Imperial College of Science, Technology and Medicine	Department of Computing
Computing Science (CS) / Software Engineering (SE)	BEng and MEng
Examinations Part I	Integrated Laboratory Course

Laboratory work is a continuously assessed part of the examinations

and is a required part of the degree assessment.

Laboratory work must be handed in for marking by the due date.

Late submissions may not be marked.

Exercise: 11	Working: Individual
Title: Processing Pictures in Java	
Issue date: 2nd February 2009	Due date: 9th February 2009
System: Linux	Language: Java

Aims

- To introduce writing simple programs in Java.
- To write programs that create multiple objects using Java classes.
- To explore the use of packages in Java.
- To gain further experience processing multi-dimensional arrays in Java.

Problem

- Your task is to *implement* several picture transformations by manipulating two-dimensional arrays.
- Write **PictureProcess.java**, making use of the helper classes **Color.java**, **Picture.java** and **PictureTool.java**. Your implementation class should reside in the **picture** package along with the helper classes provided.

Picture and PictureTool

- An image can be represented in memory as a bounded two-dimensional array of pixel 'values'. A colour-model is used to translate a pixel-value to colour components. In this lab, pixel-values will be interpreted using the RGB colour-model, so that each point within an image is mapped on to a red, green and blue component. These components are encapsulated in the provided class `picture.Color` which provides get and set methods for each primary colour. Each component has 256 possible intensities, ranging from 0 to 255. The final colour of each pixel depends on the intensities of the primary colour components. The coordinates (x,y) always mean "along and up", counting from $(0,0)$ at the bottom left.

Color

The class `picture.Color` provides methods for inspecting and setting the colour components of a pixel. These methods are:

```
public int getRed() ;
public int getGreen() ;
public int getBlue() ;
public void setRed(int red) ;
public void setGreen(int green) ;
public void setBlue(int blue) ;
```

Picture

The class `picture.Picture` provides the following interface for manipulating and querying images.

```
public int getWidth();
/*
 * Return the width of the Picture.
 */

public int getHeight();
/*
 * Return the height of the Picture.
 */

public Color getPixel(int x, int y);
/*
 * Return the colour of the pixel-value located at (x,y).
 */

public void setPixel(int x, int y, Color rgb);
/*
 * Update the pixel-value at the specified location.
 */

public boolean contains(int x, int y);
/*
 * Test if the specified point lies within the boundaries of this picture.
 */
```

PictureTool

- The class `picture.PictureTool` provides a convenient set of methods to create, load and display `Picture` objects. You will need to search the web to find the web address of the pictures that you want to load. If you find a web page that contains a picture you can get the URL (or web address) by viewing *page source* or *page info* on your web browser.

Pictures are usually stored as files with a *.gif*, *.jpeg* or *.jpg* suffix. For example the URL of an image of GuiLin gorge in China is *"http://images.cnd.org/Scenery/Top-10/GuiLin2.jpg"*.

```
public static Picture createPicture(int width, int height);
/*
 * Create a new instance of a Picture object of the specified width and
 * height, using the RGB colour model.
 */

public static Picture loadImage(String url);
/*
 * Create a Picture object from the the image at the specified URL.
 */

public static void toScreen(Picture picture);
/*
 * Display the specified Picture on-screen.
 */
```

Picture Transformations

You will need to write **PictureProcess.java** using the above classes to load, process and display images as specified below. You should provide the command to be executed along with any arguments required and the URL(s) of the picture(s) to be transformed and displayed as command line arguments to your main program given after the normal execution arguments explained later.

Invert

The invert transformation creates a picture where the colour components of each pixel are inverted. A colour component can be inverted by replacing the original intensity value of each primary colour, c, with the intensity (255-c).

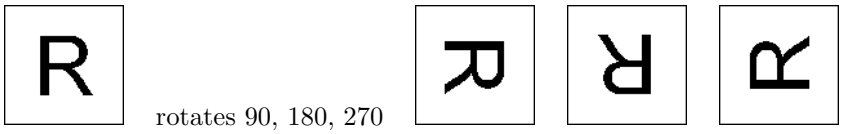
inverts to


Grayscale

The grayscale transformation creates a monochrome version of the input picture. Gray values, under the RGB colour model, are defined when the values for red, green and blue are equal. A 'gray' value can be computed by first, finding the average, avg of the three colour components, then creating a new colour with components {red=avg, green=avg, blue=avg}. (You can use integer division (by 3 in this case) freely without worrying about rounding errors.)

Rotate

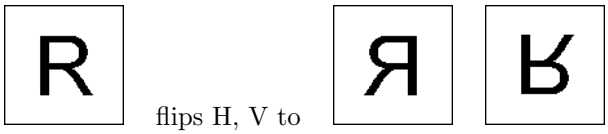
The rotate transformation creates a picture that is rotated by **90**, **180** or **270** degrees clockwise about the picture's centre. The angle of rotation should be specified as a command-line parameter to your PictureProcess implementation. e.g. `java picture.PictureProcess rotate 90` *url*.



rotates 90, 180, 270

Flip

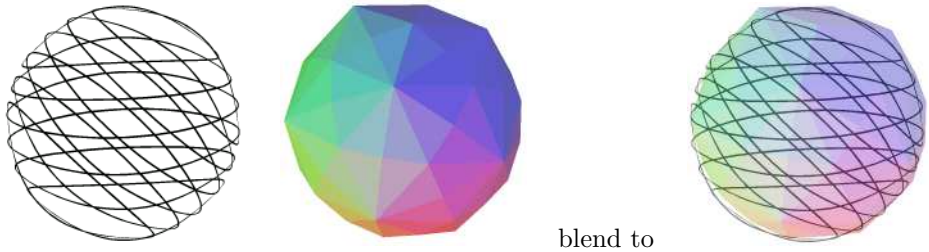
The flip transformations rotate a picture about an axis. 'Flip horizontal' mirrors the image about the y-axis, while 'Flip vertical' mirrors the image about the x-axis. The direction of reflection horizontal, **H** or vertical, **V** should be specified as a a command-line parameter to your PictureProcess implementation. e.g. `java picture.PictureProcess flip H` *url*.



flips H, V to

Blend

The blend transformation takes a *list* of pictures and combines them together so they appear layered on top of each other. The output picture will have dimensions corresponding to the *smallest* individual width and individual height within the set of specified pictures to be blended. A blended pixel is computed by finding the average colour component of each pixel across the list of pictures at any point. The list of pictures should be parameterised as a list of URLs passed to the PictureProcess command. e.g. `java picture.PictureProcess` url_1 url_2 `...`url_n.



blend to

Blur

The blur transformation creates a blurred version of the input picture. A blurred-pixel-value is computed by setting its pixel-value to the average value of its surrounding 'neighbourhood' of pixels. For example, the average of the neighbourhood:

$$\text{new value for } e \quad = \quad average \begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} \quad = \quad \frac{\sum\{a,b,c,d,e,f,g,h,i\}}{9}$$

Boundary pixels, where a 3x3 neighbourhood is not defined, should not be changed.

 blurs to 

Submit by Monday February 9th 2009

What to Do

- Copy the helper files: **Color.java**, **Picture.java** and **PictureTool.java** from CATE or with the command **exercise 11**. Note that these files begin with the line:
  `package picture ;`
  which indicates that these classes are part of the package **picture**. As these classes are part of the package **picture** they will need to be put into a subdirectory called **picture**. (The "exercise" command does this for you; if you fetch the files from CATE you will need to do this yourself.) You should place your main program file **PictureProcess.java** in the package **picture** as well. This file should be where you write a **main()** method and any supporting methods.

- To compile your program you should use the **javac** compiler in the directory above the **picture** subdirectory. You should compile your program using the **javac** compiler using the command: `javac picture/*.java`. The should compile the Java classes in this package and produce files with the same base name but with a *.class* suffix. You can then run your program with `java -ea picture/PictureProcess <arguments-to-main>`.

- Before writing your various methods you are advised to write a simple main method that loads and displys a URL without doing any processing. Once you have done this and are satisfied that you are familiar with the Java syntax necessary for writing main program classes you should continue to write the rest of your methods incrementally - testing each one as you complete it.

- **PictureTool.java** provides access methods to create, load and display **Picture** objects. Your implementation may only use Picture objects to represent images.

- Write **PictureProcess.java** within the package **picture**, implementing the transforms: *invert, grayscale, rotate, flip, blend, blur* and optionally *mosaic* (see the unassessed section below). Each transform should be invoked via the command-line. e.g.

```
java -ea picture.PictureProcess invert [url]
java -ea picture.PictureProcess grayscale [url]
java -ea picture.PictureProcess rotate [90|180|270] [url]
```

```
java -ea picture.PictureProcess flip [H|V] [url]
java -ea picture.PictureProcess blend [url_1] [url_2] ... [url_n]
java -ea picture.PictureProcess blur [url]

java ea picture.PictureProcess mosaic tile-size [url_1] [url_2] ... [url_n]
```

- Java passes any command line arguments to the main method in an array of Strings:

  ```
  public static void main ( String [] args )
  ```

  The first argument given in the array is the command to be executed and the following arguments are those appropriate to the command. Note that the "universal" invocation arguments, i.e. **java -ea PictureProcess**, will not be present in the **String** [] array - just the "extra" ones (command and any further arguments) that come after. Note that a String is a class in Java and it provides methods for comparing Strings for equality etc.

- **Test your code thoroughly** with a variety of pictures. Each transform *must* display the processed image on screen using the supplied `PictureTool.toScreen(..)` method.

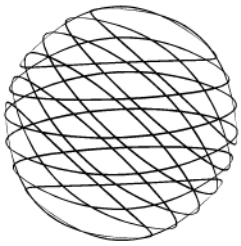
Unassessed

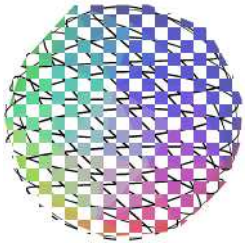
Mosaic

The mosaic transformation takes a *list* of pictures and combines them together to create a mosaic. The mosaic transform takes a integer parameter, **tile-size**, which specifies the size of a single square mosaic tile. The output picture will have dimensions corresponding to the *smallest* individual width and individual height within the set of specified pictures, *trimmed to be a multiple of the tile-size*.

The tiles in the picture are arranged so that for every tile, the neighbouring tiles to the east and south come from the next picture in the list, (wrapping round as appropriate). The top-left tile comes from the first picture. e.g. Consider the blending of pictures a, b and c (of different size):

$$\begin{pmatrix} a_1 & a_2 & a_3 \\ a_4 & a_4 & a_6 \\ a_7 & a_8 & a_9 \end{pmatrix} + \begin{pmatrix} b_1 & b_2 & b_3 & b_4 \\ b_5 & b_6 & b_7 & b_8 \\ b_9 & b_{10} & b_{11} & b_{12} \end{pmatrix} + \begin{pmatrix} c_1 & c_2 & c_3 \\ c_4 & c_5 & c_6 \\ c_7 & c_8 & c_9 \\ c_{10} & c_{11} & c_{12} \end{pmatrix} = \begin{pmatrix} a_1 & b_2 & c_3 \\ b_5 & c_5 & a_6 \\ c_7 & a_8 & b_{11} \end{pmatrix}$$



mosaics to


Submission

- This lab wil be marked by demonstration and you should arrange a mutually convenient time to demonstrate your program to your PPT.
- Before submitting, you should **test your programs on a wide range of user inputs**.

  Submit your class:
  PictureProcess.java using the **CATE** system in the usual way.

Assessment

main method and input processing	1
invert	0.5
grayscale	0.5
rotate	1
flip	1
blend	1
blur	1
Design, style, readability	4
Total	10