Australian National University

ASSIGNMENT COVER SHEET

ANU College of Engineering and Computer Science
Australian National University
Canberra ACT 0200 Australia
www.anu.edu.au

+61 2 6125 5254

Student ID	U6611178		
Name	Yanming Feng (Danny)		
Course Code & Name	COMP1100 - Programming as Problem Solving		
Assignment Topic	Technical Report for Assignment 2		
Lab Time	Monday 16:00 – 18:00		
Tutor	Jack Kelly <jack@jackkelly.name>		
Word count	1200	Due Date	7 May 2018
Last Edited	6 May 2018	Extension Granted	

Initials	YMF
For group assignments, each student must initial.	

Technical Report for Assignment 2

Yanming Feng

1. Program Introduction

This Assignment is about computer graphics. It uses Haskell to draw raster images. Most of the tasks are to implement the Bresenham Algorithm[1] to draw line, polygon, and circles. To run the program, the user should run *“cabal run”*, and then type the initial character of the shape and the colour they what to draw. Left click the mouse, drag and release to draw. For drawing a polygon, left click the mouse to create the vertex and press space bar to draw.

2. Functions Objectives

polygonCord: This function I made is called by *shapeToRaster*. It is adding the Smooth Boolean value to the list of point. However, I can improve this function by just using the *map* function in *shapeToRaster*

bresenHam[2]: This function is to implement the BresenHam Algorithm to draw a line. The ideas are inspired by Wikipedia, reddit forum and Haskell wiki. I understood what it does and implemented it.

polyLineRaster': I created two functions to draw the polygon. *polyLineRaster* is the main body that calls the *lineRaster* to draw the head and last points. Then it calls *polyLineRaster’* to draw the rest of the polygon.

isqrt: This function I made is used to calculate in square root for integer.

circleRaster[3] : Using midpoint circle algorithm to draw to circle. The ideas are taken from the Internet. I edited it to make it easier to understand.

3. Efficiency

My *rectangleRaster* used min to max method to map the value from minimum to maximum in order to draw all the points of the rectangle. However, when I did the doctest. I found out that my function will draw some points twice. This is the side effect of this method, because *zip min x1 x2* and *zip[x1,x2]* might give the same point. Although the side effect won’t influence the drawing result, the code is not the efficiency. To improve it, the better way is using list comprehension.

```haskell
rectangleRaster :: Coord -> Coord -> Raster
rectangleRaster (x1,y1) (x2,y2) =
    zip (zip [(min x1 x2)..(max x1 x2)] [y1,y1..]
  ++ zip [(min x1 x2)..(max x1 x2)] [y2,y2..]
  ++ zip [x1,x1..] [(min y1 y2)..(max y1 y2)]
  ++ zip [x2,x2..] [(min y1 y2)..(max y1 y2)]) [1,1..]
```

4. Issues Encountered, Solutions, and Potential Improvement

1. Out of memory issues. This is the first version of my *rectangleRaster* function, but it seems goes into an infinity output because when I ran it, it took all my memory space. I tried but didn’t find out what was going wrong. I thought it is doing the right thing. So, I used *zip* function built-in in Haskell and rewrote this function completely.

```
rectangleRaster (x1,y1) (x2,y2) = [((x,y),1)|x<-[x1,x1..],y<-
[minimum(y1,y2)..maximum(y1,y2)]]
  ++[((x,y),1)|x<-[x2,x2..],y<-[minimum(y1,y2)..maximum(y1,y2)]]
  ++[((x,y),1) |x<-[minimum(x1,x2)..maximum(x1,x2)],y<-[y1,y1..]]
  ++[((x,y),1) |x<-[minimum(x1,x2)..maximum(x1,x2)],y<-[y2,y2..]]
```

Similarly, the *lineRaster* function I cited from the Internet also gave an infinity output.

```
lineRaster :: Smooth -> Coord -> Coord -> Raster
balancedWord :: Int -> Int -> Int -> [Int]
balancedWord p q eps | eps + p < q = 0 : balancedWord p q (eps + p)
balancedWord p q eps               = 1 : balancedWord p q (eps + p - q)

lineRaster s (x0,y0) (x1,y1) =
  let (dx, dy) = (x1 - x0, y1 - y0)
      xyStep b (x, y) = (x + signum dx,     y + signum dy * b)
      yxStep b (x, y) = (x + signum dx * b, y + signum dy)
      (p, q, step) | abs dx > abs dy = (abs dy, abs dx, xyStep)
                   | otherwise       = (abs dx, abs dy, yxStep)
      walk w xy = xy : walk (tail w) (step (head w) xy)
  in  zip (walk (balancedWord p q 0) (x0, y0)) [1,1..]
```

2. Misunderstood the high order function.

When I did *polyLineRaster*, I used two functions to draw.

```
polyLineRaster:: Smooth -> [Coord] -> (Smooth -> [Coord] -> Raster) -> Raster
polyLineRaster z p _ = lineRaster z (head p) (last p) ++ polyLineRaster' z p
```

I found out that I actually don’t need to write *(Smooth -> [Coord] -> Raster)* because it is not using the function *polyLineRaster*` as an input.

3. I was not familiar with the *map* function. I wrote my own function *polygonCord* to map. But Haskell gave me a tip so I replaced it to use map and makes it concise.

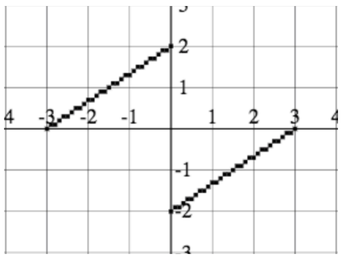
```
Polygon p -> polyLineRaster s (polygonCord z p)
polygonCord :: Resolution -> [Point] -> [Coord]
polygonCord z [] = []
polygonCord z (p:ps) = pointToCoord z p: polygonCord z ps

--Doing the same thing here
Polygon p -> polyLineRaster s (map (pointToCoord z) p)
```

4. When I first did *polyLineRaster* function, I forgot to consider the situation that all points are in a line. That led to the redundant drawing (draw the same line multiple times).

```
polyLineRaster z [p1,p2] = lineRaster z p1 p2
polyLineRaster z p = lineRaster z (head p) (last p) ++ polyLineRaster' z p
```

5. Some issues met when doing polygon

Using the example provided in the instruction. User wants to draw the polygon make from these four points. (-3,0),(0,2),(3,0),(0,-2)

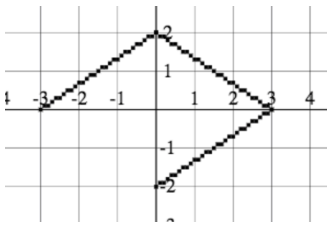


a.The first issue was that, I can only connect parts of the points. The reason is I didn't implement the recursion correctly. I wrote

```
polyLineRaster z (p1:ps) = lineRaster z p1 p2 ++ polyLineRaster z ps
```

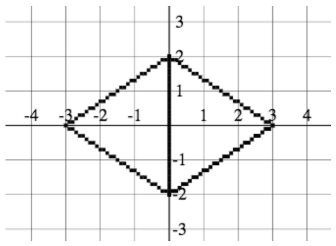
Giving p1 p2 p3 p4, this recursion always connects p1 p2, and p3 p4. It skipped p2 to p3, and p4 to p1. It can be simply fixed like this. Just used the next element and the rest of the elements to do the recursion.

```
polyLineRaster z (p1:p2:ps) = lineRaster z p1 p2 ++ polyLineRaster z (p2:ps)
```



b. Cannot connect head and last points. This is because the recursion stops at p4. It won't go from p4 to p1. I fixed this by asked Haskell to connect the first and last elements in the list like this.

```
polyLineRaster z (p1:p2:ps) = lineRaster z (head (p1:p2:ps)) (last (p1:p2:ps))++lineRaster
z p1 p2 ++ polyLineRaster z (p2:ps)
```



c. However, this brought me a new issue. It appeared an extra line inside of the polygon. After I traced the value, I found out that I shouldn't connect the head and last inside of the recursion because the head is changed in each recursion. Therefore, it connected the wrong line inside of the graph. I fixed this issue by moving the head and list drawing method to a separated function.

References

[1] J.E. Bresenham, Algorithm for computer control of a digital plotter, IBM Systems Journal 4(1):25–30, 1965, http://dx.doi.org/10.1147/sj.41.0025.
[2] 'Thoughts on Bresenham's Algorithm in Haskell' 2010, in reddit inc., viewed 5 May 2018, <https://www.reddit.com/r/haskell/comments/9mp6j/thoughts_on_bresenhams_algorithm_in_haskell/>.
[3]'Midpoint circle algorithm' 2018, in Rosetta Code, viewed 5 May 2018, <https://rosettacode.org/wiki/Bitmap/Midpoint_circle_algorithm#Haskell>.

Acknowledgement

Thanks to Alex. We studied together at CSIT, and shared just ideas of this assignment.