A Haskell Roadshow

By Joachim Breitner, for The Karlsruhe Functional Programmers Meetup Group, December 18, 2012.

The following is a rough transcript of the code that I develop duing the talk. It does not contain all the steps and the comments, but gives a good overview. Also, while here I append numbers to function names when I improve them, during the talk, the functions are just modified. So when reading the code, just ignore the appended numbers. Also ignore any ocurrence of `undefined`; these are there to fill in fields that I would not have added at that point during the talk. Stage directions are in *italics.*

Step 1: First picture

```
import Data.List
import Data.Char
import Data.Ord
import Data.Function
import Data.List.Split
import Data.Data
import Graphics.Gloss
-- Add in step 8
import Optimisation.CirclePacking
-- Add in step 10
import Graphics.Gloss.Interface.Pure.Game
```

Develop `findNames` with ghci feedback

```
findNames :: String -> [String]
findNames input =
    [ surname |
        l <- lines input ,
        let fullname = (endBy ":" l) !! 4,
        not (null (words fullname)),
        let surname = last (words fullname),
        head surname ‘elem‘ [’A’..’Z’]]
```

Next develop `frequencies` in GHCi

```
frequencies :: [Char] -> [(Char, Int)]
frequencies = map (\l -> (head l, length l)) . group . sort
```

We will have to put more data in, so lets create a data type. *At first, no color, no pos, no nextPos!*

```
data CharCircle = CharCircle {
    char :: Char,
    col :: Color,
    count :: Int,
    pos :: (Double, Double),
    nextPos :: (Double, Double)
    }
```

Every circle needs a radius. Do not put this in the value, but store it separately.

```
radius :: CharCircle -> Double
radius c = sqrt (fromIntegral (count c))
```

We can convert our stats to a CharCircle. *Ignore undefined*

```
toCircle1 :: (Char, Int) -> CharCircle
toCircle1 (c,n) = CharCircle c undefined n undefined undefined
```

Time to get drawing. Looking through hackage for something relating *vector graphics* and *simple*, we stumble on gloss. Judging from the docs, this code should do it. *Add Text and center bit by bit, draw one circle first. Use `snippet-centerText.txt`*

```
drawCircle1 :: CharCircle -> Picture
drawCircle1 c =
    pictures [ circleSolid (realToFrac (radius c)) ,
               color black $ circle (realToFrac (radius c)) ,
               centerText (char c) ]

centerText :: Char -> Picture
centerText c =
    color black $
    scale 0.1 0.1 $
    translate (-50) (-50) $
    text [c]

main1 = do
    input <- readFile "passwd"
    let stats = frequencies $ map head $ findNames input
    let circles = map toCircle1 stats
    let pic = color blue $ pictures $ map drawCircle1 circles
    display (FullScreen (1024,768)) white pic
```

Step 2: Positioning circles

Clearly not satisfiying. We need position the circles better. So lets add a function for placing a circle, storing the position in the circle and lets put them all on a line:

```
putAt2 :: Double -> Double -> CharCircle -> CharCircle
putAt2 x y c = c { pos = (x,y) }

toCircle2 :: (Char, Int) -> CharCircle
toCircle2 (c,n) = CharCircle c undefined n (0,0) undefined

drawCircle2 :: CharCircle -> Picture
drawCircle2 c =
    let (x,y) = pos c in
    translate (realToFrac x) (realToFrac y) $
    pictures [ circleSolid (realToFrac (radius c)) ,
               color black $ circle (realToFrac (radius c)) ,
               centerText (char c) ]

placeLine2 :: Double -> [CharCircle] -> [CharCircle]
placeLine2 dist = zipWith
    (\i c -> putAt2 (dist * fromIntegral i) 0 c)
    [0..]

main2 = do
    input <- readFile "passwd"
    let stats = frequencies $ map head $ findNames input
    let circles = map toCircle2 stats
    let placed = placeLine2 30 $ circles
    let pic = color blue $ pictures $ map drawCircle2 placed
    display (FullScreen (1024,768)) white pic
```

Step 3: Adding color

The coloring gives me the creeps. We need more colors! Lets define some (`snippet-colors.txt`):

```
colors :: [Color]
colors = [red, green, blue, yellow, cyan, magenta,
          violet, azure, aquamarine, orange]
```

We also want to store them. So let us extend the data type and assign them to the circles.

```
toCircle3 :: (Char, Int) -> Color -> CharCircle
toCircle3 (c,n) col = CharCircle c col n (0,0) undefined

drawCircle3 :: CharCircle -> Picture
drawCircle3 c =
    let (x,y) = pos c in
    translate (realToFrac x) (realToFrac y) $
    pictures [ color (col c) $ circleSolid (realToFrac (radius c)) ,
               color black $ circle (realToFrac (radius c)) ,
               centerText (char c) ]

main3 = do
    input <- readFile "passwd"
    let stats = frequencies $ map head $ findNames input
    let circles = zipWith toCircle3 stats colors
    let placed = placeLine2 30 $ circles
    let pic = pictures $ map drawCircle3 placed
    display (FullScreen (1024,768)) white pic
```

Step 4: Infinite lists!

Looks better! But we are missing some letters. Looks like we have not enough colors. So lets make sure we have enough, no matter how many cycles we are drawing!

```
main4 = do
    input <- readFile "passwd"
    let stats = frequencies $ map head $ findNames input
    let circles = zipWith toCircle3 stats (cycle colors)
    let placed = placeLine2 30 $ circles
    let pic = pictures $ map drawCircle3 placed
    display (FullScreen (1024,768)) white pic
```

Step 5 and 6: More dimensions

This layout clearly does not use the space well. Let us try to use the second dimention:

```
placeRectangle5 :: [CharCircle] -> [CharCircle]
placeRectangle5 = zipWith
    (\(i,j) -> putAt2 (-250 + 100 * i) (250 - 100 * j))
    [ (i,j) | j <- [0..5] , i <- [0..5] ]
```

And we can make the code a bit more fancy – pure vanity

```
placeRectangle6 :: [CharCircle] -> [CharCircle]
placeRectangle6 = zipWith id
    [ putAt2 (-250 + 100 * i) (250 - 100 * j) | j <- [0..5] , i <- [0..5] ]

main6 = do
    input <- readFile "passwd"
    let stats = frequencies $ map head $ findNames input
    let circles = zipWith toCircle3 stats (cycle colors)
    let placed = placeRectangle6 circles
    let pic = pictures $ map drawCircle3 placed
    display (FullScreen (1024,768)) white pic
```

Step 7: No overlaps, please.

Much better. But we dont want overlapping circles. Lets try to arrange them without overlap, one after another, on a row:

```
placeAutoLine7 :: Double -> [CharCircle] -> [CharCircle]
placeAutoLine7 _ [] = []
placeAutoLine7 x (c:cs) =
      putAt2 (x + radius c) 0 c :
      placeAutoLine7 (x + 2*radius c) cs

main7 = do
    input <- readFile "passwd"
    let stats = frequencies $ map head $ findNames input
    let circles = zipWith toCircle3 stats (cycle colors)
    let placed = placeAutoLine7 (-512) circles
    let pic = pictures $ map drawCircle3 placed
    display (FullScreen (1024,768)) white pic
```

Step 8: Fancy layout!

Good, but again we only use one dimension. Can we pack them tightly, but without overlap? Sounds not trivial anymore, lets see if there is a ready-made package. What a coincidence, there is one, circle-packing! The API looks simple enough, so here we go:

```
placePacked8 :: [CharCircle] -> [CharCircle]
placePacked8 cs = map (\(c,(x,y)) -> putAt2 x y c) $ packCircles radius cs

main8 = do
    input <- readFile "passwd"
    let stats = frequencies $ map head $ findNames input
    let circles = zipWith toCircle3 stats (cycle colors)
    let placed = placePacked8 circles
    let pic = pictures $ map drawCircle3 placed
    display (FullScreen (1024,768)) white pic
```

Step 9: Lets have it all.

Beautiful! But I cannot really decide which layout I like the most; I want them all. Luckily gloss makes it easy to also create interactive programs in the game mode, so lets try that:

First we assemble a list of all our circle placers. We use this as the state of our program. The simulation step does nothing, so we need to handle key events to select the next place from the list, and also provide a draw function.

```
placers9 = cycle [ placeLine2 30, placeAutoLine7 (-500),
                   placePacked8, placeRectangle6 ]

change9 (EventKey (SpecialKey KeySpace) Up _ _) ps = tail ps
change9 _ circles = circles

main9 = do
    input <- readFile "passwd"
    let names = findNames input
    let stats = frequencies (map head names)
    let circles = zipWith toCircle3 stats (cycle colors)

    let draw (placer:_) =
            pictures $ map drawCircle3 $ placer circles

    play (FullScreen (1024,768)) white 25
         placers9
         draw
         change9
         (const id)
```

Step 10: Animation

That works fine. But it could look even slicker. Lets try to animate the transition. For that we need to extend the datatype to store the current and the desired position, and set only the latter in the placers. *Placers code does not change besides updating the names of the used functions.*

```
toCircle10 :: (Char, Int) -> Color -> CharCircle
toCircle10 (c,n) col = CharCircle c col n (0,0) (0,0)
```

```
putAt10 :: Double -> Double -> CharCircle -> CharCircle
putAt10 x y c = c { nextPos = (x,y) }


placeLine10 :: Double -> [CharCircle] -> [CharCircle]
placeLine10 dist = zipWith
    (\i c -> putAt10 (dist * fromIntegral i) 0 c)
    [0..]
placeAutoLine10 :: Double -> [CharCircle] -> [CharCircle]
placeAutoLine10 _ [] = []
placeAutoLine10 x (c:cs) = putAt10 (x + radius c) 0 c : placeAutoLine10 (x + 2*radius c) cs
placePacked10 :: [CharCircle] -> [CharCircle]
placePacked10 cs = map (\(c,(x,y)) -> putAt10 x y c) $ packCircles radius cs
placeRectangle10 :: [CharCircle] -> [CharCircle]
placeRectangle10 = zipWith id
    [ putAt10 (-250 + 100 * i) (250 - 100 * j) | j <- [0..5] , i <- [0..5] ]
placers10 = cycle [placeLine10 30, placeAutoLine10 (-500), placePacked10, placeRectangle10 ]
```

Then we need a fuction that advances the circles towards their target position, depending on the time passed since the last advancement. A little geometry:

```
moveCircle10 seconds c =
      if dist <= adv
      then c { pos = nextPos c }
      else c { pos = (x',y')}
  where
    (x1,y1) = pos c
    (x2,y2) = nextPos c
    dist = sqrt ((x2 - x1)^2 + (y2-y1)^2)
    adv = speed * realToFrac seconds
    x' = x1 + (x2 - x1) * adv / dist
    y' = y1 + (y2 - y1) * adv / dist
    speed = 150
```

Our state is now both the placer list and the list of circles

In the change event, we not only pop the placer from the list, but also apply it. And we sort by character, as that influences the stacking order.

```
change10 (EventKey (SpecialKey KeySpace) Up _ _) (p:ps, circles) =
      (ps, sortBy (comparing char) $ p circles)
change10 _ state = state

moveCircles10 seconds (p, cs) = (p, map (moveCircle10 seconds) cs)
```

```
main10 = do
    input <- readFile "passwd"
    let names = findNames input
    let stats = frequencies (map head names)
    let circles = zipWith toCircle10 stats (cycle colors)

    let draw = pictures . map drawCircle3 . snd

    play (FullScreen (1024,768)) white 25
         (placers10, circles)
         draw
         change10
         moveCircles10
```

What main function to test at the moment.

```
main = main8
```