Image processing

A Web App written in JavaScript that combines various image processing techniques and methods built with React.

Luke Ellul

Introduction

This project uses JavaScript and the power of the browser to manipulate images using the techniques provided in class. ReactJS is used to design the UI. The README file located in the root directory of the project contains installation instructions and some other technicalities.

The main directory is divided into the following directories:

* **src:** contains the source code of the web app
* **build**: contains static files (JavaScript, HTML and CSS files) that can be served to the user
* **public:** contains assets used by the web app such as images used
* **node\_modules**: contains all the dependencies that are used by the project (this directory appears after you issue **npm install**)

The **src** directory contains a file called **App.js**. This is like the **main** file in a traditional app. It combines all the components together of the web app to show you the page you see when you run the web app.

Now, the way the app works is you have a **state** that is shared among all components. This state is a simple JavaScript object. The state holds various information about the web app but most importantly, it holds information about the image data of the original image and the image data of the processed image. By image data, we mean an array representing the pixels of an image. So that’s the state.

The various components representing different parts of the web app, such as buttons, sliders, etc…they dispatch actions that update the **state**. For example, the grayscale button gets the image data of the original image from the state, processes that image data, and updates the state with the processed data. The actual original image and the processed image are (magically) connected to the state and they are each connected to their image data property. So, when the processed image data in the state is updated with new image data, the processed image changes accordingly (because it’s connected to the image data). Same with the histogram chart that keeps updating itself with every modification to the processed image. The histogram is connected to the image data in the **state** and therefore, when the state is updated the histogram is updated too!

This magic stuff is actually handled in the **redux** folder in the **src** directory. The **ui** directory contains **React** components that render the UI. For example **PointProcessing.js** in the **ui** folder contains components that render the buttons that do point processing modifications on the image.

Techniques

All the techniques listed in the Assessment spec except for Line/Edge detection and Morphology were implemented in the web app. The reason why I left the last two out is because I ran out of time.

These techniques are stored in the **Functions** folder located in the **src** directory. Each technique is represented as a function that takes an array of image pixel values as an argument, along with other parameters, and returns a processed array of image pixels. These functions are then used by the components mentioned above.

It’s important to note that an image in this context is not represented as a 2-dimensional array but rather as a 1-dimensional array of numbers. Each number represents an RGB value (or alpha). So 1 pixel is represented by 4 numbers, the first number represents the **R**ed intensity, the second number the **G**reenintensity, the third number the **B**lue intensity and the fourth number represents the alpha value of that pixel. For example, this array represents an image with 4 pixels:

[23,45,67,255,23,87,34,255,34,89,09,255,23,56,57,255]

To process this 1-d array like a 2-d array the following technique is used:

Let’s say we want to read the blue component’s value of a pixel at column 200, row 50 of a 2-d array of an image. We get that value out of the 1-d array as follows:

blueComponent = **imagedata**[((**50** \* (**width** \* 4)) + (**200** \* 4)) + **2**];

Where **imagedata** is the 1-d array and **width** is the width of the image.

Point Processing

Functions that perform point processing operations are located in **PointProcessing.js** in the **PointProcessing** directory.

Invert

The invert function just loops over each pixel component and subtracts it from 255.

Grayscale

I used the average method for the grayscale function. The function loops over each pixel and for each pixel it calculates the average of all its color components, then it sets each color component of that pixel to the calculated average value.

Binarize

The binarize function accepts a threshold and checks of each component of every pixel is above the threshold. If it is, then it sets the component to 255, else it sets it to 0.

Otsu Binarize

The Otsu Binarize function performs automatic binarization by finding the threshold automatically and applying the above **Binarize** function. To find this threshold we use the have to find the Otsu Level and this is done using the **otsuLevel** function that takes the histogram counts of an image as input and outputs the Otsu level. The code for finding the Otsu Level was converted to JavaScript from Matlab code found on this Wikipedia article about the Otsu Method: <https://en.wikipedia.org/wiki/Otsu%27s_method>.

The histogram counts of an image are found using the **histogramGrayLevel** function that is found in **HistogramProcessing.js** under the **Histogram** directory.

Log Transform

Log Transform is performed by iterating over each pixel and setting it’s components’ values to a constant **c**, multiplied by the log of the pixel intensity plus 1.

Power Transform

Power Transform is applied in a similar fashion to Log Transform but instead, we multiply a constant **c** with the intensity of a pixel raised to a parameter **y**.