**Cosmetic foundations**

I implemented the scripts needed to find the RGB reflectance (the reflectance for red, green, and blue light) of a cosmetic from a patch image pulled off the web, and to show a face with a foundation cosmetic applied. The key script is colorDriver.py. It first calls getPatchColor(*patch*), where *patch* is the file name of the patch image. The function returns the RGB reflectance of the darkest (thickest) region of the patch, which shows the true color of the cosmetic, which is independent of the color of the substrate. This function, like most of the functions discussed in this document, is defined in colorLib.py.

It then eventually calls showFace which calculates the appearance of a face with a cosmetic applied. The RGB reflectance of the cosmetic is one of the parameters. However showFace needs two other pieces of information that would have to be extracted from a selfie of a face by code that is the responsibility of other contributors. They are the RGB reflectance of each point on the skin and the amount of light reflected from it. For this piece of work, I have just made some reasonable assumptions to derive these two pieces of information in a way that gives a result that is reasonably consistent with the original selfie. To do this I’ll assume that the skin reflectance is constant, so the illumination variation will give us shape from shading. It is this illumination variation that the human brain uses to derive the three dimensional shape of an object from a two dimensional image. Then I can add some blemishes to the skin reflectance image and cover it up with cosmetic.

The first step is to be able to calculate a reasonable looking bare face image with:

RGBface = Yshad \* RGBaveref

Where Yshad is a gray image giving us shape from shading, RGBface is the amount of red, green, and blue light at that point in the image, and RGBaveref is the reflectivity of the skin to red, green, and blue light. In fact, the illuminant might not be neutral, but we can get a reasonable looking simulation assuming it is. All we know is RGBface so we can get this result with a big Yshad and a small RGBaveref or a big RGBaveref and a small Yshad. To get reasonable values for Yshad note that the above equation implies:

Yface = Yshad \* Yskin

Where Yskin is the neutral reflectance of the average skin color. Furthermore, we could determine Yskin if there was an object of known reflectance illuminated the same way as the skin. One possibility would be the sclera of the eye and the surrounding skin illuminated without obvious shadows. This is clearly future work. For now we will just pick a reasonable value for Yskin. Then:

Yshad = Yface / Yskin

We now have to pick an RGBaveref consistent with the assumed Y. To do this we can calculate:

RGBref = RGBface / Yshad

for each pixel, where RGBref is the skin reflectance at that pixel. Hopefully RGBref will be reasonably constant, but in any case we should calculate its weighted average, since the bright areas of the face will give more accurate values than the darker ones.

RGBaveref = ∑( RGBref \* Yshad) / ∑ Yshad = ∑( (RGBface / Yshad ) \* Yshad) / ∑ Yshad = ∑ RGBface / ∑ Yshad

However all I have to work with are images from the net and cell phones, and they are usually in a color space called sRGB. I therefore need to go from sRGB to RGB brightness (RGBface) to do these calculations. Unfortunately OpenCV doesn’t seem to know about sRGB, so I had to do the transformation myself. sRGB is a CRT based color space defined in terms of the brightness of the CRT sRGB phosphors and a nonlinear transformation between the sRGB R, G, and B values, and the brightness of the R, G, and B phosphors. Since there is a branch in the nonlinear transformation function, this would be hard to do with array processing. Therefore I just used ‘for’ loops and sRGB to RGB subroutines, and it goes fast enough. Note that the linear RGB space used by OpenCV also assumes the sRGB phosphors.

I next add a pimple to the face reflectance, and combine the face reflectance with the shading to get the appearance of the bare skin.

Finally I use Kubelka-Munk theory to calculate the reflectance of the skin with a layer of cosmetic with the desired color and the desired thickness (opacity), and combine that reflectance with shading to get the RGB color. I’ve chosen the cosmetic color to be equal to the average color of the skin, which would be the usual usage. The final step is to convert RGB to sRGB so it will look right on a monitor.

Since the user may want to pick a cosmetic with an RGB reflectance that matches the RGBaveref of their skin, the application might want to go through the RGB reflectances of the available cosmetics and pick the one with the closest match. However, the degree of match should be measured using a metric that matches human perception. Differences in RGB reflectance, which is what we have been working with up to now, do not match human perception very well. In some areas of RGB reflectance space large changes correspond to small perceptual difference, and in other areas, small differences correspond to large perceptual differences. Differences in a metric called CIELAB are a much better match to human perception. We therefore also wrote a function (difLAB(color1, color2)) to calculate the CIELAB difference between two colors described by the RGB reflectance space we have been using up to now. colorDriver.py also calls this function to illustrate its use.

The difLAB function first uses the transformation from linear RGB to XYZ found in <https://en.wikipedia.org/wiki/SRGB>. XYZ is a color space that describes the response of the human eye to a given spectral distribution of light. All color spaces are defined by their relation to XYZ. The transformation in <https://en.wikipedia.org/wiki/Lab_color_space> is then used to go from XYZ to CIELAB.

Note that it is possible for the RGB color to be greater than 1.0, and this does strange things on the conversion to sRGB. In places where this might happen, we slightly scale down the offending separation so it never exceeds 1.0.

I’ve enclosed a paper on the Kubelka-Munk theory. I’ve played with the algebra to make it easier to implement. The final equations are:

P = T \* (1 - R∞2) / R∞

F = exp(-P)

G = R∞ \* R∞ + Δ \* R∞ - 1

R = (Δ \* F – G \* R∞) / (Δ \* F \* R∞ - G)

These equations are applied for each separation. Δ is the difference between the skin reflectance and R∞. R∞ is the red, green, or blue reflectance of a very thick layer of the cosmetic. R is the red, green, or blue reflectance of the skin with the cosmetic applied. T is the effective thickness of the cosmetic, chosen by the user of the software.

**A better cosmetic company web page processing procedure**

To simplify the process of extracting RGB reflectance values from cosmetic company web pages, I wrote a script called processPatches.py. I started off with the Sephora page, because it is easy to work with. You can go to a foundation collection, and right click on each patch. The “save picture as” menu item can be used to put the pictures in an appropriate directory, giving each patch an appropriate file name. I put the patches in each foundation collection named below in a directory with a corresponding name:

Sephora bare minerals – bare minerals original foundation broad spectrum SPF 15

Sephora Fenty beauty by Rihanna – pro filter soft matte long wear foundation

Sephora IT cosmetics standard size – Skin but better CC+ Cream with SPF 50+

Sephora Kat Von D – Lock it powder foundation.

Sephora Clinique – stay matte sheer pressed powder

Sephora Lancome – Teint Idole Ultra Long wear Cushion foundation SPF 50

Sephora Guerlain – Lingerie de Peau Aqua Nude Foundation

Sephora Este Lauder – Double wear nude cushion stick radiant makeup

The processPatches.py script processes all the images in a specified directory and puts a file called rgb.text file with the cosmetic names and RGB reflectances in that directory. See the code for a detailed explanation of how to use it.

**Eye shadow**

Eye shadow is more complicated. The shadow is only applied to the eyelid, and is usually thinned out towards the top of the eyelid. Furthermore some eye shadows are textured. We will handle this using a modification of showFace() called showFaceGen() where thickness and color will be described by images that capture the spatial variation. The necessary modifications to showFace() are straightforward.

eyeShadowDriver.py shows how to use the new function. Quite a few modifications to colorDriver.py are necessary. We have to calculate the Yshad weighted average skin reflectance for just the eyelid. This is defined by the region of floodfill.bmp, a one separation image, where the pixels have been set to a value of 200. The thickness of the eye shadow is read in from an image named, in this example, feathered.bmp. It is a one channel image where the thickness is a value between 0 and 254. The line that surrounds the eyelid has a value == 255, and is removed from the image before it is used.

The eye shadow color can either be set to a constant value (specified by the eyeShadow parameter for the non textured case) or an image (texture.bmp for the textured case). showFaceGen() expects an image for the cosmetic color, so we generate one for the non-textured case. The textured case is a bit trickier, since it turns out that the Kubelka-Munk math blows up if a separation of the cosmetic reflectance is either 0 or 1. We therefore slightly shrink its range to prevent this from happening.

Below I explain how I generated floodfill.bmp, texture.bmp, and feathered.bmp. The final application will have to do it differently, but I will explain how I did it, since some of the ideas might be useful in the final application.

First I drew a pure red line on the upper part of the eyelid using paint. I called this eye1feather.bmp. Then I copied this image and added a second line on the lower part of the eyelid. I called this eye1circle.bmp. The two lines completely enclose the eyelid. Then using a convenient third party tool I found a pixel within the eyelid which will be the seed for a region growing algorithm. These steps would have to be done without human intervention in the final application. I then used redSense.py to make corresponding mask images just containing the lines, white on black. The eye1feather.bmp image was used to make eye1featherLine.bmp, and eye1circle.bmp was used to make eyelid.bmp.

I then entered the names of the mask images and the seed value in floodfill.py and ran it. The script first calls floodfill(*image*, *seed*, *value*) which identifies the pixels in the *image* that are part of the eyelid by setting their values to *value*. It then writes out this *image* with the name floodfill.bmp.

It also runs floodfillFunc(*image*, *seed*, *valObj*) which uses an object with a getVal(*pixel*) method to determine the value to place in each *pixel* of the flood filled region of the *image*. If *valObj* is an instance of the constVal class, getVal always returns the value set during the construction of the object. This is then just another implementation of the functionality of floodfill. Its output is not used, but the function is enclosed for illustrative purposes.

The featheredVal class is an extension of constVal. If *valObj* is an instance of the featheredVal class, getVal(*pixel*) returns a value that goes from a large value at the lower part of the eyelid, to zero at its upper limit. The sharpness of the transition is set by the *pwr* parameter in the getVal(*pixel*) method. A value of 0.3 seems to work pretty well. The upper limit of the eyelid is stored in *valObj* by reading in eye1featherLine.bmp before running floodfillFunc. After running floodfillFunc, the *valObj*.renorm(*image*) method is run to scale the values in the eyelid region of the *image* so they go from 0 to 254. Finally the *image* is read out as feathered.bmp, which will be used to control the thickness of the eye shadow.

The color and possible texture of the desired eye shadow is obtained from an image of the cosmetic. If the eye shadow is not textured, use processPatches.py to get its color, and use eyeShadowDriver.py in the constant color mode. If it is textured a rescaling will probably be necessary to make the scale of the eye shadow image match the scale of the selfie. In addition, the image of the eye shadow will probably have to be tiled to cover the entire selfie. Knitted.py takes care of both tasks. It takes the selfie and textured eye shadow images as well as a scale parameter, and makes the texture.bmp image. The scale parameter is the number of pixels per inch in the selfie divided by the number of pixels per inch in the eye shadow image. Therefore there should be a ruler in the eye shadow image, and both eyes should be visible in the selfie. The distance between the pupils of adult humans is about two inches.

**Required elements in release:**

colorLib.py

colorDriver.py

input: armaniPatch.png

input: crop4.jpg (selfie)

output: simulation.png

output: withPimple.png

output: withCosmetic.png

processPatches.py

Sephora data directories

eyeShadowDriver.py

input: eye1.bmp (selfie)

input: floodfill.bmp

input: feathered.bmp

input: texture.bmp

output: simulation.png

output: withCosmetic.png

redSense.py

input: eye1feather.bmp

output: eye1featherLine.bmp

input: eye1circle.bmp

output: eyelid.bmp

floodfill.py

input: eyelid.bmp

output: floodfill.bmp

input: eye1featherLine.bmp

output: feathered.bmp

knitted.py

input: 20171106\_120437a.jpg (un-scaled eye shadow)

input: eye1.bmp

output: texture.bmp