Advanced Programming Handout 8

Drawing Regions
(SOE Chapter 10)

Pictures

- Drawing Pictures
  - Pictures are composed of Regions (which are composed of shapes)
  - Pictures add <u>color</u> and <u>layering</u>

```
data Picture = Region Color Region
             | Picture `Over` Picture
             | EmptyPic
     deriving Show
```

Digression on Importing

- We need to use SOEGraphics for drawing things on the screen, but SOEGraphics has its own Region datatype, leading to a name clash when we try to import both SOEGraphics and our Region module.
- We can work around this as follows:

```
import SOEGraphics hiding (Region)
import qualified SOEGraphics as G (Region)
```

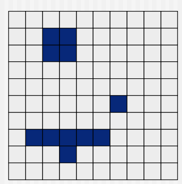
- The effect of these declarations is that all the names from SOEGraphics *except* Region can be used in unqualified form, and we can say G.Region to refer to the one from SOEGraphics.

Recall the Region Datatype

```
data Region =
   Shape Shape                  -- primitive shape
 | Translate Vector Region      -- translated region
 | Scale     Vector Region      -- scaled region
 | Complement Region            -- inverse of a region
 | Region `Union` Region        -- union of regions
 | Region `Intersect` Region -- intersection of regions
 | Empty
```

- How do we draw things like the intersection of two regions, or the complement of a region? These are hard to do efficiently. Fortunately, the `G.Region` interface uses lower-level support to do this for us.

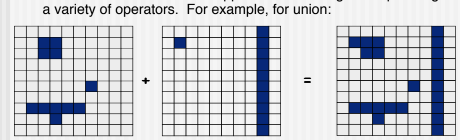
G.Region



- The `G.Region` datatype interfaces more directly to the underlying hardware. It is essentially a two-dimensional array or "bit-map", storing a binary value for each pixel in the window.

Efficient Bit-Map Operations

- There is efficient low-level support for combining bit-maps using a variety of operators. For example, for union:



- Making these operations fast requires detailed control over data layout in memory -- a job for a lower-level language. This part of the SOEGraphics module is therefore just a "wrapper" for an external library (probably written in C).

G.Region Interface

Why IO here?

```
createRectangle :: Point -> Point -> IO G.Region
createEllipse   :: Point -> Point -> IO G.Region
createPolygon   :: [Point] -> IO G.Region

andRegion       :: G.Region -> G.Region -> IO G.Region
orRegion        :: G.Region -> G.Region -> IO G.Region
xorRegion       :: G.Region -> G.Region -> IO G.Region
diffRegion      :: G.Region -> G.Region -> IO G.Region
deleteRegion    :: G.Region -> IO ()

drawRegion      :: G.Region -> Graphic
```

These functions are defined in the SOEGraphics library module.

Drawing G.Region

- To render things involving intersections and unions quickly, we perform these calculations in a `G.Region`, then turn the `G.Region` into a graphic object, and then use the machinery we have seen in earlier chapters to display the object.

```
drawRegionInWindow ::
      Window -> Color -> Region -> IO ()

 drawRegionInWindow w c r =
   drawInWindow w
     (withColor c (drawRegion (regionToGRegion r)))
```

- To finish this off, we still need to define `regionToGRegion`.
- But first let's complete the big picture by writing the (straightforward) function that uses `drawRegionInWindow` to draw Pictures.

Drawing Pictures

- Pictures combine multiple regions into one big picture. They provide a mechanism for placing one sub-picture on top of another.

```
drawPic :: Window -> Picture -> IO ()

drawPic w (Region c r)    = drawRegionInWindow w c r
drawPic w (p1 `Over` p2)  = do drawPic w p2
                               drawPic w p1
drawPic w EmptyPic        = return ()
```

- Note that `p2` is drawn before `p1`, since we want `p1` to appear "over" `p2`.

Now back to the code for rendering Regions as G.Regions...

Turning a Region into a G.Region

Let's first experiment with a simplified variant of the problem to illustrate an efficiency issue...

```
data NewRegion = Rect Side Side   <- instead of G.Region

regToNReg :: Region -> NewRegion
regToNReg (Shape (Rectangle sx sy))
      = Rect sx sy
regToNReg (Scale (x,y) r)
      = regToNReg (scaleReg (x,y) r)
  where scaleReg (x,y) (Shape (Rectangle sx sy))
           = Shape (Rectangle (x*sx) (y*sy))
        scaleReg (x,y) (Scale s r)
           = Scale s (scaleReg (x,y) r)
```

omitting cases for other Region constructors

A Problem

- Consider

```
(Scale (x1,y1)
       (Scale (x2,y2)
              (Scale (x3,y3)
                     ... (Shape (Rectangle sx sy))
                      ... )))
```

- If the scaling is *n* levels deep, how many traversals does `regToNReg` perform over the `Region` tree?

We've Seen This Before

- We have encountered this problem before in a different setting. Recall the naive definition of `reverse`:

```
reverse []     = []
reverse (x:xs) = (reverse xs) ++ [x]
  where []     ++ zs = zs
        (y:ys) ++ zs = y : (ys ++ zs)
```

- How did we solve this? We used an extra accumulating parameter:

```
reverse xs = loop xs []
  where loop [] zs     = zs
        loop (x:xs) zs = loop xs (x:zs)
```

- We can do the same thing for `Regions`.

N.b.: A good compiler (like GHC) really will implement this function call as a jump!

Accumulating the Scaling Factor

```
regToNReg2 :: Region -> NewRegion
regToNReg2 r = rToNR (1,1) r
  where rToNR :: (Float,Float) -> Region -> NewRegion
        rToNR (x1,y1) (Shape (Rectangle sx sy))
               = Rect (x1*sx) (y1*sy)
        rToNR (x1,y1) (Scale (x2,y2) r)
               = rToNR (x1*x2,y1*y2) r
```

- To solve our original problem, repeat this for all the constructors of `Region` (not just `Shape` and `Scale`) and use `G.Region` instead of `NewRegion`. We also need to handle translation as well as scaling.

Final Version

accumulated scaling

accumulated translation

```
regToGReg :: Vector -> Vector -> Region -> G.Region
regToGReg loc sca (Shape s)
  = shapeToGRegion loc sca s
regToGReg loc (sx,sy) (Scale (u,v) r)
  = regToGReg loc (sx*u, sy*v) r
regToGReg (lx,ly) (sx,sy) (Translate (u,v) r)
  = regToGReg (lx+u*sx, ly+v*sy) sca r
regToGReg loc sca Empty
  = createRectangle (0,0) (0,0)
regToGReg loc sca (r1 `Union` r2)
  = let gr1 = regToGReg loc sca r1
        gr2 = regToGReg loc sca r2
    in orRegion gr1 gr2
```

To finish, we need to write similar clauses for `Intersect`, `Complement` etc. and define

```
shapeToGRegion :: Vector -> Vector -> Shape -> G.Region
```

A Matter of Style

- While the function on the previous page does the job correctly, there are several stylistic issues that could make it more readable and understandable.
- For one thing, the style of defining a function by patterns becomes cluttered when there are many parameters (other than the one which has the patterns).
- For another, the pattern of explicitly allocating and deallocating (bit-map) `G.Region`'s will be repeated in cases for intersection and for complement, so we should abstract it, and give it a name.

Abstracting Out a Common Pattern

```
primGReg loc sca r1 r2 op
  = let gr1 = regToGReg loc sca r1
        gr2 = regToGReg loc sca r2
    in op gr1 gr2
```

Definition by cases with a Case Expression

```
regToGReg :: Vector -> Vector -> Region -> G.Region
regToGReg (loc@(lx,ly)) (sca@(sx,sy)) shape =
 case shape of
  Shape s            -> shapeToGRegion loc sca s
  Translate (u,v) r -> regToGReg (lx+u*sx,ly+u*sy) sca r
  Scale (u,v) r      -> regToGReg loc (sx*u, sy*v) r
  Empty              -> createRectangle (0,0) (0,0)
  r1 `Union` r2      -> primGReg loc sca r1 r2 orRegion
  r1 `Intersect` r2 -> primGReg loc sca r1 r2 andRegion
  Complement r       -> primGReg loc sca winRect r diffRegion

regionToGRegion :: Region -> G.Region
regionToGRegion r = regToGReg (0,0) (1,1) r
```

Pattern renaming

A Region representing the whole graphics window

Drawing Pictures

```
draw :: Picture -> IO ()
draw p = runGraphics (
         do w <- openWindow "Region Test" (xWin,yWin)
            drawPic w p
            spaceClose w
         )
```

A Better Definition

```
($) :: (a->b) -> a -> b
f ($) x = f x

draw :: Picture -> IO ()
draw p = runGraphics $
         do w <- openWindow "Region Test" (xWin,yWin)
            drawPic w p
            spaceClose w
```

In effect, we've introduced a second syntax for application, with lower precedence than the standard one

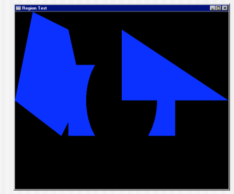
Some Sample Regions

```
r1 = Shape (Rectangle 3 2)
r2 = Shape (Ellipse 1 1.5)
r3 = Shape (RtTriangle 3 2)
r4 = Shape (Polygon [(-2.5,2.5), (-3.0,0),
                     (-1.7,-1.0),
                     (-1.1,0.2), (-1.5,2.0)] )
```

Sample Pictures

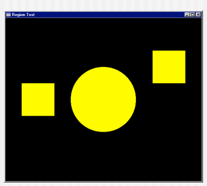
```
reg = r3 `Union`                 -- RtTriangle
       (r1 `Intersect`           -- Rectangle
       Complement r2 `Union`     -- Ellipse
       r4)                       -- Polygon

pic1 = Region Cyan reg
Main1 = draw pic1
```



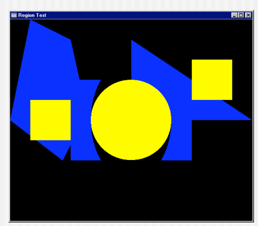
More Pictures

```
reg2 = let circle = Shape (Ellipse 0.5 0.5)
           square = Shape (Rectangle 1 1)
       in (Scale (2,2) circle)
          `Union` (Translate (2,1) square)
          `Union` (Translate (-2,0) square)
pic2 = Region Yellow reg2
main2 = draw pic2
```



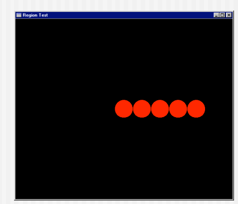
Another Picture

```
pic3 = pic2 `Over` pic1
main3 = draw pic3
```



Separating Computation From Action

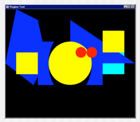
```
oneCircle   = Shape (Ellipse 1 1)
manyCircles = [ Translate (x,0) oneCircle | x <- [0,2..] ]
fiveCircles = foldr Union Empty (take 5 manyCircles)
pic4 = Region Magenta
        (Scale (0.25,0.25)
               fiveCircles)
main4 = draw pic4
```



Ordering Pictures

```
pictToList :: Picture -> [(Color,Region)]

pictToList  EmptyPic      = []
pictToList (Region c r)   = [(c,r)]
pictToList (p1 `Over` p2)
      = pictToList p1 ++ pictToList p2
```



Lists the Regions in a Picture from top to bottom.

(Note that this is possible because Picture is a datatype that can be analyzed. Would not work with, e.g., a characteristic function representation.)

A Suggestive Analogy

```
pictToList  EmptyPic      = []
pictToList (Region c r)   = [(c,r)]
pictToList (p1 `Over` p2) = pictToList p1 ++ pictToList p2

drawPic w (Region c r)   = drawRegionInWindow w c r
drawPic w (p1 `Over` p2) = do drawPic w p2
                              drawPic w p1
drawPic w EmptyPic       = return ()
```

We'll have (much) more to say about this later...

Pictures that React

- Goal: Find the topmost Region in a Picture that "covers" the position of the mouse when the left button is clicked.
- Implementation: Search the picture (represented as a list) for the first Region that contains the mouse position.
- Then (just for fun) re-arrange the list, bringing that one to the top.

```
adjust :: [(Color,Region)] -> Vertex ->
          (Maybe(Color,Region), [(Color,Region)])
             selected picture       reordered list

adjust []             p = (Nothing, [])
adjust ((c,r):regs) p =
   if r `containsR` p
      then (Just (c,r), regs)
      else let (hit, rs) = adjust regs p
           in  (hit, (c,r) : rs)
```

Doing it Non-recursively

From the Prelude:
```
break:: (a -> Bool) -> [a] -> ([a],[a])
```

For example:
```
break even [1,3,5,4,7,6,12] ➔ ([1,3,5],[4,7,6,12])
```

So:
```
adjust2 regs p
  = case (break (\(_,r) -> r `containsR` p) regs)
    of
       (top,hit:rest) -> (Just hit, top++rest)
       (_,[])         -> (Nothing, regs)
```

Putting it all Together

```
loop :: Window -> [(Color,Region)] -> IO ()
loop w regs =
 do clearWindow w
    sequence [ drawRegionInWindow w c r |
                      (c,r) <- reverse regs ]
    (x,y) <- getLBP w
    case (adjust regs (pixelToInch (x - xWin2),
                       pixelToInch (yWin2 - y) )) of
       (Nothing,  _       ) -> closeWindow w
       (Just hit, newRegs) -> loop w (hit : newRegs)

draw2 :: Picture -> IO ()
draw2 pic = runGraphics $
            do w <- openWindow "Picture demo" (xWin,yWin)
               loop w (pictToList pic)
```

Try it Out

```
p1,p2,p3,p4 :: Picture
p1 = Region Magenta r1
p2 = Region Cyan r2
p3 = Region Green r3
p4 = Region Yellow r4

pic :: Picture
pic = foldl Over EmptyPic [p1,p2,p3,p4]
main = draw2 pic
```

A Matter of Style, 3

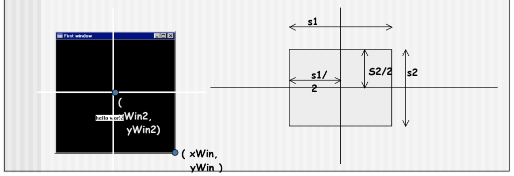
```
loop2 w regs
    = do clearWindow w
         sequence [ drawRegionInWindow w c r |
                    (c,r) <- reverse regs ]
         (x,y) <- getLBP w
         let aux (_,r) = r `containsR`
                         ( pixelToInch (x-xWin2),
                           pixelToInch (yWin2-y) )
         case (break aux regs) of
           (_,[])         -> closeWindow w
           (top,hit:bot) -> loop w (hit : (top++bot))

draw3 pic = runGraphics $
            do w <- openWindow "Picture demo" (xWin,yWin)
               loop2 w (pictToList pic)
```

Extra slides...

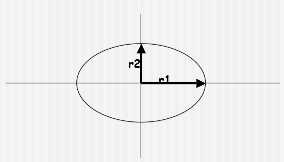
Implementing ShapeToGRegion

```
shapeToGRegion
  :: Vector -> Vector -> Shape -> IO G.Region
shapeToGRegion (lx,ly) (sx,sy) (Rectangle s1 s2)
  = createRectangle (trans(-s1/2,-s2/2)) (trans (s1/2,s2/2))
    where trans (x,y) = ( xWin2 + inchToPixel (lx+x*sx),
                          yWin2 - inchToPixel (ly+y*sy) )
```



The Ellipse Case

```
shapeToGRegion (lx,ly) (sx,sy) (Ellipse r1 r2)
  = createEllipse (trans (-r1,-r2)) (trans ( r1, r2))
    where trans (x,y) =
            ( xWin2 + inchToPixel (lx+x*sx),
              yWin2 - inchToPixel (ly+y*sy) )
```



Polygon and RtTriangle

```
shapeToGRegion (lx,ly) (sx,sy) (Polygon pts)
  = createPolygon (map trans pts)
    where trans (x,y) =
            ( xWin2 + inchToPixel (lx+x*sx),
              yWin2 - inchToPixel (ly+y*sy) )

shapeToGRegion (lx,ly) (sx,sy) (RtTriangle s1 s2)
     = createPolygon (map trans [(0,0),(s1,0),(0,s2)])
       where trans (x,y) =
               ( xWin2 + inchToPixel (lx+x*sx),
                 yWin2 - inchToPixel (ly+y*sy) )
```

A Matter of Style, 2

```
shapeToGRegion (lx,ly) (sx,sy) s = case s of
  Rectangle s1 s2  -> createRectangle (trans (-s1/2,-s2/2))
                                      (trans ( s1/2, s2/2))
  Ellipse r1 r2    -> createEllipse   (trans (-r1,-r2))
                                      (trans ( r1, r2))
  Polygon pts      -> createPolygon (map trans pts)
  RtTriangle s1 s2 -> createPolygon
                        (map trans [(0,0),(s1,0),(0,s2)])
   where trans (x,y) = ( xWin2 + inchToPixel (lx+x*sx),
                         yWin2 - inchToPixel (ly+y*sy) )
```

- shapeToGRegion has the same problems as regToGReg
  - The extra parameters obscure the pattern matching.
  - There is a repeated pattern; we should give it a name.