Detailed Instructions for CO2016 Coursework 1

Please note: the demonstration video is an example of image effect linearBox and phase-Shift.

When the template code ImageManipulation.java is run, an image appears on the screen as part of a GUI. The image coordinates of your mouse are (x,y); and note that x and y are also code variables. We define A(x,y) to be a square of width 2*size with center (x,y), where size is also a code variable. There is a code variable n too; more on that later.

You are going to write code for methods linearBox and linearOct. These should both produce image effects, as discussed in the lectures: thus they have one input parameter being the image, but also input parameters x, y, size, and n. Each ``effect'' is determined by ALL these parameters. The original image should change as you move your mouse around the image (see video): the value of (x,y) alters, and for each new value one of the methods is called, producing an image effect.

These effects involve moving/recoloring pixels as outlined in the lectures. Please read the slides carefully! Each method should not go through each pixel q and "move" it to its new position q' = Move(q). Each method should, however, go through each pixel p and either color it correctly, or calculate which pixel prep "will be moved" to the position of p. We call prep the PREIMAGE of p, so prep = MoveInverse(p). We then perform the ``move'' by setting the RGB of p to be the RGB of prep.

Both of these methods apply a specific image transformation, IMGTRANS. To explain IMGTRANS, consider a pixel-line with end-points O and P, and a point D on the line:

```
O      i               D                                    P
xxxxxxxxxxxxyyyyyyyzzzzzAAAAAAAAAAAAAAAABBBCCCCCCCCCCCCCCCCCC
```

After applying the transformation, the (new) pixels between O and D are coloured GRAY (g below). The original pixel at O moves to P, the pixel at D is fixed, and the pixels in between O and D are MOVED AND SCALED LINEARLY (*), each i to mi. So the output looks like

```
O                       D                           mi      P
ggggggggggggggggggggggggzzzzzyyyyyyyyyyxxxxxxxxxxxxxxxxxxxxxx
```

and (*) means by definition that either of the ratios of distances below are equal

```
      mi to P : D to P    =    O to i : O to D    (*)
      D to mi : D to P    =    i to D : O to D    (*)
```

Note that, informally, the output pixels from D to P are like a stretched reflection (mirror at D facing O) of the input pixels from O to D - this idea will help you visually when testing your code. mi is the reflection of i.

**** linearBox ****

linearBox parameters are

@param image	the image you are transforming
@param x	the x-coordinate of the centre of the square A(x,y)
@param y	the y-coordinate of the centre of the square A(x,y)
@param size	half the length/width of the square
@param n	see below

```
=======================================================================
linearBox SPECIFICATION: Check if A(x,y) is contained within the given
image.

If not the method has no effect (i.e. identity effect).

If A(x,y) is contained within the image, then visible changes
should be seen (i.e. a non-identity effect) but ONLY within
A(x,y). These changes result from applying IMGTRANS to each horizontal
line of pixels in A(x,y).
=======================================================================

To code up the SPECIFICATION

(i) Write code to check the containment of A(x,y)

(ii) Hence, if required, use a loop to visit each pixel (i,j) in A(x,y).

(iii) Apply IMGTRANS: For (i,j) there is a horizontal line of pixels
in A(x,y) where

          O is at the left side of the square A(x,y)
          D is at x+n (for a parameter int n)
          P is at the right side of the square A(x,y)

and each pixel in the line has a downwards image coordinate j.

The tricky part is calculating the pixel movements; selecting and
coloring grey pixels is easy. For the movements, if pixel p (on the
line) at coordinate (i,j) is between D and P, then pixel prep
(on the line between O and D) has coordinate (prei,j). Thus, crucial
to the coding is working out prei in terms of i, and this is what
linTrans does.
```

```
        *******

        Exercise: Work out prei in terms of i and O, D and
        P. EITHER

        --- by making use of (*) OR
        --- since the scaling is LINEAR this means that

            prei =  g*i + k

            a STRAIGHT LINE GRAPH with gradient g and ``prei
            intercept'' k. You know that if i=D then prei=D and if
            i=P then prei=O; from this you can work out g and k.

        This should give you the code for method linTrans.

        *******

Note: Due to rounding/integer division effects you should compute prei
with doubles and return the final result for prei as an int.

Note: prei is calculated ONLY for pixels between D and P. The original
pixels between O and D are changed to gray.

Note: In the skeleton program, a copy, called temp, of the
BufferedImage is made.  Use the temp image to get the RGB values of
pixels prep and set the RGB values of the BufferedImage image pixels p
to these values; see lecture slides.


***************************************************************
```

```
**** linear Oct ****

==========================================================================
linearOct SPECIFICATION: Check if A(x,y) is contained within the given
image.

If not the method has no effect (i.e. identity effect).

If A(x,y) is contained within the image, then visible changes
should be seen (i.e. a non-identity effect) but ONLY within
A(x,y). These changes result from applying IMGTRANS to each
line of pixels ODP in A(x,y) where (see DIAGRAM 1)

          O is at the center of the square A(x,y)
          P ranges from point G to H, and from F to K. P is given by the
            intersection of the line from O passing through (i,j),
            and the left or right sides of A(x,y) (ie GH and FK).
          D is at the intersection of a circle centered at O with
            radius size/3.

This means that each of the four octants OGV, OVH, OKU and OUF are
visibly transformed.
==========================================================================

To code up the SPECIFICATION

(i) Write code to check the containment of A(x,y)

(ii) Hence, if required, use a loop to visit each pixel (i,j) in A(x,y).

(iii) Apply IMGTRANS to each line ODP specified by an (i,j).

Again, the tricky part is calculating the pixel movements; selecting and
coloring grey pixels is easy.
For the movements, if pixel p (on the line) at coordinate (i,j) is
```

between D and P, then pixel prep (on the line between O and D) has
coordinate (prei,prej). So we must calculate (prei,prej) from (i,j);
you can then "perform the move". If (i,j) is on OD make it gray.

To do so we need to use mathematical cartesian coordinate geometry!
You can see this in DIAGRAM2.
And all calculations will be easier if we translate A(x,y) (with image
coordinates), to A(0,0) in the Cartesian coordinate system. The axes
are now labelled I and J. We write (I,J) for the cartesian coords in
A(0,0) of image coordinate (i,j) in A(x,y).
We then compute (preI,PreJ) with cartesian coords in A(0,0) and
translate back to image coords (prei,prej) in A(x,y). The special
method octlinTrans computes (preI,preJ) from (I,J).

The remaining paragraphs elaborate on these basic ideas.

Translating A(x,y) in image coords to A(0,0) in cartesian coords:

```
      Example: suppose items Q and T are on a straight line with
      origin O (say at distances D and E from O)

      _________________O______________Q______T

      and Q is translated to O, with T moved relative to Q. Then T is
      now at ** distance E-D **  from the origin with Q at distance D-D=0.

      _________________X______Y_______________

      Q moves distance D to O; so all other distances change by
      relative amount D too.
```

In the case of A(x,y), the pixel at (x,y) is translated to cartesian
coords, and THEN to (0,0). All other points of A(x,y) are also
translated to cartesian coordinates, and then translated relative to

(0,0). In the case of (x,y) we have (x,y) --> (x,-y) -->
(x-x,(-y)-y) --> (0+x,0+(-y)) --> (x,-(-y)) ie (x,y). You'll need to work out
the same process for (i,j).

We can now work with cartesian coordinates (I,J) and the pixels inside
the square A(0,0) at the origin.

More on octlinTrans.
Now consider DIAGRAM2, and only octant OGV. We have to look at each
pixel at (I,J) on its line DP and work out (preI,preJ).
In DIAGRAM3 picturing OPV, if we know the distances of D, (I,J) and P
from O, we can then apply linTrans, and work out (preI,preJ). Note: a
slight cheat here: let's write d for the distance of (I,J), but
overload O, D, P as variables for the distances. Then

```
        theta = arctan(J/I)
            O = 0                    ie O is at the origin
            P = ?                    ie distance O to P
            D = size/3               ie distance O to D
            d = ??
```

Use some coord geometry to compute P and d. Then call linTrans to
compute pred. And finally use coord geometry to compute preI and
preJ. You can then complete the method octlinTrans. Don't forget to
calculate preI and preJ using doubles and finally coerce to int.

For each (I,J) you should work out which octant it is in, and change
coordinates so that (I,J) "moves to" OGV. Note that octant OGV is
defined by 0 < J < I, which is codeable. See DIAGRAM2 for hints on one
other quadrant. For the move, here is one example. If (I,J) is in OVH
then (I,-J) is in OGV: so you can call octlinTrans on (I,-J).

We should now be able to write code to apply IMGTRANS in all octants.

Note: The parameter n is not actually used in linearOct.

```
*****************************************************************

      **** phaseShift ****

phaseShift performs a shift of color on a specified area.

The specified area is a subset S of a square A(x,y) with center (x,y)
and corners (x-size, y-size), (x+size,y-size), (x+size, y+size),
(x-size, y+size).  The subset S is A(x,y) intersect (image -
boundary), where boundary consists of those pixels that are less than
2*n pixels away from any image edge.  The pixels p of S will be
changed in the following way: If p is at (i,j) and i >= 300 then the
RGB of p should be set to PURE maximum blue.  Otherwise the red, green
and blue components of p will all be equal to the green component of
the pixel that is 2*n pixels vertically up from p. Thus the changed pixels of
will always be RGB grayscale if i<300.

      @param image if the image you are transforming
      @param n as defined above
      @param x the x-coordinate of the centre of the square A(x,y)
      @param y the y-coordinate of the centre of the square A(x,y)
      @param size half the length/width of the square
```