Image Pipeline Report

https://github.com/aryanc1027/790_ISP

Introduction

This assignment dives into image processing implementing the fundamental principles of image formation and editing. The coding part of this assignment is completed in python and uses many builtin and imported libraries such as *skimage, numpy, matplotlib, and colour_demosaicing* to manipulate images, perform complex mathematical operations, and allow for this project to construct a basic image pipeline. The fully fledged out version of this pipeline is essential for transforming RAW images into visually appealing images suitable for display, and it is used in many modern DSLR cameras.

Part 1: Developing RAW Images

The very first part of this project deals with using the CLI tool *dcraw* to convert the RAW image file into a *.tiff* format. In specific, the command *dcraw -4 -d -v -w -t <RAW_filename>* is run to produce the required *.tiff* file that retains the image's raw data for further processing. After this function is run, important details about the RAW image file is produce such as:

- *<black> = 0*
- *<white> = 16383*
- *<r_scale> = 1.628906*
- *<b_scale> = 1.000000*
- *<g_scale> = 1.386719*

After these values are found, the python code calls the *skimage* function *imread* to load images into the python code. These functions change the RAW image into a numpy 2D-array of unsigned integers. For the specific image used in this project, the dimensions are found to be a width of 4284 pixels and a height of 2844 pixels, and the bit depth of the image is 16 bits per pixel. Once we have our 2D array, the next part is to linearize the array to correct for dark noise and over-exposure by using the *clip* function to map all negative values to 0 and all values greater than 1 to 1.

One of the most critical steps in image processing is identifying the correct bayer pattern, as it determines how colour information is interpreted and reconstructed from the sensor data. The python function I defined looks at the top left 2x2 square of the image file, analyses the mean values of the 2x2 square, and compares them against a predefined set of mean values expected for each Bayer pattern. In the case of our testing image, the pattern *'rggb'* was selected. Furthermore, when we save the resulting image from the bayer pattern function, we got this image:

which further shows us that the pattern *'rggb'* should be selected.

The next step in the pipeline is the white balancing through different channels. The white balancing algorithm assumes the average of all the colours in the scene and then uses that assumption to balance all the colours to their desired values. The grey balancing algorithm operates under a similar premise but focuses on achieving a balance where the average of the red, green, and blue channels across the image approximates a neutral grey. These two algorithms differ from the last as it doesn't take into account the lighting conditions. The last algorithm uses the camera's white balancing presets to apply a more tailored approach by adjusting the image's colour channels according to the recorded *<r_scale>, <g_scale>,* and *<b_scale>* values. After looking at all three balancing algorithms, and assuming that the *scale* values are accurate, the best image was found using the camera balancing algorithm. All three of these algorithms went through a manual white balancing phase as well and the final *manually_white_balanced_camera_algorithm* image was deemed the best image, as it resembles the original image the best without being too flushed out or too bright.

Now that we have a fully fledged RGB image, we need to change the colours of the image in order to make it look like how we want to. The below matrices are balanced images after they go through this correction using linear algebra. This correction applies the following linear transformation to the RGB vector $[R\_cam,\ G\_cam,\ B\_cam]$ to each pixel of the demosaiced image.

$$\text{White balanced image after color space correction}\begin{bmatrix} 1 & 0.62428032 & 0.15198579 \\ 0.72704274 & 1 & 0.07840931 \\ 1 & 1 & 0.32774511 \\ \vdots & \vdots & \vdots \\ 0 & 1 & 1 \end{bmatrix}$$

$$\text{Grey balanced image after color space correction}\begin{bmatrix} 1 & 0.62428032 & 0.15198579 \\ 0.72704274 & 1 & 0.07840931 \\ 1 & 1 & 0.32774511 \\ \vdots & \vdots & \vdots \\ 0 & 1 & 1 \end{bmatrix}$$

$$\text{Camera image after color space correction}\begin{bmatrix} 0.593869084 & 0.149479233 & 0.0363918549 \\ 0.174084924 & 0.49057043 & 0.0187745203 \\ 0.302689875 & 0.232339379 & 0.0594079891 \\ \vdots & \vdots & \vdots \\ 0 & 1 & 0.568028261 \end{bmatrix}$$

Once we have the correct RGB image, we need to adjust the brightness. This process involves linearly scaling the 16-bit, full-resolution, linear RGB image to correct its brightness for display purposes. After much trial and error, the recommended percentage where my image looked the best was 25% (*mean of .25)*. Following brightness adjustment, gamma encoding is applied using a non-linear transformation to ensure the image displays correctly. This process adheres to a sRGB standard image type.

Finally, we're onto the step of compression which takes the final sRGB image and compresses it into various *.png* and *.jpg* images. The image is to be saved in .PNG format, which typically does not apply compression, and in .JPEG format with a quality parameter set to 95, which introduces some level of compression. The compression ratio that was applied is calculated by comparing the file size of the uncompressed image to the file size of the compressed image. Based on the provided image, the compression ratio for a JPEG image with the quality parameter set to 95 is 22.63. This ratio remains the same even when the JPEG quality is set to 100, which suggests that there is no additional compression occurring at the highest quality setting for JPEG in this case. Now, this part is subjective as *"indistinguishable from the original"* is a very subjective statement but based on my observations, the lowest acceptable compression ratio with a quality parameter of 100 is 16.55.

Moving on from the automatic image pipeline, the next part of the code sets up the manual white balancing which normalises RGB channels based on a manually selected white patch. This was done in a two step process. First, the code reads the original image, translates it into a double precision array and clips the values like we mentioned before. It then displays the image and allows the user to click on two points using *plt.ginput(2)*. The function then calculates

the coordinates of the selected region and returns the patch coordinates to be used in the white balancing function. This function balances the image and then saves the original and the white balanced image to their respective locations. After multiple trial and error, the patch that works the best for the test image is the top left corner. This can vary based on the particular image used.

The last part of the pipeline was to use the *dcraw* CLI tool to take the raw image and convert it into its final form. The exact code that was used is: *dcraw -v -w -q 3 -o 1 -g 2.4 12.92 data/baby.nef.*

- *-v*: Verbose output
- *-w*: Camera white balance
- *-q 3*: Highest quality demosaicing
- *-o 1*: Converting to the sRGB colour space
- *-g 2.4 12.92*: Applying a specific gamma correction
- *data/baby.nef*: RAW image file path

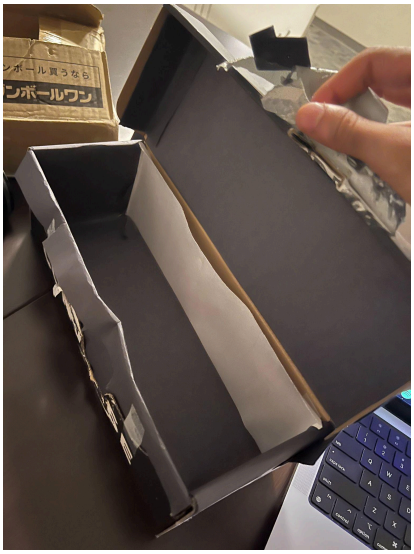
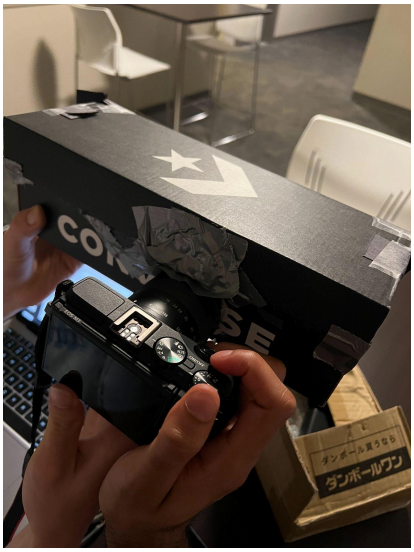
Looking at all the final developed images, I believe that the *dcraw* version looks the best as it correctly performs all the necessary steps to the image pipeline. A close contender would be the manually white balance image using the camera balanced channel, but the brightness is a bit off which would have to do with the manually chosen location performed during the manual white balancing.

**** All images discussed in part one are in the data folder of the github repository ****

Part 2: Developing RAW Images

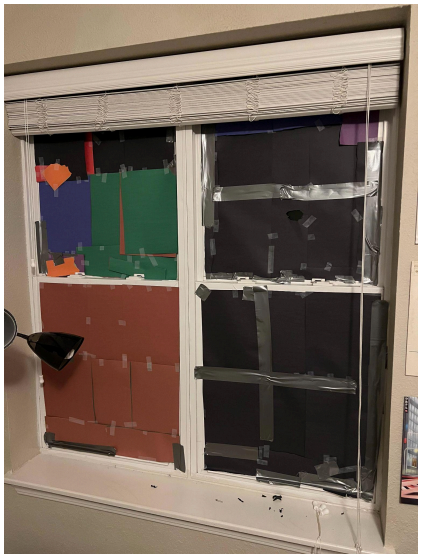
The second part of this project throws one into the world of the basic principles of image formation and exposure control by constructing a pinhole camera and a camera obscura. This hands-on experience gives practical experience with the fundamentals behind image capturing and manipulating light, and your environment, to create the most optimal scene for an image to be taken.

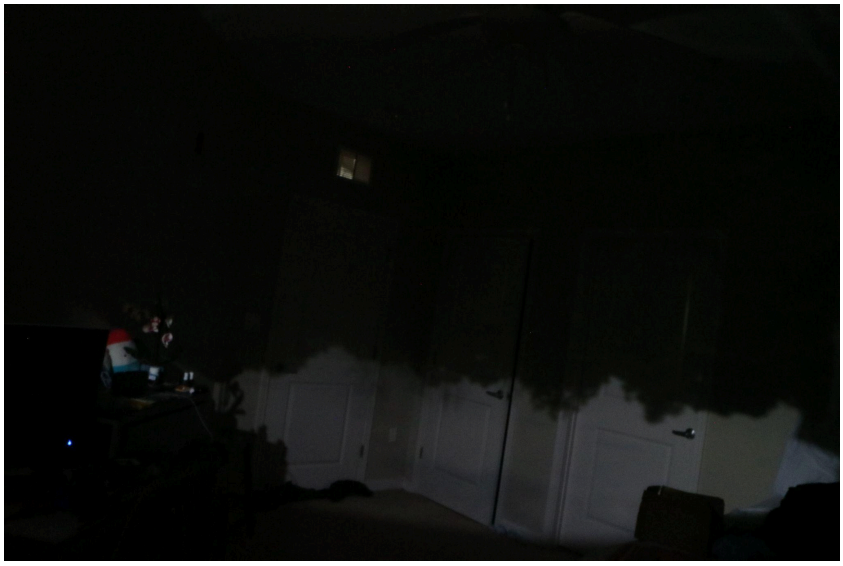
Shown below is the first step to this part, the pinhole camera:

Looking at these two images, we can see some very clear design choices made in order to get the best images possible. First off, we covered the inside, besides the back wall, in all black paper to prevent any extra light from reflecting. The back wall was then covered in white paper which serves like a screen for the to be placed. Furthermore, the box was chosen carefully to ensure that no extra light could come through providing us with the best images possible. The coroners and any places that light seems to leak in through were covered in duct tape and extra construction paper to form the most “light-proof” box possible. A small pinhole was poked in order to let a small amount of light in (this is where the image comes from). Finally, a camera hole was made in the side of the box and covered in duct tape to ensure that a minimal amount of extra light comes in. Looking at the pinhole camera, the screen size is at max around a 4 x 12 screen. The only caveat is that the entire screen was not used to take the image so the actual screen size may vary. Next, the distance between the screen and the camera (the focal length) is around 4 inches. Lastly, the field of view of the image is any light (so any area) that enters the pinhole. For the pinhole itself, the different diameters that were used are .52, 1.3, and 5 millimetres. As the diameter got bigger, more light got let into the hole which made the image more bright. Unfortunately, the extra light distorted the image and made it more blurry.

Finally, the last part of this project was to design a camera obscura which is when you make a room as dark as possible by covering up a window and shutting off all other lights. Then, one creates a hole in the covered window and the light coming in creates an inverse image of anything outside the window. Shown below is the camera obscura and a “picture” taken with the “camera”:





Clearly the images show a completely covered window and the corresponding background “image” that has the outside forest. If you look at the image on the right, it is clear that there is a treeline captured on the wall.

Shown below are 9 different images taken with the pinhole camera: