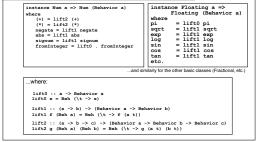
> lift0 :: a -> Behavior a lift0 x = Beh (t -> x)

Dependent Behaviors

 Given a function f, we can lift it to a function on behaviors that, at a given time t, samples its argument and passes the result through f:

lift1 :: (a -> b) -> (Behavior a -> Behavior b) lift1 f (Beh a) = Beh (t -> f (a t))

Numeric Behaviors



<section-header><section-header><code-block><code-block><code-block><code-block></code></code></code></code>

New Type Classes

 Besides lifting existing type classes such as Num to behaviors, we can define new classes for manipulating behaviors. For example:

> class Combine a where empty :: a over :: a -> a -> a

instance Combine Picture where empty = EmptyPic over = Over

instance Combine a => Combine (Behavior a) where empty = lift0 empty over = lift2 over

overMany :: Combine a => [a] -> a
overMany = foldr over empty

Hiding More Detail

• We have not yet hidden all the "practical" details of animation - in particular time itself.

- But through more aggressive lifting...

tintogin more aggressive mining...
reg = lift1 Shape
ell = lift2 Shipse
red = lift2 Slipse
red = lift0 Ked
yellow = lift0 Yellow
Eranslate (Beh al, Beh a2) (Beh r) -- mis existsy here
= Beh (\t-> Translate (al t, a2 t) (r t))

...we can redefine our red revolving ball without referring to time at all:

revolvingBallB :: Behavior Picture revolvingBallB = let ball = shape (ell 0.2 0.2) in reg red (translate (sin time, cos time) ball)

More Liftings

Comparison operators:

(>*) :: Ord a => Behavior a -> Behavior a -> Behavior Bool (>*) = lift2 (>)

Conditional behaviors:

cond :: Behavior Bool -> Behavior a -> Behavior a -> Behavior a cond = lift3 (\p c a \rightarrow if p then c else a)

For example, a flashing color: flash :: Behavior Color
flash = cond (sin time >* 0) red yellow

Time Travel

- A function for translating a behavior through time: timeTrans :: Behavior Time -> Behavior a -> Behavior a timeTrans (Beh f) (Beh a) = Beh $(a \cdot f)$
- For example: timeTrans (Petime) anim (timeTrans (5+time) anim) 'over' anim timeTrans (negate time) anim --- double speed --- double speed --- one anim 5 sec behind another --- go backwards
- Any kind of behavior can be time transformed:

flashingBall :: Behavior Picture
flashingBall =
let ball = shape (ell 0.2 0.2)
in reg (timeTrans (8*time) flash)
(translate (sin time, cos time) ball)

Final Example

revolvingBalls :: Behavior Picture

revolvingBalls = overMany [timeTrans (time + t*pi/4) flashingBall | t <- map lift0 [0..7]]

See SOE for a more substantial example: a kaleidoscope program. (The details of its construction can be skimmed, but you may enjoy running it...)