AP Computer Science
Picture Lab 6: Edge Detection

Name _____
Date _____ Period _____

Objective: To work on methods to do simple edge detection.

Background:
Detecting edges is a common image processing problem. For example, digital cameras often feature face detection. Some robotic competitions require the robots to find a ball using a digital camera, so the robot needs to be able to "see" a ball.

One way to look for an edge in a picture is to compare the color at the current pixel with the pixel in the next column to the right. If the colors differ by more than some specified amount, this indicates that an edge has been detected and the current pixel color should be set to black. Otherwise, the current pixel is not part of an edge and its color should be set to white (see the figures below). How do you calculate the difference between two colors? The formula for the difference between two points (x_1, y_1) and (x_2, y_2) is $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$. Likewise, the difference between two colors $(red_1, green_1, blue_1)$ and $(red_2, green_2, blue_2)$ is $\sqrt{(red_2 - red_1)^2 + (green_2 - green_1)^2 + (blue_2 - blue_1)^2}$. The **colorDistance** method in the **Pixel** class uses this calculation to return the difference between the current pixel color and a passed color.

Below is a swan picture and its edge detected version to the right. The method is **edgeDetection** in **Picture.java**. The edge detection algorithm measures the color distance between each pixel and its immediate right (next column) neighbor to determine if the pixel is at an edge. If the color distance between the two pixels is greater than a set threshold value, then an edge is detected and the resulting picture's pixel is turned black. Otherwise, the resulting pixel is turned white denoting no edge. For the picture below, each pixel measured the color distance to its right pixel neighbor (next column over).

Activity:
A1) Write an edge detection method that detects edges by comparing each pixel with the pixel below.

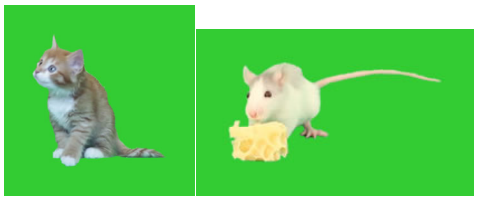
Create the new method `Picture edgeDetectionBelow(int threshold)` inside **Picture.java** that uses the pixel below (next row) to calculate the color distance and returns a black-and-white edge picture.

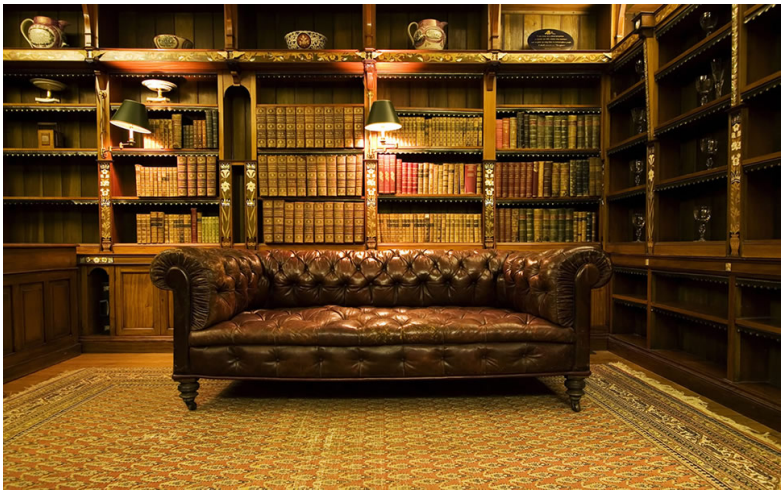
Use the following method comments, signature, and code to begin your method:

```
/** Method that creates an edge detected black/white picture
 *  @param threshold      threshold as determined by Pixel's colorDistance method
 *  @return               edge detected picture
 */
public Picture edgeDetectionBelow(int threshold)
{
  Pixel[][] pixels = this.getPixels2D();
  Picture result = new Picture(pixels.length, pixels[0].length);
  Pixel[][] resultPixels = result.getPixels2D();
```

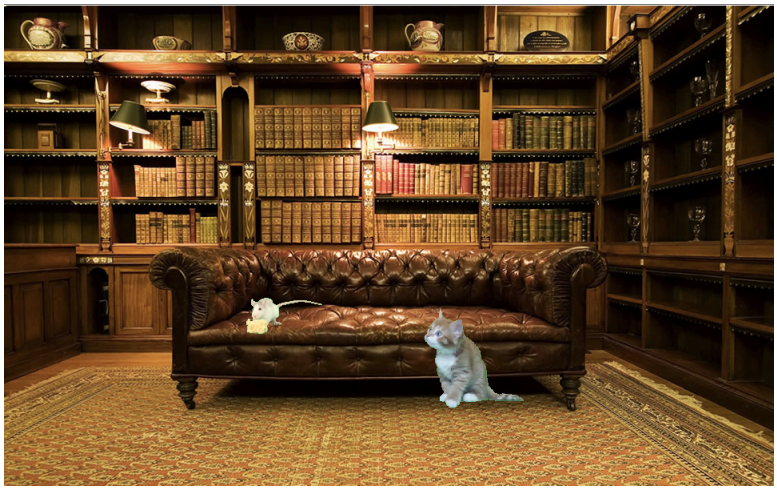
Green Screen

Another popular effect is the “green screen”. On television, they make a person or object “appear” somewhere else. This is done by superimposing a picture of the subject in front of a picture of a location. For example, below are two photos of a cat and mouse each with a green background. The third photo is a couch in a library.





By scaling and superimposing the cat and mouse onto the library background, the two animals appear to be in the same room together.



Activity:

A2) Download **greenScreenImages.zip** and extract the directory. It contains two backgrounds: IndoorHouseLibraryBackground.jpg and IndoorJapaneseRoomBackground.jpg. In addition, there are six green screen images of cats, a dog, a mouse, and Minions. Write a method that returns an image of two green screen images superimposed on the background. Everything must look natural and in scale. Points will be deducted if the images are out of scale or look unnatural. Use **PictureExplorer** on the green screen images to determine the green you must remove.

Create the new method `Picture greenScreen()` inside **Picture.java** that returns a green screen result.

Important!!! This project is different than all the others because you will need to **hardcode** the files that you use for background and green screen images inside your **greenScreen** method in **Picture.java**. Use only the pictures provided by Mr Greenstein for background and green screen images. Images must start with the directory “greenScreenImages/“ in the file name.

Here is an example of a green screen method inside **Picture.java**:

```
/** Method that creates a green screen picture
*  @return               green screen picture
 */
public Picture greenScreen()
{
  // Get background picture
  Picture bkgnd = new Picture(“greenScreenImages/IndoorHouseLibraryBackground.jpg”);
  Pixels[][] bkgndPixels = bkgnd.getPixels2D();
  // Get cat picture
  Picture cat = new Picture("greenScreenImages/kitten1GreenScreen.jpg");
  Pixel[][] catPixels = cat.getPixels2D();
  // Get mouse picture
  Picture mouse = new Picture(“greenScreenImages/mouse1GreenScreen.jpg”);
  Pixel[][] mousePixels = mouse.getPixels2D();
  …
```

Here is an example of the test method inside PictureTester.java:

```
/** Method to test greenScreen */
public static void testGreenScreen()
{
  // choose any picture to start since it will *not* be used
  Picture pic = new Picture(“images/beach.jpg");
  Picture gScreen = pic.greenScreen();
  gScreen.explore();
}
```

Activity

A3) Write a method to rotate a picture a certain angle.

Create the new method `Picture rotate(double angle)` inside **Picture.java** that rotates a picture in radian measure and returns a rotated picture. Rotation causes issues that you will need to fix as part of the method.

The transformation formulas for calculating a rotated pixel are the following:

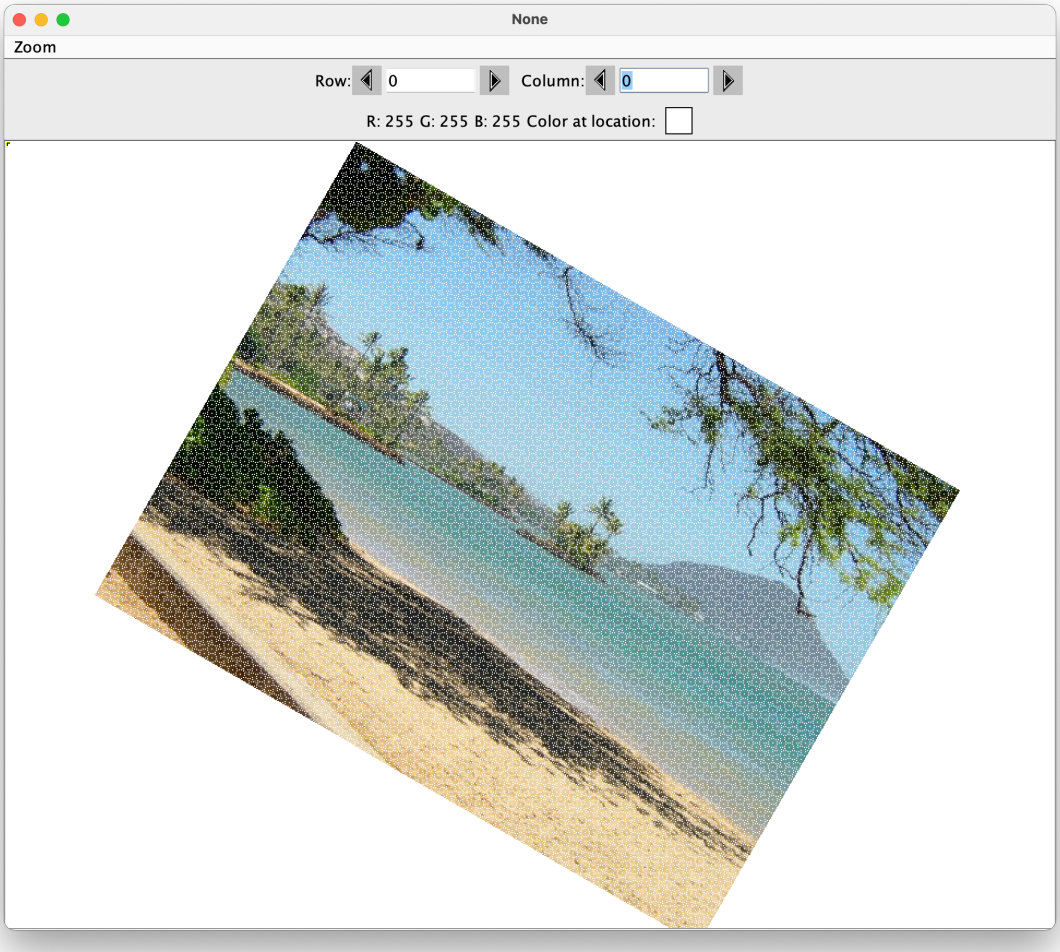
$$x_1 = x_0 \cos\theta - y_0 \sin\theta$$
$$y_1 = x_0 \sin\theta + y_0 \cos\theta$$

where (x_0, y_0) is the original location and (x_1, y_1) is the new location. Remember, columns represent the x-direction and rows represent the y-direction.

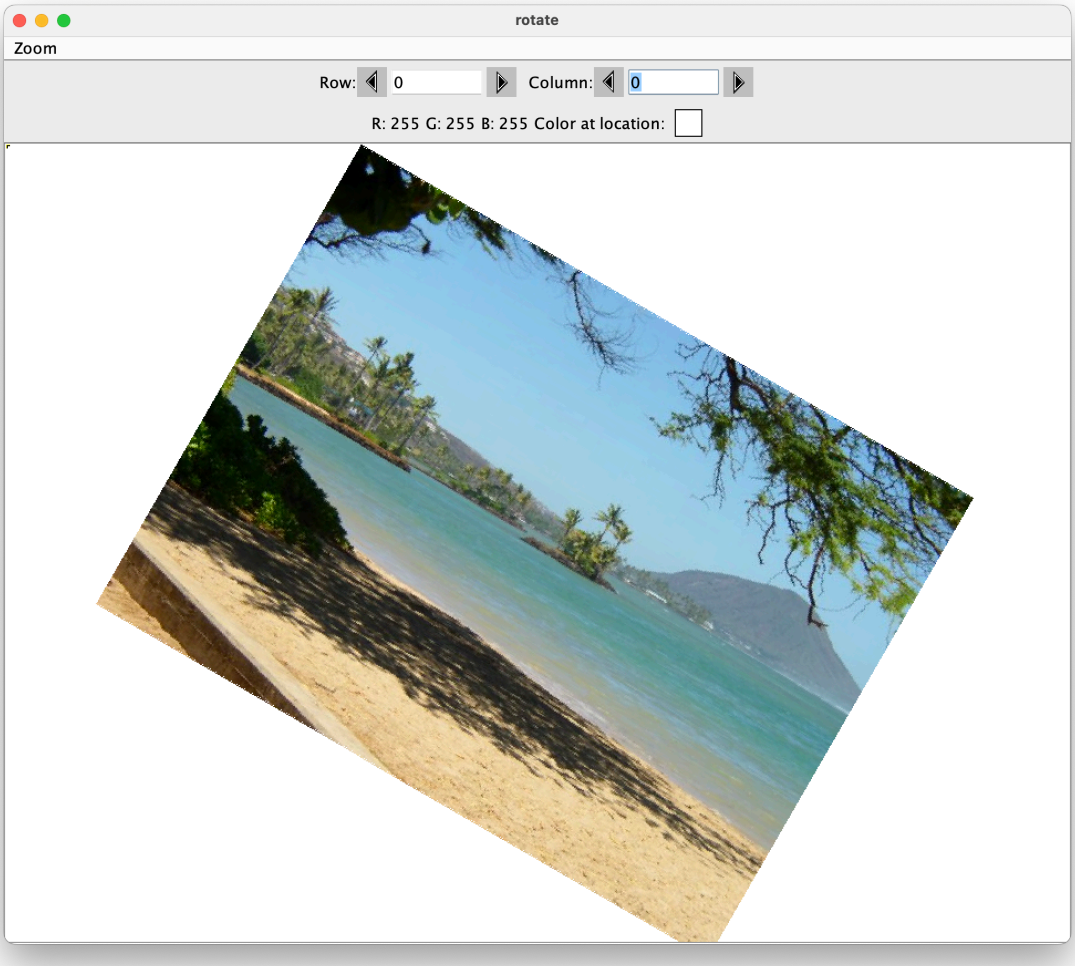
You will need to write your code to compensate and fix issues that happen when you rotate, for example:

1. Rotation happens around (0,0) as the center, so the picture will need to be moved to center it.
2. The new picture will need to be bigger to hold the rotated image. At least 90% of the picture must show in the rotated version.
3. Pixels will "drop out", meaning that rotation does not fill all of the space. "Drop out" causes pixel-sized holes throughout the picture. You will need to find a way to fill these spaces to create a complete picture.

A rotated picture with "drop out" looks like this:



You will need to “fill in” all of the pixels so the picture looks whole again, like below.



Use the following method comments, signature, and code to begin your method:

```
/**
 *    Rotate image in radians, clean up "drop-out" pixels
 *    @param angle        angle of rotation in radians
 *    @return             Picture that is rotated
 */
public Picture rotate(double angle) {
```

Mr Greenstein will check the rotations of $\frac{\pi}{6}$ radians (30˚) and $\frac{\pi}{4}$ radians (45˚). Be sure to handle both of these cases.