CS 457/557 Functional Programming

Lecture 13

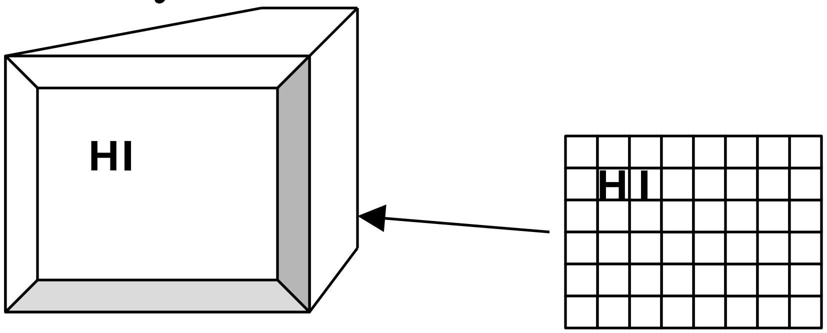
Animations

Animations

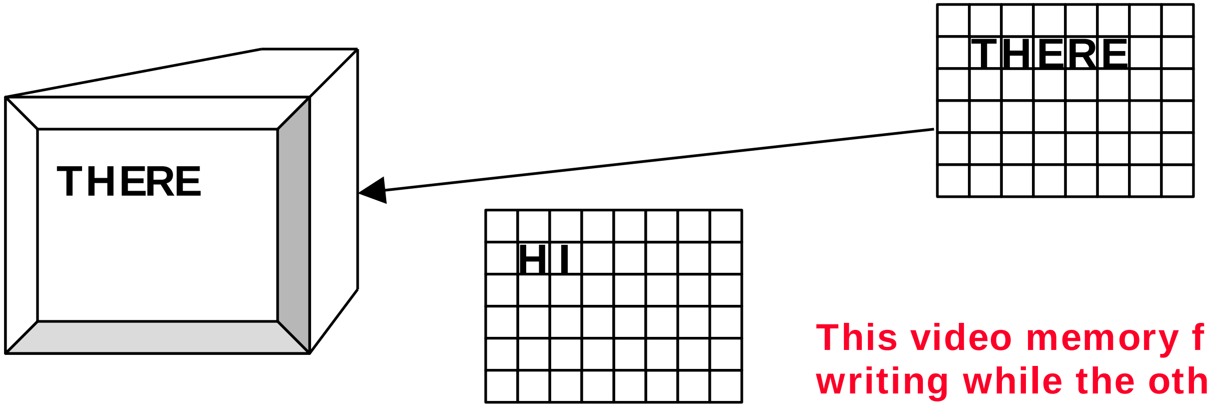
- An animation is a “moving” graphic.
  - Sometimes we say a **time-dependent** graphic, since where it “moves” to is dependent upon time.
- To create the illusion of “movement” we need draw frames with a different picture each frame.
  - A frame rate of about 30 frames a second is optimal
  - less than 15-20 appears to flicker
  - greater than 30 gives no apparent improvement
- To draw a frame we need to erase the old frame before drawing the new frame.
- All our drawings have been accumulative (we never erase anything, just draw “over” what’s already there).
- There exist several strategies for frame drawing.

Buffered graphics

- Display devices display the information stored in the video memory.



- Buffered graphics use two sets of memory, instantaneously switching from one memory to the other, so quickly that the flicker effect is unobservable.



This video memory free for writing while the other is displayed

Haskell interface to buffered graphics

Usual tick rate = 30 times per second

- `getWindowTick :: Window -> IO()`
  - Every window has an internal timer. getWindowTick “waits” for the next “tick” (since the last call to getWindowTick) before it returns. If the next “tick” has already occurred it returns immediately.
- `getTime :: IO Integer`
  - Returns the current time, measured in milliseconds, counting from some arbitrary point. By itself, means nothing, but the **difference** between successive calls accurately measures elapsed time.
- `setGraphic :: Window -> Graphic -> IO()`
  - Writes the graphic into the “free” video graphic buffer. At the next frame “tick” what’s in the “free” video buffer will be drawn, and the current buffer will become the free buffer.

Interface to the richer window interface

Old interface:

```
openWindow :: String -> Point -> IO Window
```

e.g. `openWindow “title” (width,height)`

Richer interface:

```
openWindowEx :: String -> Maybe Point ->
 Maybe Point -> (Graphic -> DrawFun) ->
  Maybe Word32 -> IO Window
```

```
openWindowEx “title”
             (Just(x,y))    -- upper left corner
             (Just(width,height))
             drawBufferedGraphic  -- drawing mode
             (Just 30)      -- refresh rate
```

Animations in Haskell

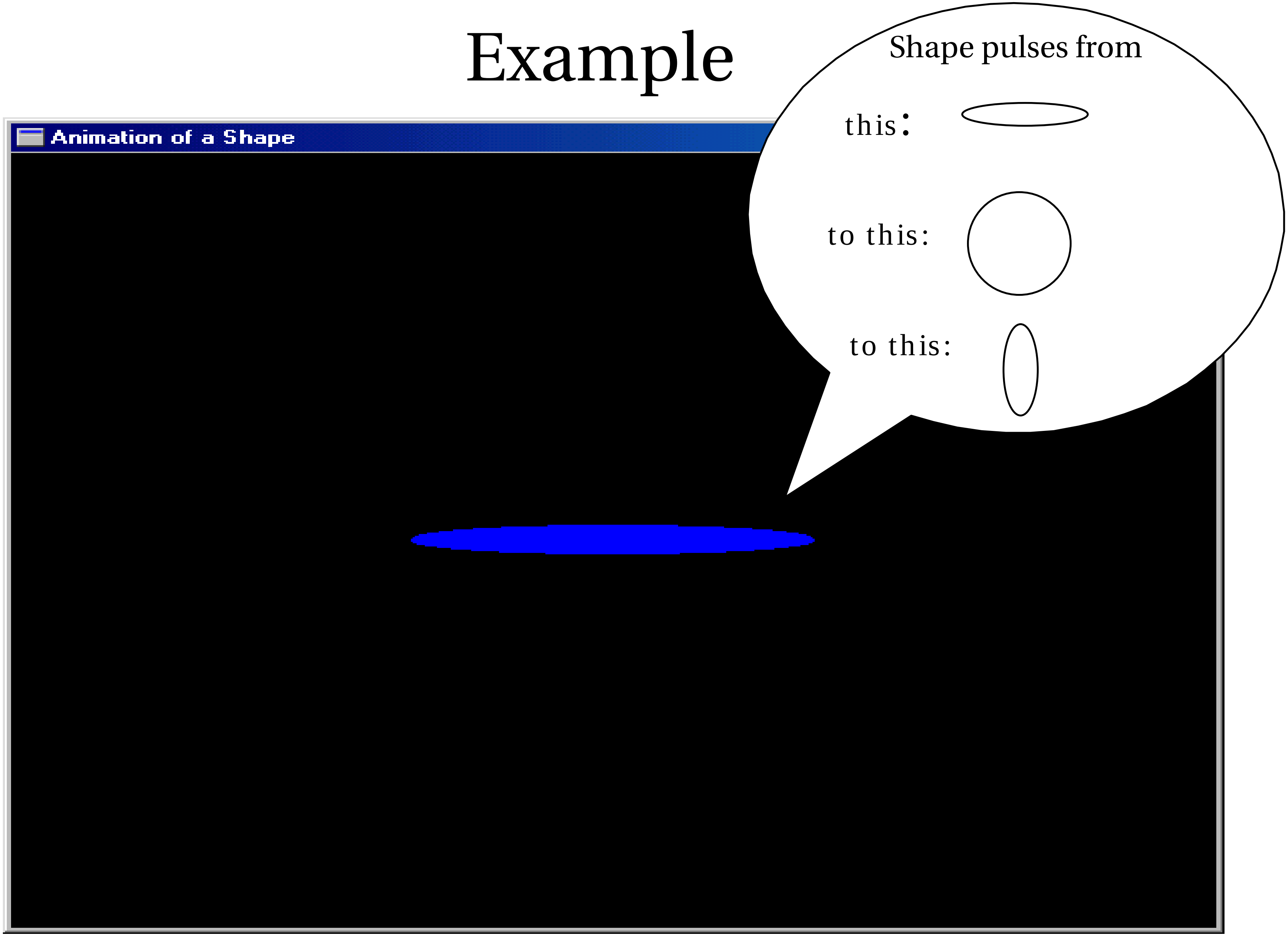
```
type Animation a = Time -> a
type Time = Float

blueRubberBall :: Animation Graphic
blueRubberBall t = withColorBlue (
                    shapeToGraphic (
                     Ellipse (sin t) (cos t)))

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

main1 = animate
 "Animation of a Shape" blueRubberBall
```

Example



Shape pulses from

this:

to this:

to this:

The `animate` function

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

animate title anim  = runGraphics (
  do w <- openWindowEx title (Just (0,0)) (Just(xWin,yWin))
            drawBufferedGraphic (Just 30)
     t0 <- getTime
     let loop =
           do t <- getTime
              let ft = fromInteger (t-t0) / 1000
              setGraphic w (anim ft)
              getWindowTick w
              loop
     loop)
```

Alternative Definition

- We made `animation` a polymorphic type constructor so that we could describe time-varying behaviors of types other than `Graphic`.
- Could rewrite example like this:

```
rubberBall :: Animation Shape
rubberBall t = Ellipse (sin t) (cos t)

main1 :: IO ()
main1 = animate “Animation of a Shape”
           (withColor Blue .
            shapeToGraphic .
            rubberBall)
```

  - » Note convenience of composition here.

Complex Animations

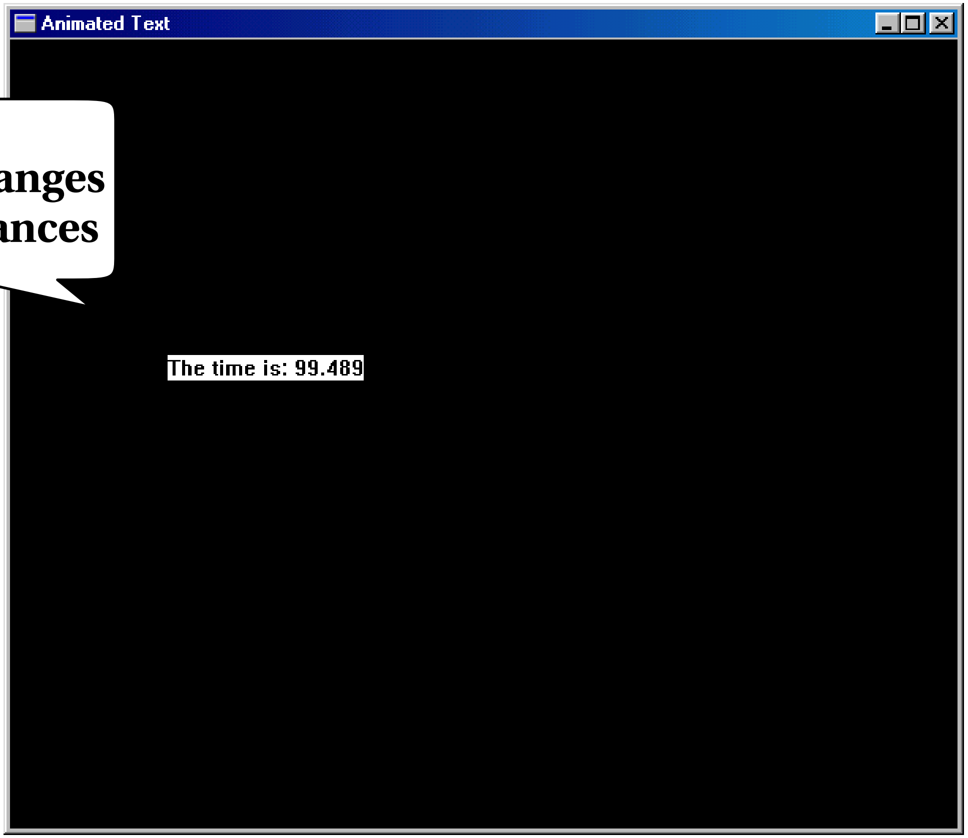
```
revolvingBall :: Animation Region
revolvingBall t
  = let ball = Shape (Ellipse 0.2 0.2)
    in Translate (sin t, cos t) ball

planets :: Animation Picture
planets t
  = let p1 = Region Red (Shape (rubberBall t))
        p2 = Region Yellow (revolvingBall t)
    in p1 `Over` p2

tellTime :: Animation String
tellTime t = "The time is: " ++ show t
```

Telling Time

```
main2 = animate "Animated Text”
                  (text (100,200) . tellTime)
```



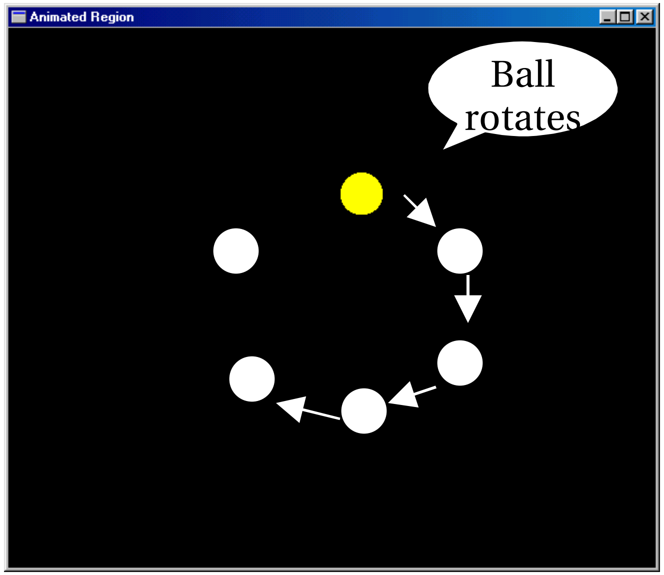
The time is: 99.489

The time changes as time advances

Revolving Circle

```
regionToGraphic :: Region -> Graphic
regionToGraphic = drawRegion . regionToGRegion


main3 = animate "Animated Region"
     (withColor Yellow . regionToGraphic .
      revolvingBall)
```



Animating Pictures

```
picToGraphic :: Picture -> Graphic
picToGraphic (Region c r)
  = withColor c (regionToGraphic r)
picToGraphic (p1 `Over` p2)
  = picToGraphic p1 `overGraphic` picToGraphic p2
picToGraphic EmptyPic = emptyGraphic
```

Case analysis over structure of picture.

Use the primitives `overGraphic` & emptyGraphic

```
main4 = animate "Animated Picture"
       (picToGraphic . planets)
```

Lifting primitives to animations

- It's useful to define “time varying” primitives, e.g.

```
type Anim = Animation Picture
```

- First an `Anim` which doesn’t really vary

```
emptyA :: Anim
emptyA t = EmptyPic
```

Recall
Anim =
Animation Picture =
Time -> Picture
hence the time
parameter t

- Combining time varying pictures

```
overA :: Anim -> Anim -> Anim
overA a1 a2 t = a1 t `Over` a2 t

overManyA :: [Anim] -> Anim
overManyA = foldr overA emptyA
```

Time Translation

```
timeTransA :: (Time -> Time) ->
                Animation a -> Animation a
```

or

```
timeTransA :: Animation Time ->
                Animation a -> Animation a

timeTransA f a t = a (f t)
```

or

```
timeTransA f a = a . f

timeTransA (2*) anim    -- runs twice as fast
timeTransA (5+) anim    -- runs 5 seconds ahead
```

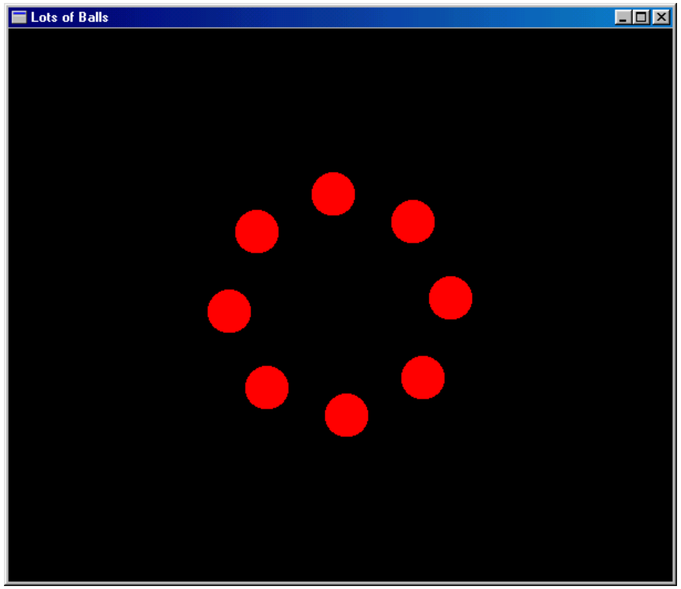
Example

```
rBall :: Anim
rBall t = let ball = Shape (Ellipse 0.2 0.2)
          in Region Red (Translate (sin t, cos t) ball)

rBalls :: Anim
rBalls = overManyA
           [ timeTransA ((t*pi/4)+) rBall | t <- [0..7]]

main5 = animate "Lots of Balls"
         (picToGraphic . rBalls)
```

Each ball rotates
pi/4 seconds behind
the one in front of it


Lots of Balls