Assignment 1: Raw Image Decoder

Guilherme Gomes Haetinger - 00274702

Proposal: *The goal of this assignment is to familiarize the students with how digital cameras acquire and process images. For this, you will be implementing a raw image decoder. A CCD is a monochromatic sensor and in order to capture color images, digital cameras use color filters on top of the CCD. To reduce cost and simplify design and construction, most cameras use a single CCD with a color filter array (CFA), such as the Bayer filter (see Figure 1). Thus, the image processing module of a digital camera has to convert the captured raw image data into a full-color image.*

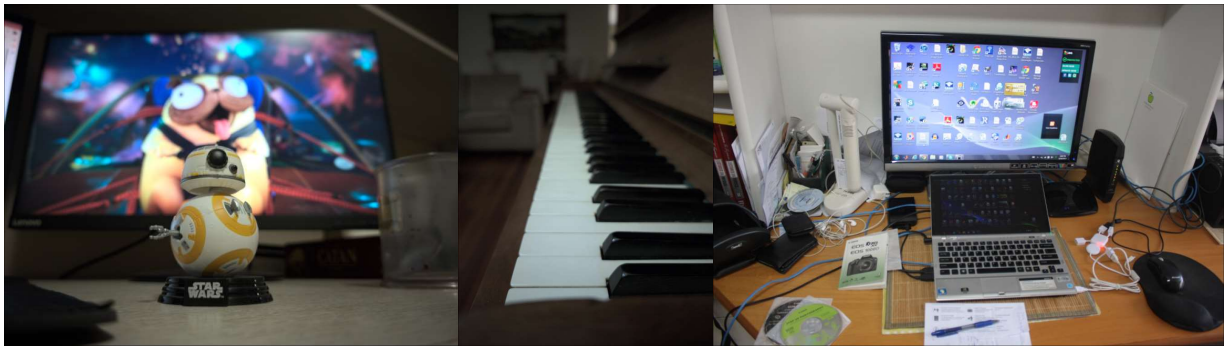
Introduction

Building the raw image decoder designated to us, students, takes essentially four steps:

- **Reading an image** avoiding all libraries that do all the work for you, *i. e.* finding a library that will give you a correct mosaic of values;
- **Demosaic the image** in a way that it respect the image mosaic pattern (which should be also fetched from the raw image metadata);
- **Apply whitebalancing** to the image in a way that we get a more real like color distribution (not too green, red or blue). This can be made through a couple of ways and we'll be exploring a single one: *Scaling Camera RGB*;
- **Apply γ-encoding** to produce a more real-like picture by enhancing the light effect in the dark areas.

I've setup three pictures to undergo this whole process. One of them was given with the assignment and the other two were taken with my cellphone, which has the ability of taking *.dng* raw photos.

The image given along the assignment was converted from *.CR2* to *.dng* by Adobe's Digital Negative converter sofware. We'll be using this as our main picture to go along the illustrative examples.



Reading Raw Images

I used **Julia** to program this whole assignment, but took advantage of Python's *rawpy* library to read raw image information. This way we can simply extract a variable `raw_pixels` from the read image.

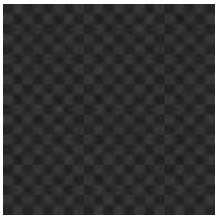
```
rawpy = PyCall.pyimport("rawpy");
```

```
function read_image_and_normalize_content(filename)
    raw = rawpy.imread("./res/dngs/$filename");
    raw_pixels = raw.raw_image;
    norm = Int.(raw_pixels) / maximum(raw_pixels);
    Gray.(norm)
end;
```

`grayscale =`

- `grayscale = read_image_and_normalize_content(filename)`

If we zoom in, we can actually see how mosaicked this image is even when monochromatic, since different colors have different influence values on the output image.



Filling in the Color Mosaic

To understand how exactly we should be filling the mosaic, *i. e.* assigning each pixel to a given color, we need to first pick a pattern out of two possible Bayer filter patterns. To do so, we use the `raw_pattern` value from *rawpy*.

The possible patterns are the following:

$$pattern_1 = \begin{bmatrix} R & G \\ G & B \end{bmatrix}$$
$$pattern2 = \begin{bmatrix} G & R \\ B & G \end{bmatrix}$$

The format defines in which diagonal the green value takes over. In our example images, we have $pattern_1$ in our assignment image and $pattern_2$ in the rest.

We can calculate the pattern for an image by computing the determinant of the `raw_pattern` matrix from *rawpy* and checking if it determinant is positive or negative.

```
function get_file_pattern(filename)
    raw = rawpy.imread("./res/dngs/$filename");
    pattern = raw.raw_pattern
    det(pattern) < 0 ? "pattern_1" : "pattern_2"
end;
```

```
scene_pattern = "pattern_1"
```

Now that we know the pattern for the image, we can simply setup the indices that populate each channel. In **Julia**, we can do that by having an array of *CartesianIndex* for each color channel.

To populate them, we can move around the image indices by moving a $2x2$ matrix and filling in the correct indices.

```
function create_color_channel_idxs(image, pattern)
    (rows, cols) = size(image);
    red_idxs = []
    green_idxs = []
    blue_idxs = []
    for i ∈ (1:2:rows)
        for j ∈ (1:2:cols)
            if (pattern == "pattern_1")
                push!(red_idxs, (i, j))
                push!(green_idxs, (i, j + 1))
                push!(green_idxs, (i + 1, j))
                push!(blue_idxs, (i + 1, j + 1))
            else
                push!(green_idxs, (i, j))
                push!(red_idxs, (i, j + 1))
                push!(blue_idxs, (i + 1, j))
                push!(green_idxs, (i + 1, j + 1))
            end
        end
    end
    red_idxs = red_idxs .|> CartesianIndex
    green_idxs = green_idxs .|> CartesianIndex
    blue_idxs = blue_idxs .|> CartesianIndex
    return (red_idxs, green_idxs, blue_idxs)
end;
```

Generating the color channel given the indices should be quite easy now. We can generate a blank version the same size of the image using `zeros`, fill it up given the `CartesianIndex` array and make sure it's properly shaped with the original dimensions.

```
function setup_color_channel(idxs, values, rows, cols)
    img = zeros((rows, cols))
    img[idxs] = values[idxs]
    return reshape(img, (rows, cols))
end;
```

Doing this for every channel should be as straightforward as the following function:

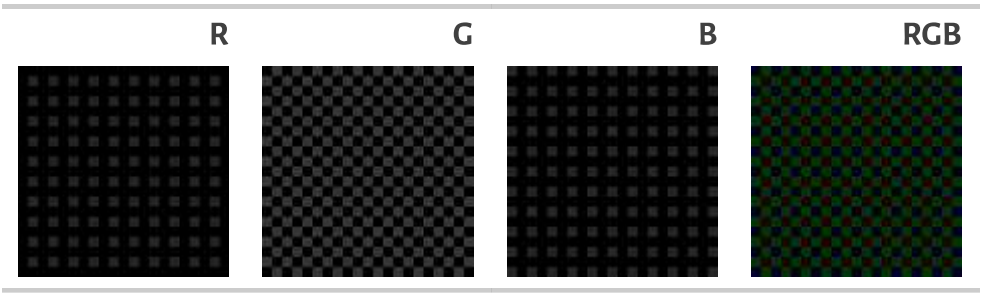
```
function setup_color_channels(image, red_idxs, green_idxs, blue_idxs)
    (rows, cols) = size(image);
    R = setup_color_channel(red_idxs, grayscale, rows, cols)
    G = setup_color_channel(green_idxs, grayscale, rows, cols)
    B = setup_color_channel(blue_idxs, grayscale, rows, cols)
    return (R, G, B)
end;
```

From now on, we'll be working on channels separately as they are completely independent from each other. Once we have a decent result, we can then put them together with the `colorview` function. Thus, we can retrieve the color channels from the pattern using the code below.

```
(red_idxs, green_idxs, blue_idxs) =
    create_color_channel_idxs(grayscale, scene_pattern);
```

```
(R, G, B) = setup_color_channels(grayscale, red_idxs, green_idxs, blue_idxs);
```

We can see how the pattern is reflected by plotting the values of each channel!



```
colorview(RGB, R, G, B)
```

Demosaicking

Now that we have the separate channels, we can move on to Demosaicking. This process is done by finding the values of the missing pixels in each channel. We'll be using Bilinear Interpolation to do so. This can be achieved by applying matrix convolution to each channel. The kernels used for this will be the same for the red and blue channel, but will be different for the green channel, which has more pixels in the image.

It's easier to think of them in the following way:

- The green channel's missing pixels will be either red or blue, both of which have green neighbors in all points but the diagonals. Hence, their kernel will be missing weights for the corners.
- The red and blue channels need to look for either pixels on the diagonal, where they need to weight 4 pixels, or look for pixels on the sides, where they'll only have 2 pixels weighted.

Thus, they are represented as follows:

$$K_r = K_b = \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

$$K_g = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

We can apply the convolution by using the `conv` function, dividing by the defined weight (in this case, 4), clipping out the resulted borders and clamping the results to keep all values in the range $[0, 1]$.

```
conv_norm (generic function with 1 method)
```

```
function conv_norm(K, M)
    convoluted = conv(K, M)[2:end-1, 2:end-1] / 4.0
    return clamp.(convoluted, 0, 1)
end
```

conv_image

```
conv_image(R, G, B)
```

Given the three color channels, apply the respective kernels.

```
"""
    conv_image(R, G, B)

Given the three color channels, apply the respective kernels.
"""
function conv_image(R, G, B)
    K_r = [ 1 2 1; 2 4 2; 1 2 1 ];
    K_b = K_r;
    K_g = [ 0 1 0; 1 4 1; 0 1 0 ];
    R_c = conv_norm(K_r, R)
    G_c = conv_norm(K_g, G)
    B_c = conv_norm(K_b, B)
    return (R_c, G_c, B_c)
end
```

```
(R_c, G_c, B_c) = conv_image(R, G, B);
```

After applying the demosaicking process, we can see that the image is no longer as dark and green, getting more of other colors as well, *e. g.* the blue of the desktop background and the purple on BB8's core. However, it's pretty clear things are still a long way from perfect. Green is predominant and dark objects have almost no inner contrast.

demosaicked =

- `demosaicked = colorview(RGB, R_c, G_c, B_c)`

White Balance

To achieve more real-like colors, we rely on a technique called **White Balance**. This technique is based on normalizing each pixel given the assumption that another pixel `P_w` of color `[R_w, B_w, G_w]` is actually white. Therefore, we can create a transformation that maps `[R_w, B_w, G_w]` to `[1, 1, 1]`. This is thus applied to every pixel.

There are some ways of getting the coordinates to `P_w`. Here, we'll just be using a Pluto widget that allows us to click where we want the white pixel to be set. This *Interactive Example* section will only be interactive if on a running Pluto session.

apply_white_balancing

```
apply_white_balancing(image, P_w)
```

Apply white balancing to `image` when `P_w` is the considered true white color.

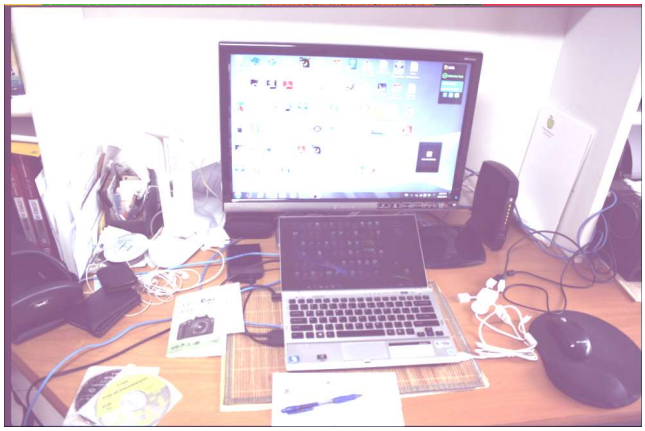
```
"""
    apply_white_balancing(image, P_w)

Apply white balancing to `image` when `P_w` is the considered true white color.
"""
function apply_white_balancing(image, P_w)
    channels = channelview(image)
    R_wb = channels[1, :, :] / P_w.r
    G_wb = channels[2, :, :] / P_w.g
    B_wb = channels[3, :, :] / P_w.b
    colorview(RGB, R_wb, G_wb, B_wb)
end
```

white_balanced =

- `white_balanced = apply_white_balancing(demosaicked, demosaicked[754, 1746])`

Results



Wall (3336x271)

Paper (2329x2482)

File (1746x754)

Charger (1298x1448)

It's clear that the picked pixel has a huge influence on how the image will turn out. Not only is the overall brightness of the image influenced, but also the color tone. The two best results, in my opinion, would be the *File* and the *Charger*. The *File* is a bit darker but holds contrast in the desktop with a somewhat warmer tone. The *Charger* holds a colder tone and has more contrast on the darker areas of the image.

BB8

BB8's core (2796x3034)

The Pug's eyeball (3043x1094)

Piano

Background Light (175x1780)

Lightest Key (3027x2571)

These two extra example images have faulty setups for this *pick your pixel* algorithm: both of them don't have a clear white element. Thus, the colors can assume tones that aren't real-like even with white-balancing. Although I was able to get a better result on *BB8*'s picture by selecting the *Pug's eyeball*, it got too dark (which will be fixed by gamma encoding) and too warm. The *Piano* picture was taken in a very dark room with warm lighting, so selecting the white-ish key means every other pixel will have to increase their blue value and so the scene will be a bit bluer.

Gamma Encoding

The human eye is more sensible to changes in dark areas of pictures. This variation isn't present in digital picture capture, which captures the signals linearly. Thus, we must find a way to compensate and enhance dark signals while compressing bright ones. This can be done by raising every value to a γ. This process is called *Gamma Encoding*. We'll be doing this just the same as the other functionalities, processing per channel.

apply_gamma

```
apply_gamma(image, γ)
```

Apply gamma correction given an `image` and a `γ`.

```
"""
    apply_gamma(image, γ)

Apply gamma correction given an `image` and a `γ`.
"""
function apply_gamma(image, γ)
    channels = channelview(image)
    R_gc = (channels[1, :, :]) .^ γ
    G_gc = (channels[2, :, :]) .^ γ
    B_gc = (channels[3, :, :]) .^ γ
    colorview(RGB, R_gc, G_gc, B_gc)
end
```

```
apply_gamma(white_balanced, 0.75)
```

As a first impression, we can see how the picture isn't as dark and uniform anymore. We can see more of it and the colors are more natural. The result isn't perfect, but it's still much better than what we had as an input.

Results

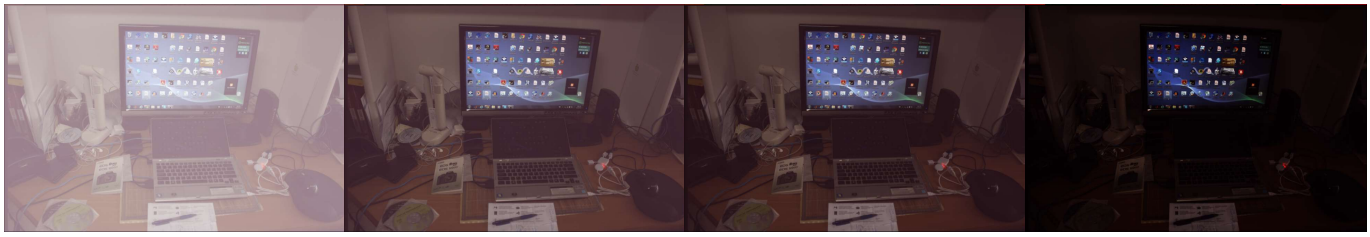
The results below show how much the γ variation can change the picture.

The computer with a high γ of 2.0 shows the monitor in high contrast with colorful icons, but with a dark and almost uniform background. When applying a very small γ, however, we see that all the picture becomes unrealistically bright. We then settle for a small γ that helps us see the laptop monitor is displaying something while also keeping the overall contrast of the picture.

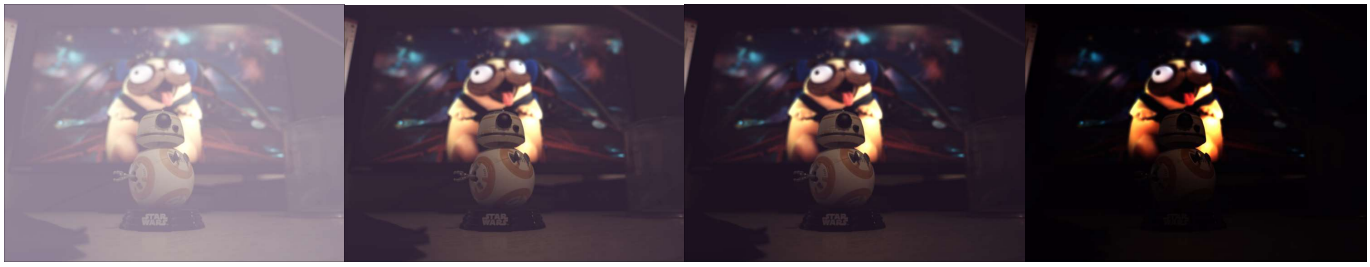
The same goes for *BB8*'s picture, there the high γ offuscates the toy and highlights the highly contrasted pug, whereas with $\gamma = 0.75$ they both seem to have the same amount of contrast, arranging the picture in a much better way.

The Piano picture still shows to have a bad result. With the failed attempt of whitebalancing, the picture looks extremely bluish. The $\gamma = 1.0$ seems to give out the best result in terms of visibilty, though. This is probably due to the fact that the image was captured in a very dark room. Even though this isn't the best taken photo out of the three, the piano's $\gamma = 2.0$ result is a very artistic output in my opinion.

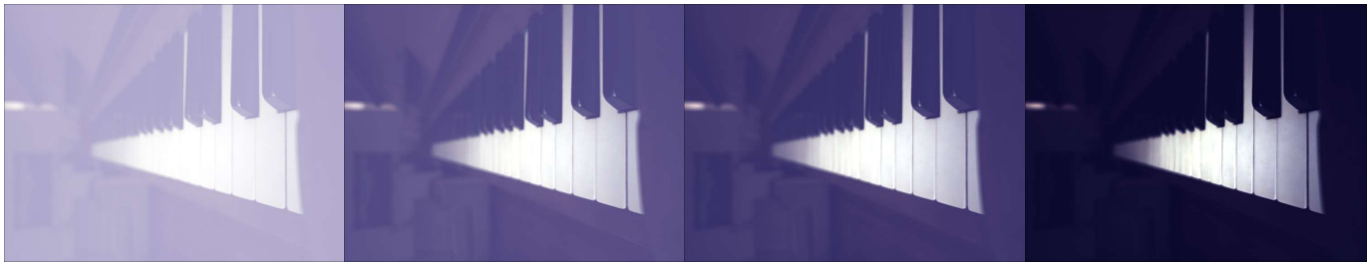
γ = 0.25 γ = 0.75 γ = 1.0 γ = 2.0



γ = 0.25 γ = 0.75 γ = 1.0 γ = 2.0

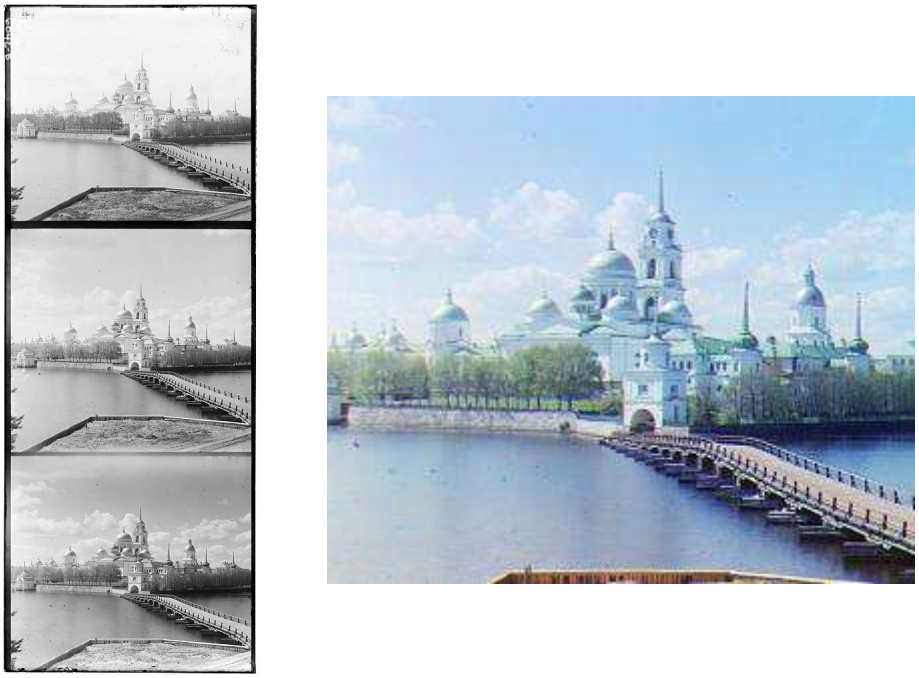


γ = 0.25 γ = 0.75 γ = 1.0 γ = 2.0



Conclusion

This assignment was very interesting and insightful to take. Not only was I able to relearn the things we talked about in class, but I was able to put them to practice, understanding the influence of each variable in the process of creating a digital photography, from taking a picture to producing the proper color channels for it. I think the most fascinating was to finish it and think *"how did people take pictures that look better than the ones I produced on the early 20th century?"* Creative people like **Sergei Mikhailovich Prokudin-Gorskii** took pictures and elaborated concepts that would precede these optimizations and post-processing factors we have today.



Set of images taken with separate filters `R`, `G`, `B` (left) reconstructed by me into a full colored picture (right) using alignment algorithms. Pictures were taken by **Sergei** with his three lensed camera.

As a retrospect to what I was able to achieve with my examples, I'd say that both my pictures could look better if I had added a piece of white paper on the picture, just to make sure I had a proper white reference. Other than that, maybe I could have achieve better white balance results on the Piano image if I had approached it with the Gray world algorithm, which finds a proper pixel instead of having the user select it. Maybe in the future I can go back to this and apply more complex and automatic processes to image post-processing.

It's funny to see how complex digital image capturing actually is. I grew up taking pictures with my father's camera and then my phone, and then sharing it with whomever I wanted in any format I wanted to do so. This assignment shows the depth that digital image capturing has, as we've seemed to have just scratched the surface since I am not able to say confidently that my images are real-like. I'm excited to see what's next.

Appendix: Use this on a running Pluto instance!

Interaction

Select a photo

scene_raw.dng

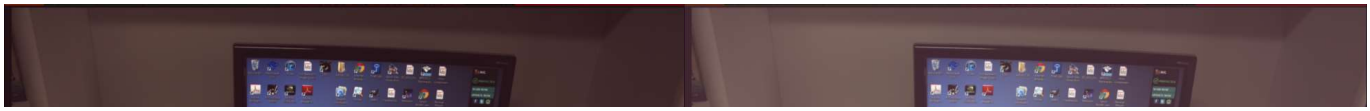
true_white =

0.7

```
@bind γ Slider(0:0.1:3; show_value=true, default=0.7)
```

Click on the image on the top

Adjust γ with the slider



Setup

ImageClicker

```
ImageClicker(img_src, id; default=Dict("x" => 0.5, "y" => 0.5))
```

Creates a widget that takes in the `cell_id` of a pluto displayed image and a unique `id` to enable other `ImageClicker`s. It can also have a default coordinate set.

clicker =

```
clicker = @bind coord ImageClicker("a2e61c94-415d-4018-8437-098de32364d4", "whitebalance_img")
```

```
white_applied = apply_white_balancing(demosaicked, true_white);
```

```
gamma_applied = apply_gamma(white_applied, γ);
```

Environment Setup

Libraries

```
using Images, PyCall, HypertextLiteral, PlutoUI, LinearAlgebra, DSP, Latexify
```

Python environment fix

```
link_dir = "/usr/sbin/python"
```

```
python_environment = "/usr/bin"
```

```
pushfirst!(
    pyimport("sys")."path",
    joinpath(python_environment, "lib/python3.9/site-packages")
);
```