CIS username: skww86

For problem 1, I chose to darken the image by subtracting a fixed value from every pixel value across the three colour channels (could be anywhere between 0 and 255 depending on user input). I chose this technique because it would ensure a consistent darkening across the entire image which seemed like the most appropriate method so that the light leak itself would stand out in the image when we eventually add it. I have generated the light leak masks in an external application. To blend the light leak mask with the image, I chose to add together the darkened image with a fraction of the light leak mask - with said fraction being determined from the user input. I thought it would be better to add the whole darkened image to just a fraction of the light leak mask, because we had just darkened the image to an appropriate level, and it didn't make sense to adjust this darkening again in the blending process by an alternate method which, say, could involve using just a fraction of the image. In my masks, the background pixels are black (i.e value 0), so the addition of these pixels to the image has no effect. This ensures the darkened background on the image remains the same, but the pixels where the light leak lies blend with the light leak. This ultimately ensures the final light leak stands out in the image. Note the darkening and the blending with the mask are done concurrently on a pixel by pixel basis to reduce algorithm running time. The algorithm iterates once through every single pixel in the image and changes the value. For an mxn image, this will take m*n time. In the images above, m = n and so the running time is O(n^2).

For problem 2, for the monochrome effect, we first create a new empty image (a noise texture) of the same dimensions as the input image. Salt and pepper noise is applied to the image (with equal probability of salt and pepper). I chose to salt and pepper with a fairly high probability (0.4) because from testing, the more noise, the more ‘grainy’ the image eventually appears which is what we want, so long as the grain isn’t too dominant. All the other pixels are initialised to grey (127) so that they appear ‘neutral’ in the noise texture. Motion blur is then applied to the noise texture to turn the noise into ‘strokes’ - I have done this by applying a convolution filter, as this allows me to easily control the direction of the blur. I have implemented diagonal blur because in real life someone would most likely draw/shade with a pencil in a diagonal direction. Finally, the image is blended with the noise texture; this is done by adding together a user determined fraction of the noise texture with (1-this fraction) of the grayscale image. This ensures the more noise texture, the less image, and vice versa, so the blending is fully adjustable, and adding a fraction of A to 1-fraction of B automatically ensures there can be no value overflow.

For the coloured pencil effect, similar principles apply except we create 2 separate noise textures which we blend into 2 corresponding colour channels. I chose to apply this to the green and red colour channels, with the intention of having red as the dominant colour in the output image, with hints of green (being the opposite hue of red and therefore complementing it well) seeping through to create a ‘crayon’ blend effect. The two noise textures need to be different to do this. So for the red channel noise texture, instead of initialising all pixels to a neutral value

(127) like we do for the green one, we set it to 177 to ensure a richer red base layer. Then when salt & peppering the noise textures, there is a higher probability of a pepper value for the red noise texture, and a lower probability of a pepper value for the green noise texture. Again this is to ensure the output image is more red. We also apply the salt & pepper noise for each noise texture based on two separate independent random variables so as to increase the randomness in the noise between the two textures; we don't want them to be 'in sync'. We then apply motion blur and blend them into the associated channels as before. This algorithm will see us iterating through the image pixels of the empty noise texture to transform each pixel to either 255 or 0 (salt or pepper) depending on the value of a randomly generated number. Since the noise texture is the same size as the input image, for an input image of size n*n like the ones above this will take n^2 time. We undertake some other operations in this algorithm but none of them are asymptotically larger than this salt & pepper iteration, so the total running time is O(n^2). This is the case for both the monochrome and the coloured pencil modes - because in the colour modes although we salt & pepper two noise textures we do this concurrently within the same iteration loops.

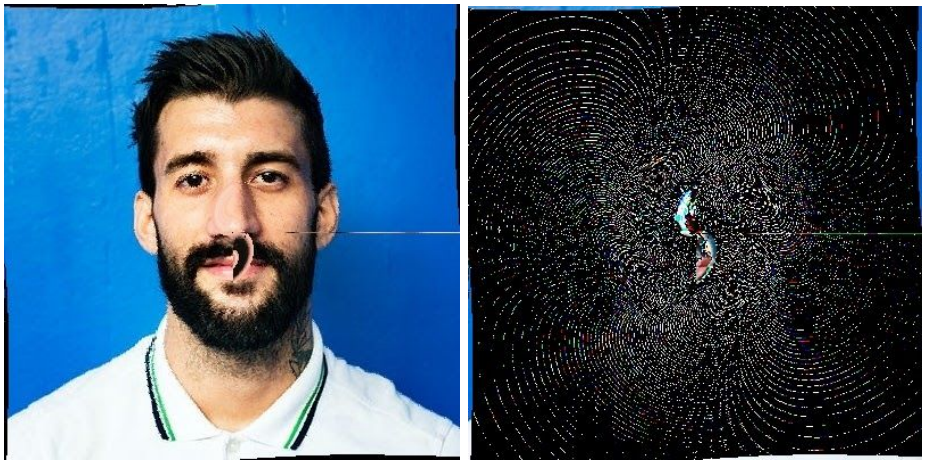

For problem 3, the initial smoothing is done via a Gaussian Blur. I chose this method because the degree of edge blurring is not as significant in comparison to other blurring filters, it is symmetric in the sense that the operation is performed equally in all directions, and the degree of blurring is controllable via the input parameters. I found that kernel dimensions of 5x5, and standard deviation value of 1 (in both directions) seems to work well and does not blur the image too much; we simply want to smooth the image so excessive blurring would not help to beautify the image. These parameters also ensure the brightness of the image is not affected in any noticeable manner which again would not be desirable for what we want to do here. Once the image has been smoothed, we have designed three different colour curves, which will be applied to each of the corresponding channels. For the colour curve that will be applied to the red channel, we have increased the values from the low to mid tones upwards. The purpose of this is to make the skin tones have a 'warmer' look to them, as for the input images in DUO the skin tones will fall in these low-mid to high ranges. There is no need to increase the warmth on the lower tones for this channel as we will be treating these tones differently since the skin tone elements of the images will not fall in these tones. Instead, we have actually slightly decreased the values in the low tone range, as this will make the dark tones a bit more teal in colour, since teal appears in the absence of red. We do this because the DUO input images have dark hair

and so the intention is to add some very subtle ‘cold’ tones to the hair, to make it look a bit more interesting than just natural dark hair, and also because teal and orange are opposites and therefore complement each other well (we will mention orange later). For the colour curve that will be applied to the blue channel, we slightly decrease the values in the higher tones, meaning that the effect will mostly be applied to the skin tone colours for these input images. Yellow appears in the absence of blue, so adding some yellow to the already more red skin tones will give them a slightly more orange look. We increase the values in the lower tones, which should introduce more blue in the darker tones, and therefore for these input images, in the hair, which will go hand in hand with the enhanced levels of teal in these tones as a result of what we did in the red colour curve.
For the colour curve that will be applied to the green channel, we have a minor decrease of the values in the lower tones. Magenta appears in the absence of green, so this will add a touch of magenta to the darker regions, mainly the hair. We increase the values from the low to mid tones upwards, similarly to what we did on the red colour curve, but to a lesser degree. We know that a mix of red and green, that is more favoured towards red (hence the lesser degree of green increase), produces an orange, and we want this applied to these particular ranges of tones because the skin tones fall in this range for these input images. The bulk of this algorithm involves the process of using the lookup table to transform pixel values. The code must iterate through every single pixel in the image and read the lookup table to find its new value (and this is done for 3 colour channels). For an nxn image, this means 3*n^2 iterations in total, so the time complexity is O(n^2).

Note: Bilinear interpolation has been used in the above images

For problem 4, a new empty image is created, and we apply a reverse mapping whereby we iterate through each pixel in the new, empty image, and use the geometric transformation mapping function to generate the corresponding pixel in the original image whose value that pixel should take. The mapping function is a trigonometric function based on polar coordinates. So relative to the selected swirl centre, the polar coordinates of each pixel is established. We then calculate an angle which changes the further we move away from the swirl centre, which creates the whirlpool effect around the centre.

The swirl algorithm iterates through every pixel in the image in order to calculate the value that each one should take. For an n*n image this is n^2 iterations. We do carry out other operations in the algorithm - for example, when doing bilinear interpolation, we sort the 4 nearest pixels - this operation will take O(nlogn) - but neither this nor any other operation is asymptotically larger than O(n^2) so the overall time complexity is O(n^2). The rightmost image above shows where low pass filtering has been applied to the image prior to transformation. Personally I was not able to observe how this process helped to suppress aliasing artifacts, because I was unable to identify any aliasing artifacts in the first image where low pass filtering has not been applied prior to transformation. But I understand how this would help in principle because the blurring would ‘smooth out’ any outlying pixels such that their new values depend on their neighbours (the nature of which depends on the type of blurring filter used). These outlying pixels are the most likely candidates to end up as aliasing artifacts so making the image more consistent by blurring it should suppress such artifacts.



The leftmost image above is the result of the inverse transformation being applied to the already swirled image, and the rightmost image is the subtraction of the leftmost image from the original source image. The subtraction image is mainly black, as I would expect because I would expect the vast majority of pixels to be the same as they are in the input image, because you are simply reversing a transformation back to its original state. Due to a random mapping anomaly, there are some pixels near the centre of the image (just below the nose) that look off - and this difference is picked up in the subtraction image. Other than that, there are some random pixels dotted around the image which imply differences in those pixel values. I would expect this because in the first transformation, we are iterating through integer pixel values and mapping them to new values, which are then approximated to a whole number (via the interpolation method) so that they can be transcribed to a pixel value. But when we carry out the inverse transformation, the approximation is done in the other direction, so the mappings will not always be exactly the same one way to the other. A lot of the pixel values that are different are appearing in a curve/swirl motion in the subtraction image - which is in line with the fact that we are carrying out geometric transformations using trigonometric functions which should map values in the shape of curves.