**PROJECT A EXPLANATION:**

The code that was given was unfinished and had missing methods in the java script. Below shows each of the code implementation of java in Single Thread and Multi Thread:

For single thread, the image is processed sequentially. Here are some of the snippets that is related to this type of implementation:

**public** **static** BufferedImage convertTo(colourImage imgStruct) {

**int** width= imgStruct.width;

**int** height= imgStruct.height;

**short**[][][] pixels= imgStruct.pixels;

BufferedImage greyimg= **new** BufferedImage(width, height, BufferedImage.TYPE\_INT\_RGB);

**for** (**int** y= 0; y < height; y++) {

**for** (**int** x= 0; x < width; x++) {

**short** red= pixels[y][x][0];

**short** green= pixels[y][x][1];

**short** blue= pixels[y][x][2];

// here the grey scale is calculated with the given requirements

**int** greys= ((red >> 16) & 255) + ((green >> 8) & 255) + (blue & 255);

**int** rgbgrey= (greys << 16) | (greys << 8) | greys;

// the calculated grey scale value is now applied to the new image

greyimg.setRGB(x, y, rgbgrey);

}

}

**return** greyimg;

}

Here the ‘convertTo()’ method takes the ‘colourImage’ class and converts the image to a greyscale. This method iterates each pixel of the image calculating the greyscale values. These calculations are then applied to the newly created image file (‘Wr’) later. This is all single thread implementation and is all run sequentially.

**public** **static** BufferedImage histogramequalize(BufferedImage image) {

**int** width= image.getWidth();

**int** height= image.getHeight();

**int**[] histogram= histogramcalc(image); // histogram calculation

**int**[] cumulativehistogram= calccumulativehist(histogram);

BufferedImage equalizedimg= **new** BufferedImage(width, height, BufferedImage.TYPE\_INT\_RGB); // histogram equalization is applied on the new image after creation

**int** totalpixels= width\*height;

**for** (**int** y= 0; y<height; y++) {

**for** (**int** x= 0; x<width; x++) {

**int** pixel= image.getRGB(x, y);

**int** greys= pixel & 255;

**int** equalizegrey= (**int**) (cumulativehistogram[greys]\*255.0 / totalpixels); // histogram equlization applied to grey scale value

**int** eqRGB= (equalizegrey << 16) | (equalizegrey << 8) | equalizegrey;

equalizedimg.setRGB(x, y, eqRGB); //new grey scaled pixel into the new image 'Wr'

}

}

**return** equalizedimg;

}

This is also another snippet of a single thread implementation of the program where the ‘histogramequalize’ method applies the histogram equalization of the calculated greyscale image. Not only does it calculate the histogram but also calculates the cumulative histogram of the grey scaled image therefore, histogram equalization is applied to each pixel from the cumulative histogram results. This is again another single thread implementation and therefore the operation is executed sequentially.

The next type of implementation is multi-threading. This type of implementation aims to improve the performance when the image is processed independently by each thread separately. These can be modified based on the available hardware.

Firstly, in multi-threading there are many ways this can be implemented such as callable method is created for encapsulating the image processing for specific regions. These types include greyscale conversion, histogram calculation and histogram equalization. Another way is by enabling parallel processing. Here the image is divided into smaller regions. This then processes each region independently by separate threads.

In conclusion, the performance is better in multi-thread implementation than in single-thread implementation due parallel processing. However, this performance depends on many factors such as the image size, hardware capabilities, and available processing cores. In addition, not to mention that many considerations or compromises can occur such as thread synchronization and the different part of the program competing for the same resources (resource contention).

Overall, both implementations of the threads (single and multi) each provides its advantages and its different approaches to image processing. In short, the single thread sequentially executes tasks whereas multi-threading uses different features such as parallel processing which improves its performance.

Below are the before and after results of the image that was given (‘Rain\_Tree.jpg) and the newly created image (‘Wr.jpg’) using the single threading implementations:

**Rain\_Tree**

A large tree with no leaves

Description automatically generated with low confidence

**Wr**

**AFTER**

**BEFORE**