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
- Picture `Over` Picture

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
- 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:

Efficient Bit-Map Operations
- 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

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.

Why IO here?

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

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)]
... (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 [] ++ yz = yz
(y:ys) ++ z = y : (ys ++ z)

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

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

We can do the same thing for Regions.
Accumulating the Scaling Factor

\[
\text{regToNReg2} :: \text{Region} \rightarrow \text{NewRegion} \\
\text{regToNReg2} \mathbf{r} = \text{rToNR} (1,1) \mathbf{r} \\
\text{where} \\
\text{rToNR} :: (\text{Float},\text{Float}) \rightarrow \text{Region} \rightarrow \text{NewRegion} \\
\text{rToNR} (x1,y1) (\text{Shape} (\text{Rectangle} sx sy)) = \text{Rect} (x1*sx) (y1*sy) \\
\text{rToNR} (x1,y1) (\text{Scale} (x2,y2) \mathbf{r}) = \text{rToNR} (x1*x2,y1*y2) \mathbf{r} \\
\]

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

\[
\text{regToGReg} :: \text{Vector} \rightarrow \text{Vector} \rightarrow \text{Region} \rightarrow \text{G.Region} \\
\text{regToGReg} \mathbf{loc} \mathbf{sca} (\text{Shape} \mathbf{s}) = \text{shapeToGRegion} \mathbf{loc} \mathbf{sca} \mathbf{s} \\
\text{regToGReg} \mathbf{loc} (sx,sy) (\text{Scale} (u,v) \mathbf{r}) = \text{regToGReg} \mathbf{loc} (sx*u, sy*v) \mathbf{r} \\
\text{regToGReg} \mathbf{loc} (lx,ly) (sx,sy) (\text{Translate} (u,v) \mathbf{r}) = \text{regToGReg} \mathbf{loc} (lx+u*sx, ly+v*sy) \mathbf{r} \\
\text{regToGReg} \mathbf{loc} \mathbf{sca} \text{Empty} = \text{createRectangle} (0,0) (0,0) \\
\text{regToGReg} \mathbf{loc} \mathbf{sca} (\mathbf{r1} \cup \mathbf{r2}) = \text{orRegion} \mathbf{gr1} \mathbf{gr2} \\
\text{regToGReg} \mathbf{loc} \mathbf{sca} (\mathbf{r1} \cap \mathbf{r2}) = \text{andRegion} \mathbf{gr1} \mathbf{gr2} \\
\text{regToGReg} \mathbf{loc} \mathbf{sca} \text{Complement} \mathbf{r} = \text{diffRegion} \mathbf{createRectangle} (0,0) (0,0) \text{winRect} \mathbf{r} \\
\]

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

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

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

\[
\text{primGReg} \mathbf{loc} \mathbf{sca} \mathbf{r1} \mathbf{r2} \mathbf{op} \\
= \text{let} \mathbf{gr1} = \text{regToGReg} \mathbf{loc} \mathbf{sca} \mathbf{r1} \\
\quad \mathbf{gr2} = \text{regToGReg} \mathbf{loc} \mathbf{sca} \mathbf{r2} \\
\quad \mathbf{in} \mathbf{op} \mathbf{gr1} \mathbf{gr2} \\
\]

Definition by cases with a Case Expression

\[
\text{regToGReg} :: \text{Vector} \rightarrow \text{Vector} \rightarrow \text{Region} \rightarrow \text{G.Region} \\
\text{regToGReg} \mathbf{loc} (lx,ly) \mathbf{shape} = \text{case} \mathbf{shape} \text{ of} \\
\quad \text{Shape} \mathbf{s} \rightarrow \text{shapeToGRegion} \mathbf{loc} \mathbf{sca} \mathbf{s} \\
\quad \text{Translate} (u,v) \mathbf{x} \rightarrow \text{regToGReg} \mathbf{loc} \mathbf{sca} ((x*u,y*v)) \mathbf{x} \mathbf{r} \\
\quad \text{Scale} (u,v) \mathbf{x} \rightarrow \text{regToGReg} \mathbf{loc} \mathbf{sca} ((x*u,y*v)) \mathbf{r} \\
\quad \text{Empty} \mathbf{r} \rightarrow \text{createRectangle} (0,0) (0,0) \\
\quad \text{Union} \mathbf{r} \rightarrow \text{regToGReg} \mathbf{loc} \mathbf{sca} (\text{shapeToGRegion} \mathbf{loc} \mathbf{sca} \mathbf{r1} \mathbf{r2}) \\
\quad \text{Intersect} \mathbf{r} \rightarrow \text{regToGReg} \mathbf{loc} \mathbf{sca} (\text{shapeToGRegion} \mathbf{loc} \mathbf{sca} \mathbf{r1} \mathbf{r2}) \\
\quad \text{Complement} \mathbf{r} \rightarrow \text{regToGReg} \mathbf{loc} \mathbf{sca} \text{winRect} \mathbf{r} \\
\]

Drawing Pictures

\[
\text{draw} :: \text{Picture} \rightarrow \text{IO} () \\
\text{draw} \mathbf{p} = \text{runGraphics} \mathbf{p} \\
\mathbf{p} \mathbf{w} <\left\{ \begin{array}{l} \\
\quad \text{openWindow} \text{"Region Test"} (xWin, yWin) \\
\quad \text{drawPic} \mathbf{w} \mathbf{p} \\
\quad \text{spaceClose} \mathbf{w} \\
\end{array} \right. \\
\]

Page <#>
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`
     (r1 `Intersect`
      Complement r2 `Union`
      r4)

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 = \x -> oneCircle
fiveCircles = fold Union Empty (take 5 manyCircles)

Main4 = draw pic4
Ordering Pictures

Lists the Region 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

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.

Putting it all Together

Try it Out

Drawing Pictures

```
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" [Win,Win]
     loop w (pictToList pic)
```

```
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 ]
  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 <-
loop2 w (pictToList pic)

Extra slides...
Implementing ShapeToGRegion

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) )

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) )

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) )

The Ellipse Case

shapeToGRegion (lx,ly) (sx,sy) (Ellipse s1 s2) =
  createEllipse (trans (-r1,-r2)) (trans (r1, r2))
  where trans (x,y) = ( xWin2 + inchToPixel (lx+x*sx),
                       yWin2 - inchToPixel (ly+y*sy) )

Polygon and RtTriangle

A Matter of Style, 2