Non-Photorealistic Rendering - Paintings

6.865 - Computational Photography. November 23rd, 2020.

TAREQ EL DANDACHI

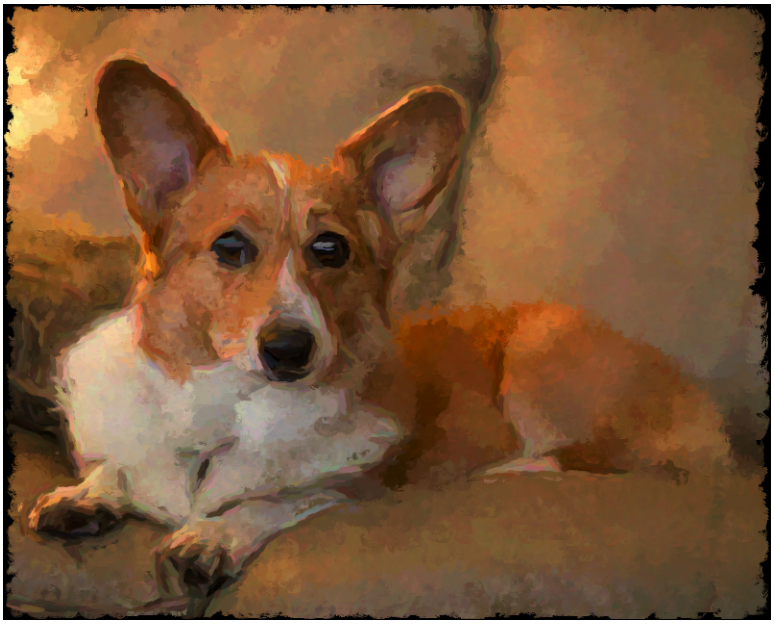


Fig. 1. Painterly rendering of a dog using the multiGammaPaint method.

1 INTRODUCTION

For my final project, I worked on implementing a C++ program for a previous year's PSet, namely the Non-Photorealistic Rendering (Painterly Rendering) PSet. The main idea behind the program is given an input image, it will render a version that looks like a painting. In addition to the basic implementation, I added new methods that add subjects to the painting, use multiple brushes, a way to reduce the colorspace and recolor the image. I also experimented with multiple methods to make the painting look better.

I chose to work on this PSet for two main reasons. Firstly, I have always admired the amount of work that goes into paintings and the artistic creativity that goes into it and I wanted to fiddle around with making paintings and manipulating their styles. Secondly, I always wondered what realistic photos would look like if they were painted, however, I am not very good at painting, therefore having a program that can paint in multiple styles and color palettes seemed like a really cool project to undertake.

I ended up implementing:

(1) painterly: a function that repaints by sampling points and splatting a brush.
(2) orientedPaint: a function similar to painterly but orients the stroke direction to follow the angles in the original image.
(3) paintMultiBrush: a function similar to orientedPaint but uses three types of brushes, one for each "layer" of the image, producing a sharper and more visually pleasung image.
(4) multiGammaPaint: a function that dramatizes an image by playing with its gamma before applying a procedure similar to paintMultiBrush.
(5) An assortment of functions driven by the k-means algorithm to separate an image into a color subspace.

See Figure 2 for an example of 1-4. The functions that use k-means (5) are special, in that they allow for recoloring an image, modifying the mood of the image and repainting an image based on a reference image.

Given an input image, the program attempts to find the "flow" of pixels, identifying the edges of surfaces in the picture and how they point. The program then applies a rotated brush that is aligned with the direction field we extracted. The program first uses a thick brush to draw the background of the painting, it applies randomly positioned blotches of brush strokes. It then applies smaller brush strokes with a second kind of brush that are aligned with the edges of the change in direction field, this helps define the edges more. Then the algorithm continues by applying a third kind of brush stroke along the edges of the direction field, this helps create straight lines

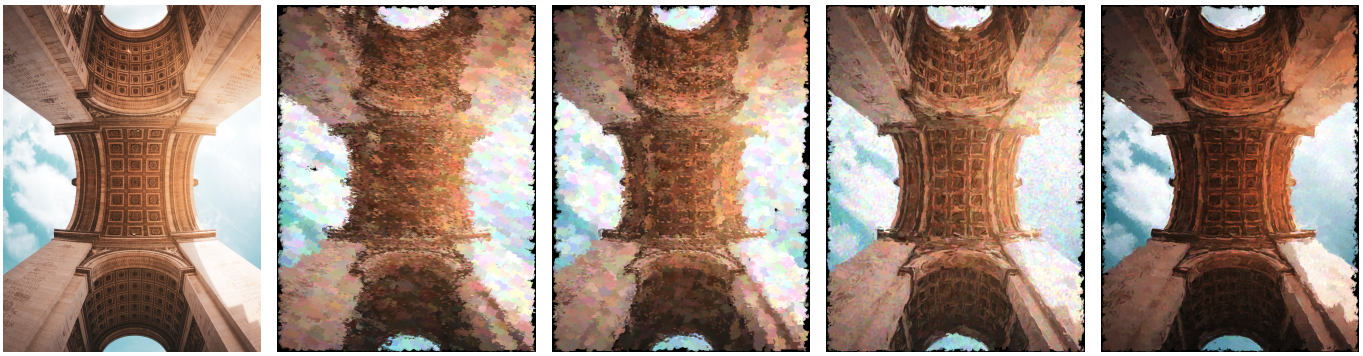


Fig. 2. From left to right: (1) input image, (2) image produced by painterly, (3) image produced by orientedPaint, (4) image produced by paintMultiBrush and (5) image produced by multiGammaPaint

along lines in the picture, see picture 4 in Figure 2.

To give the program more artistic creativity, simulate the dynamics of painting more and remove high frequency color noise, this method reformulates the color-space into an optimization problem. This was inspired by the reformulation of an image into an 8D space from the paper "Portrait Lighting Transfer Using a Mass Transport Approach". The program does that by grouping the input image into a small subspace of colors that it can use to paint the new image. The program uses the k-means algorithm, which tries to group points in some high dimensional space into new subspaces which contain points that are relatively close to each other. It does that by minimizing the distance in the higher dimensional space (which is a combination of a color parameter and position parameter) between points as much as it can and the mean of the subspaces it finds.

The program can then assign colors to these subspaces by either their mean or some new color. Therefore when the program repaints the image it simulates an artists smaller color palette and removes the high frequency noise. It also allows for changing the image's colors. One use case for the algorithm is painting a recolored version of an image using a reference image to choose the colors from, therefore you can paint a night scene using day colors, and get a morning scene, see Figure 9.

2 BACKGROUND

There are lots of paper's related to painterly rendering specifically, and there is a wide range of methods they follow to produce visually pleasing results. The paper "Painterly Rendering with Curved Brush Strokes of Multiple Sizes" worked by varying the size of the brush and uses curved brush strokes as opposed to blots of paint like the method followed in this paper. Another paper titled "Paint By Relaxation" uses a relaxation algorithm and heuristics to find a photo with the least energy, which corresponds to a painting. This method is especially nice since the author is able to vary terms and achieve a new painting style. "Orientable Textures for Image-Based Pen-and-Ink Illustration" uses the geometry of a surface in the photo to find a direction field and apply "orientable textures." This paper has some similarities with this work, since it also depends on the use of the direction field, through the structure tensor, and can use different brushes depending on the field, however it targets paintings as opposed to drawing illustrations. I was also inspired by the paper "Portrait Lighting Transfer Using a Mass Transport Approach", in the paper, the authors decompose an image into an 8-dimensional subspace composed of a color subspace, a normal subspace and a distance subspace, and then they use an approximate solver for a mass transport problem to minimize a reformulation of the transfer of lighting from one image to another. Although the method followed in this problem to segment and extract colors is very different, it was inspired by the formulation shown in that paper.

3 METHOD

3.1 Subspace separation using k-means

Given an input image I, we create a 5 dimensional subspace S; 3 dimensions for the color and 2 for the position, $\mathbf{v}_{x,y} = (c_1, c_2, c_3, kx, ky)^T$, where k is the smallest dimension of the image. This helps characterize any two pixels in an image by 2 types of distances, d_c, the color distance and d_t, the total distance.

$$d_c(I_{x1,y1}, I_{x2,y2}) = \sqrt{(I^2_{x1,y1} - I^2_{x2,y2}) \cdot [1\ 1\ 1\ 0\ 0]^T} \tag{1}$$

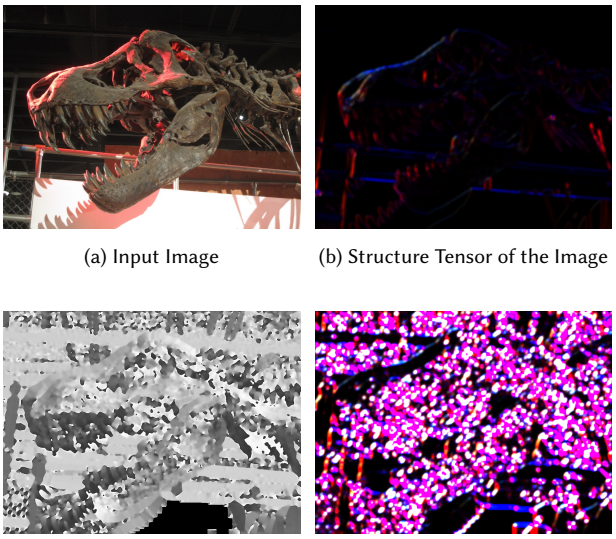
$$d_t(I_{x1,y1}, I_{x2,y2}) = \sqrt{(I^2_{x1,y1} - I^2_{x2,y2}) \cdot [1\ 1\ 1\ 1\ 1]^T} \tag{2}$$

This idea of distance helps us in our clustering problem. We want to find k means that are the center of k clusters, C_i, $i = 1...k$. We can find the clusters by the minimization formula of k-means clustering, where μ_i is the mean of the cluster C_i:

$$\arg\min_S \sum_{i=1}^{k} \sum_{\mathbf{x} \in C_i} ||\mathbf{x} - \mu_i||^2 \tag{3}$$

$$\arg\min_S \sum_{i=1}^{k} \frac{1}{2|C_i|} \sum_{\mathbf{x},\mathbf{y} \in C_i} ||\mathbf{x} - \mathbf{y}||^2 \tag{4}$$

Fig. 3. Colors of the 5 means of the clusters formed by k-means on the above image.



(a) Input Image (b) Structure Tensor of the Image

(c) Direction Field of the Image (d) Structure Tensor of the Direction Field

Fig. 4. The different types of fields calculated for an input image.

We can modify this minimization to include either d_c or d_t as the distance function term, $||\mathbf{x} - \mathbf{y}||^2$. We then map every cluster C_i to a color which is represented by the first 3 terms of μ_i. Figure 3 is an example of the code running on an input to extract $k = 5$ colors. We can replace the color of every point in C_i by the color of μ_i and we will get a visually similar photo, as the value of k increases, the photo will resemble the ground truth more, if k is the number of pixels in the image it will be identical. To get interesting results, we need to set k to some small constant to get a much smaller color space wrt. the images original color space, otherwise we will get an image that is too similar that is drowned in the noise we will later introduce in painterly.

The k-means algorithm is implemented using Eigen. It takes in the following inputs: a matrix A, which is a column vector of $\mathbf{v}_{x,y}$ for all the possible x, y pairs, i.e. $\dim A = (x \cdot y, 5)$, a number of clusters to find k and the number of iterations i. The algorithm is an approximation that achieves the solution perfectly as $i \rightarrow x \cdot y$. The algorithm usually sets $i = 5k$ since this results in an acceptable solution without taking too much time, especially since the values of k are small, although ideally, it would be exponential in the input size.

The algorithm initially chooses k random rows from A and uses them as the means, A^k represents a version of A in which copies of A are stacked k times on top of each other. It then proceeds by calculating $\min \sum_{\text{rows}} d(A_c^k, \mu)^2$, where $d \in \{d_c, d_t\}$, and updates the mean by the taking the average of the result in the column space of A^k. This way after i iterations, we have an approximation of what the new means are and can use them to classify points based on their distance away from each other.

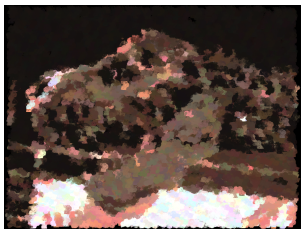
3.2 Structure Tensor and Direction Fields

The structure tensor of an image helps represent the edges of an image, in this formulation, I will focus on the structure tensor composed of the derivative of the luminance channel L. The structure tensor T written in terms of the gradient of L, $\nabla L = [\partial_x L \;\; \partial_y L]$:

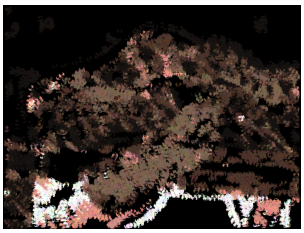
$$T = (\nabla L)(\nabla L)^T \tag{5}$$

From T, we are able to use a self-adjoint eigensolver to extract the eigenvectors, since we know that T is adjoint, $T = T^*$, therefore the angle has to be real. Once we extract the eigenvalue-eigenvector pairs, we will search for the eigenvector with the smallest eigenvalue, $\mathbf{v}$, and get the angle of that vector. In the implementation, I used Eigen to extract the eigenvalue-eigenvector pairs. We end up with a direction field that depicts the angle of the minimum eigenvector for each pixel that ranges between 0 and 2π.

The algorithm also references the magnitude tensor, which is nothing but the trace of the matrix $\frac{1}{2}T$. This is because the algorithm doesn't depend on the direction of the structure tensor, but is trying to find an overall difference in direction of the tensor instead. We also attempt to remove high frequency noise from the magnitude tensor, we do that by creating a 5×5 neighbourhood for every pixel and add up the value of the magnitude tensor at each pixel in the neighbourhood, if it surpasses a cutoff $\delta = 0.15$, then the pixel gets to keep its magnitude tensor value, otherwise it is set to 0. See Figure 4 for a visualisation of the different fields.

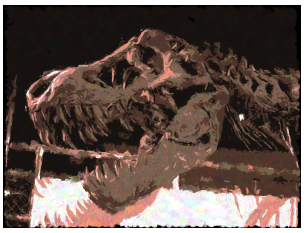


(a) Background (low-frequency) layer



(b) Filling (Angle Tensor Structure) layer

(c) Detail (high-frequency) layer



(d) Sum of all three layers

Fig. 5. The individual layers that make up a painting produced by the code on the input image from Figure 4a

3.3 Painting Low Frequencies | The Background

The color reduced image represents a possible color to use when we paint near this pixel, since the brush size should be larger than 1 to get interesting results. To paint the low frequencies, or what I will refer to as the "background" of the image, we don't need high fidelity in our blotches, so firstly the brush of size $d \times d$. Then we generate random (x, y) coordinate pairs within the image and apply a brush stroke on that position of color $f(c_{x,y}, n)$, where $c_{x,y}$ is the color of the pixel (x, y) in the original image and f is a function that introduces a noise of magnitude n as follows:

$$f(c_{x,y}) = c_{x,y} \cdot \left(1 - \frac{n}{2} + n\delta\right) \tag{6}$$

where δ is a randomly generated 3 entry column vector. The reason the application of this background corresponds to painting low frequencies is that we are randomly sampling a space of dimension $W \times H$ (the image dimensions) by N samples, and each sample has size $d \times d$. Assuming the overlap of randomly generated values is minimal, the space is reduced into a maximum of N colors (if $k = N$), this smoothens out the content of the image, if we fill out any empty spaces in that space using some form of interpolation, the result will be very similar to blurring the image, which corresponds to the low frequencies of the image. An example of the application of the low-frequency layer can be seen in Figure 5a.

When placing a blotch of the brush, we make sure we rotate the brush by $\theta \in \frac{2\pi}{\delta\theta} \times \mathbb{Z}$, where $\delta\theta$ is a constant that defaults to 36 in the code, this means there are 36 angles the brush can be in. The angle at which we rotate the brush can be extracted from the direction field of the image.

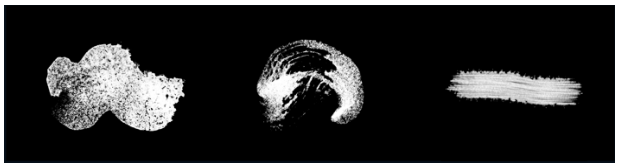


Fig. 6. The brushes used in the multiBrush code. From left to right: The first brush is the background brush, the second is used on the tensor structures of the angles and the third is used on the tensor structures of the image.

3.4 Painting Inside the Angle Tensor Structure | Filling in the Painting

Given the direction field of the image, we can further calculate the tensor of that to find the regions where the angle changes a lot. Placing brush strokes inside regions where the direction field changes by a lot gives a higher probability of placing a blot in the correct orientation on a point that could've been oriented incorrectly when width of the brush is large enough. We give priority to these patches by using a random number generator to perform rejection sampling. Whenever we choose a pair (x, y) to place a brush stroke on, we also generate a number $0 \leq r \leq 1$, and if that number happens to be less than the reference field, $r < R_{x,y}$, in this case $R_{x,y}$ is the angle tensor at (x, y), we place a blotch there.

As the separation between values in the field increases, we can see that the probability of finding a paint blotch on a lower value approaches 0, while the probability of all the strokes being in a high region will approach 1. This helps fill in regions that need more attention such as filling in the high change regions we described above. See figure 5b for the placement of brushstrokes within the angle tensor structure.

3.5 Painting High Frequencies | Fine Touches on Edges

To color in the high frequency regions, i.e. the edges of objects in the image and areas which correspond to higher sharpness values, we will use the magnitude tensor as described in section 3.2. The magnitude tensor highlights areas where the second derivative changes sharply, the direction field at these edges is usually perpendicular to the magnitude tensor seams. In figure 5c, the brush strokes on the fossil align perfectly with the edges of the skull and the teeth from 4a. It also adds in lines highlighting the bone structure of the skull, these are all considered high frequency features.

The implementation of this follows exactly from filling in the angle tensor structure, in that it also uses a random number generator and a cutoff, with the reference field here being the magnitude tensor.

3.6 Modifying by Gamma Encoding

While trying to make the photos look more artistic, I first tried to make the shot more dramatic by non-linearly modifying the image pixels by a gamma of 0.25 and then quantizing the pixels into 2^4

bins to get a resultant more dramatic image. Figure 7 has an example of the results of the algorithm. It makes the photos look more like paintings since smaller details that are often focused on in a painting are more pronounced while using less colors in an exponential map, for example, Figure 7d shows the castle with more clearly than 7c. The function 'multiGammaPaint' implements this.

3.7 Using the k-means when recoloring

Now that we see that the gamma makes photos look more dramatic, we can show that we can achieve something similar using the k-means algorithm we described above. When dividing the color into different bins, we can change the individual colors each bin by some mapping function f and get a recolored image. We can specify the function f by multiple ways, either as a function on the input ($f(c) = 1 - c$), a new mapping from manual colors we choose (I used https://coolors.co/ to choose nice palettes while working on this project) and we can extract colors (using k-means again) from a reference image to recolor.

I will focus on showing the results of recoloring from another image, since it is easier to present the results in the report that way. Figure 8 shows how we can change the appearance of the biome using a reference image, in the example, the reference photo (8a) has paler greens than the bright greens in the input (8b), resulting in different looking paintings. This has the effect of changing the color palette a painter is using when painting. A cooler example is shown in figure 9 where the recoloring is able to reproduce the morning photo from a night photo and colors close to that of a shot in the morning (which could've been chosen manually by a palette).

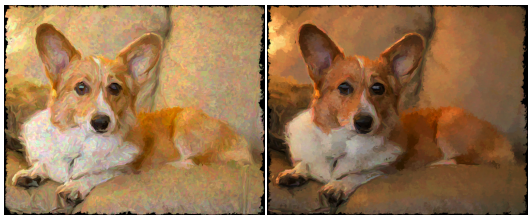
The distance function can also be applied to the chrominance of the photo instead of the RGB image, this can help to classify colors better. The function 'reduce_color_space' takes in the parameters use_distance and chrominance which decide whether to use the distance and whether to use RGB or chrominance.

4 RESULTS

The algorithm was tested on approximately 30 unique photos to repaint and 10 brushes to paint with. The repainting without k-means runs in an average of 1.98 seconds, with the fastest time being 1.8 and the slowest being 2.1 on inputs of resolution 1080×764. The repainting with k-means algorithm runs in an average of 2.34 seconds, with the fastest time being 1.9 and the slowest being 2.7 on inputs of resolution 1080×764. Color extraction (via k-means) and repainting with k-means averages at 3.4 seconds.

Photos used to test can fall in 7 main categories:

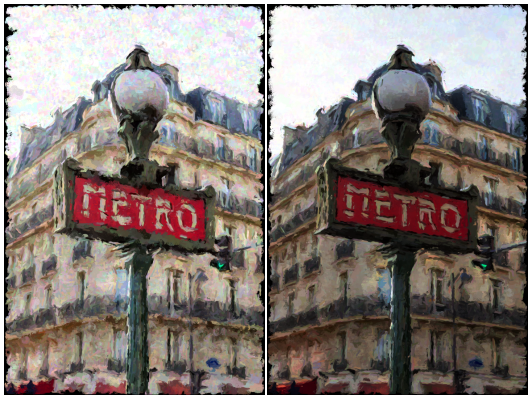
(1) Photos taken with an iPhone 8's back camera (and rescaled in software).
(2) Photos taken with a Canon PowerShot SD780 (and rescaled in software).
(3) Vector graphics created manually using Affinity Designer to test specific features such as angle detection and k-means sampling. I also used this to add more brushes.



(a) 'Archie.png' without modifying gamma (b) 'Archie.png' with modifying gamma

(c) 'unsplash-5.png' without modifying gamma (d) 'unsplash-5.png' with modifying gamma

(e) 'unsplash-6.png' without modifying gamma (f) 'unsplash-6.png' with modifying gamma



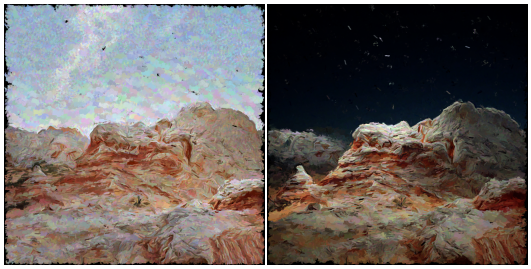
(g) 'unsplash-7.png' without modifying gamma (h) 'unsplash-7.png' with modifying gammat

Fig. 7. Comparison of photos with and without gamma modification.

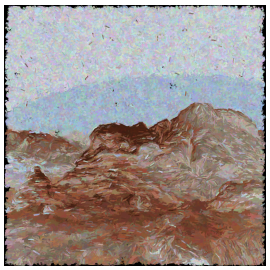
(4) Photos provided in the source from the 2013 PSet.
(5) Photos chosen from unsplash (and rescaled in software).
(6) Photos extracted from macOS's background directory.

Categories 1 and 2 should be in the Directory [./Input/photos]. The third and fourth should be distributed into both the main directory [./Input] and the brushes directory [./Input/brushes]. The

(a) Original Day Shot (b) Original Night Shot

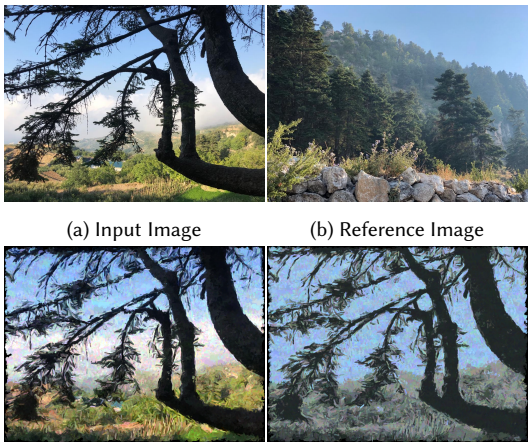


(c) Repainted Image of Day Shot (d) Repainted Image of Night Shot



(e) Repainted Image of Night Shot using the color-space extracted from the day shot

Fig. 9. Painting a day shot from the night shot of an image.



(a) Input Image (b) Reference Image

(c) Result without recoloring (d) Result recoloring with reference

Fig. 8. Painting a day shot from the night shot of an image.

fifth category is under [./Input/unsplash_photos], note that I do not own any of the photos included in that directory and I am very grateful for the contributors on Unsplash and Unsplash's License (https://unsplash.com/license). The photos in the last category are in the directory [./Input/apple_photos/], these are photos that come shipped with macOS BigSur (I do not own them, they are available in the directory [/Library/Desktop Pictures]), I thought of repainting them for fun as I've been staring at them endlessly.

The biggest challenge of working on this project is trying to tackle the repainting problem, I was very interested in changing the color or style of an image and I wasn't sure of a way to automate the process without manually editing the photo. Formulating k-means and writing efficient code in Eigen that runs the k-means algorithm to extract the bins was a very rewarding experience. I really like the results of recoloring in 9, I think it is really cool to be able to recolor a night image into its day version.

Note I did experiment with adding a function [addSubjectPainting] which takes a mask and input image and then repaints that image on top of another image. I decided to leave it out because it isn't as cool as the k-means, however, this function works, you can use the images mask, liz and china-cheat to verify it works.