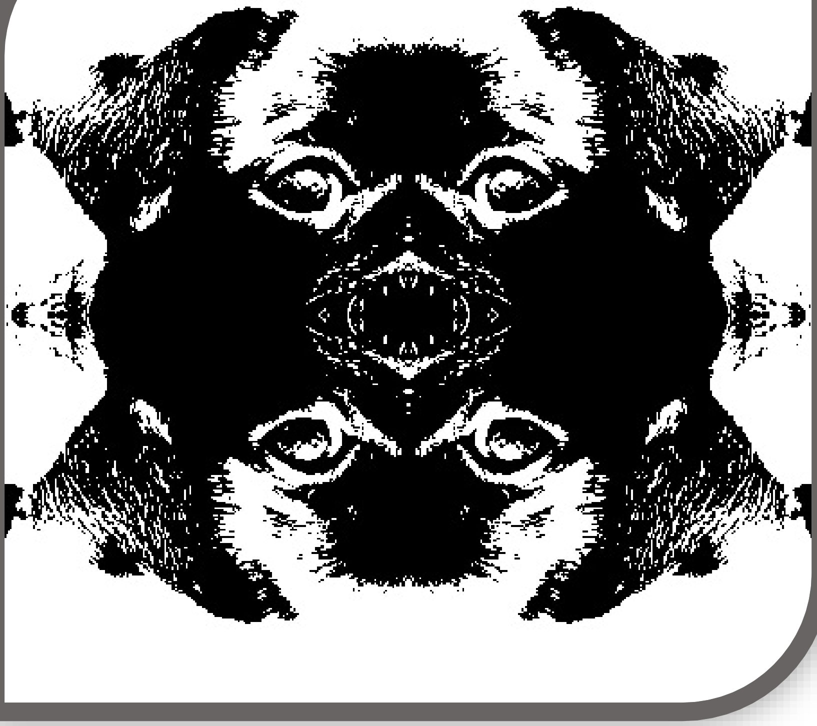
Image Manipulation with Java



Christian Goodman, Sam Carroll, Dennis Smith, Paul, Shuwa
Discrete Structures II

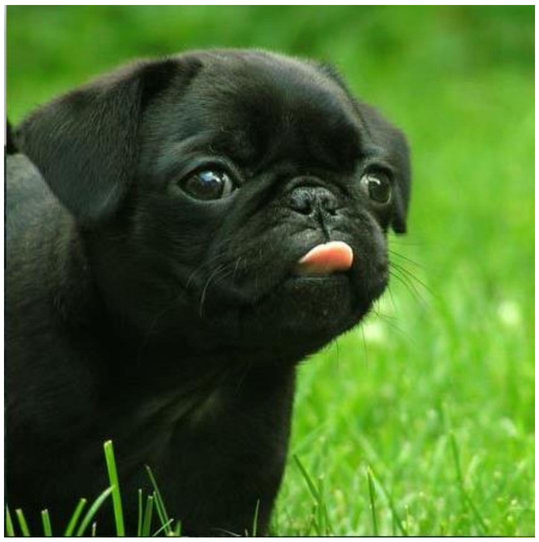
Contents

Executive Summary

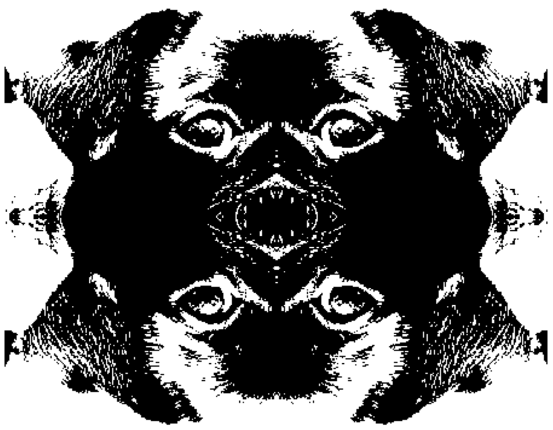
Images are everywhere and they shape how we interpret things. For example, if you see the image below:

i’m lovin’ it

You will automatically associate this image with the McDonald’s franchise. This is a pretty simple image that has not undergone any significant manipulation. There is only one color and no real background. Think now about an image with some detailed features and perhaps some overlapping colors and textures. For our purposes we will use the following image:



In this image you see a cute little puppy but this image can be used to create so much more. There are more possibilities here than that first meet the eye. At first glance you think this is a cute puppy and it would be awesome if you owned a puppy like this. What if we altered the image just a bit? Would you have the same response based on how we manipulate the image? What are your first thoughts if you encounter this image below?



Quite frightening isn't it? This is the same image only with a few manipulations. If you look closely you can see bits of the original image. The only thing done here was putting the image in grayscale and reflecting the horizontal and vertical axis's.

In this project, we hoped to explore different ways to manipulate images based on methods we could code using the Java programming language. We were able to implement a number of effects using sliders and buttons on our GUI.

Project Description

In this project we start out with and image of a puppy. This image was selected a random but our code is scalable and could accept any image. With this default image selected, we apply different effects to manipulate the image. Our goal was to see how many different effects could be completed and combined and what results would they yield. The effects we used are as follows:

- Manipulation of Red, Green, and Blue pixel values
- Grayscale
- Inverted Grayscale
- Heat Map
- Heat Map Variation
- Heat Map Outline
- Random Map
- Horizontal Reflection
- Vertical Reflection
- Color Swap
- Color Swap Variation
- Raster Manipulation
- Raster Manipulation Variation

By adding these features to our project we should be able to produce some very interesting results. We will see what we come up with later.

Background

Image manipulation is the art of transforming an image to convey what you *want*, rather than what the original image may have shown. Images in Java are defined by the java.awt.image.Image interface, which provides a basic model for controlling an image, but the real meat of the Java Image is the java.awt.image.BufferedImage. A BufferedImage is an accessible buffer of image data, essentially pixels, and their RGB colors. The BufferedImage provides a powerful way to manipulate the Image data. A BufferedImage object is made up of two parts a ColorModel object and a Raster object.

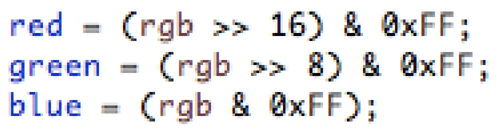
Probably the most common and most used image operation is loading an image. Loading an image is fairly straight forward, and there are many ways to do it, so I will show you one of the simplest and quickest ways, using the javax.imageio.ImageIO class. Basically what we want to do is load all the bytes from some image file, into a BufferedImage so that we can display it. This can be accomplished through the use of ImageIO. Of course loading an image is nearly completely useless unless you can display it on screen, so next ill explain how to *render*, or draw an image. In this case we will use an uploaded image from a URL. Next we set the width and the height and then display the image.

We have our main java file that drives the program. We have a painting file where all the manipulations are housed. Finally we have a view file that contains our declarations for how the GUI is set up, labels, buttons sliders, etc... All of the magic will happen in the paint file.

The manipulation functions are quite simple to understand. The data will be fetched from the image. This data consists of the Red, Green, and Blue (RGB) content of the image. These are the building blocks of all images. These building blocks will be manipulated in different ways throughout our various functions. Let's examine the first function.

Our first function, drawGrayscale(), will normalize the colors so that the image is draw in grayscale. This is done by examining the horizontal pixels and the vertical pixels using a conditional for loop and from this step we get the RGB of a given x and y coordinate image.

In the demo we use the 0xFF declaration. See figure below:



```
red = (rgb >> 16) & 0xFF;
green = (rgb >> 8) & 0xFF;
blue = (rgb & 0xFF);
```

Notice the 0xFF part of the code. 0xff is the hexadecimal number FF which has a integer value of 255. And the binary representation of FF is 00000000000000000000000011111111 (under the 32-bit integer).

The “&” operator performs a bitwise AND operation. In a nutshell, “& 0xff” effectively masks the variable so it leaves only the value in the last 8 bits, and ignores all the rest of the bits. In our case we are trying to transform color values from a particular format to standard RGB values (which is 8 bits long). Our ultimate use for this procedure is to normalize the RGB values. To accomplish this we will divide the sum of these values by three giving us an average value by which we ‘normalize’. We initialize a grayLevel function to be used as the amount of gray applied when we invoke a slider action. Then we need to change the original RGB values of a pixel to the new RGB we have created though this normalization process and application of the level of gray.

The Figure above also demonstrates the selected RGB data that will be normalized in the next section. We select this data then divide the sum of the RGB by 3 (red, green, blue) to normalize them. The variable named ‘value’ is used for storing the amount of gray applied and passed through a slider in our GUI. See the figure below:

```
int grayLevel = (red + green + blue) / value;
```

The variable ‘grayLevel’ will store the normalized amount of RGB data. Then we will set a new integer with the normalized RGB values. This changes the original RGB of a pixel to the new RGB created. Then we just repeat. The figure below demonstrates this process:

```
int gray = (grayLevel << 16) + (grayLevel << 8) + grayLevel;

image.setRGB(i, j, gray);
```

Our next function ‘drawGrayscaleInverted’ does the same thing as the grayscale function above. In this function, instead of dividing the value we use another arithmetic operation to get some interesting results. We introduce multiplication in the function to see what results we will encounter. See below:

```
int grayLevel = (red + green + blue) * value;

int gray = (grayLevel << 16) + (grayLevel << 8) + grayLevel;
```

The exact same function above only instead of dividing the RGB values by the ‘value’ variable, we are multiplying them by the ‘value’ variable.

Experimental procedure

The procedure is very simple. There are not complex steps involved after the code is completed. The procedure is as follows:

1. Select an image
2. Load into our program
3. Apply different values from the sliders
4. Click buttons
5. Note the effect
6. Record unique combinations
7. Further develop

Select an image:
They we our project is set up, the image would have to be selected and added as a supporting file in the code. You would have to notify the programmer to insert the image.

Load into our program:
The programmer takes the image converts to an appropriate format if necessary, and adds this image to the supporting files in Java.

Apply different slider values:
Manipulate the sliders in the GUI window. The slider values can be between 0 for no effect by the slider to 255 full effect of the slider.

Click buttons:
Select the buttons in any order. See what effects happen once the buttons have been selected.

Note the effect:
Observe what happens.

Record unique combinations:
Write down any notable combinations for the transitions of your images.

Further develop:
Make note of what can improve the performance and output by the program.

GUI Buttons

<u>Heat Map</u>

A heat map is a graphical representation of data where the individual values contained in a matrix are represented as colors. The heat map button draws a pseudo heat map of color by summing all the RGB values. If the total values are greater than 150, then we set the color to white. If the values fall below 150 then we set the color to black. White is represented as a value of 255 and black is represented as a value of 0. See the code below:

```
public void drawHeatMap(BufferedImage image)
   {
      for (int i = 0; i < image.getWidth(); i++) {

            for (int j = 0; j < image.getHeight(); j++) {
            color = new Color(image.getRGB(i, j));
            red = color.getRed();
            green = color.getGreen();
            blue = color.getBlue();

            if (red+green+blue > 150)
            {
               if (heatMapped == true)
               {
                  red = 255;
                  blue = 255;
                  green = 255;
               }

               if (heatMapped == false)
               {
                  red = 0;
                  blue = 0;
                  green = 0;
               }
            }
            else
            {
               if(heatMapped == true)
               {
                  red = 0;
                  blue = 0;
                  green = 0;
               }

               if (heatMapped == false)
               {
                  red = 255;
                  blue = 255;
                  green = 255;
               }


            }
```

```
              Color newColor = new Color(red, green, blue,
color.getAlpha());
              rgb = newColor.getRGB();
                  image.setRGB(i, j, rgb);
              }
         }
         heatMapped = !heatMapped;
    }
```

<u>Heat Map Variable</u>
This button uses the same concept as the above stated heat map. This new heat map button varies by inversing the map color. White becomes black and vice versa. See the code below:

```
public void drawHeatMap2(BufferedImage image)
    {
        for (int i = 0; i < image.getWidth(); i++) {

              for (int j = 0; j < image.getHeight(); j++) {
              color = new Color(image.getRGB(i, j));
              red = color.getRed();
              green = color.getGreen();
              blue = color.getBlue();

              //-- "Hot Points"
              if (red+green+blue > 150)
              {
                 heatMapped = !heatMapped;

                 if (heatMapped == true)
                 {
                    red = 255;
                    blue = 255;
                    green = 255;
                 }

                 if (heatMapped == false)
                 {
                    red = 0;
                    blue = 0;
                    green = 0;
                 }
              }
              else
              {
                 if(heatMapped == true)
                 {
                    red = 0;
                    blue = 0;
                    green = 0;
                 }

                 if (heatMapped == false)
```

```
                {
                   red = 255;
                   blue = 255;
                   green = 255;
                }


             }

             Color newColor = new Color(red, green, blue,
color.getAlpha());
             rgb = newColor.getRGB();
                image.setRGB(i, j, rgb);
             }
         }
     }
```

<u>Heat Outline</u>

This button has similar properties as the Heat Map Variable button. The modifications made under this function cause the edges of an image to stand out. By inversing the heat map's color in each column, the edges of a given image will appear different. See code below:

```
public void drawHeatMapOutline(BufferedImage image)
    {
       for (int i = 0; i < image.getWidth(); i++) {
          heatMapped = !heatMapped;
             for (int j = 0; j < image.getHeight(); j++) {
             color = new Color(image.getRGB(i, j));
             red = color.getRed();
             green = color.getGreen();
             blue = color.getBlue();


             if (red+green+blue > 150)
             {

                if (heatMapped == true)
                {
                   red = 255;
                   blue = 255;
                   green = 255;
                }

                if (heatMapped == false)
                {
                   red = 0;
                   blue = 0;
                   green = 0;
                }
             }
             else
             {
                if(heatMapped == true)
                {
                   red = 0;
                   blue = 0;
                   green = 0;
                }

                if (heatMapped == false)
                {
                   red = 255;
                   blue = 255;
                   green = 255;
                }


             }

             Color newColor = new Color(red, green, blue,
color.getAlpha());
             rgb = newColor.getRGB();
                image.setRGB(i, j, rgb);
             }
        }
    }
```

Random Map
This functions loops each pixel and assigns it a random red, green, or blue value between 0 and 255. See the code below:

```
public void drawRandomMap(BufferedImage image)
    {
        System.out.println("drew heatmap");
        for (int i = 0; i < image.getWidth(); i++) {
             for (int j = 0; j < image.getHeight(); j++) {
             Random rand = new Random();
             color = new Color(image.getRGB(i, j));

             red = rand.nextInt((255 - 0) + 1) + 0;
             blue = rand.nextInt((255 - 0) + 1) + 0;
             green = rand.nextInt((255 - 0) + 1) + 0;

             Color newColor = new Color(red, green, blue,
color.getAlpha());
             rgb = newColor.getRGB();
                image.setRGB(i, j, rgb);
             }
        }
    }
```

Horizontal Reflection
This is a simple function that reflects the image on its horizontal axis. First we get the width, then the height, lastly get the RGB values. We display the default image and right on the center of the horizontal axis, we display the mirrored image. See the code below:

```
public void drawHorizontalReflect(BufferedImage image)
   {
      int w = image.getWidth() -1;
      int h = image.getHeight() -1;

      for (int i = 0; i < image.getWidth(); i++) {
           for (int j = 0; j < image.getHeight(); j++) {
           color = new Color(image.getRGB(i, j));

           red = color.getRed();
           green = color.getGreen();
           blue = color.getBlue();

           Color newColor = new Color(red, green, blue,
color.getAlpha());
           rgb = newColor.getRGB();
               image.setRGB(w-i, j, rgb); //Mirroring done here
               image.setRGB(i, j, rgb);
           }
      }
   }
```

Vertical Reflection

Much like the horizontal function we repeat the same process but we mirror the image over its vertical axis. As before, we need the width, height, and RGB values. Once these values are collected, the default image is displayed and the mirrored image is also displayed mirrored from the vertical axis. See the code below:

```
public void drawVerticalReflect(BufferedImage image)
   {
      int w = image.getWidth() -1;
      int h = image.getHeight() -1;

      for (int i = 0; i < image.getWidth(); i++) {
           for (int j = 0; j < image.getHeight(); j++) {
           color = new Color(image.getRGB(i, j));

           red = color.getRed();
           green = color.getGreen();
           blue = color.getBlue();

           Color newColor = new Color(red, green, blue,
color.getAlpha());
           rgb = newColor.getRGB();
               image.setRGB(i, h-j, rgb); //Mirroring done here
               image.setRGB(i, j, rgb);

           }
      }
   }
```

Color Swap

The color swap button takes the values collected for RGB and swaps them. Red becomes blue, green becomes red, and blue becomes green. See the code below:

```
public void drawColorSwap(BufferedImage image)
   {
      System.out.println("drew heatmap");
      int w = image.getWidth() -1;
      int h = image.getHeight() -1;

      for (int i = 0; i < image.getWidth(); i++) {
           for (int j = 0; j < image.getHeight(); j++) {
           color = new Color(image.getRGB(i, j));

           red = color.getRed();
           green = color.getGreen();
           blue = color.getBlue();


           Color newColor = new Color(blue, red, green,
color.getAlpha());
           rgb = newColor.getRGB();
               image.setRGB(i, j, rgb);

           }
      }
   }
```

Color Swap 2

The color swap 2 button is similar to above but instead of just the color, this function focuses on the pixel. In doing this, red pixels becomes blue pixels, green pixels become red pixels, and blue pixels become red pixels. Our default image has a green background. When this filter is applied it is interesting to see how the green background now becomes red as can be seen in Figure 9 of the image appendix. Further application of the color swap 2 button will change this red back ground into a blue one as demonstrated in Figure 10 of the image appendix. See the code below:

```
public void drawColorSwap2(BufferedImage image)
   {
      // -- loop each pixel and modify the RGB --
      for (int i = 0; i < image.getWidth(); i++) {
           for (int j = 0; j < image.getHeight(); j++) {
           color = new Color(image.getRGB(i, j));

           blue = color.getRed();
           red = color.getGreen();
           green = color.getRed();


           Color newColor = new Color(red, green, blue,
color.getAlpha());
           rgb = newColor.getRGB();
               image.setRGB(i, j, rgb);
```

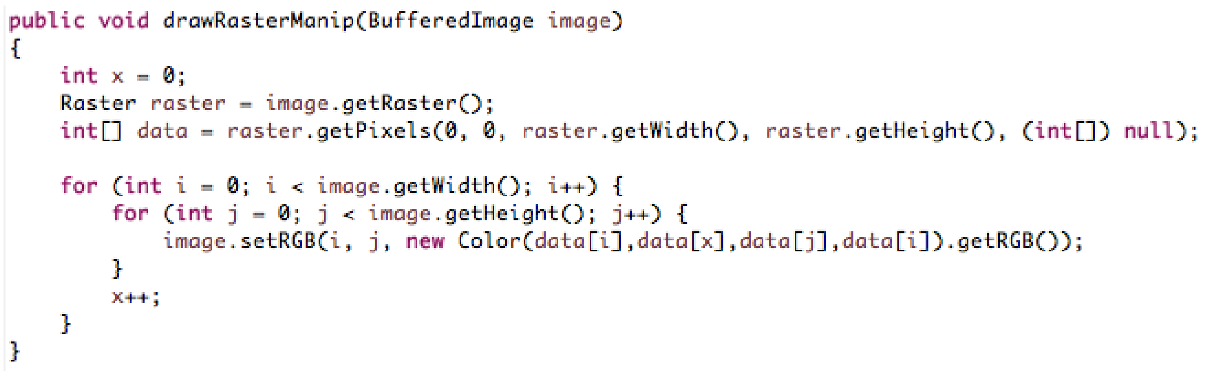
```
                }
            }
        }
```

Raster Manipulation 1

Before we can explain what the button does, let's examine what a Raster is in detail. A Raster defines values for pixels occupying a particular rectangular area of the plane, not necessarily including (0, 0).

As the programmer, it is our responsibility to avoid accessing such pixels. Although a Raster may live anywhere in the plane and therefore contains a translation factor that allows pixel locations to be mapped between the Raster's coordinate system and that of the default image.

In short, we have used the Raster Manipulation to manipulate the default image to product a new effect. It is similar to other functions where we need the width, height, but now we also need an integer array for storing, and the pixel coordinates that we are writing our changes to. Figure 13 illustrates what effect happens when applied to the default image. There is no visible evidence of the default image after this filter has been applied. This is quite interesting to note. Note the figure below:



```
public void drawRasterManip(BufferedImage image)
{
    int x = 0;
    Raster raster = image.getRaster();
    int[] data = raster.getPixels(0, 0, raster.getWidth(), raster.getHeight(), (int[]) null);

    for (int i = 0; i < image.getWidth(); i++) {
        for (int j = 0; j < image.getHeight(); j++) {
            image.setRGB(i, j, new Color(data[i],data[x],data[j],data[i]).getRGB());
        }
        x++;
    }
}
```

The getPixles method returns an int array containing all samples for a rectangle of pixels, one sample per array element. An ArrayIndexOutOfBoundsException may be thrown if the coordinates are not in bounds. However, explicit bounds checking is not guaranteed.

Parameters:

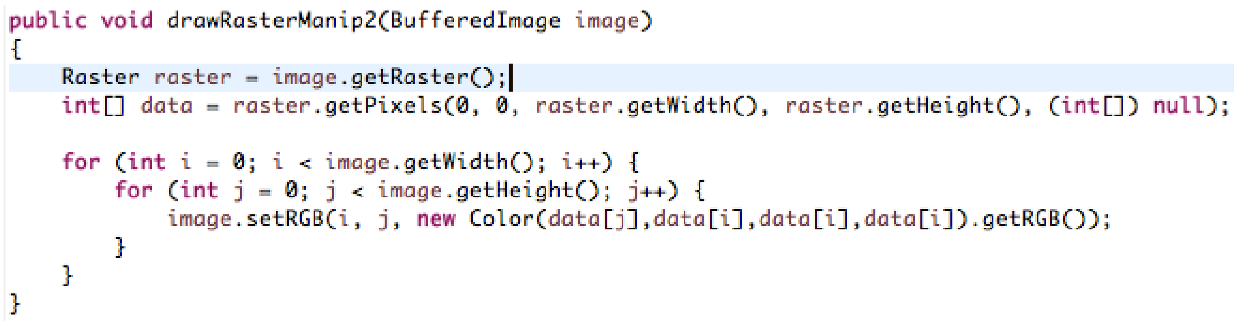
x - The X coordinate of the upper-left pixel location
y - The Y coordinate of the upper-left pixel location
w - Width of the pixel rectangle
h - Height of the pixel rectangle

Returns:

the samples for the specified rectangle of pixels.

Raster Manipulation 2

This manipulation follows the same concept as above. There is only a slight modification. The effect is changing a small piece of the code from above. Everything is the same only that we are not incrementing the "x" variable at the end of the block. See the figure below:



```
public void drawRasterManip2(BufferedImage image)
{
    Raster raster = image.getRaster();
    int[] data = raster.getPixels(0, 0, raster.getWidth(), raster.getHeight(), (int[]) null);

    for (int i = 0; i < image.getWidth(); i++) {
        for (int j = 0; j < image.getHeight(); j++) {
            image.setRGB(i, j, new Color(data[j],data[i],data[i],data[i]).getRGB());
        }
    }
}
```

This very subtle change has interesting results as can be seen in Figure 14 of the image appendix.

drawNew

The new function removes the effects imposed by the sliders. If any effects from the sliders are present and you wish them removed, you only need to press the new button for them to be removed. See the code below:

```
public void drawNew(BufferedImage image)
   {
      // -- loop each pixel and modify the RGB --
      for (int i = 0; i < image.getWidth(); i++) {
             for (int j = 0; j < image.getHeight(); j++) {
             color = new Color(image.getRGB(i, j));


             blue = color.getRed();
             red = color.getGreen();
             green = color.getRed();


             Color newColor = new Color(red, green, blue,
color.getAlpha());
             rgb = newColor.getRGB();

             image.setRGB(i, j, rgb);
             }
      }
   }
```

Reset

The reset button restores the default image to its original condition. If any filters or effects have been applied, the reset button will remove these and display the default image. See the code below:

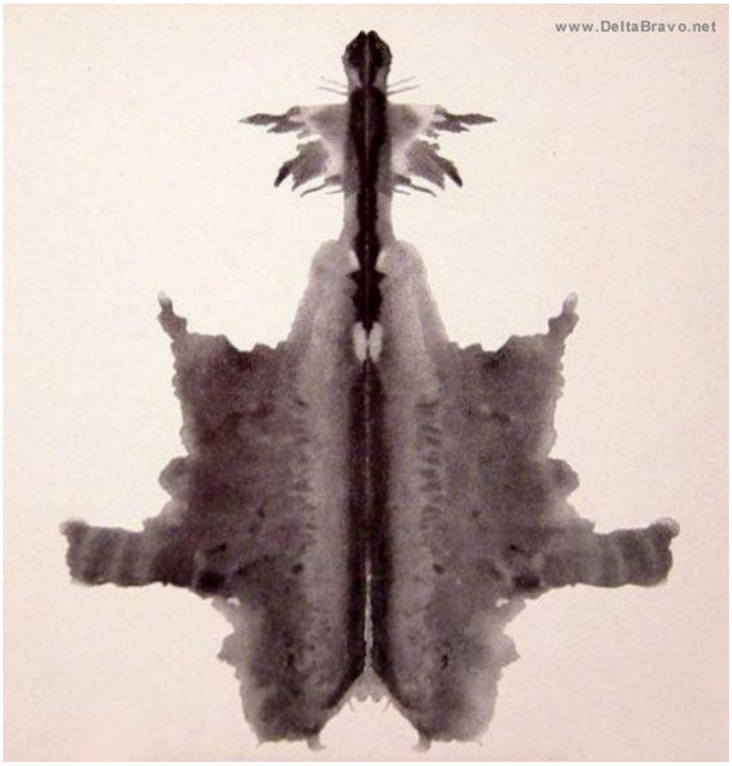
```
public void drawReset()
      {
         for (int i = 0; i < originalImage.getWidth(); i++) {
                for (int j = 0; j < originalImage.getHeight(); j++)
{

                int rgb = originalImage.getRGB(i, j);
                    image.setRGB(i, j, rgb);
```

```
                                        }
                                }
                        }
                }
```

Analysis and Conclusions

This project started as just an image editor. The initial approach was to manipulate the image from its default condition to the current condition. For example, the default image to a heat map image. Having done further analysis, the default image can be manipulated a number of times more creating some very interesting results. It would be interesting to know how many combinations of this image would produce interesting results. If you notice in figure one in the figure appendix, it is an image of a puppy.

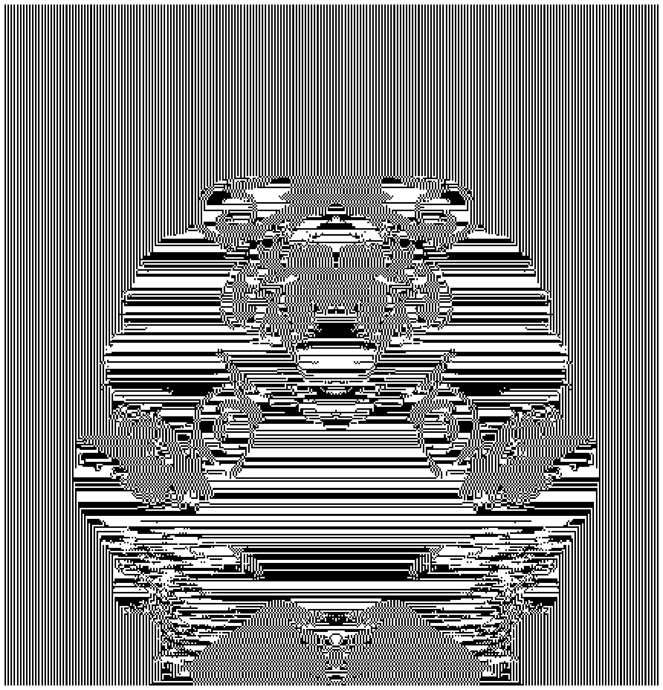
It is further interesting to note how these images could be used in science. Some of the images seem consistent with those of the Rorschach ink blot test. For example, see the image below:



According to the ink blot test this is a card with black ink showing an amorphous "splat" shape. This one can be hard to see anything in. Occasionally described as a foreshortened view of a person with their arms outstretched.

Possible Sexual Imagery: The head of the male sex organ (the portion at the top of the card) or alternately, a female sex organ (middle and bottom part of the card).

This is an image from our manipulation:



This image seems to favor the previous image some. With a few more filters and manipulations we might be able to replicate these images more accurately.

Going Forward

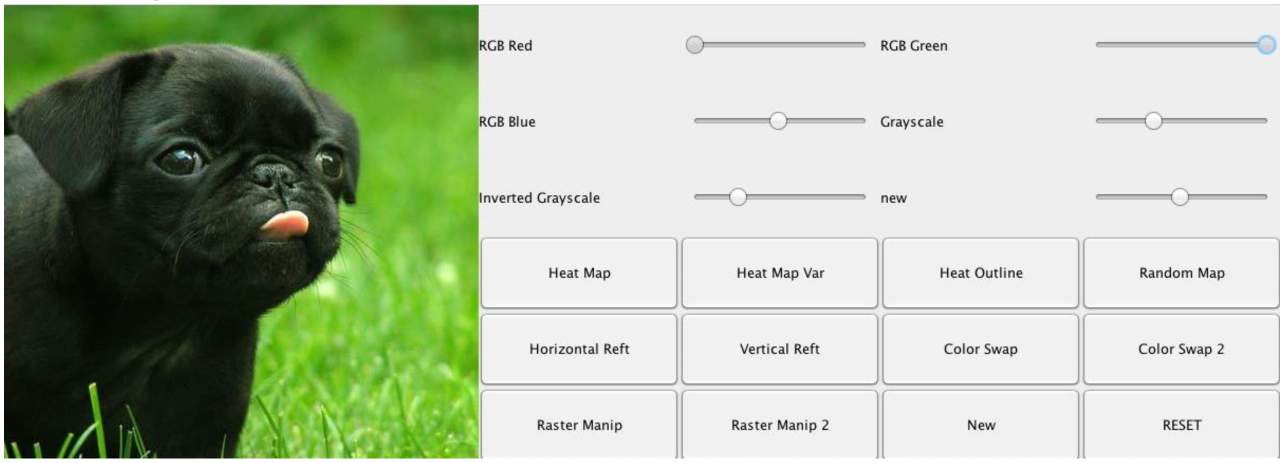
It is already interesting seeing how the effects created affect a given image. This image is predetermined. What we would like to implement is a way to upload images from a url. This enables this project to be utilized with any type of image.

It would be quite interesting to see what other effects could be created. Perhaps using a brightness, opacity, sharpness, etc…. Combining these in different combinations to see how the effects could affect the default image.

The introduction of movement and action with the editor effects might be interesting. Perhaps the changing of effect while in motion. This could possible develop into a video creator by utilizing several images through different transitions and manipulations creating a storyline.

This project may not solve or hinge on the heels of some scientific enigma, but it is fun to see what you can do with a few lines of code and making slight variations to produce big changes.

GUI Image



The sliders are located in the upper portion of the interface and the buttons are located in the lower porting of the interface. The default image is the picture of the puppy on the left.

Image Appendix

Figure 0: Default Image

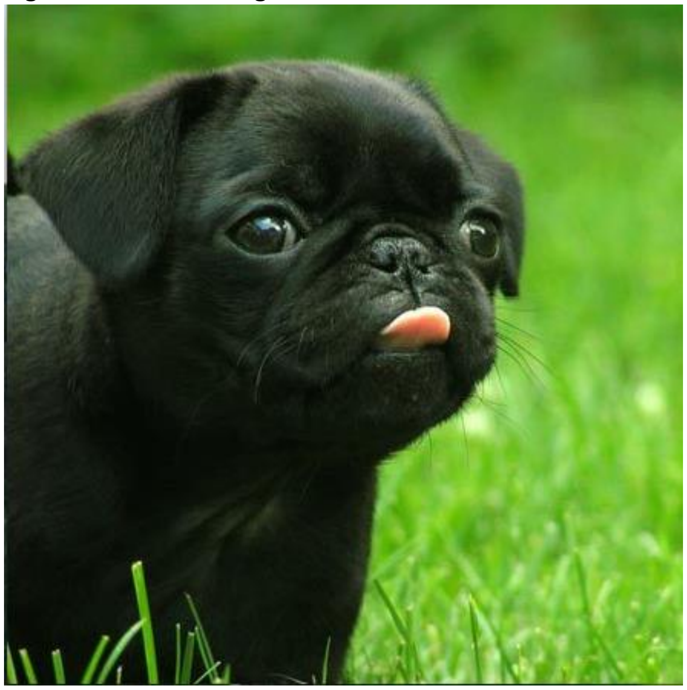


Figure 1: Image with normal Heat Map applied

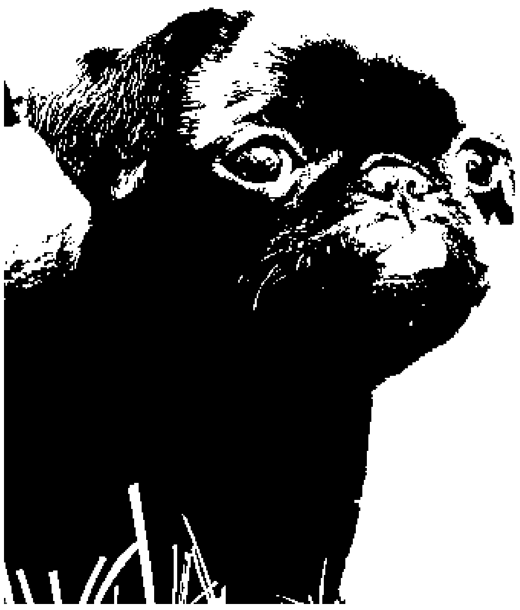


Figure 2: Heat Map Variable

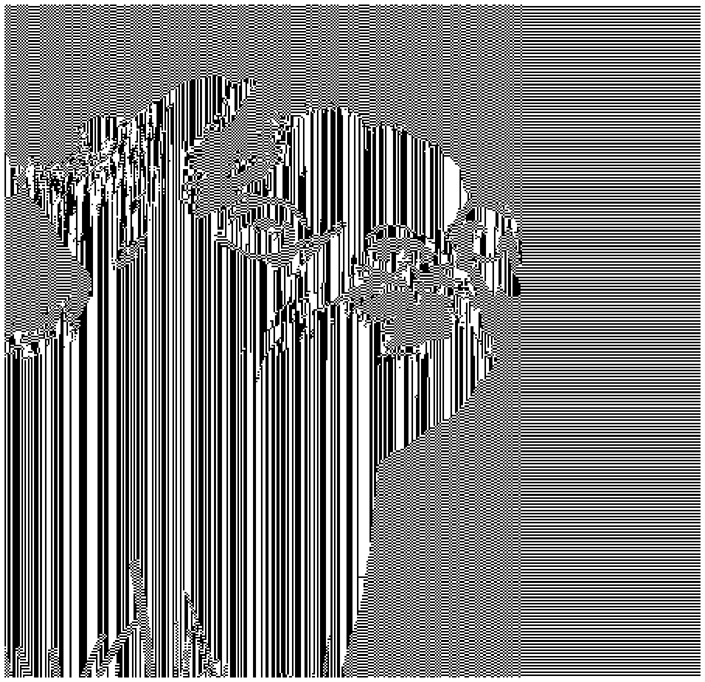


Figure 3: Heat Map Outline

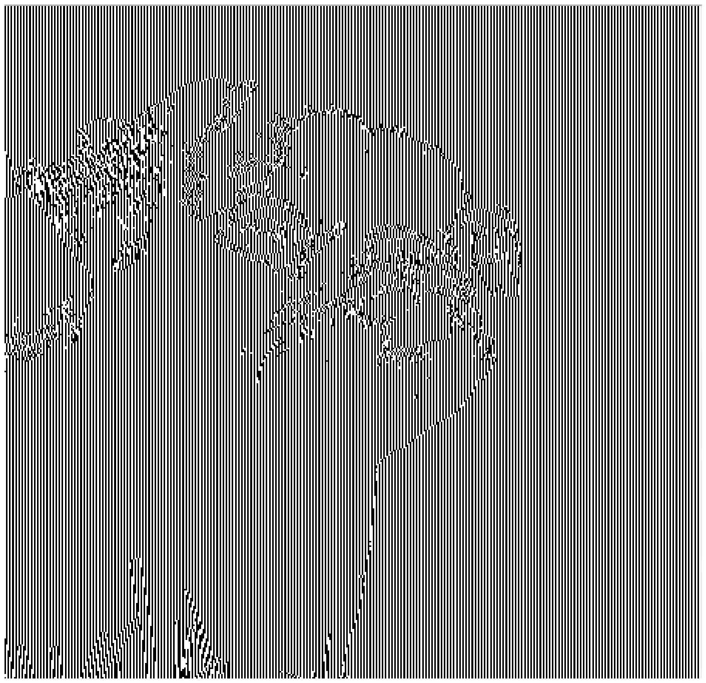


Figure 4: Random Map

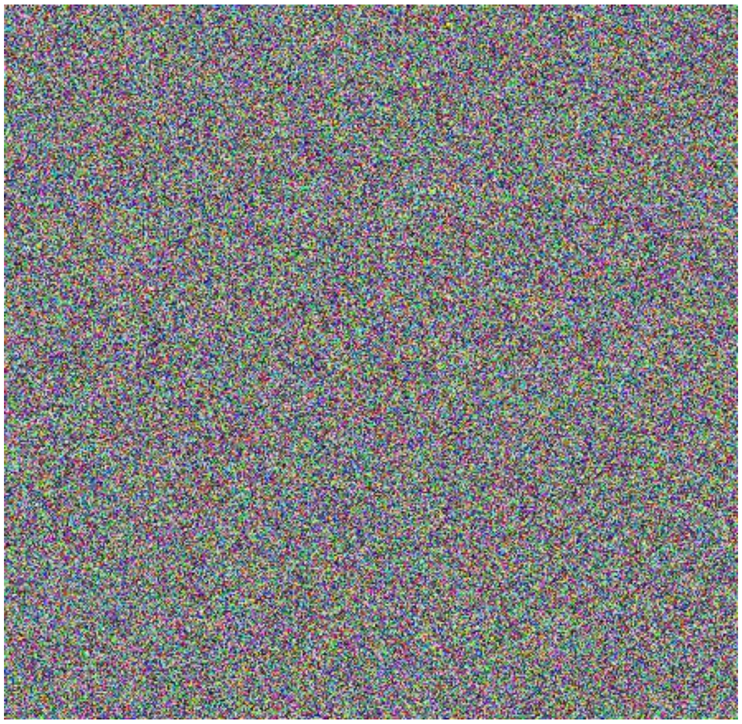


Figure 5: Horizontal Reflection

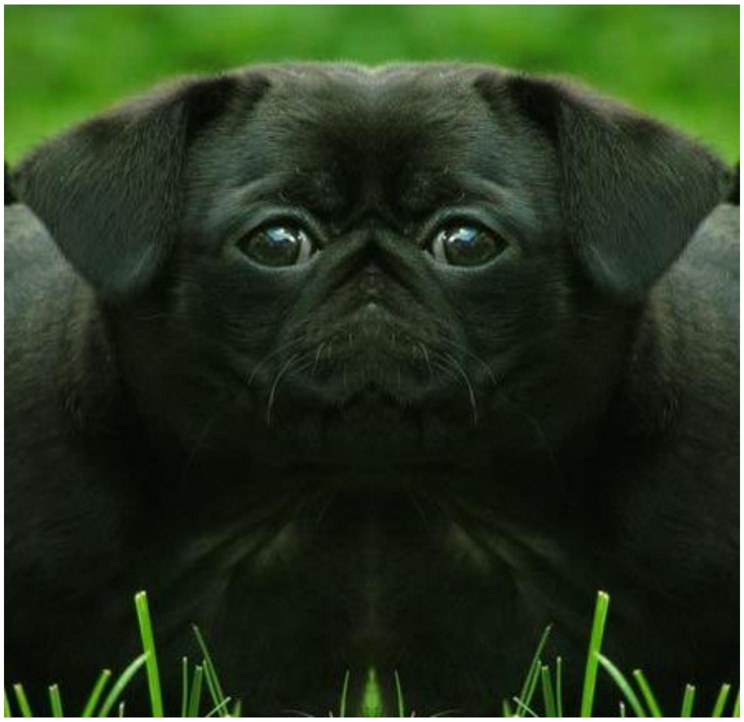


Figure 6: Vertical Reflection

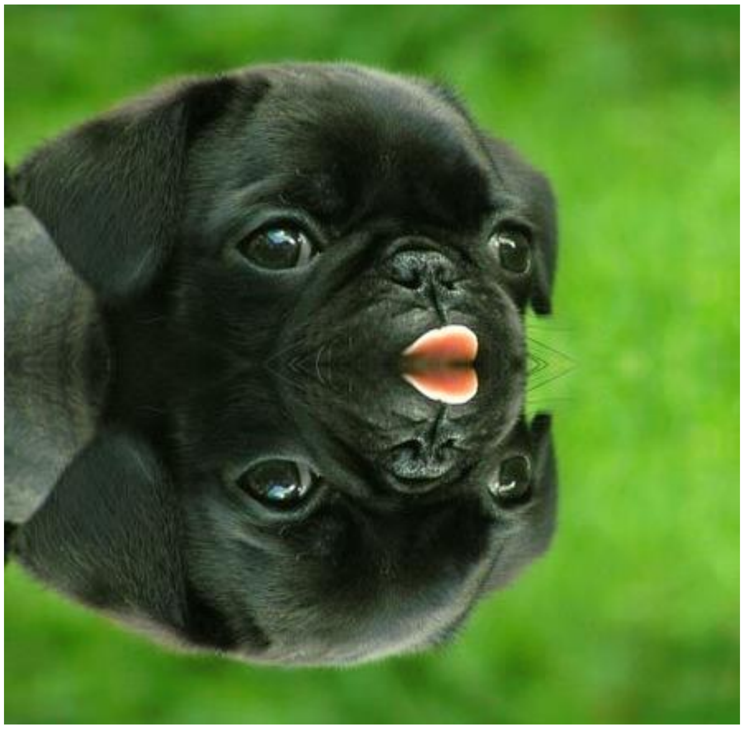


Figure 7: Color Swap (Red)

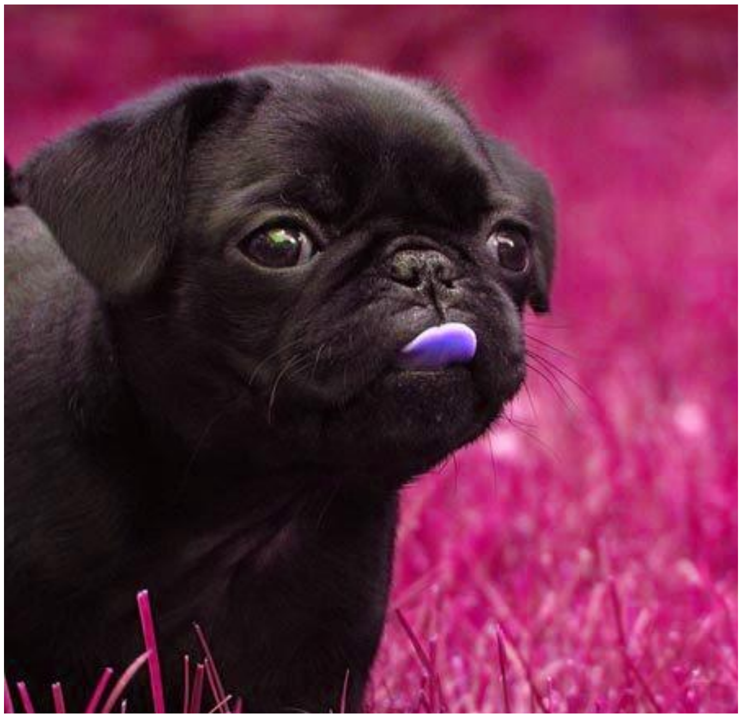


Figure 8: Color Swap (Blue)

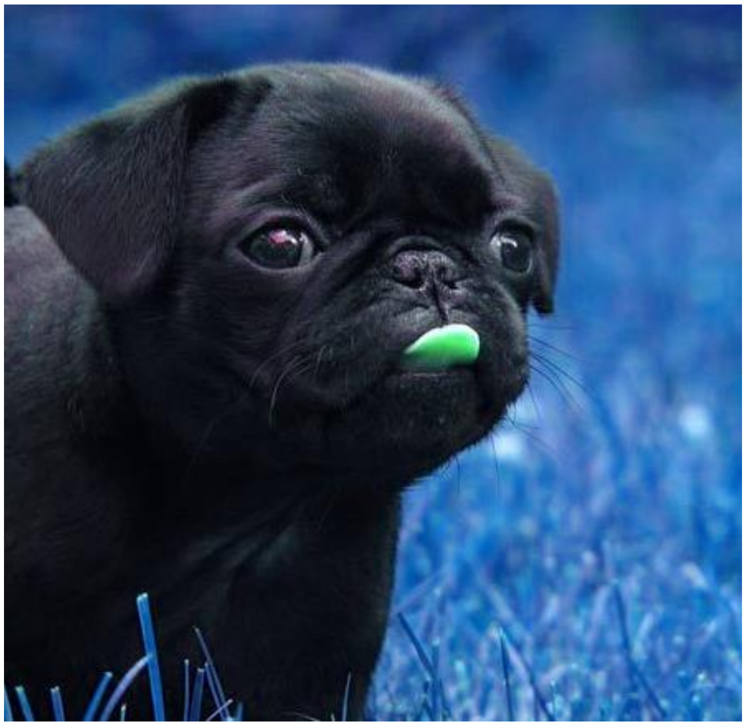


Figure 9: Color Swap 2 (Red)

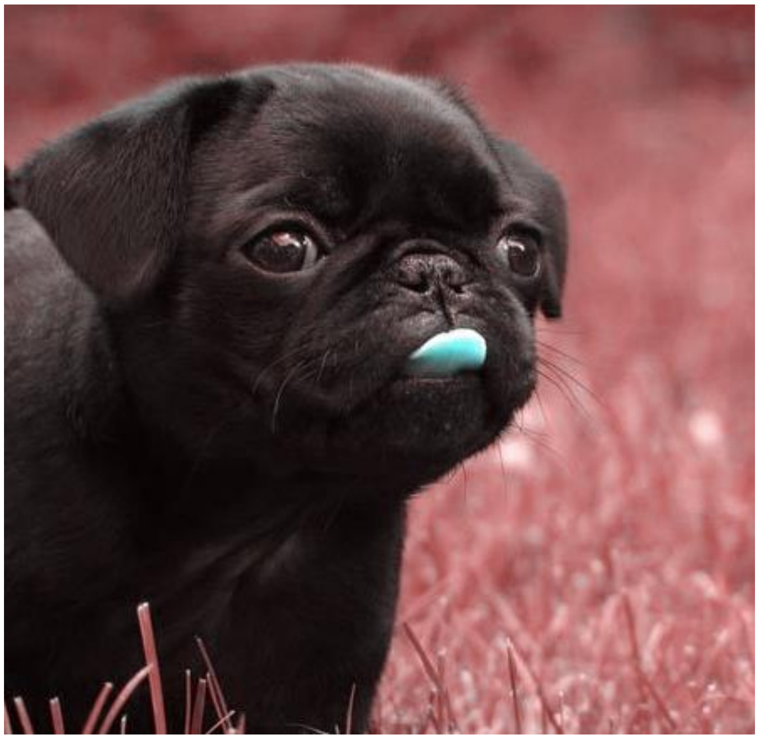


Figure 10: Color Swap 2 (Blue)

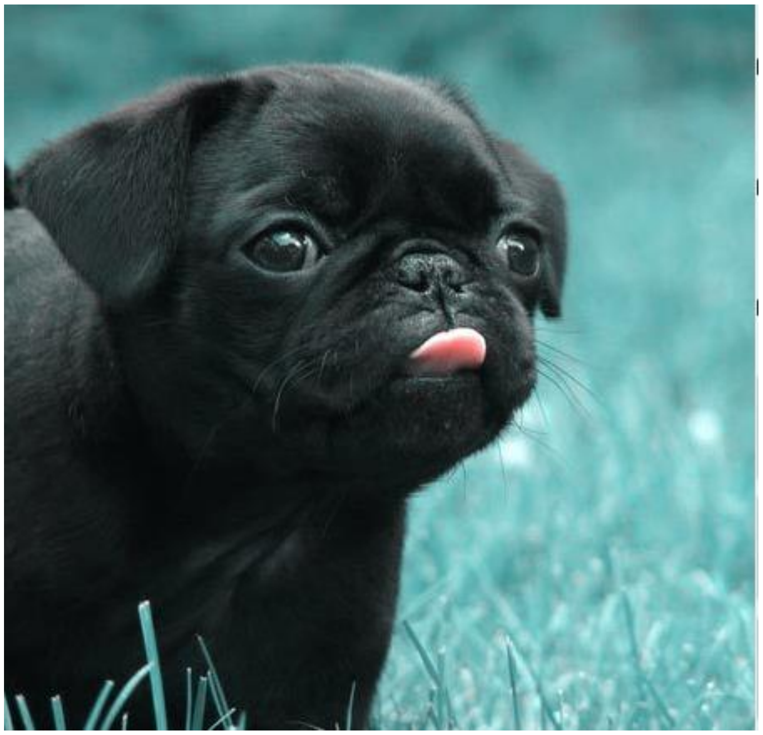


Figure 11: Color Swap 1 then Color Swap 2

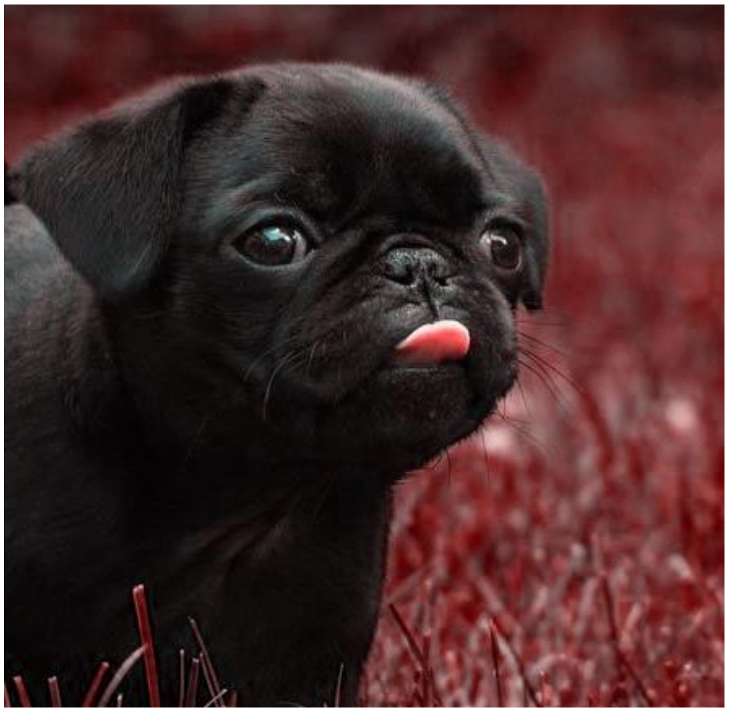


Figure 12: Color Swap 2 and Color Swap 1

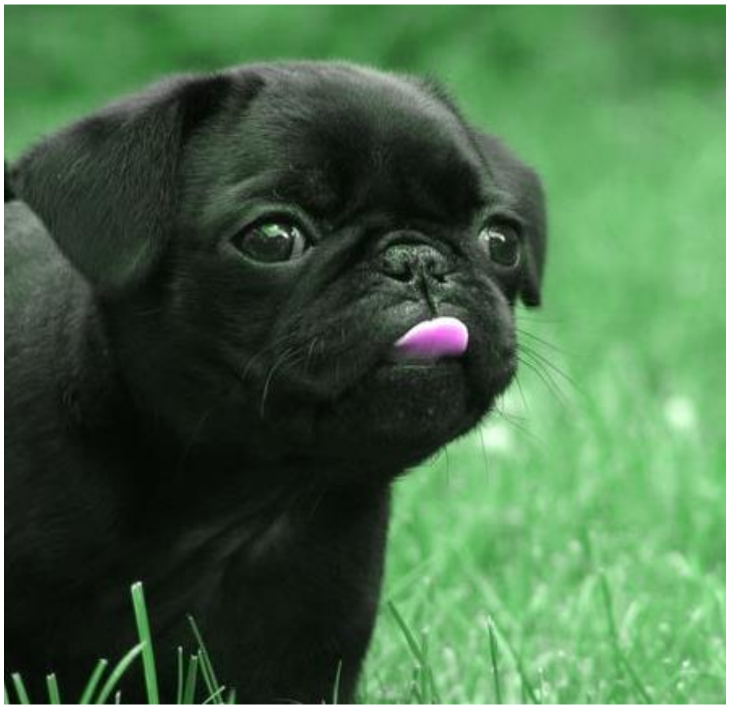


Figure 13: Raster Manipulation

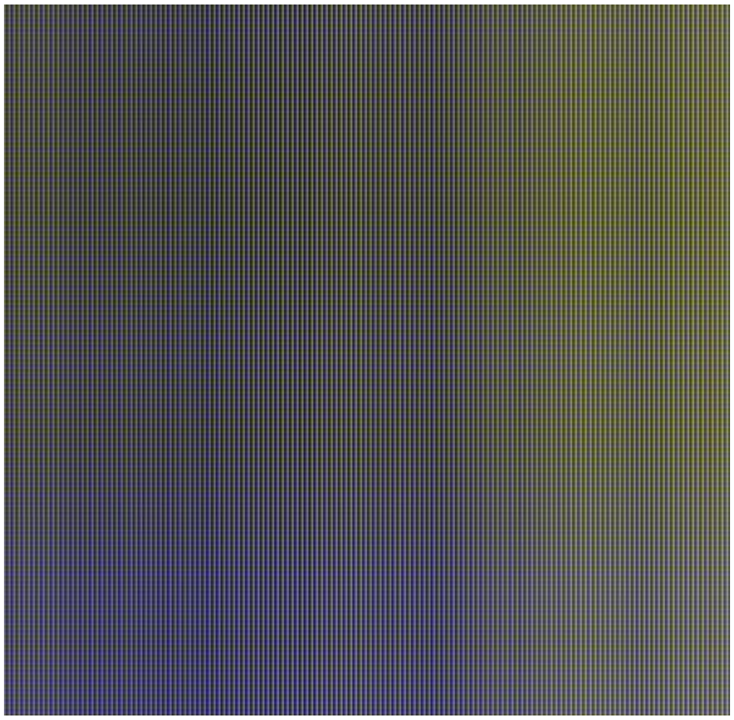
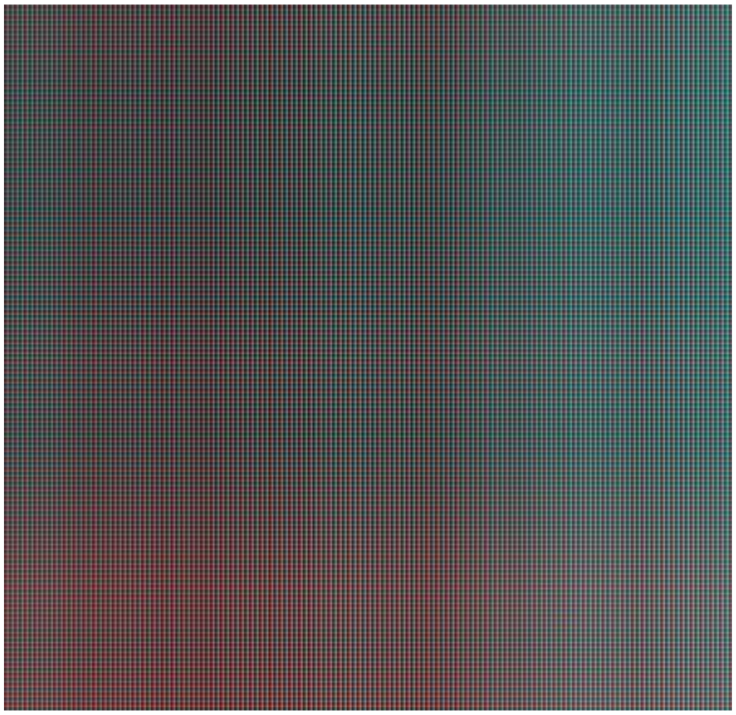


Figure 14: Raster Manipulation 2



Other Interesting images: