SIMPLE IMAGE PROCESSOR

IMAGE PROCESSING (CS-3712)

H.A.KUSAL
140331D

Table of Contents

INTRODUCTION

The developed tool is intended to read digital images from the memory and manipulate the image to enhance or modify the properties. The program was developed in JAVA environment.

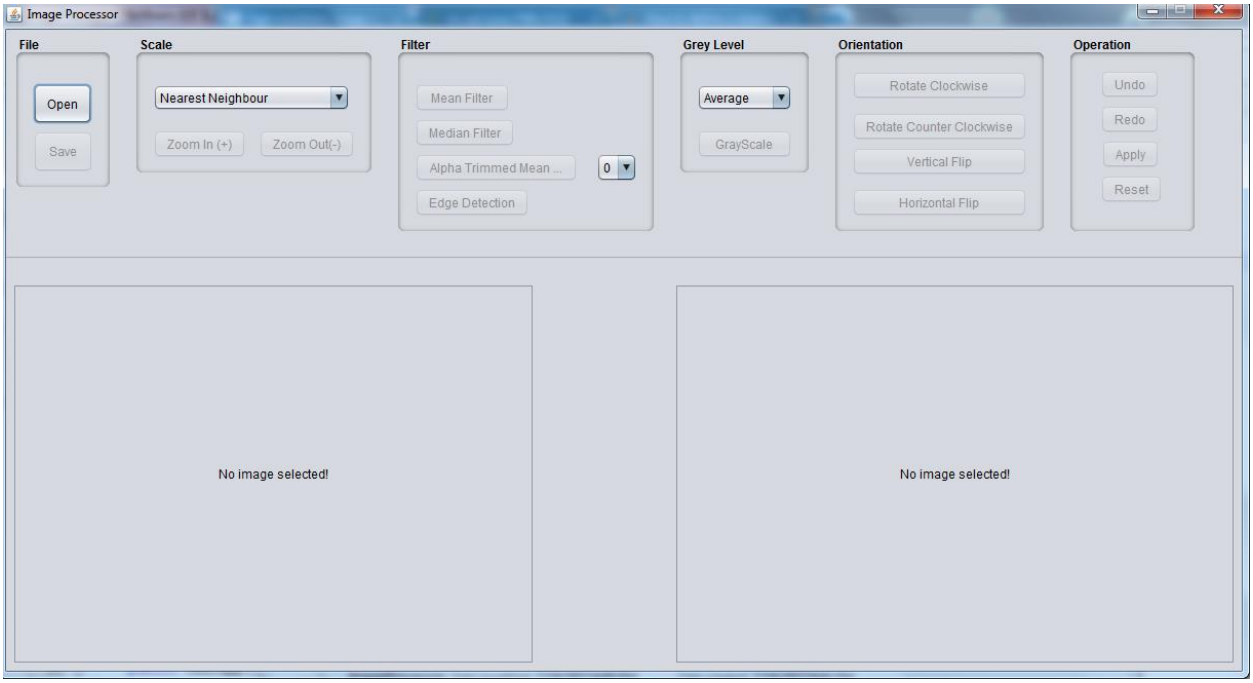

Figure 1: Image Processor

The tool is capable of performing the following operations on a selected image:

- Scaling
  - Sample up
  - Sample down
- Noise reduction
  - Mean filter
  - Median filter
  - Alpha trimmed mean filter
- Point operations
  - Grayscale
- Transformation
  - Flip (vertical and horizontal)
  - Rotation (clockwise and counter-clockwise)
- Edge detection

  Apart from these operations which can be performed on an image, operations such as undo, redo, apply and reset operations are implemented in the tool.

IMPLEMENTATION APPROACH

IMAGE ACQUISITION

Digital images which are stored in the computer can be read with JAVA ImageIO class. This class enables the implementation to acquire a digital image as a 2D array of Color values. This Color values include the RGB values of the pixel and thus these pixel values can be manipulated to process the image at large.
Reading the stored image files in to the program and writing the processed image files to a JPEG format image is handled via FileHandler class.

```
import javax.swing.JFileChooser;
import javax.swing.filechooser.FileFilter;
import javax.swing.filechooser.FileNameExtensionFilter;

/**
 *
 * @author kusalh
 */
public class FileHandler {

    private static final String ROOT_PATH = new File("").getAbsolutePath();
    static JFileChooser filechooser = new JFileChooser(ROOT_PATH.concat("\\src\\resources"));
    static FileFilter imagefilter = new FileNameExtensionFilter("Image Files", ImageIO.getReaderFileSuffixes());

    public static String openFile(){...13 lines }

    public static boolean SaveAsFile(){...22 lines }


}
```

Figure 2: FileHandler Class

Editor class is then responsible for creating a new BufferedImage object and processing the input image file to produce the targeted output image file. All the image processing methods are implemented within this class and can be accessed via the static Editor object “Editor”.

```
public class Editor {

    private static Editor Editor;
    String imagepath;
    //Stack for storing the temporal images while processing
    List<BufferedImage> versions;
    Deque<BufferedImage> undoStack;
    Deque<BufferedImage> redoStack;
    HashMap<List<Integer>,BufferedImage> scaledImages;
    BufferedImage image = null;
    BufferedImage originalImage;
    static boolean processed = false;

    public Editor(String path){...18 lines }

    public static Editor getEditor() {...3 lines }

    public static void setEditor(Editor editor) {...3 lines }

    public BufferedImage getOriginalImage() {...3 lines }

    public void undo(){...15 lines }

    public void redo(){...15 lines }


    public String getImagepath() {...3 lines }

    public BufferedImage getImage() {...3 lines }

    public void addImage(BufferedImage newImage){...4 lines }
```

Figure 3: Editor Class

IMPLEMENTED OPERATIONS

IMAGE SCALING

Image scaling, which is also known as image resampling is basically changing the resolution of the image. This operation either increases the number of pixels that represent the digital image or reduce it. When Up scaling and image (increasing the resolution) new pixels has to be introduced. And similarly when down scaling the image a set of pixels are lost. This operations are to be implemented in a way that the image details are preserved.
Two approaches are known for scaling a digital image.

1. Nearest Neighbor Method
2. Bi-Linear Interpolation Method

Nearest neighbor method which essentially duplicated or remove the neighbor pixels is implemented in the tool. As a result the scaling operation is carried out as a multiplication of 2.

```
public BufferedImage scaleDown(){

    int divisor = 2;
    int [] size = this.getSize();
    List<Integer> newSize = new ArrayList<>(Arrays.asList((int)(size[0]/divisor)/10, (int)(size[1]/divisor)/10));
    BufferedImage newImage = new BufferedImage((int)(size[0]/divisor), (int)(size[1]/divisor), BufferedImage.TYPE_INT_RGB);

    if(scaledImages.containsKey(newSize) && !processed){

        return scaledImages.get(newSize);

    }else{

        for(int j=0; j<newImage.getHeight(); j++){
            for(int i=0; i<newImage.getWidth(); i++){
                try{
                    Color currentColor = getColorAt((int)(i*divisor), (int)(j*divisor));
                    newImage.setRGB(i, j, currentColor.getRGB());
                }catch(Exception e){
                    Color currentColor = getColorAt((int)((i-1)*divisor), (int)((j-1)*divisor));
                    newImage.setRGB(i, j, currentColor.getRGB());
                }
            }
        }
        scaledImages.put(newSize,newImage);
/         processed = false;
        return newImage;
    }
}
```

Figure 4: Down Sample Algorithm

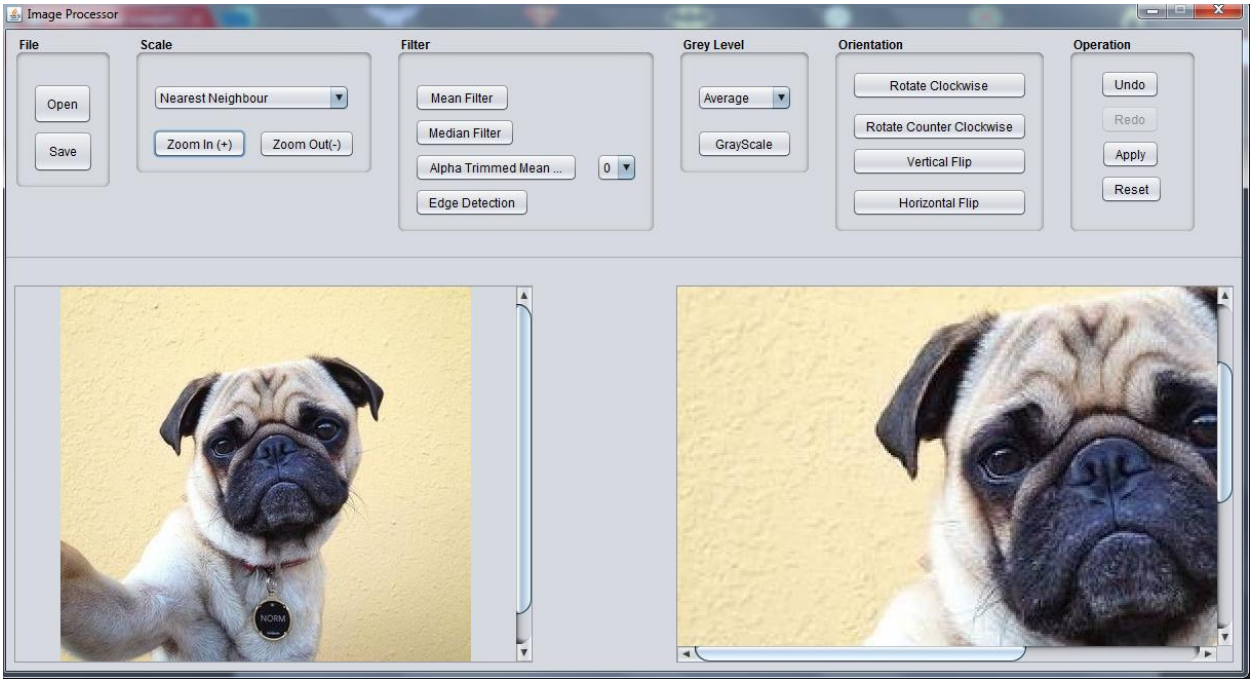

Figure 5: Up Scaled Image

NOISE REDUCTION

Noise is defined as the random occurrences of pixels with deviated pixels values compared to the neighborhood. Neighborhood operations are carried out to correct the pixel values and thus remove the noise present in the digital image.

Three types of filters are implemented in the tool.

Mean Filter

This filter uses the mean value of a defined neighborhood of pixel to correct the noise pixels. For the implementation, “8 neighbors” neighborhood was selected.

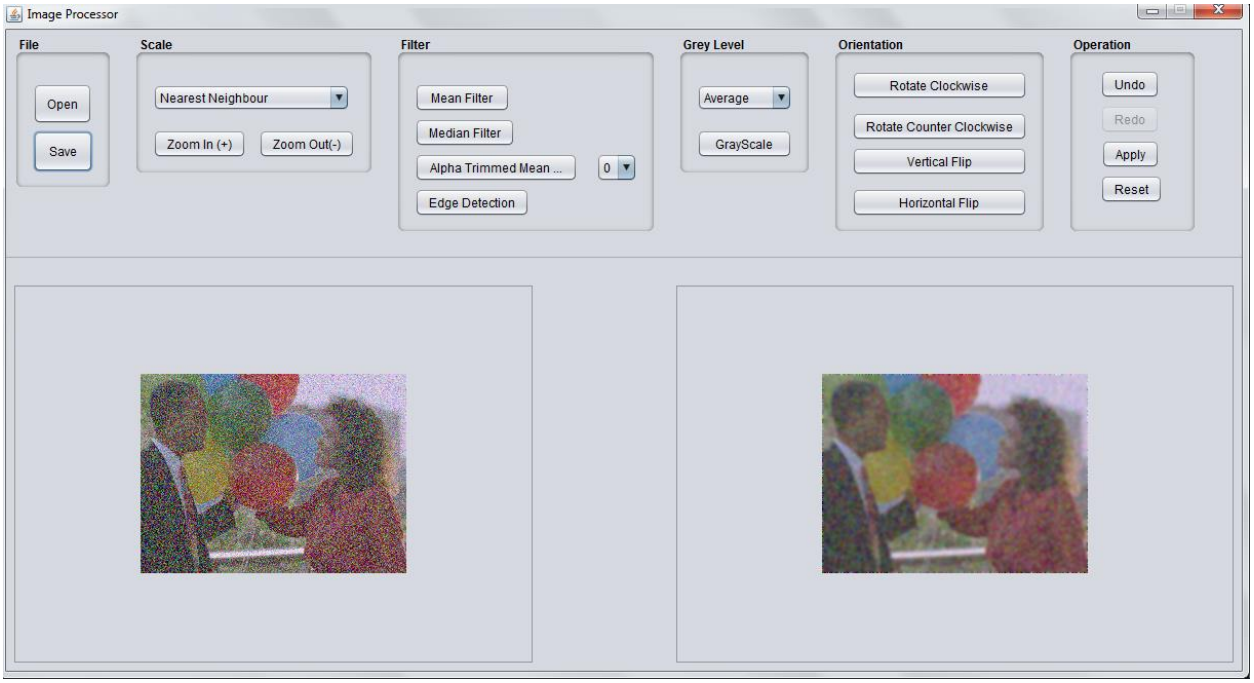

Figure 6: Mean filtered noisy image

Median Filter

Median filter corrects the noise pixels by applying the median pixel value of the selected neighborhood. This filter does not change the value of a pixel to a value which was not present in the original image. Apart from that the blurring which is imminent in Mean Filtering is reduced too.

Alpha Trimmed Mean Filter

This filter is essentially a mixture of Mean and Median filters. After sorting the neighborhood pixel values, this algorithm uses a “p” value to decide the size of the pixel set to calculate the mean value. If the p value is equal to 0, the whole neighborhood is contributed to calculate the mean. Thus the filter acts as a Mean Filter for the p value of 0. On the other hand if the p value is equal to 1, the algorithm only uses the middle pixel in the sorted set to calculate the mean value. Which would return the pixel value of the particular pixel itself. Thus the filter acts as the Median Filter for the p value of 1.

```
public BufferedImage getAlphaTrimFilter(int p){
    /*
     * value of p decides the type of the filter to apply
     * window size fixed and it is 9. N=9
     * */
    int [] size = this.getSize();
    BufferedImage newImage = new BufferedImage((int)(size[0]), (int)(size[1]), BufferedImage.TYPE_INT_RGB);
    processed = true;

    for(int j=0; j<size[1]; j++){
        for(int i=0; i<size[0]; i++){
                //body
            ArrayList <Integer> intensityBuffer = new ArrayList<Integer>();
            HashMap<Integer,Color> colorIntensityMap = new HashMap<>();
            for(int row=0; row<3; row++){
                for(int col=0; col<3; col++){

                    try{
                        Color color = getColorAt(i-1+col, j-1+row);
                        Integer intensity = getColorIntensity(color);

                        colorIntensityMap.put(intensity, color);
                        intensityBuffer.add(intensity);
                        intensityBuffer.sort(null);
                    }
                    catch(ArrayIndexOutOfBoundsException e){

                    }
                }
            }
                int divisor = 9-2*p;

                double red = 0;
                double green = 0;
                double blue = 0;

                if((9-p) <= intensityBuffer.size()){

                    try{
                        for (int t=p; t<9-p; t++){
                            red += colorIntensityMap.get(intensityBuffer.get(t)).getRed();
                            green += colorIntensityMap.get(intensityBuffer.get(t)).getGreen();
                            blue += colorIntensityMap.get(intensityBuffer.get(t)).getBlue();
                        }
                        red = red/divisor;
                        green = green/divisor;
                        blue = blue/divisor;
                        Color newColor = new Color((int)red, (int)green, (int)blue);
                        newImage.setRGB(i, j, newColor.getRGB());
                    }catch(ArrayIndexOutOfBoundsException e){
                        newImage.setRGB(i, j, getColorAt(i, j).getRGB());
                    }
                }else{

                    newImage.setRGB(i, j, getColorAt(i, j).getRGB());

                }
            }
        }
    return newImage;
}
```

Figure 7: Algorithm of Alpha trimmed mean filter

POINT OPERATIONS

Point operations are carried out to modify the intensity values of the pixels independent of the neighboring pixels. Transformations are also a form of point operations. These point operations can be either performed dependent on the location of the pixel or independent of the location of the pixel.

Grayscale

Grayscale value for a given pixel is obtained by taking the average of RGB values of the pixel.

$$Grey\ value = (R\ value + G\ value + B\ value) / 3$$

Three modes of gray scaling is introduced to the tool.

Average Grayscale

Following code snippet shows the implementation of the Average Gray scaling implemented in the tool.

```
public BufferedImage getAverageGrayScale(int[] size, BufferedImage newImage){

    for(int j=0; j<size[1]; j++){
        for(int i=0; i<size[0]; i++ ){
            int avg  = (this.getColorAt(i, j).getBlue() + this.getColorAt(i, j).getGreen() + this.getColorAt(i, j).getRed())/3;
            Color avgColor = new Color(avg, avg, avg);
            newImage.setRGB(i, j, avgColor.getRGB());
        }
    }
    return newImage;
}
```

Figure 8: Average Gray Scaling

Lightness Grayscale

Following code snippet shows the implementation of the Lightness Gray scaling implemented in the tool.

```
public BufferedImage getLightnessGrayScale(int[] size, BufferedImage newImage){

    for(int j=0; j<size[1]; j++){
        for(int i=0; i<size[0]; i++ ){
            Color currentColor = this.getColorAt(i, j);

            //get max and min values
            int max = Math.max(currentColor.getBlue(), Math.max(currentColor.getGreen(), currentColor.getRed()));
            int min = Math.min(currentColor.getBlue(), Math.min(currentColor.getGreen(), currentColor.getRed()));

            int avg = (max + min)/2;
            Color avgColor = new Color(avg, avg, avg);

            newImage.setRGB(i, j, avgColor.getRGB());
        }
    }
    return newImage;

}
```

Figure 9: Lightness Gray Scaling

Luminosity Grayscale

Below is an example of the application of luminosity gray scale. Average value of the pixel is more biased towards the Green component of the RGB value of the pixel in this implementation. Human eye is more sensitive to the green color than the other two components. Thus the filter is more biased towards the green component.

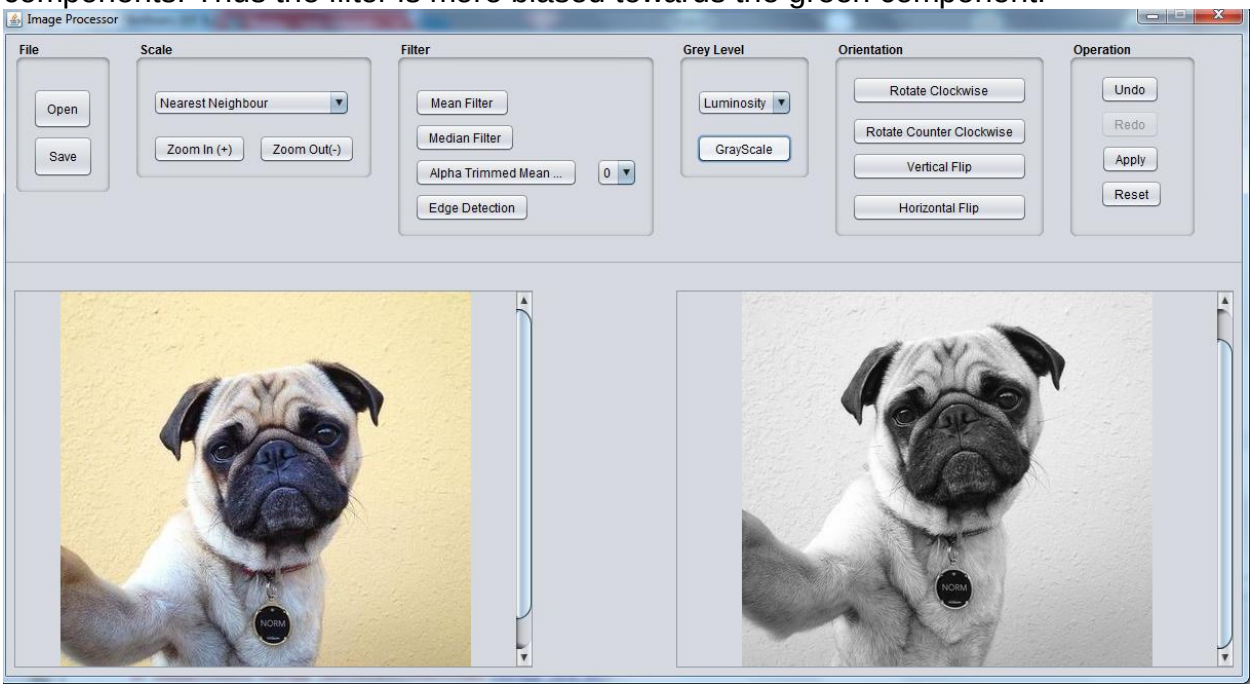

Figure 10: Luminosity Gray Scaling

TRANSFORMATIONS

Transformation is basically changing the orientation of the image. The tool can rotate the image in both clockwise and counter-clockwise direction by 90 degrees at a time. Other than that the tool can flip the image vertically and horizontally.

Flip

Flipping can be done around a vertical axis or a horizontal axis. Flipping an image around a vertical axis is widely known as mirroring.

```
public BufferedImage getFliped(int axis){
    /* axis=0 => horixontal flip*/
    int [] size = this.getSize();
    BufferedImage newImage = new BufferedImage((int)(size[0]), (int)(size[1]), BufferedImage.TYPE_INT_RGB);
    processed = true;

    if(axis==0){
        for (int j=0; j<size[1]; j++){
            for(int i=0; i<size[0]; i++){
                newImage.setRGB(i, size[1]-1-j, getColorAt(i, j).getRGB());
            }
        }
        return newImage;
    }else{
        for (int j=0; j<size[1]; j++){
            for(int i=0; i<size[0]; i++){
                newImage.setRGB(size[0]-1-i, j, getColorAt(i, j).getRGB());
            }
        }
        return newImage;
    }
}
```

Figure 11: Algorithm for flipping.

Rotate

Rotating an image is implemented in both directions clockwise and counter-clockwise.

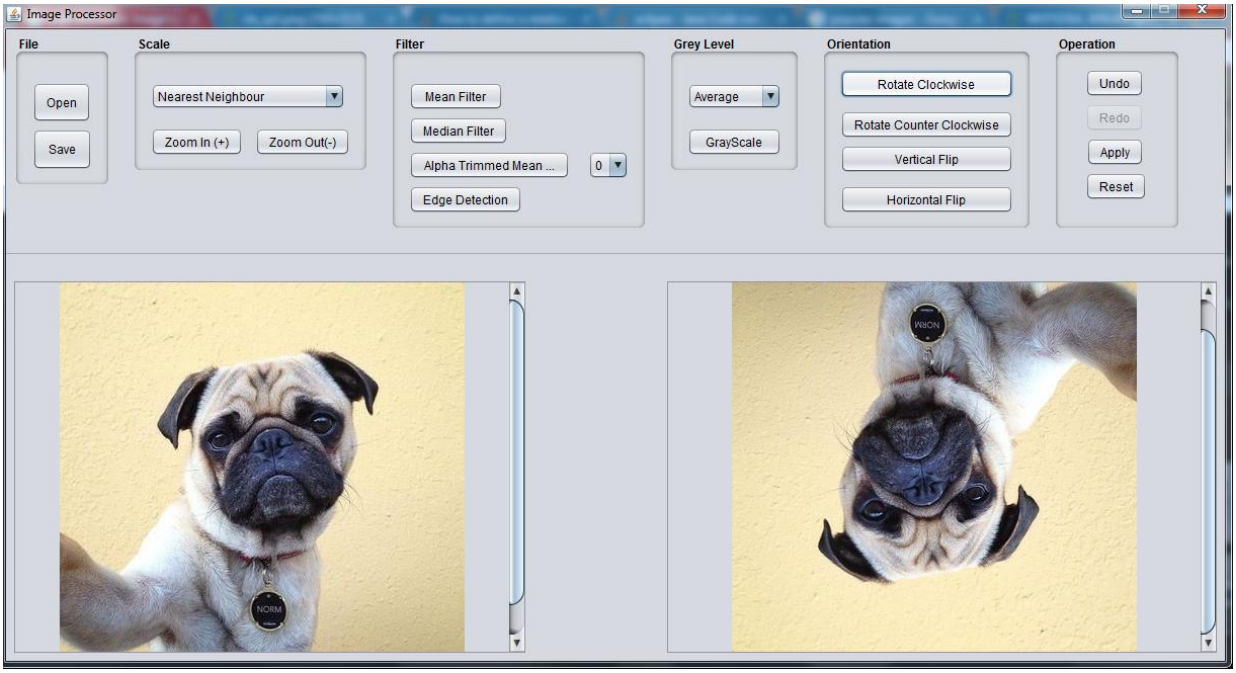

Figure 12: Rotating clockwise 180 degrees

```
public BufferedImage getRotated(int direction){
    /*
     * rotation = 0 clockwise;
     * rotation = 1 counter clockwise;
     * */

    int [] size = this.getSize();
    BufferedImage newImage = new BufferedImage((int)(size[1]), (int)(size[0]), BufferedImage.TYPE_INT_RGB);
    processed = true;

    switch(direction){

        case 0:
            for (int j=0; j<size[1]; j++){
                    for(int i=0; i<size[0]; i++){
                            newImage.setRGB(size[1]-1-j, i, getColorAt(i, j).getRGB());
                    }
            }
            return newImage;

        case 1:
            for (int j=0; j<size[1]; j++){
                        for(int i=0; i<size[0]; i++){
                                newImage.setRGB(j, size[0]-1-i, getColorAt(i, j).getRGB());
                        }
                }
                return newImage;
        default:
            return this.getImage();
    }
}
```

Figure 13: Algorithm for rotation

EDGE DETETCTION

A sudden change or discontinuity in the intensity value of a pixel with regarding to its neighbors is defined as an edge. The gradient of the pixel intensity values along several directions are monitored to identify an edge. Once a sudden change of the gradient beyond a certain pre-defined threshold is detected, an edge is discovered. In the program “Laplacian Filter” is implemented. This filter uses the following mask and can be used to detect both inward and outward edges.

0	−1	0
−1	4	−1
0	−1	0

Figure 14: Laplasian Mask

Following code snippet shows the implementation of edge detection in the tool.

```
public BufferedImage getEdgeDetection(){
    int [] size = this.getSize();
    BufferedImage newImage = new BufferedImage((int)(size[0]), (int)(size[1]), BufferedImage.TYPE_INT_RGB);
    processed = true;

    for(int j=0; j<size[1]; j++){
        for(int i=0; i<size[0]; i++){
            try{
                int[][] EDGE_MASK = new int[][]{new int[]{-1, 0, 1}, new int[]{-2, 0, 2}, new int[]{-1, 0, 1}};
                double colorRed=0;
                double colorGreen=0;
                double colorBlue=0;
                for(int row=0; row<3; row++){
                    for(int col=0; col<3; col++){
                        //X direction convolution process
                        colorRed = colorRed + ((double)(getColorAt(i-1+col, j-1+row).getRed()*EDGE_MASK[row][col]));
                        colorGreen = colorGreen + ((double)(getColorAt(i-1+col, j-1+row).getGreen()*EDGE_MASK[row][col]));
                        colorBlue = colorBlue + ((double)(getColorAt(i-1+col, j-1+row).getBlue()*EDGE_MASK[row][col]));

                        //Y direction convolution process
                        colorRed = colorRed + ((double)(getColorAt(i-1+col, j-1+row).getRed()*EDGE_MASK[2-col][row]));
                        colorGreen = colorGreen + ((double)(getColorAt(i-1+col, j-1+row).getGreen()*EDGE_MASK[2-col][row]));
                        colorBlue = colorBlue + ((double)(getColorAt(i-1+col, j-1+row).getBlue()*EDGE_MASK[2-col][row]));
                    }
                }
                newImage.setRGB(i, j, new Color((int)(colorRed), (int)colorGreen, (int)colorBlue).getRGB());
            }catch(ArrayIndexOutOfBoundsException e){}
            }
    }
    return newImage;
}
```

Figure 15: Algorithm for edge detection.

CONCLUSION

The theories of image processing can be effectively implemented to process the digital images efficiently. The algorithms implemented in the tool are primitive. Further modifications can be done and several aspects of the tool can be improved to produce images of better quality. Furthermore few operations which are not included on the tool can be implemented in the future.