Digital Image Processing
Homework #4

Osama Salman – 0171927

Problem Understanding and Approach

We are given in this problem an image consisting of a number of shapes (lines, circles, squares, stars, and color spots) where each shape is colored in a specific color. However, some of the shapes in this image are overlapping making it harder to differentiate between them while also being spread in a way that makes shape-guessing techniques take wrong turns. It is worth mentioning that the “circles” in this image are not perfect circles – more like ellipses – which is an issue that we need to adapt to later on. The image is furthermore corrupted with what appears to be random noise that distorts the general shape of the lines and circles, which are our main interest here.

We are asked to produce a single new color image containing the longest red, green, and blue lines, and the largest red, green, and blue circles each filled and colored with previously-specified, solid colors. We also need to report the lengths of the longest lines in addition to the radii and centers of the largest circles we find.

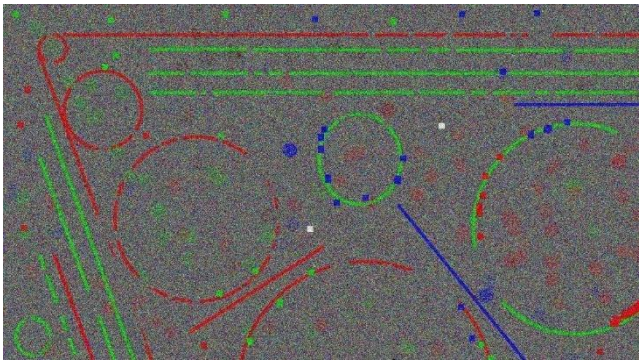
Visually, it is quite simple to guess and point out the required shapes. However, it is a hard task for the computer to do. So, in this approach we tend to make it easier for the computer by eliminating noise in each step whether it is straight-up random noise or just objects that may get in our way while performing the most important step which is finding the required lines and circles.

Steps and Outcomes

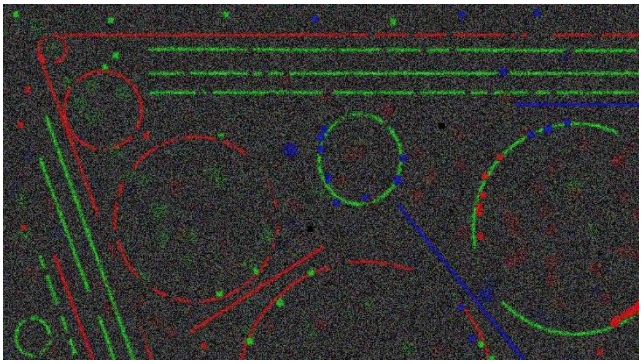
The main idea of our approach is to eliminate noise and unwanted objects in each step until the image is clear enough for line and circle detection. Our first step was to remove the random, white spots and pixels in the image. We perform this operation using color slicing with the prototypical color being pure white with the RGB values of (255, 255, 255) and with a radius of (250). We implemented color slicing using the function below:

```
def replace_color_with(image, center, radius, color):
    distance = find_color_distance(image, center)
    color_replaced_image = np.full_like(image, color)
    return np.where(distance <= radius, color_replaced_image, image)
```

This operation gave the following result:



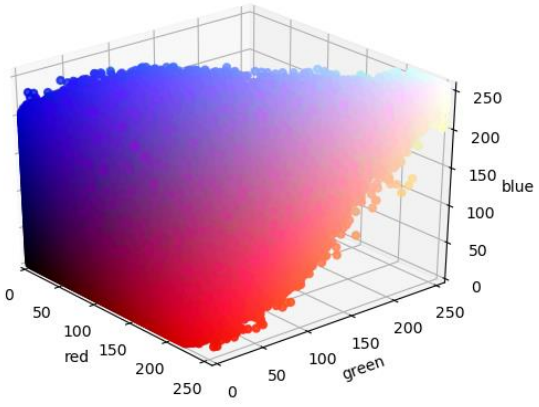
Zoomed-in, original image



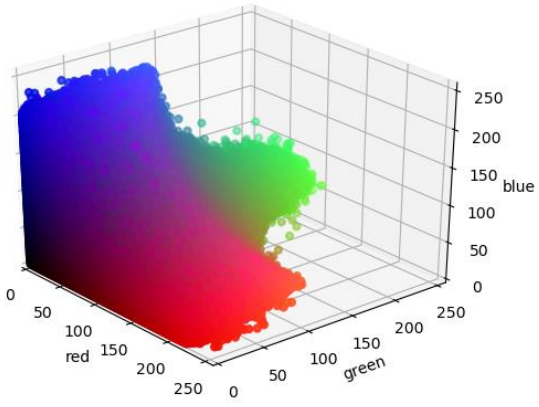
Zoomed-in, color-sliced image

As we can see, the image has got much more dim as all the bright pixels were eliminated along with the white squares all around the image. Calculations confirm this too as the average of the channels’ means dropped from (100.6) to (50). This operation is useful for removing the white pixels as they have high values of red, green, and blue which could be mistaken for another color when splitting

and separating the channels of the image. We can see from the visual representation of the RGB color space of the image – before and after performing color slicing – that most of the high-valued pixel colors were eliminated except for the red, green, and blue portions, which are the required colors for the segmentation operation. An opposite approach of the same idea could have been done, which suggests performing color slicing on the image with prototypical centers of pure red, green, and blue, which would produce somewhat similar similar results.

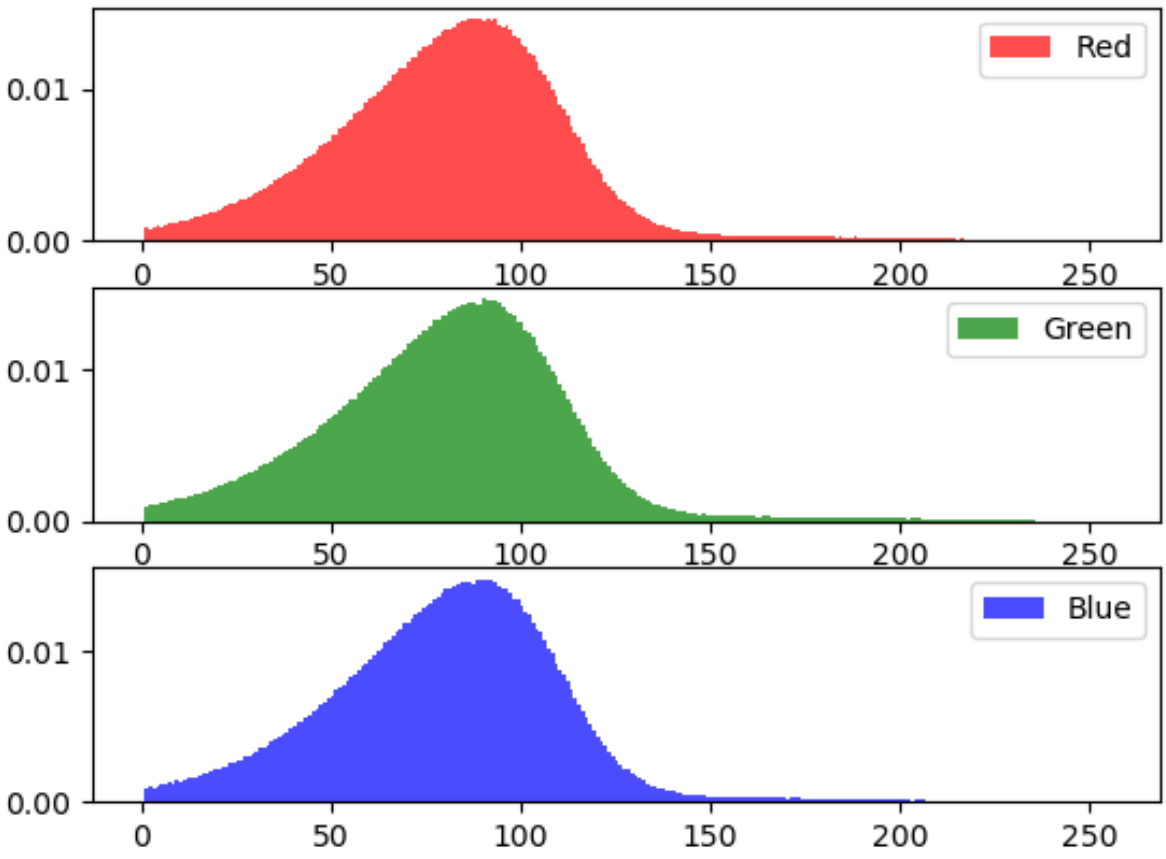


Color space of original image



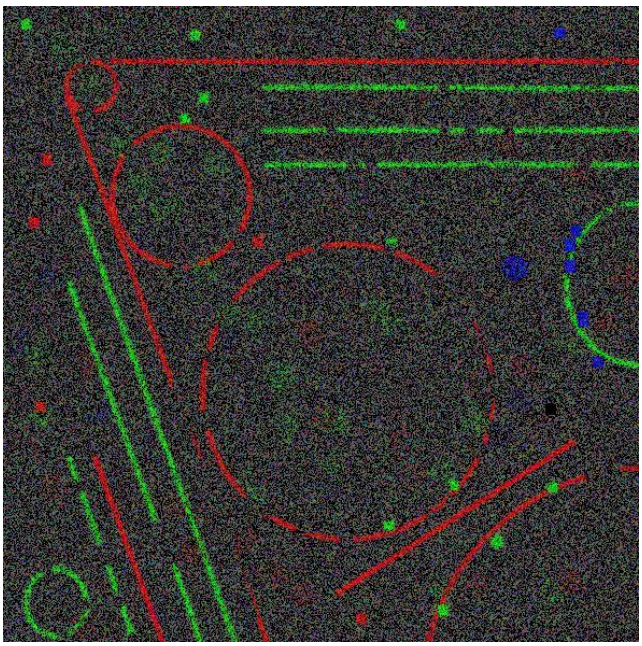
Color space of color-sliced image

The second step is to reduce the noise in the image making it easier to threshold the image and eliminate the random noise. Checking out the histogram of the color-sliced image (without counting the black pixels), the noise seems to be Gaussian.

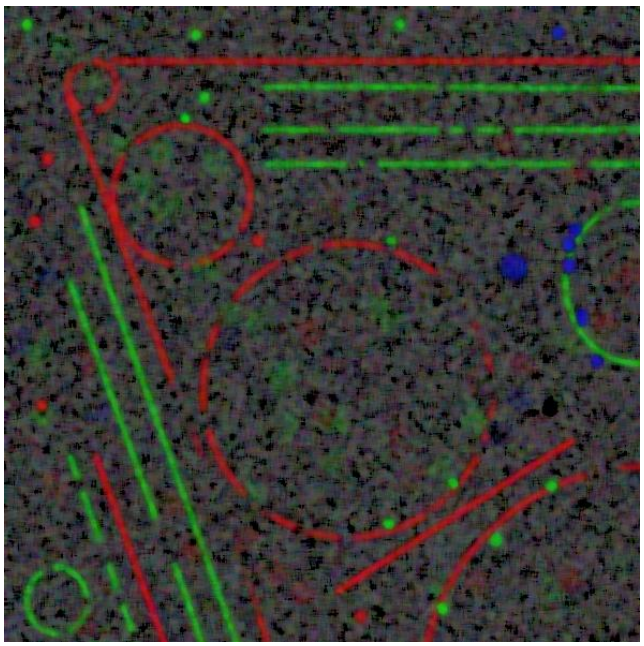


Red, green, and blue histograms of the color-sliced image

Gaussian noise can be reduced using spatial smoothing. We used the medianBlur function given by the OpenCV 2 Python library. This is the result we got after smoothing the image with a 7x7 median mask:



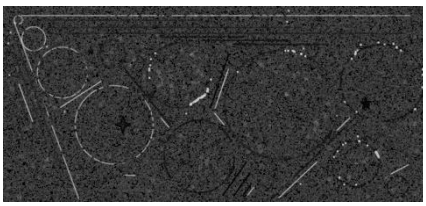
Zoomed-in, color-sliced image (before blurring)



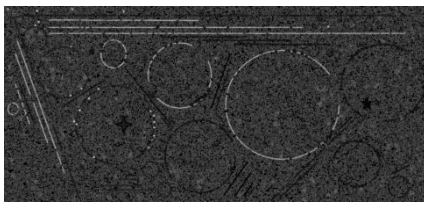
Zoomed-in, median-blurred image

We can see from the resultant image that the solid-color portions of the image got even more solid by reducing the noise inside them, while also reducing the contrast of the noise which is very visible visually and is also confirmed by the numbers as after blurring the image, the average of the standard deviation (contrast) of the three channels of the image went down from (45.89) to (22.71). This operation is important as it fixes and thickens the solid shapes and reduces the high-valued noise by bringing it closer to the mean, which makes the image ready for the threshold operation.

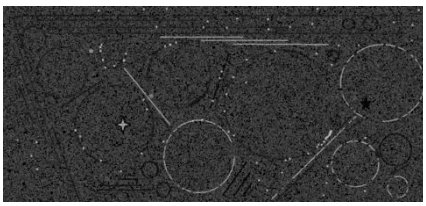
The image is now broken down and split into the three channels (red, green, and blue) in order to find and detect shapes of a single color. These are the greyscales of each color channel in this image:



Red channel



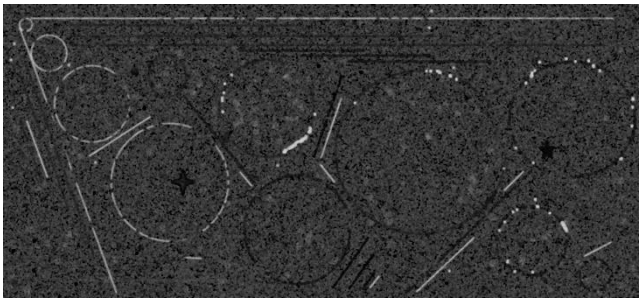
Green channel



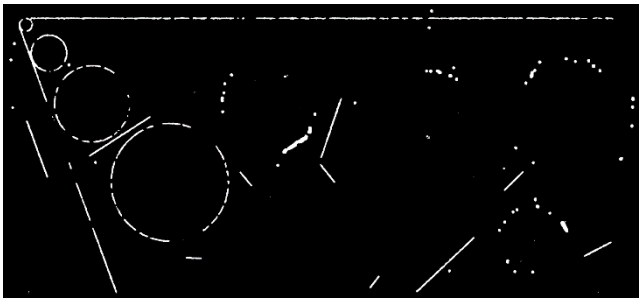
Blue channel

We are going to discuss performed operations on the red channel only as the exact same operations with the same values are also performed on the other two channels.

After eliminating the white pixels and reducing the noise, we need to threshold the image to completely isolate the red objects from the background and other colors in the image. The resultant red channel greyscale is of the range (0 – 255), so for fine results we performed the threshold operation at a value of (115). This is what the thresholding produced:



Red channel

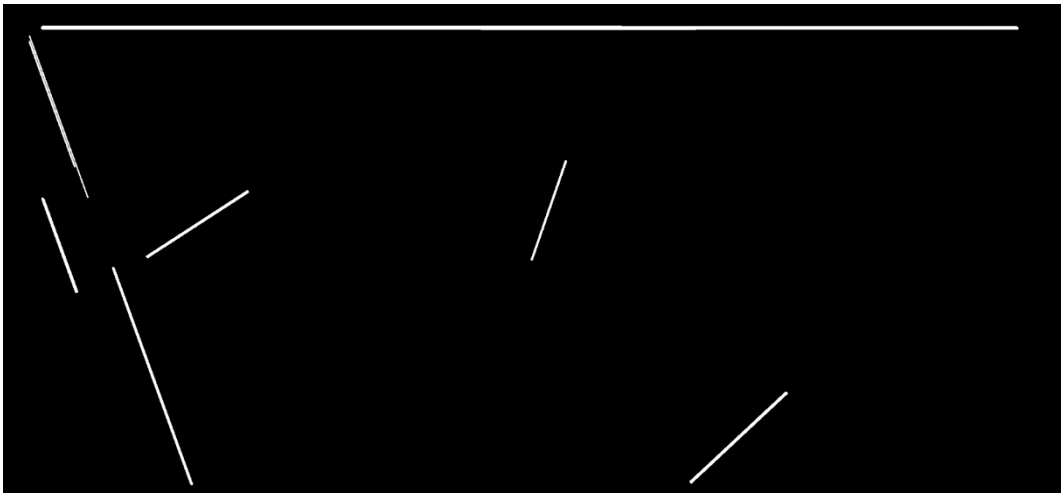


Red channel after thresholding

The red lines in the image are now very clear and could be easily identified using the Hough transform. OpenCV 2 offers a function that uses the Hough transform to find lines in the image and *edge linking* to link between the broken line pieces at or less than a certain gap value. These are the parameters that were used for this function where the parameters *rho* and *theta* indicate the resolution of the quantization of ρ and θ values, and *threshold* indicates the minimum number of "votes" in the accumulator that a line has to get in order to be considered a valid line.

```
cv2.HoughLinesP(thresholded_image, rho=1, theta=np.pi / 180, threshold=200,
minLineLength=200,maxLineGap=60)
```

This function saves up the task of finding the end points of the lines for us and returns the end points of the lines that it was able to find. These are the lines that it returned in the red channel:



Lines returned by the HoughLinesP function

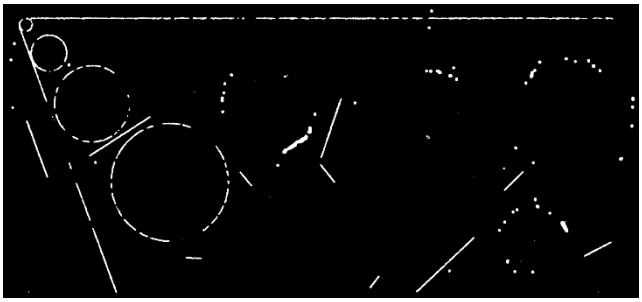
As we can see, the found lines are very precise despite having much noise all over the image, which shows how effective the Hough transform is. In order to find the longest line, we simply apply the distance equation ($length = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$) on the endpoints of each line and compare them to find the largest distance.

Line color	Length of the longest line (in pixels)
Red	2113
Green	1922
Blue	656.29

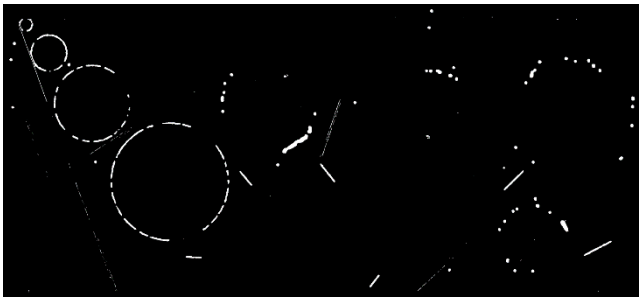
Length of the longest lines found

Now that we have found the lines in the image, the next step is to find the circles. For this process, we are going to use another function from OpenCV 2 called HoughCircles. This function is computationally heavy and is not always accurate in the case of existence of noise and pixels which are not part of a circle. So, we first try to reduce the noise even more and eliminate any object that is

not a circle. From the information we achieved when finding the lines, we can now eliminate all the lines found in the image by subtracting them from the thresholded image.



Thresholded red channel



Thresholded red channel after subtracting the lines

This operation yielded a great result as it succeeded in eliminating most of the lines in the image. Although, as noticed, some of the lines left very thin remnants behind due to the inaccuracy of the drawn lines' thickness. This could be solved by increasing the drawn lines' thickness, although, this could lead to distorting some of the circles in the image. Alternatively, as the remnants are very thin, they could be eliminated by applying *erosion* to the image.

Red channel (before applying erosion)

Eroded red channel

By applying *erosion* to the image using a 3x3 mask, the line remnants were gone along with the very small noise pixels and unwanted, extra details. However, this function made the circles thinner than they actually are, which would be an obstacle for us when applying the Hough Circle Transform. Thus, we apply a 3x3-masked *dilation* back on the image to restore the original circles' thicknesses.

Eroded red channel

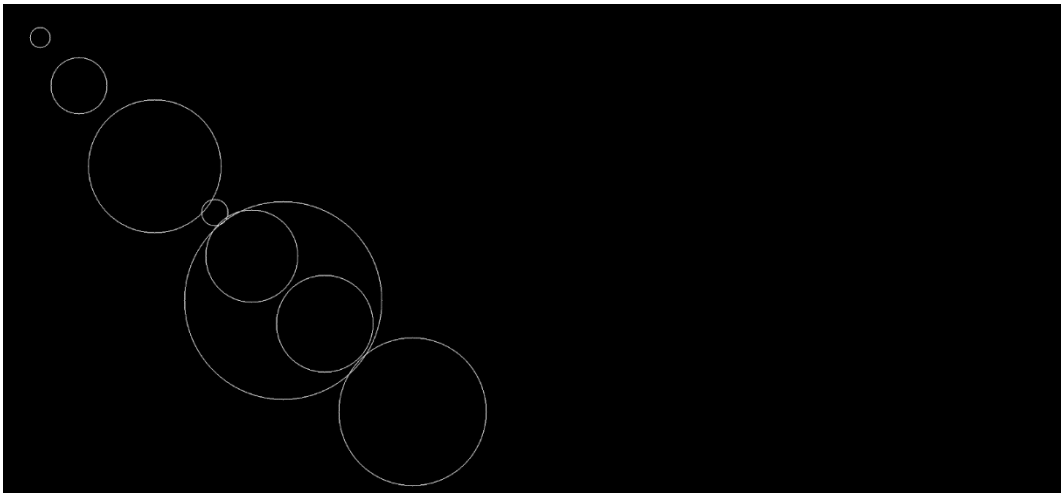
Dilated red channel

Applying *erosion* and *dilation* on the image reduced the amount of noise in the image for us. Comparing the red channel before and after these two operations, the difference is not that huge visually, but it yields far better performance and results in the step of finding the circles in the image. OpenCV 2 offers a function called HoughCircles which uses the Hough Circles Transform to find the most circle-like objects in the image. The Hough Circles Transform works in the same concept as the Hough Lines Transform, as it stores the circles that can be drawn from each *on* pixel in different radii, each radius on a different accumulator, then it sorts the circles found by the most *votes* in each accumulator and returns the centers and radii of the most fit circles according to some rules and parameters. These are the parameters that we used for this function where the parameter *dp*

determines the accumulator resolution, the *minDist* parameter determines the minimum distance between the returned circles' centers, and *param2* determines the threshold for centers in the accumulator at which circles are considered valid circles.

```
cv2.HoughCircles(dilated_image, cv2.HOUGH_GRADIENT, dp=1, minDist=100, param2=22)
```

These are the circles that the function was able to find according to these parameters:



Returned circles by the HoughCircles function

As we can see, the function was able to find the main circles in the red channel along with some other false circles which is due to the noise and the imperfections in the image. In order to find the largest circles in the image, we simply compare their radii and pick the largest one.

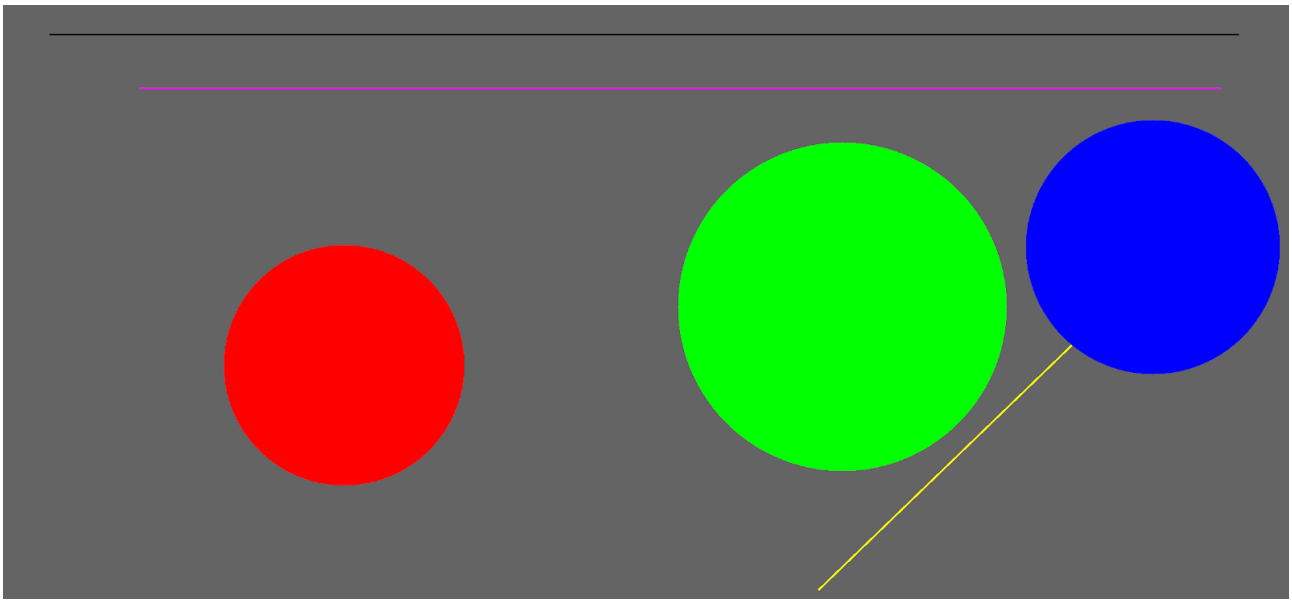
Circle color	Center of the circle (x, y)	Radius of the circle (in pixels)
Red	(606, 640)	214
Green	(1492, 536)	292
Blue	(2044, 430)	226

Centers and radii of the largest circles found

The next step is to draw these circles and fill each circle with a specific color. For filling the circles, we implemented and used the *Hole Filling* algorithm while passing it the center as the 'known pixel' inside the shape we are willing to fill as it was found to be the most effective in the process filling a circle. This is how we implemented the function:

```
def fill_hole(image, known_pixel):
    height, width = image.shape
    dimensions = (height, width)
    kernel = [(0, 255, 0), (255, 255, 255), (0, 255, 0)]
    kernel = np.array(kernel, 'uint8')
    compliment = image ^ 0xFF
    array = np.zeros(dimensions, 'uint8')
    array[known_pixel] = 255
    while True:
        result = np.bitwise_and(cv2.dilate(array, kernel), compliment)
        if np.array_equal(result, array):
            break
        array = result
    return np.bitwise_or(array, image)
```

Lastly, we combine the colored, largest lines and circles we found in one image to show our results. We chose the grey background color of RGB (100, 100, 100) for contrast issues. This is the final result we ended up with:



Final result

This is a simple flowchart demonstrating the processing steps done to this image: