Andrew Fernandez

CAP4401

Project 3

Project 3 Report

This project was very difficult. The instructions felt vague so I definitely had to do more research online and using the textbook to figure out exactly what I needed to do. I re-implemented the way I used my ROIs because I figured out a more efficient way to write them and the parameters file. This report will cover the parameters file, compilation and execution instructions, and each function implemented in this project, including examples with images and their timings that are printed out every time.

I. Parameters

There is one file that is read by the program. It has very strict options for each line in the file. It includes the information for each ROI, parameters for the various functions for each ROI, and the images the appropriate functions should be run on. There are cycles for the lines. For every image/function that you want to have operated on, the first line will contain the name of the source image, what you wish to name the target image, the function that you want to operate over the image on, and the number of ROIs (n) used for the function. The following n lines will be information given for the parameters for each ROI for each function. Each of these n lines will contain the following information in the following order: x, y, sx, sy, the threshold, and the direction. Everything is separated by spaces. The first lines will be in this form: input_image output_image function. Where the function can be gray_edge (for gray edge detection), rgb_edge (for RGB color edge detection), hsi_edge (for HSI color edge detection). A side-note is that the parameters file should be in the following path: /Project2/project/bin/. The following image is an example of a properly constructed parameters file called parameters.txt:

project > bin > parameters.txt

```
baboon.pgm baboon_edge_d.pgm gray_edge 2
0 0 100 300 10 45
320 80 150 150 15 100
wheel.pgm wheel_d.pgm gray_edge 1
0 0 300 300 10 45
ball.ppm ball_edge_d.ppm rgb_edge 2
100 50 100 100  10 45
210 200 280 215 10 45
wine.ppm wine_edge_d.ppm rgb_edge 3
0 0 100 300 10 45
200 600 250 100 10 45
400 400 100 100 10 45
ball.ppm ball_hsi_edge_d.ppm hsi_edge 2
100 50 100 100  10 45
210 200 280 215 10 45
wine.ppm wine_hsi_edge_d.ppm hsi_edge 3
0 0 100 300 10 45
200 600 250 100 10 45
400 400 100 100 10 45
```

II. Compilation and Execution

In order to compile this program, make needs to be run twice. Once inside the Project2 directory, run make inside of /Project2/iptools to compile both image.cpp and utility.cpp. Then travel to /Project2/project and run make to compile iptool.cpp. This will create the executable in /Project2/project/bin of the name iptool. To run the program, travel to /Project2/project/bin/ and execute ./iptool <file_name> where <file_name> is the name of your parameters file located inside /Project2/project/bin/.

III. Functions

A. Gray Edge Detection

This function generates 3 images. One that uses the gradient amplitude calculated as the intensity of the image. The name of this image is the target one specified in the parameters file. One that thresholds the gradient amplitude. The name of the image is the same as the target

output file name, prepended with “grad_amplitude_thresh”. The last image that it generates that thresholds the direction. The name of the image is the same as the target output file name, prepended by “direction_thresh”.

I start this function by creating the images I will be generating and resizing them to the source’s dimensions. Then I loop through all my ROIS, extract the region information and the function’s parameters. Then I use a double loop to go through the source image. If the current pixel is inside the region, I call a function call getGradientXY that calculates the gx and gy given the source image, the indices of the pixel, and the current region being looked at. Then I calculate the gradient amplitude and the pixel’s angle direction. I set the one image’s intensity value to the gradient amplitude. Then I threshold to binarize by the amplitude for the second image. And finally, I threshold by the direction that was given. Here is my function:

```
void utility::grayEdgeDetection(image& src, image& tgt, const vector<roi>& regions, char* outfile) {
    tgt.resize(src.getNumberOfRows(), src.getNumberOfColumns());

    image amplitude_threshold, direction_threshold;
    amplitude_threshold.resize(src.getNumberOfRows(), src.getNumberOfColumns());
    direction_threshold.resize(src.getNumberOfRows(), src.getNumberOfColumns());

    image temp_img, temp_img_gat, temp_img_dir;
    double gradient_amplitude;
    double pixel_angle;

    //  sobel operator
    temp_img.copyImage(src);
    temp_img_gat.copyImage(src);
    temp_img_dir.copyImage(src);
    for (int r = 0; r < regions.size(); r++) {
        int x = regions.at(r).x;
        int y = regions.at(r).y;
        int sx = regions.at(r).sx;
        int sy = regions.at(r).sy;
        int T = regions.at(r).gray_threshold;
        int angle = regions.at(r).gray_direction;

        for (int i = 0; i < temp_img.getNumberOfRows(); i++) {
            for (int j = 0; j < temp_img.getNumberOfColumns(); j++) {
                if (
                    i >= y &&
                    i < (y + sy) &&
                    j >= x &&
                    j < (x + sx)
                ) { // inside the region
                    // calculating gx, gy, gradient amplitude, and edge direction using Sobel Operator 3x3
                    int x_gradient = getGradientXY(temp_img, i, j, regions.at(r)).gx;
                    int y_gradient = getGradientXY(temp_img, i, j, regions.at(r)).gy;

                    gradient_amplitude = sqrt(pow(x_gradient, 2) + pow(y_gradient, 2));
                    pixel_angle = atan((double)y_gradient/(double)x_gradient) * (180/PI);

                    // setting intensity image
                    tgt.setPixel(i, j, checkValue(gradient_amplitude));

                    // setting amplitude thresholed img
                    if (gradient_amplitude < T) {
                        amplitude_threshold.setPixel(i, j, MINRGB);
                    }
                    else {
                        amplitude_threshold.setPixel(i, j, MAXRGB);
                    }
```

```
200
201                     // setting direction threshold image
202                     if (gradient_amplitude > T) {
203                         if (
204                             pixel_angle >= (angle - 10) &&
205                             pixel_angle <= (angle + 10)
206                         ) {
207                             direction_threshold.setPixel(i, j, MAXRGB);
208                         }
209                         else {
210                             direction_threshold.setPixel(i, j, MINRGB);
211                         }
212                     }
213                 }
214                 else {
215                     tgt.setPixel(i, j, checkValue(temp_img.getPixel(i, j)));
216                     amplitude_threshold.setPixel(i, j, checkValue(temp_img_gat.getPixel(i, j)));
217                     direction_threshold.setPixel(i, j, checkValue(temp_img_dir.getPixel(i, j)));
218                 }
219             }
220         }
221         temp_img.copyImage(tgt);
222         temp_img_gat.copyImage(amplitude_threshold);
223         temp_img_dir.copyImage(direction_threshold);
224     }
225
226     char gat_img_name[100] = "grad_amplitude_thresh_";
227     amplitude_threshold.save(strcat(gat_img_name, outfile));
228
229     char dir_img_name[100] = "direction_thresh_";
230     direction_threshold.save(strcat(dir_img_name, outfile));
231 }
```

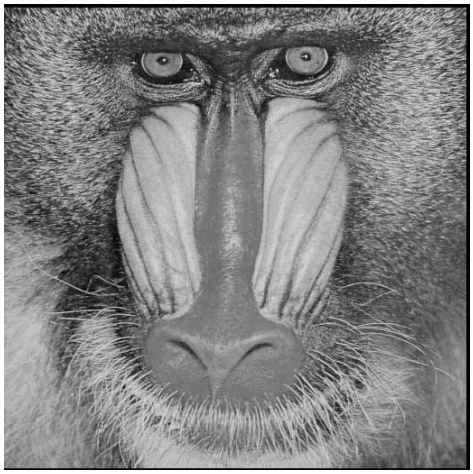
Here is the getGradientXY helper function:

```
121  gradient_amplitude getGradientXY(image& src, int i, int j, roi reg) {
122      int x_grad = 0, y_grad = 0;
123      gradient_amplitude ga;
124
125      x_grad = (
126          getPixelIfInROI(src, i + 1, j - 1, reg) +
127          getPixelIfInROI(src, i + 1, j, reg) * 2 +
128          getPixelIfInROI(src, i + 1, j + 1, reg) -
129          getPixelIfInROI(src, i - 1, j - 1, reg) -
130          getPixelIfInROI(src, i - 1, j, reg) * 2 -
131          getPixelIfInROI(src, i - 1, j + 1, reg)
132      );
133
134      y_grad = (
135          getPixelIfInROI(src, i + 1, j + 1, reg) +
136          getPixelIfInROI(src, i, j + 1, reg) * 2 +
137          getPixelIfInROI(src, i - 1, j + 1, reg) -
138          getPixelIfInROI(src, i - 1, j - 1, reg) -
139          getPixelIfInROI(src, i, j - 1, reg) * 2 -
140          getPixelIfInROI(src, i + 1, j - 1, reg)
141      );
142
143      x_grad /= 8;
144      ga.gx = x_grad;
145
146      y_grad /= 8;
147      ga.gy = y_grad;
148
149      return ga;
150  }
```

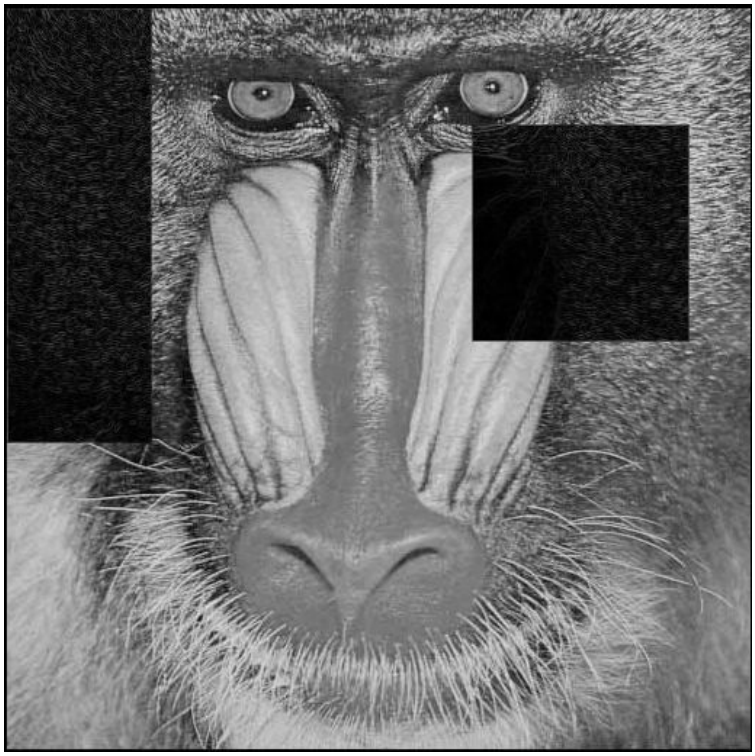
And here is the getPixelIfInROI helper function:

```
107  int getPixelIfInROI(image& src, int i, int j, roi& reg) {
108      if (
109          i >= reg.y &&
110          i < (reg.y + reg.sy) &&
111          j >= reg.x &&
112          j < (reg.x + reg.sx)
113      ) { // In the ROI
114          return src.getPixel(i, j);
115      }
116      else {
117          return 0;
118      }
119  }
```

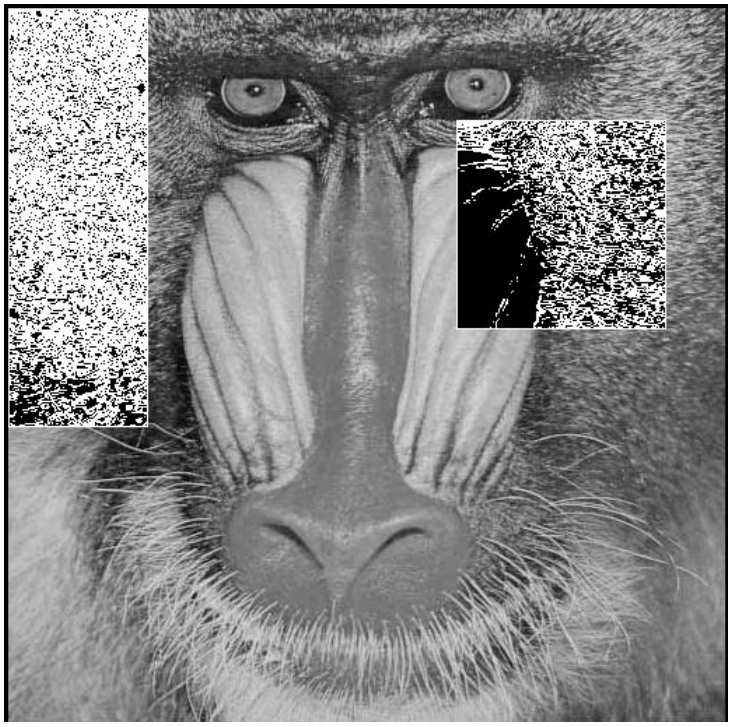
Here is baboon.pgm as the input:



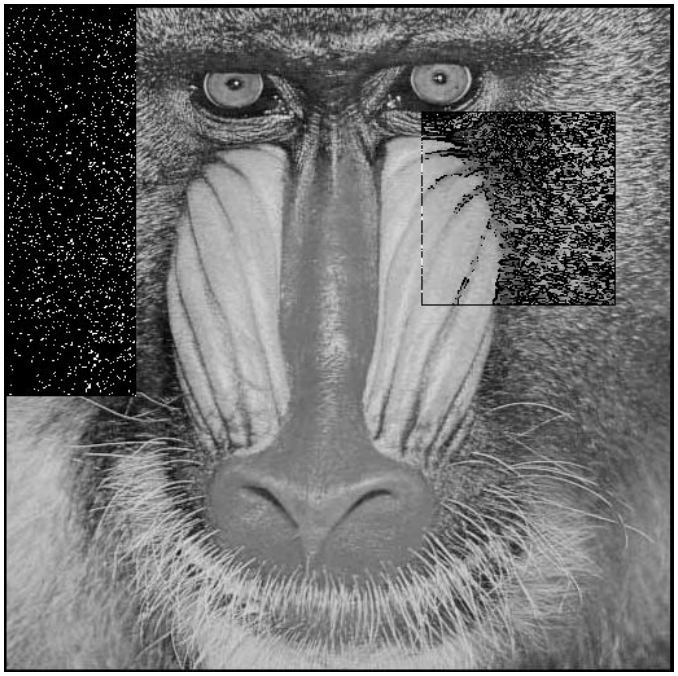
Here is baboon_edge_d output:



Here is grad_amplitude_thresh_baboon_edge.pgm output:



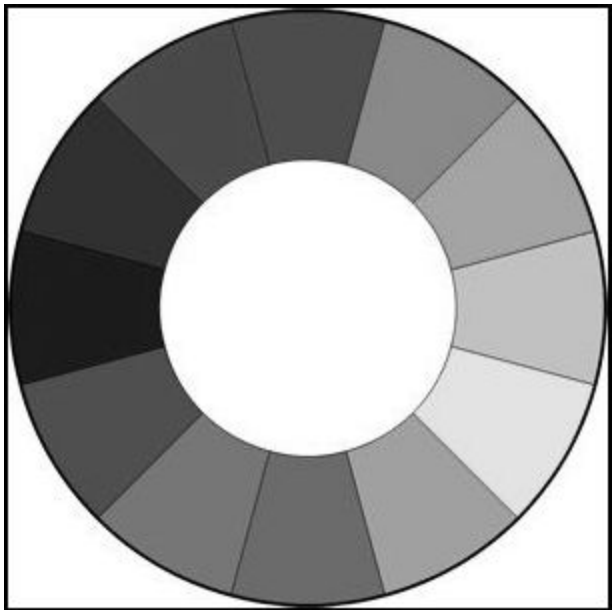
Here is direction_thresh_baboon_edge_d.pgm output:



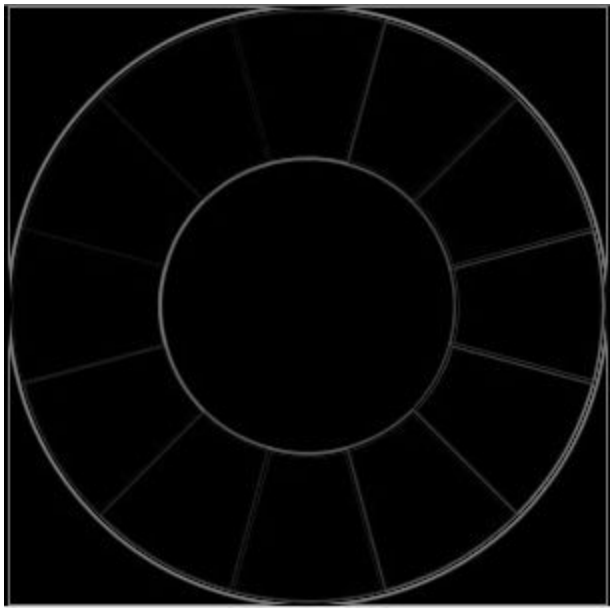
Here is the timing for this function to generate these three images:

```
Gray Edge time for baboon.pgm = 74ms
```

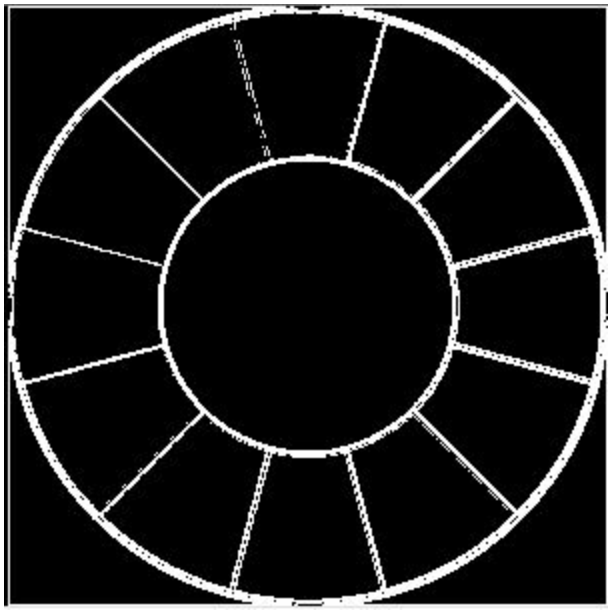
Here is the input for wheel.pgm:



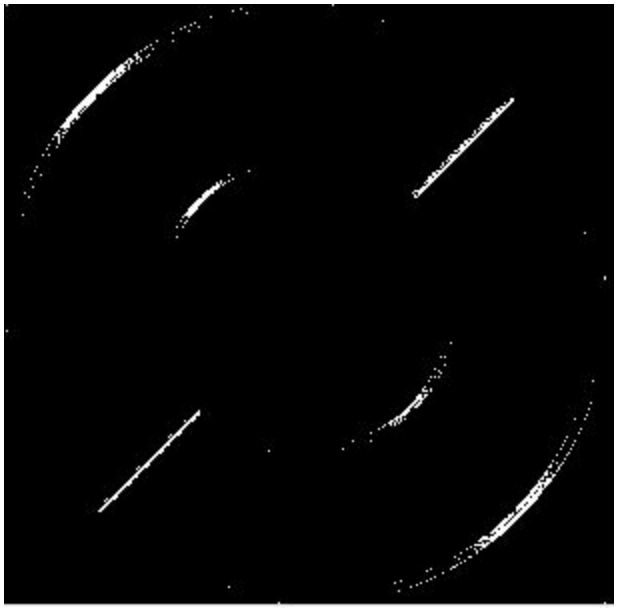
Here is wheel_d.pgm for output:



Here is the grad_amplitude_thresh_wheel_d.pgm output file:



Here is the directdion_thresh_wheel_d.pgm output:



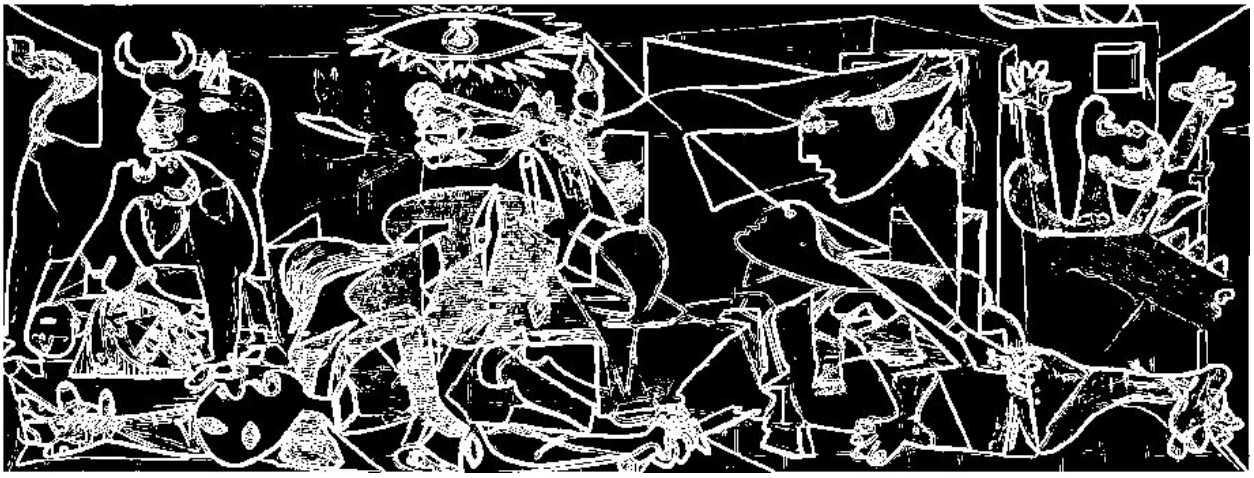
Here is the timing for this input image:

```
Gray Edge time for wheel.pgm = 38ms
```

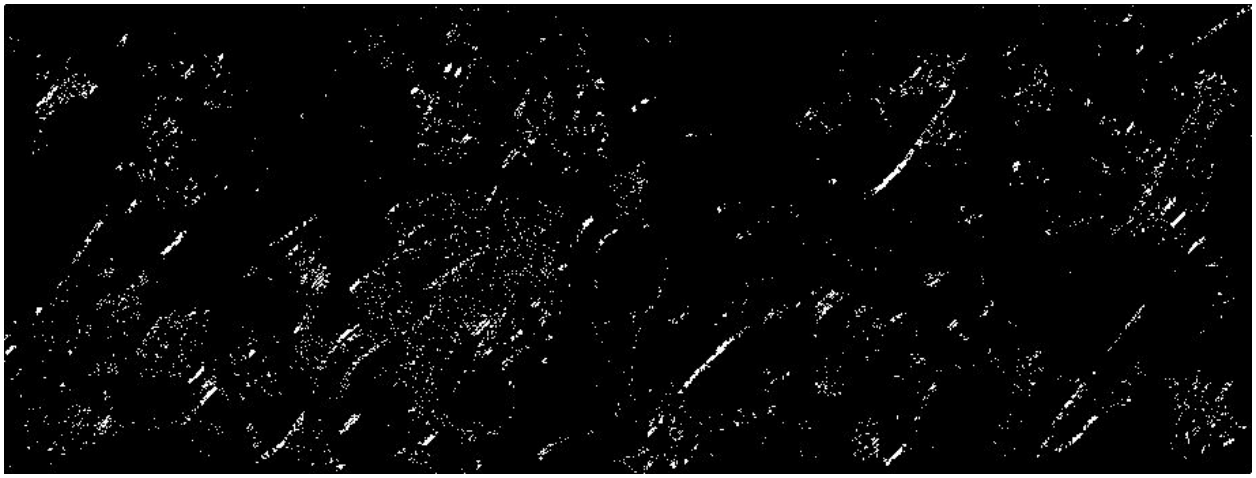
Here is the input for picasso.pgm:

Here is the output for picasso_edge_d.pgm:

Here is the grad_amplitude_thresh_picasso_edge_d.pgm output:



Here is the direction_thresh_picasso_edge_d.pgm output:



Here is the timing for this input image:

This was the slowest of the 3 gray scale images that my function executed. This is because it probably had more edges it had to detect and operate on than the other images. It is not necessarily bigger than baboon.pgm.

B. RGB Edge Detection

This function generates 4 images based on a color input image. Three of these images are for the different RGB channels that were edge detected. In this report, I will only be including the final image with these RGB images combined.

I start off by creating all the images the function needs to generate and resizing them based on the dimensions of the source image. I loop through my regions, extract the parameters, then loop through the source image. If the pixel is within the current region, I calculate the gx and gy for each of the 3 color channels. I do this using an overloaded version of getGradientXY that takes in a color as additional input. Then I calculate the gradient amplitude for each channel respectively. Then, I threshold the combination of the three channels, checking to see if they are less than the threshold given as a parameter. That is where I binarize the images. If it is not in the ROI, I just set the image's pixel to the source image's pixel. Here is that function:

```
281  void::utility::RGBEdgeDetection(image& src, image& tgt, const vector<roi>& regions, char* outfile) {
282      tgt.resize(src.getNumberOfRows(), src.getNumberOfColumns());
283      image red_edge_detection, green_edge_detection, blue_edge_detection;
284      red_edge_detection.resize(src.getNumberOfRows(), src.getNumberOfColumns());
285      green_edge_detection.resize(src.getNumberOfRows(), src.getNumberOfColumns());
286      blue_edge_detection.resize(src.getNumberOfRows(), src.getNumberOfColumns());
287
288      image temp_img, temp_img_red, temp_img_green, temp_img_blue;
289      temp_img.copyImage(src);
290      temp_img_red.copyImage(src);
291      temp_img_green.copyImage(src);
292      temp_img_blue.copyImage(src);
293
294      for (int r = 0; r < regions.size(); r++) {
295          int x = regions.at(r).x;
296          int y = regions.at(r).y;
297          int sx = regions.at(r).sx;
298          int sy = regions.at(r).sy;
299          int T = regions.at(r).color_threshold;
300          int angle = regions.at(r).color_direction;
301
302          double red_gradient_amplitude, green_gradient_amplitude, blue_gradient_amplitude;
303          double red_pixel_angle, green_pixel_angle, blue_pixel_angle;
304
305          for (int i = 0; i < temp_img.getNumberOfRows(); i++) {
306              for (int j = 0; j < temp_img.getNumberOfColumns(); j++) {
307                  if (
308                      i >= y &&
309                      i < (y + sy) &&
310                      j >= x &&
311                      j < (x + sx)
312                  ) { // inside the region
313                      int red_gx = getGradientXY(temp_img, i, j, regions.at(r), RED).gx;
314                      int red_gy = getGradientXY(temp_img, i, j, regions.at(r), RED).gy;
315                      int green_gx = getGradientXY(temp_img, i, j, regions.at(r), GREEN).gx;
316                      int green_gy = getGradientXY(temp_img, i, j, regions.at(r), GREEN).gy;
317                      int blue_gx = getGradientXY(temp_img, i, j, regions.at(r), BLUE).gx;
318                      int blue_gy = getGradientXY(temp_img, i, j, regions.at(r), BLUE).gy;
319
320                      red_gradient_amplitude = sqrt(pow(red_gx, 2) + pow(red_gy, 2));
321                      green_gradient_amplitude = sqrt(pow(green_gx, 2) + pow(green_gy, 2));
322                      blue_gradient_amplitude = sqrt(pow(blue_gx, 2) + pow(blue_gy, 2));
323
324                      red_pixel_angle = atan((double)red_gy/(double)red_gx) * (180/PI);
325                      green_pixel_angle = atan((double)green_gy/(double)green_gx) * (180/PI);
326                      blue_pixel_angle = atan((double)blue_gy/(double)blue_gx) * (180/PI);
327
```

```
328                     // thresholding gradient amplitude for three channels
329                     if (red_gradient_amplitude < T) {
330                         red_edge_detection.setPixel(i, j, RED, MINRGB);
331                         red_edge_detection.setPixel(i, j, GREEN, MINRGB);
332                         red_edge_detection.setPixel(i, j, BLUE, MINRGB);
333                     }
334                     else {
335                         red_edge_detection.setPixel(i, j, RED, MAXRGB);
336                         red_edge_detection.setPixel(i, j, GREEN, MAXRGB);
337                         red_edge_detection.setPixel(i, j, BLUE, MAXRGB);
338                     }
339
340                     if (green_gradient_amplitude < T) {
341                         green_edge_detection.setPixel(i, j, RED, MINRGB);
342                         green_edge_detection.setPixel(i, j, GREEN, MINRGB);
343                         green_edge_detection.setPixel(i, j, BLUE, MINRGB);
344                     }
345                     else {
346                         green_edge_detection.setPixel(i, j, RED, MAXRGB);
347                         green_edge_detection.setPixel(i, j, GREEN, MAXRGB);
348                         green_edge_detection.setPixel(i, j, BLUE, MAXRGB);
349                     }
350
351                     if (blue_gradient_amplitude < T) {
352                         blue_edge_detection.setPixel(i, j, RED, MINRGB);
353                         blue_edge_detection.setPixel(i, j, GREEN, MINRGB);
354                         blue_edge_detection.setPixel(i, j, BLUE, MINRGB);
355                     }
356                     else {
357                         blue_edge_detection.setPixel(i, j, RED, MAXRGB);
358                         blue_edge_detection.setPixel(i, j, GREEN, MAXRGB);
359                         blue_edge_detection.setPixel(i, j, BLUE, MAXRGB);
360                     }
361
362                     // thresholding the combination of the three channels
363                     if (
364                         red_gradient_amplitude < T &&
365                         green_gradient_amplitude < T &&
366                         blue_gradient_amplitude < T
367                     ) {
368                         tgt.setPixel(i, j, RED, MINRGB);
369                         tgt.setPixel(i, j, GREEN, MINRGB);
370                         tgt.setPixel(i, j, BLUE, MINRGB);
371                     }
```

```
                    }
                    else {
                        tgt.setPixel(i, j, RED, MAXRGB);
                        tgt.setPixel(i, j, GREEN, MAXRGB);
                        tgt.setPixel(i, j, BLUE, MAXRGB);
                    }
                }
                else {
                    tgt.setPixel(i, j, RED, checkValue(temp_img.getPixel(i, j, RED)));
                    tgt.setPixel(i, j, GREEN, checkValue(temp_img.getPixel(i, j, GREEN)));
                    tgt.setPixel(i, j, BLUE, checkValue(temp_img.getPixel(i, j, BLUE)));
                    red_edge_detection.setPixel(i, j, RED, checkValue(temp_img_red.getPixel(i, j, RED)));
                    red_edge_detection.setPixel(i, j, GREEN, checkValue(temp_img_red.getPixel(i, j, GREEN)));
                    red_edge_detection.setPixel(i, j, BLUE, checkValue(temp_img_red.getPixel(i, j, BLUE)));
                    green_edge_detection.setPixel(i, j, RED, checkValue(temp_img_green.getPixel(i, j, RED)));
                    green_edge_detection.setPixel(i, j, GREEN, checkValue(temp_img_green.getPixel(i, j, GREEN)));
                    green_edge_detection.setPixel(i, j, BLUE, checkValue(temp_img_green.getPixel(i, j, BLUE)));
                    blue_edge_detection.setPixel(i, j, RED, checkValue(temp_img_blue.getPixel(i, j, RED)));
                    blue_edge_detection.setPixel(i, j, GREEN, checkValue(temp_img_blue.getPixel(i, j, GREEN)));
                    blue_edge_detection.setPixel(i, j, BLUE, checkValue(temp_img_blue.getPixel(i, j, BLUE)));
                }
            }
        }
        temp_img.copyImage(tgt);
        temp_img_red.copyImage(red_edge_detection);
        temp_img_green.copyImage(green_edge_detection);
        temp_img_blue.copyImage(blue_edge_detection);
    }

    char red_img_name[100] = "red_ed_";
    red_edge_detection.save(strcat(red_img_name, outfile));

    char green_img_name[100] = "green_ed_";
    green_edge_detection.save(strcat(green_img_name, outfile));

    char blue_img_name[100] = "blue_ed_";
    blue_edge_detection.save(strcat(blue_img_name, outfile));
}
```

Here is the overloaded version of getGradientXY Helper function:

```
// overloaded function to get the gradient amplitude
gradient_amplitude getGradientXY(image& src, int i, int j, roi reg, channel color) {
    int x_grad = 0, y_grad = 0;
    gradient_amplitude ga;

    x_grad = (
        getPixelIfInROI(src, i + 1, j - 1, reg, color) +
        getPixelIfInROI(src, i + 1, j, reg, color) * 2 +
        getPixelIfInROI(src, i + 1, j + 1, reg, color) -
        getPixelIfInROI(src, i - 1, j - 1, reg, color) -
        getPixelIfInROI(src, i - 1, j, reg, color) * 2 -
        getPixelIfInROI(src, i - 1, j + 1, reg, color)
    );

    y_grad = (
        getPixelIfInROI(src, i + 1, j + 1, reg, color) +
        getPixelIfInROI(src, i, j + 1, reg, color) * 2 +
        getPixelIfInROI(src, i - 1, j + 1, reg, color) -
        getPixelIfInROI(src, i - 1, j - 1, reg, color) -
        getPixelIfInROI(src, i, j - 1, reg, color) * 2 -
        getPixelIfInROI(src, i + 1, j - 1, reg, color)
    );

    x_grad /= 8;
    ga.gx = x_grad;

    y_grad /= 8;
    ga.gy = y_grad;

    return ga;
}
```

Here is the overloaded version of getPixelIfInROI function:

```
// overloaded function to get the pixel if in ROI when also given a particular color
int getPixelIfInROI(image& src, int i, int j, roi& reg, channel color) {
    if (
        i >= reg.y &&
        i < (reg.y + reg.sy) &&
        j >= reg.x &&
        j < (reg.x + reg.sx)
    ) { // In the ROI
        return src.getPixel(i, j, color);
    }
    else {
        return 0;
    }
}
```

Here is the input for ball.ppm:

Here is the output ball_edge_d.ppm:

Here is the timing for this input:

```
RGB Edge time for ball.ppm = 50ms
```

Here is the input for wine.ppm:

Here is the output wine_edge_d.ppm:

Here is the timing for this input:

This was the slowest image that ran. This is probably because it is the largest one and for every operation it has to also calculate for RGB channels.

C. HSI Edge Detection

This picture generates 2 images. The first sets the gradient amplitude that it calculated to the pixel's intensity image. The next one thresholds the RGB values after converting back from HSI. This function is interesting because it converts each pixel in the given region to HSI, performs operations using those HSI values, converts it back to RGB, and finally uses those values to set the new values.

This function starts off by creating the images and resizing them based off the source's dimensions. Then I loop through the regions, extract the necessary parameters, then loop through the source image. If the current pixel is in the image, I convert the RGB pixel to HSI using RGBtoHSI function. Then I calculate the gx and gy using the intensity values that are calculated. Then I calculate the gradient amplitude and set the intensity value to this for the HSI. Then I use these values to convert back to RGB using the HSItoRGB function. Then, I set the value for the regular gradient_amplitude image to each respective RGB color channel that I got back from the conversion. Then I threshold the RGB values and binarize the image, similar to how I did it in the RGB function. If the pixel is not in the region, I just set it to the sources pixel at that location. Here is what that function looks like:

```
void utility::HSIEdgeDetection(image& src, image& tgt, const vector<roi>& regions, char* outfile) {
    tgt.resize(src.getNumberOfRows(), src.getNumberOfColumns());

    image grad_amplitude_img;
    grad_amplitude_img.resize(src.getNumberOfRows(), src.getNumberOfColumns());

    image temp_img, temp_img_grad_amp;

    temp_img.copyImage(src);
    temp_img_grad_amp.copyImage(src);
    for (int r = 0; r < regions.size(); r++) {
        int x = regions.at(r).x;
        int y = regions.at(r).y;
        int sx = regions.at(r).sx;
        int sy = regions.at(r).sy;
        int T = regions.at(r).color_threshold;

        for (int i = 0; i < temp_img.getNumberOfRows(); i++) {
            for (int j = 0; j < temp_img.getNumberOfColumns(); j++) {
                if (
                    i >= y &&
                    i < (y + sy) &&
                    j >= x &&
                    j < (x + sx)
                ) { // inside the region
                    HSI hsi_pixel = RGBtoHSI(
                        temp_img.getPixel(i, j, RED),
                        temp_img.getPixel(i, j, GREEN),
                        temp_img.getPixel(i, j, BLUE)
                    );

                    double gx = 0, gy = 0;

                    gx = (
                        calculateIValue(temp_img, i + 1, j - 1, regions.at(r)) +
                        calculateIValue(temp_img, i + 1, j, regions.at(r)) * 2 +
                        calculateIValue(temp_img, i + 1, j + 1, regions.at(r)) -
                        calculateIValue(temp_img, i - 1, j - 1, regions.at(r)) -
                        calculateIValue(temp_img, i - 1, j, regions.at(r)) * 2 -
                        calculateIValue(temp_img, i - 1, j + 1, regions.at(r))
                    );

```

```
                    gy = (
                        calculateIValue(temp_img, i + 1, j + 1, regions.at(r)) +
                        calculateIValue(temp_img, i, j + 1, regions.at(r)) * 2 +
                        calculateIValue(temp_img, i - 1, j + 1, regions.at(r)) -
                        calculateIValue(temp_img, i - 1, j - 1, regions.at(r)) -
                        calculateIValue(temp_img, i, j - 1, regions.at(r)) * 2 -
                        calculateIValue(temp_img, i + 1, j - 1, regions.at(r))
                    );

                    gx /= 8;
                    gy /= 8;
                    double ga_val = sqrt(pow(gx, 2) + pow(gy, 2));

                    // setting image to gradient amplitude intensity
                    HSI new_hsi_pix;
                    new_hsi_pix.h = hsi_pixel.h;
                    new_hsi_pix.s = hsi_pixel.s;
                    new_hsi_pix.i = ga_val;
                    RGB new_rgb_pixel = HSItoRGB(new_hsi_pix);
                    grad_amplitude_img.setPixel(i, j, RED, checkValue(new_rgb_pixel.r));
                    grad_amplitude_img.setPixel(i, j, GREEN, checkValue(new_rgb_pixel.g));
                    grad_amplitude_img.setPixel(i, j, BLUE, checkValue(new_rgb_pixel.b));

                    // setting image to binarized values
                    if (
                        new_rgb_pixel.r < T &&
                        new_rgb_pixel.g < T &&
                        new_rgb_pixel.b < T
                    ) {
                        tgt.setPixel(i, j, RED, MINRGB);
                        tgt.setPixel(i, j, GREEN, MINRGB);
                        tgt.setPixel(i, j, BLUE, MINRGB);
                    }
                    else {
                        tgt.setPixel(i, j, RED, MAXRGB);
                        tgt.setPixel(i, j, GREEN, MAXRGB);
                        tgt.setPixel(i, j, BLUE, MAXRGB);
                    }
                }
                else {
                    tgt.setPixel(i, j, RED, temp_img.getPixel(i, j, RED));
                    tgt.setPixel(i, j, GREEN, temp_img.getPixel(i, j, GREEN));
                    tgt.setPixel(i, j, BLUE, temp_img.getPixel(i, j, BLUE));
                    grad_amplitude_img.setPixel(i, j, RED, temp_img_grad_amp.getPixel(i, j, RED));
                    grad_amplitude_img.setPixel(i, j, GREEN, temp_img_grad_amp.getPixel(i, j, GREEN));
                    grad_amplitude_img.setPixel(i, j, BLUE, temp_img_grad_amp.getPixel(i, j, BLUE));
                }
            }
        }
```

```
591             temp_img.copyImage(tgt);
592             temp_img_grad_amp.copyImage(grad_amplitude_img);
593         }
594
595         char grad_amp_img_name[100] = "grad_ampltiude_";
596         grad_amplitude_img.save(strcat(grad_amp_img_name, outfile));
597     }
```

Here is the calculateIValue function used:

```
478 double calculateIValue(image& src, int i, int j, const roi& reg) {
479     if (
480         i >= reg.y &&
481         i < (reg.y + reg.sy) &&
482         j >= reg.x &&
483         j < (reg.x + reg.sx)
484     ) { // In the ROI
485         int rgb_sum = (
486             src.getPixel(i, j, RED) +
487             src.getPixel(i, j, GREEN) +
488             src.getPixel(i, j, BLUE)
489         );
490
491         double i_val = (double)rgb_sum / (double)(3 * 255);
492
493         return i_val;
494     }
495     else {
496         return 0;
497     }
498 }
```

Here is my RGBtoHSI conversion function:

```
411  HSI RGBtoHSI(int r, int g, int b) {
412      double h, s, i;
413      double norm_r = (double)r / (double)(r + g + b);
414      double norm_g = (double)g / (double)(r + g + b);
415      double norm_b = (double)b / (double)(r + g + b);
416
417      double num = 0.5 * ((norm_r - norm_g) + (norm_r - norm_b));
418      double den = sqrt(pow(norm_r - norm_g, 2) + ((norm_r - norm_b) * (norm_g - norm_b)));
419
420      if (b <= g) {
421          h = acos(num/den);
422      }
423      else {
424          h = ((2 * PI) - (acos(num/den)));
425      }
426
427      s = 1 - (3 * min(min(norm_r, norm_g), norm_b));
428      i = (double)(r + g + b) / (double)(3 * 255);
429
430      HSI pix;
431      pix.h = h;
432      pix.i = i;
433      pix.s = s;
434      return pix;
435  }
```

Here is my HSItoRGB conversion function:

```
437  RGB HSItoRGB(HSI pix) {
438      double h, s, i;
439      int r, g, b;
440
441      h = pix.h;
442      s = pix.s;
443      i = pix.i;
444
445      double x = i * (1 - s);
446
447      if (h < ((2 * PI) / 3)) {
448          RGB rgb_pix;
449          double y = i * (1 + (s * cos(h) / cos((PI/3) - h)));
450          double z = (3 * i) - (x + y);
451          rgb_pix.r = y * 255;
452          rgb_pix.g = z * 255;
453          rgb_pix.b = x * 255;
454          return rgb_pix;
455      }
456      else if (h < ((4 * PI) / 3)) {
457          auto new_h = h - ((2 * PI) / 3);
458          double y = i * (1 + (s * cos(new_h) / cos((PI/3) - new_h)));
459          double z = (3 * i) - (x + y);
460          RGB rgb_pix;
461          rgb_pix.r = x * 255;
462          rgb_pix.g = y * 255;
463          rgb_pix.b = z * 255;
464          return rgb_pix;
465      }
466      else  {
467          auto new_h = h - ((4 * PI) / 3);
468          double y = i * (1 + (s * cos(new_h) / cos((PI/3) - new_h)));
469          double z = (3 * i) - (x + y);
470          RGB rgb_pix;
471          rgb_pix.r = z * 255;
472          rgb_pix.g = x * 255;
473          rgb_pix.b = y * 255;
474          return rgb_pix;
475      }
476  }
```

Here is the input image ball.ppm:

Here is the output image ball_hsi_edge_d.ppm:

Here is the output image grad_amplitude_ball_hsi_edge.ppm:

The timing for this function is here:

```
HSI Edge time for ball.ppm = 24ms
```

Here is the input image wine.ppm:

Here is the output image wine_hsi_edge_d.ppm:

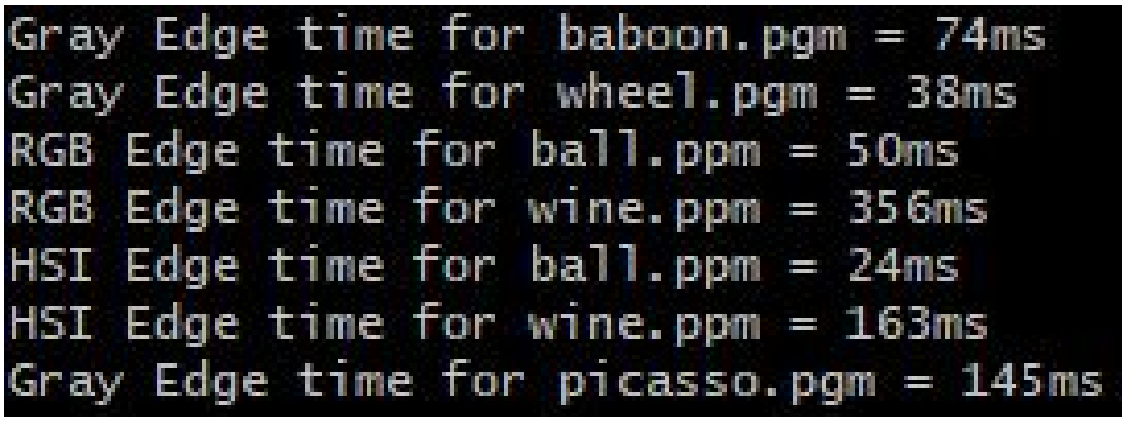
Here is the output image grad_amplitude_wine_hsi_edge_d.ppm:

Here is the timing for this input:

```
HSI Edge time for wine.ppm = 163ms
```

IV. Performance

Here are all my metrics for the functions and inputs:



The gray scale edge detectors were fairly normal time. They were quick as expected. I think the biggest surprise found from reviewing the metrics is that HSI and RGB edge detectors do the same thing, but the HSI function cuts the time required to accomplish this in half. This is pretty interesting because I would have hypothesized it would be the other way around since the RGB does not have to do all these conversions. However, the HSI halves the time it takes to detect the edges in these images. This is probably because it only has to consider one intensity value, rather than the 3 from each of the color channels when doing its calculations. It is a pretty significant time difference though. I thought that was pretty cool.

V. Conclusion

This was probably the most interesting image processing project I have done thus far. I really enjoyed doing this one more than the previous ones. I liked to see the images that were output by this one. I thought they look really cool. I feel like I gained a firm foundation of knowledge on edge detection as well as benefits of HSI over RGB. I thought it was really cool how HSI improved the performance of edge detection so much compared to the RGB. As always, I had my hard times with this project, but at the end, I believe I was able to get everything perfectly.