Advanced Programming
Handout 8

Drawing Regions
(SOE Chapter 10)
Pictures

- Drawing Pictures
  - Pictures are composed of Regions (which are composed of shapes)
  - Pictures add color and layering

```
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

```haskell
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.
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).
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.

\[
\text{drawRegionInWindow} ::
\text{Window} \to \text{Color} \to \text{Region} \to \text{IO} ()
\]

\[
\text{drawRegionInWindow} \ w \ c \ r =
\text{drawInWindow} \ w
\quad (\text{withColor} \ c \ (\text{drawRegion} \ (\text{regionToGRegion} \ r)))
\]

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.

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

```haskell
data NewRegion = Rect Side Side

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)
```

Instead of G.Region

Omitting cases for other Region constructors
A Problem

- Consider

\[(\text{Scale} \ (x_1, y_1) \ \\
\text{ } \ \\
\text{Scale} \ (x_2, y_2) \ \\
\text{ } \ \\
\text{Scale} \ (x_3, y_3) \ \\
\ldots \ \text{(Shape} \ (\text{Rectangle} \ sx \ sy))) \ \\
\ldots \ )])\]

- If the scaling is \( n \) levels deep, how many traversals does \text{regToNReg} perform over the \text{Region} tree?
We’ve Seen This Before

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

\[
\begin{align*}
\text{reverse } [] & = [] \\
\text{reverse } (x:xs) & = (\text{reverse } xs) ++ [x] \\
\text{where } [] & ++ zs = zs \\
(y:ys) & ++ zs = y : (ys ++ zs)
\end{align*}
\]

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

\[
\begin{align*}
\text{reverse } xs & = \text{loop } xs [] \\
\text{where } \text{loop } [] zs & = zs \\
\text{loop } (x:xs) zs & = \text{loop } xs (x:zs)
\end{align*}
\]

- 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.
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
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) $G.\text{Region}$’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

\[
\text{primGReg loc sca r1 r2 op} = \text{let gr1 = regToGReg loc sca r1} \\
\hspace{1em} gr2 = \text{regToGReg loc sca r2} \\
\hspace{2em} \text{in op gr1 gr2}
\]
Definition by cases with a Case Expression

\[
\text{regToGReg} :: \text{Vector} \rightarrow \text{Vector} \rightarrow \text{Region} \rightarrow \text{G.Region}
\]

\[
\text{regToGReg} \ (\text{loc}@(\text{lx},\text{ly})) \ (\text{sca}@\text{(sx,sy)}) \ \text{shape} = \begin{cases} 
\text{shapeToGRegion loc sca s} \\
\text{regToGReg} \ (\text{lx}+u*\text{sx},\text{ly}+u*\text{sy}) \ \text{sca} \ r \\
\text{regToGReg} \ \text{loc} \ (\text{sx}u, \text{sy}v) \ r \\
\text{createRectangle} \ (0,0) \ (0,0) \\
\text{primGReg} \ \text{loc} \ \text{sca} \ r1 \ r2 \ \text{orRegion} \\
\text{primGReg} \ \text{loc} \ \text{sca} \ r1 \ r2 \ \text{andRegion} \\
\text{primGReg} \ \text{loc} \ \text{sca} \ \text{winRect} \ r \ \text{diffRegion} 
\end{cases}
\]

\[
\text{regionToGRegion} :: \text{Region} \rightarrow \text{G.Region}
\]

\[
\text{regionToGRegion} \ r = \text{regToGReg} \ (0,0) \ (1,1) \ r
\]
Drawing Pictures

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

\[(\$) :: (a\rightarrow b) \rightarrow a \rightarrow b\]
\[f (\$) x = f x\]

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

\[
\text{reg} = r3 \ `\text{Union}` \\
(\ r1 \ `\text{Intersect}` \ \\
\ \text{Complement} \ r2 \ `\text{Union}` \ \\
r4) \ \\
\text{-- RtTriangle} \ \\
\text{-- Rectangle} \ \\
\text{-- Ellipse} \ \\
\text{-- Polygon}
\]

\[
\text{pic1} = \text{Region Cyan} \ \text{reg} \\
\text{Main1} = \text{draw pic1}
\]
More Pictures

```haskell
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

\[ \text{pic3} = \text{pic2 \ `Over` \ pic1} \]
\[ \text{main3} = \text{draw \ pic3} \]
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

\[
pictToList :: \text{Picture} \rightarrow [(\text{Color},\text{Region})]
\]

\[
pictToList \ \text{EmptyPic} \quad = \quad []
\]

\[
pictToList \ (\text{Region} \ c \ r) \quad = \quad [(c,r)]
\]

\[
pictToList \ (\text{p1 `Over` p2})
\quad = \quad \text{pictToList p1} ++ \text{pictToList p2}
\]

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

\[
\begin{align*}
pictToList\;\text{EmptyPic} &= [] \\
pictToList\;\text{(Region c r)} &= [(c,r)] \\
pictToList\;\text{(p1 `Over` p2)} &= \text{pictToList p1 ++ pictToList p2} \\
drawPic\;w\;\text{(Region c r)} &= \text{drawRegionInWindow w c r} \\
drawPic\;w\;\text{(p1 `Over` p2)} &= \text{do drawPic w p2} \\
& \quad \quad \quad \quad \text{drawPic w p1} \\
drawPic\;w\;\text{EmptyPic} &= \text{return ()}
\end{align*}
\]

We’ll have (much) more to say about this later...
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)])
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:
\[
\text{break}:: (a -> \text{Bool}) -> [a] -> ([a],[a])
\]

For example:
\[
\text{break even} \ [1,3,5,4,7,6,12] \Rightarrow ([1,3,5],[4,7,6,12])
\]

So:
\[
\text{adjust2} \ \text{regs} \ p = \text{case} \ (\text{break} \ \langle\_,r\rangle \rightarrow r \ `\text{containsR}` \ p) \ \text{regs}) \ \\
of \langle\text{top, hit: rest}\rangle \rightarrow (\text{Just} \ \text{hit, top++rest}) \ \\
\langle\_,[\]\rangle \rightarrow (\text{Nothing, } \text{regs})
\]
Putting it all Together

\[
\text{loop :: Window} \to [(\text{Color, Region})] \to \text{IO ()}
\]

\[
\text{loop w regs =}
\]
\[
do \text{clearWindow w}
\]
\[
\text{sequence [ } \text{drawRegionInWindow w c r } |\\n\quad (c,r) \leftarrow \text{reverse regs } ]
\]
\[
(x,y) \leftarrow \text{getLBP w}
\]
\[
\text{case (adjust regs (pixelToInch (x - xWin2),}
\]
\[
\quad \text{pixelToInch (yWin2 - y) )) of}
\]
\[
(\text{Nothing, _ } ) \rightarrow \text{closeWindow w}
\]
\[
(\text{Just hit, newRegs) } \rightarrow \text{loop w (hit : newRegs)}
\]

\[
\text{draw2 :: Picture} \to \text{IO ()}
\]

\[
\text{draw2 pic = runGraphics }$
\]
\[
\text{do w } \leftarrow \text{openWindow "Picture demo" (xWin,yWin)}
\]
\[
\text{loop w (pictToList pic)}
\]
A Matter of Style, Redux

loop2 w regs
  = do clearWindow w
    sequence [ drawRegionInWindow w c r | (c,r) <- reverse regs ]
    (x,y) <- getLBP w
    let (px,py) = (pixelToInch (x-xWin2),
                  pixelToInch (yWin2-y))
    let testHit (_,r) = r `containsR` (px,py)
    case (break testHit 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)
Try it Out

\[
p1, p2, p3, p4 :: \text{Picture} \\
p1 = \text{Region Magenta } r1 \\
p2 = \text{Region Cyan } r2 \\
p3 = \text{Region Green } r3 \\
p4 = \text{Region Yellow } r4 \\
\]

\[
pic :: \text{Picture} \\
pic = \text{foldl Over } \text{EmptyPic } [p1, p2, p3, p4] \\
main = \text{draw3 pic} 
\]
Extra slides...
Implementing ShapeToGRegion

\[
\text{shapeToGRegion} :: \text{Vector} \rightarrow \text{Vector} \rightarrow \text{Shape} \rightarrow \text{IO G.Region}
\]

\[
\text{shapeToGRegion} \ (lx, ly) \ (sx, sy) \ (\text{Rectangle} \ s1 \ s2)
\]

\[
= \text{createRectangle} \ \left( \text{trans} (-s1/2, -s2/2) \right) \ \left( \text{trans} \ (s1/2, s2/2) \right)
\]

where \(\text{trans} \ (x, y) = \left( x \ \text{Win2} + \ \text{inchToPixel} \ (lx + x \times sx), \ y \ \text{Win2} - \ \text{inchToPixel} \ (ly + y \times sy) \right)\)
The Ellipse Case

\[
\text{shapeToGRegion} \ (lx, ly) \ (sx, sy) \ (\text{Ellipse} \ r1 \ r2) \\
= \text{createEllipse} \ (\text{trans} \ (-r1, -r2)) \ (\text{trans} \ (r1, r2)) \\
\text{where} \ \text{trans} \ (x, y) = \\
( x\text{Win2} + \text{inchToPixel} \ (lx+x*sx), \\
y\text{Win2} - \text{inchToPixel} \ (ly+y*sy) )
\]
Polygon and RtTriangle

\[
\text{shapeToGRegion} \ (lx, ly) \ (sx, sy) \ ((\text{Polygon} \ pts)) \\
= \text{createPolygon} \ (\text{map} \ \text{trans} \ pts) \\
\quad \text{where} \ \text{trans} \ (x, y) = \\
\quad \quad (xWin2 + \text{inchToPixel} \ (lx + x*sx), \\
\quad \quad \quad yWin2 - \text{inchToPixel} \ (ly + y*sy))
\]

\[
\text{shapeToGRegion} \ (lx, ly) \ (sx, sy) \ ((\text{RtTriangle} \ s1 \ s2)) \\
= \text{createPolygon} \ (\text{map} \ \text{trans} \ [(0,0), (s1,0), (0, s2)]) \\
\quad \text{where} \ \text{trans} \ (x, y) = \\
\quad \quad (xWin2 + \text{inchToPixel} \ (lx + x*sx), \\
\quad \quad \quad yWin2 - \text{inchToPixel} \ (ly + y*sy))
\]
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.