Image & Video Processing: Assignment 4

Kyla Kaplan
May 2024

Contents

1 Color Palette Extraction from Images

1 Linear RGB and sRGB Color Palettes

1.1 Loading Image in RGB space

Initially, we load the image and convert it from sRGB to the linear RGB space. This is done via an implemented method `srgb_to_linear(srgb)` (see below). The image itself was loaded and normalized to a range between zero and one after being converted to double. In order to get the RGB image, we invert the assumedly applied gamma correction. The sRGB palette in typically more vibrant due to this gamma correction, whereas a linear RGB palette provides a more accurate color representation, which is more useful in application where we need a more precise color differentiation.

```
function linear_rgb = srgb_to_linear(srgb)
    % Apply inverse sRGB gamma correction
    threshold = 0.04045;
    linear_rgb = srgb / 12.92;
    mask = srgb > threshold;
    linear_rgb(mask) = ((srgb(mask) + 0.055) / 1.055) .^ 2.2;
end
```

1.2 Color Clustering

To prepare the image for clustering, we need to reshape the data that is contained within it.

```
    img_reshaped = reshape(img, [], 3);
    linear_img_reshaped = reshape(linear_img, [], 3);
```

We reshape and flatten both of the 3D image arrays into 2D arrays, where each row represents a pixel and each column represents its respective color channel (`R`, `G`, `B`).

Now, it is ready to cluster according to K-means clustering, as instructed in the assignment — using the in-built `k-means()` clustering function, divided into 7 clusters.

```
    num_clusters = 7; % as given
    [idx_srgb, centroids_srgb] = kmeans(img_reshaped, num_clusters, 'Distance', 'sqeuclidean',
        'Replicates', 5);
    [idx_linear, centroids_linear] = kmeans(linear_img_reshaped, num_clusters, 'Distance',
        'sqeuclidean', 'Replicates', 5);
```

As a result, we receive the reshaped sRGB image in 7 clusters, with the cluster indices being `idx_srgb` and the centroids themselves being `centroids_srgb`.

1.3 Image decomposition into base color layers

After having extracted the centroids of each cluster as the "average" or "representative" for our palette, we create 7 layers for each cluster, setting all other pixels to black, and then we try to draw a color strip with the centroids of each cluster.

```
    layers_srgb = zeros([size(img), num_clusters]);
    layers_linear = zeros([size(img), num_clusters]);
```

These arrays will store the layers for each cluster in both sRGB and linear RGB spaces.

```
for k = 1:num_clusters
    mask_srgb = reshape(idx_srgb == k, size(img, 1), size(img, 2));
    mask_linear = reshape(idx_linear == k, size(img, 1), size(img, 2));
    for c = 1:3 % create the layers
        layers_srgb(:,:,c,k) = img(:,:,c) .* mask_srgb;
        layers_linear(:,:,c,k) = img(:,:,c) .* mask_linear;
    end
end
```

The rest of the code can be found in `ex1.m`. Displayed below are the results of the code.

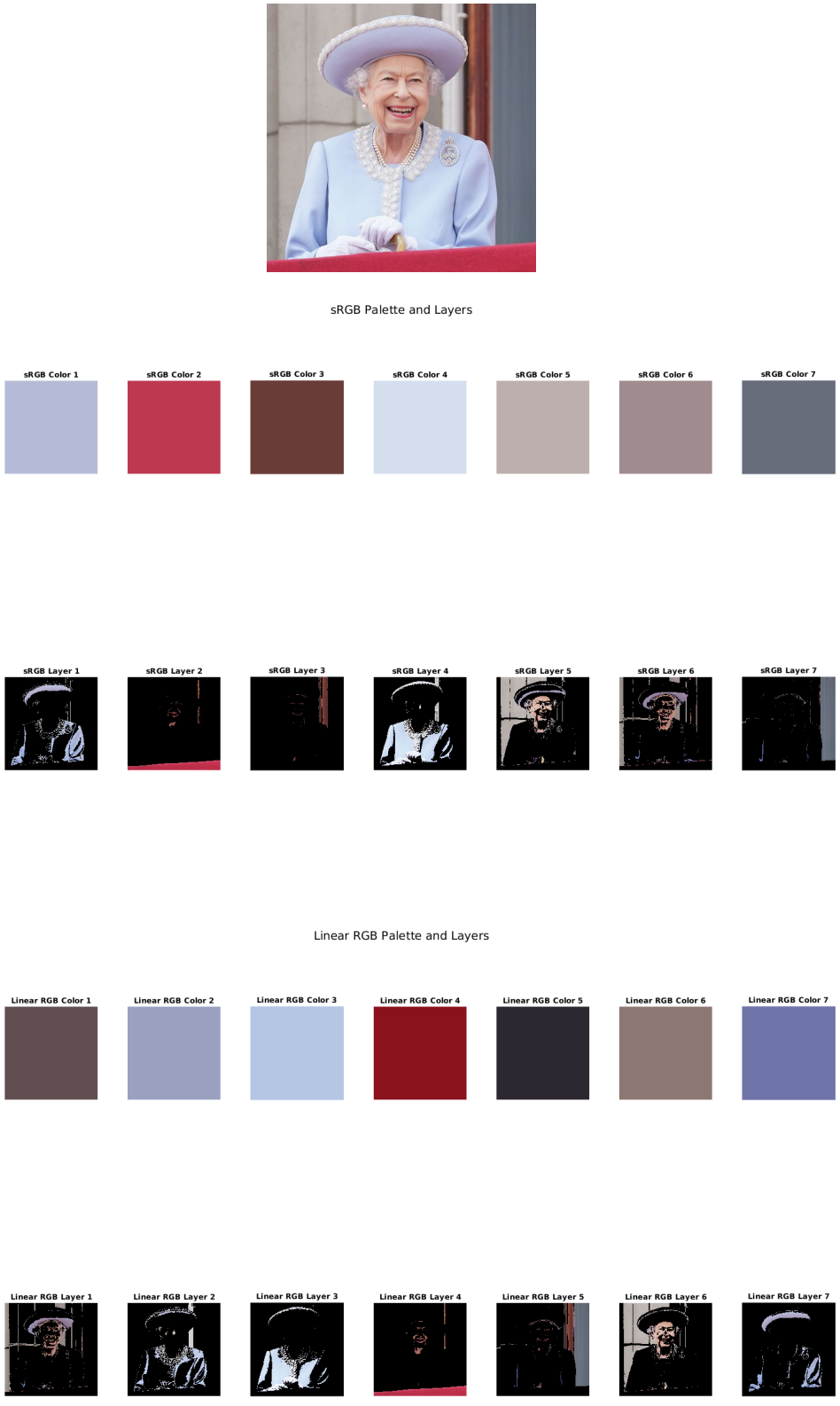

Figure 1: Original image (`queen.jpg`); sRGB vs. Linear RGB Palettes and Layers

2 CIELab Color Palette

As the assignment states: "perceptually uniform color spaces, such as CIELab, exploit computational models of human perception to create color spaces where geometric distances roughly correspond to similar distances in the perceptual space". Since the RGB space is not the most optimal for when determining the notion of distance as it does not align well with the human-perceived perceptual difference/distance between colors, the CIELab color space is suited better for human perception as it is constructed in a way that similar colors are perceptually "closer" and dissimilar ones are "farther". By clustering colors in the CIELab color space, we can obtain clusters whose points are close to each other in the CIELab space, resulting in a better color palette than the one obtained by just clustering colors in the RGB space. It can be expected that the colors of the palette are more distinct that the ones that are further than each other. The steps are similar to the first part of the exercise, and additionally this part of the exercise is also included in the file `ex1.m`.

```
lab_img = rgb2lab(img); % convert to cielab
lab_img_reshaped = reshape(lab_img, [], 3);
...
[idx_lab, centroids_lab] = kmeans(lab_img_reshaped, num_clusters, 'Distance', 'sqeuclidean',
    'Replicates', 5);
...
for k = 1:num_clusters
    ..
    mask_lab = reshape(idx_lab == k, size(img, 1), size(img, 2));
    for c = 1:3 % create the layers
        layers_lab(:,:,c,k) = lab_img(:,:,c) .* mask_lab;
    end
    ...
    % note how we convert cielab to the srgb color space to visualize the results
    layers_srgb_lab(:,:,:,k) = lab2rgb(layers_lab(:,:,:,k));
end
```

The results are displayed below in Figure 2.

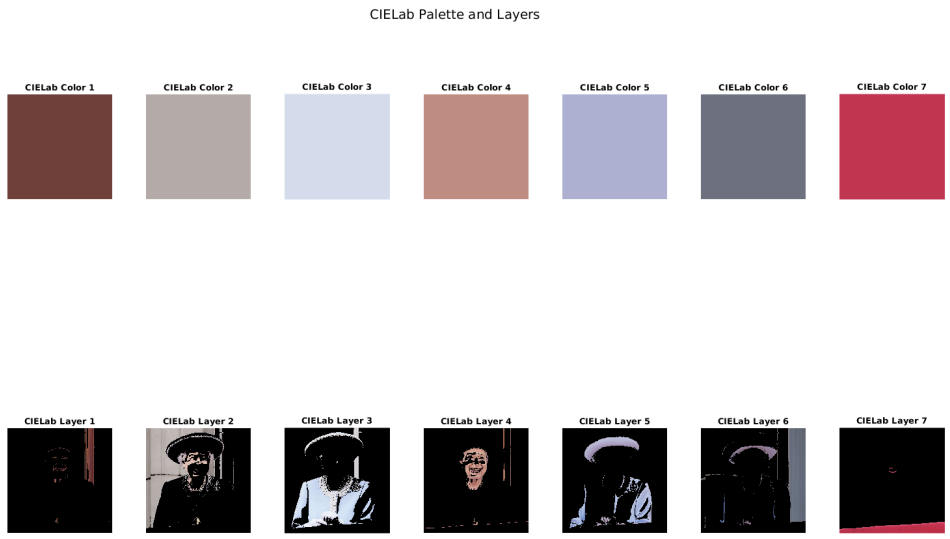

Figure 2: CIELab Palette and Layers

As an "artistic look" on one of my images, I sought to "pastel-ize" the image, by making it a more pinkish-purplish hue. This is done by:

1. Increasing the lightness of the first cluster
2. Shifting the hue of the second cluster
3. Increase the saturation of the third cluster
4. Convert the modified layers back to sRGB for displaying

These are the final results using my image (I ran the whole code with a different file at the beginning and only saved the result):

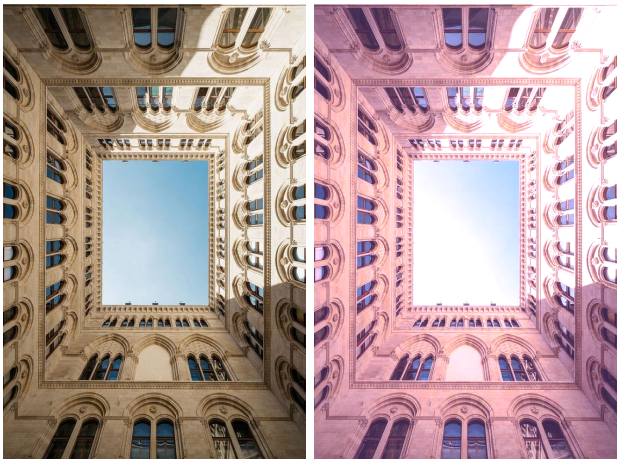

Figure 3: My own image, modified, with an "artistic touch"

This code is also included towards the bottom of `ex1.m`.

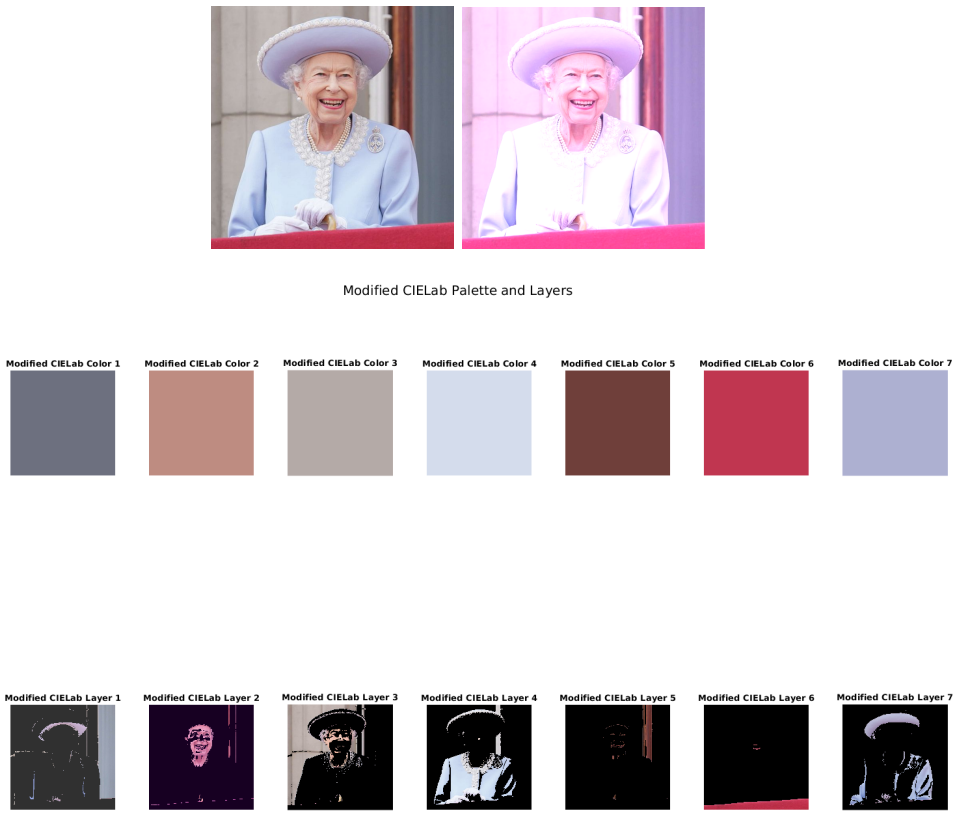

Figure 4: Modified image (`queen.jpg`) and modified CIELab Palettes and Layers

2.1 With the Bonus

Although the image in Figure 3 looks to be the desired effect I was seeking, the general "quality of the palette" could be improved by reducing "outlier colors", as the `kmeans` algorithm is quite sensitive to outliers. My attempt at a "mini-Lightroom" is included within `ex1_bonus.m`.

2 Color Quantization and lookup tables (LUTs)

For this exercise of the assignment we are asked to implement a Lookup Table (LUT) in order to map the colors of an image as previously, but this time to a reduced color palette. Similarly to how we used the `k_means` algorithm to cluster the colors (color quantization), a part of the return of that algorithm results in giving us directly the "id's" of the cluster indexes that belong to the Lookup Table. Each pixel is mapped to the cluster index. In turn, the `cluster_color` as the name insists, contains the actual color of the cluster, helping to map each pixel to the color-quantized palette.

As the bonus asks, apart from quantizing in the HSV space, we can also implement the same method in the CIELab space. We can compute the grayscale LUT by clustering an image in a certain color space and then transform it by isolating only the grayscale information by taking the luminance channel (i.e. the first channel of the image). The formula for it is as follows: $(0.2989 \times R) + (0.5870 \times G) + (0.1140 \times B)$. Results are shown below, and the respective code can be found in `ex2.m`.

Figure 5: Color quantizing for HSV and CIELab spaces, + grayscale w/ luminance for 32 color clusters

Bonus

Particularly, a difference between the quality of the HSV and CIELab spaces are noticeable such as in areas around the Queen's neck and the contrast of her blue coat against the stone wall in the background, wherein the CIELab space performs slightly better (smoother) visually.

3 Gaussian and Laplacian Pyramids

As we learned in the lectures, when speaking of "Gaussian pyramids" we speak of a sequence of images that are visualized via layers repeatedly applying Gaussian blurring to the original image. The Laplacian pyramid on the other hand, is the sequence of images obtained by subtracting from each layer of the Gaussian pyramid the prior "up-sampled" version of the image within the Gaussian pyramid. This way the Laplacian pyramid contains the artefacts and details of each respective Gaussian layer in the pyramid.

As the assignment similarly states, in order to reconstruct the original image we add up each up-sampled Laplacian pyramid and receive the image required (which is not the same case for the Gaussian pyramid).

Figure 6: Gaussian Pyramids for `sad` and `happy`

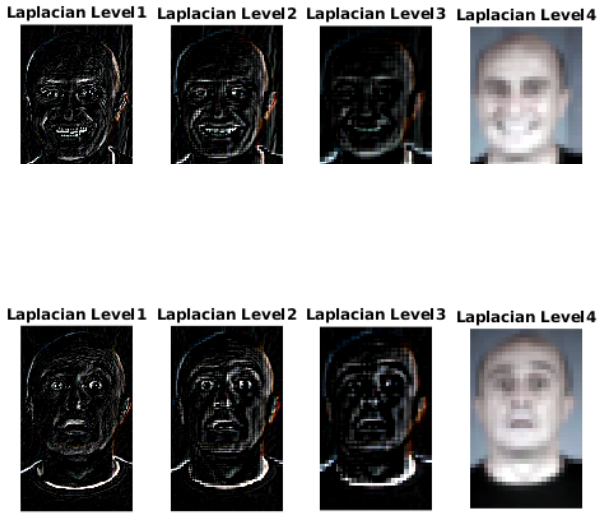

Figure 7: Laplacian Pyramids for `sad` and `happy` (artificially brightened for better visualization)

Figure 8: Restored Laplacian Pyramid's to their original images

As we can see from the results above, the Laplacian pyramid layers can be restored back into their respective original images. All the code for this respective exercise can be found in `ex3.m`.

```
% function computing the gaussiand and laplacian pyramids, as well as calling
function [gaussian_pyramid, laplacian_pyramid, reconstructed_img] = pyramids(img, levels)
    img = im2double(img);

    gaussian_pyramid = cell(levels, 1);
    gaussian_pyramid{1} = img; % first layer is original image
    for i = 2:levels
        img = imgaussfilt(gaussian_pyramid{i-1}, 1);
        img = imresize(img, 0.5, 'nearest');
        gaussian_pyramid{i} = img;
    end

    laplacian_pyramid = cell(levels, 1); %initialize
    for i = 1:levels-1
        prev_img = imresize(gaussian_pyramid{i+1}, size(gaussian_pyramid{i}(:,:,1)), 'nearest');
        laplacian_pyramid{i} = gaussian_pyramid{i} - prev_img;
    end
    laplacian_pyramid{levels} = gaussian_pyramid{levels}; % last one = gaussian

    % display gaussian
    figure;
    for i = 1:levels
        subplot(1, levels, i);
        imshow(gaussian_pyramid{i}, []);
        title(['Gaussian Level ', num2str(i)]);
    end

    % display laplacian (with brightness adjustment for visualization)
    figure;
    for i = 1:levels
        brightened_lap = imadjust(laplacian_pyramid{i}, stretchlim(laplacian_pyramid{i}), []);
        subplot(1, levels, i);
        imshow(brightened_lap, []);
        title(['Laplacian Level', num2str(i)]);
    end

    % display the reconstructed images
    figure;
    reconstructed_img = reconstruct_laplacian(laplacian_pyramid);
    imshow(reconstructed_img, []);
end
```

I use the inbuilt `imgaussfilt()` function, along with `imadjust()` to artificially brighten the Laplacian layers for the assignment paper.

4 Hybrid Images

As the assignment asks, we are presented the opportunity to utilize our understanding of Gaussian and Laplacian pyramids to create a hybrid image, wherein the high-frequency content of one image and the low-frequency image of another image blend together to create a "hybrid" image. Due to the fact that our human visual systems is more sensitive to higher-frequency content at a close distance, but diminishes as soon as we move away from said image. On the other hand, we start to be able to view lower-frequencies better. Thus these hybrid images are made by exploiting this aspect of human nature.

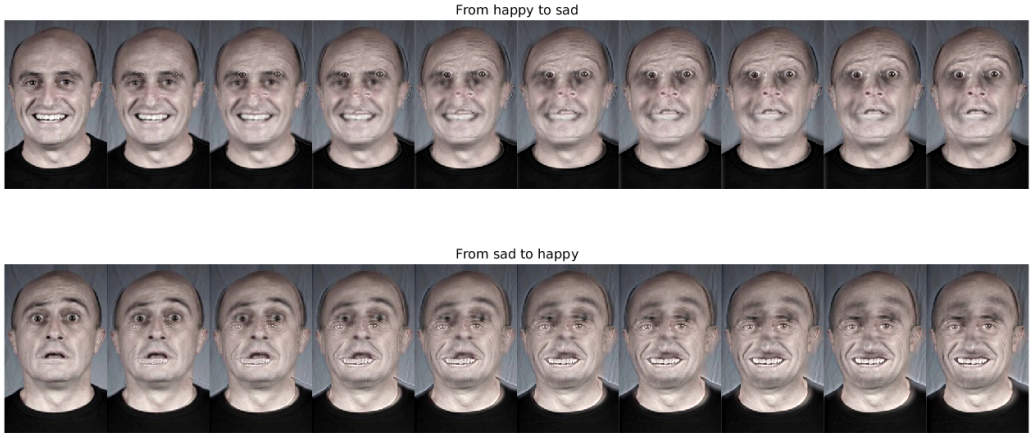

Figure 9: σ decreasing

In Figure 9 are 2 sets of hybrid images created from the given `sad.jpg` and `happy.jpg`. In both of these images I attempted to first combine the low-frequency of one image with the high-frequency component of the other (and vice-versa), under different σ cut-offs for the low- and high-pass filters.

Different cut-off frequencies for σ have been used from 0.5 to 5, from left to right. These images seem to dominate one-another depending on the distance the viewer has from the screen they are looking at them. When looking at the "sad to happy" image, the sad image persists visibly stronger, but once you look towards the right the "happy" smile dominates. These effects will depend on many factors, including the screen on which someone is looking at, the brightness of the screen, the glare, the distance from which they are looking, their eyesight, etc.

All the code for this respective exercise can be found in `ex4.m`.

Ideal Viewing Conditions

After inspecting the image properties closer (and also accounting for a slight pixel difference between the two different images), the images and hybrids are of `width:193`, `height:316`. Viewing the relatively small images on a 15-inch laptop screen, around 45-60cm is optimal to see the sad/happy face, and vice versa (the lower end of that number causing the image on the left to **not** be coordinated with the image on the right).

5 Bonus: Create your own Instagram Filter!

For this part of the assignment I have taken a humorous approach (as someone of a younger generation), and have attempted to create a "deep-fried" filter. What is a deep-fried filter? According to `knowyourmeme.com`: "deep-fried memes are a style of meme wherein an image is run through dozens of filters to the point where the image appears grainy, washed-out, and strangely colored." The figure below shows an example from the internet after clicking on the first Google search for `deep-fried meme`:

Figure 10: (Source) `https://www.wikidata.org/wiki/Q124312812`

Common features of the image are the high levels of saturation/exaggerated colors, low quality visuals (i.e. highly pixelated artefacts), red/blue-tinted flares around eyes (if the subject of the image is a person) or just seemingly random color spots.

No one exactly knows how this meme began, but I thought it would be a fun exercise to try and replicate this exact aesthetic, as it is actually very particular and has a few steps to it involving what we have learnt during this course throughout the semester.

The partial code for this exercise is included below, and is also included in the file in `ex5.m`.

```
function deep_fry(img)
    [x, y] = ginput(2); % using the picker from the first assignment
    ...
    % severely over-sharpen the image with kernel
    sharpened_img = imfilter(original, [-1 -1 -1; -1 9 -1; -1 -1 -1]);
    ...
    hsv_img(:, :, 2) = hsv_img(:, :, 2) * saturation_factor;
    ...
    % reduce image quality
    scale_factor = 1.5;
    reduced_qual_img = imresize(sat_img, 1 / scale_factor);
    ...
    % sharpen the image again for good measure
    final_img = imfilter(reduced_qual_img, [-1 -1 -1; -1 9 -1; -1 -1 -1]);
    ...
    % add flares where the ginput() was decided
    flare_radius = 50 / scale_factor; % move these based by the scale_factor previously
    flare_scale = 1.5;
    ...
    final_img(:,:,3) = final_img(:,:,3) + flare; % Add flares to blue channel
end
```

Following are the results I achieved after utilizing the `queen.jpg`, `happy.jpg` and my own included `dog.png`.

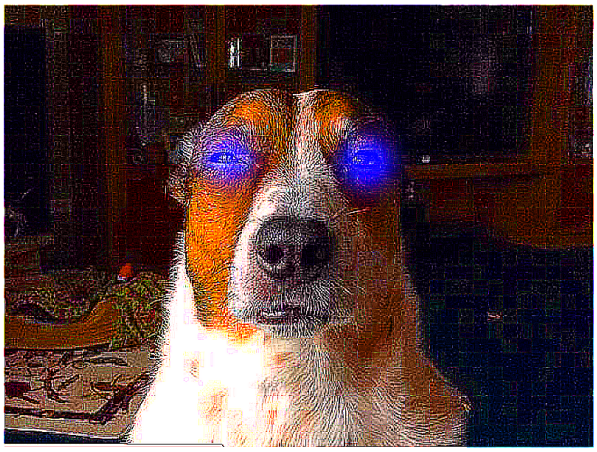

Figure 11: Given assignment images and my own, "deep-fried" edition