Image Processing Coursework
Report by htrf88

Problem 1 - Light Leak Filter

I darken the image using a multiplication point transform to reduce the intensity of each pixel by the user-specified darkening coefficient which is a value between 0 and 1 using the following formula — the darkened image = the original image * (1 - darkening coefficient).

The images above show the original image, gamma-corrected image and image darkened using point transform respectively. I chose to use multiplication point transform over gamma correction because it reduces the intensity of each pixel by a constant amount giving the impression that the photo was taken in a darker environment and I felt that the multiplication point transform gives an image that looks more like the 1st reference photo compared to gamma correction.

To generate the light beam I simply generated a mask of all pixels initially set to [0,0,0]. Then I set a narrow vertical section of the image to either white light ([255,255,255]) or rainbow (generated using the Python's coloursys module) depending on the mode. I multiplied the columns of the light beam by values of the sine function in range 0 to π to make the light beam softer around the edges for a better blending effect and then shift the columns slightly to replicate the tilted beam in the reference photo.

Finally, I use weighted addition to blend the mask and the image using the formula below to generate the images show after it

Final image = darkened image * (1 - blend coefficient) + mask * blend coefficient



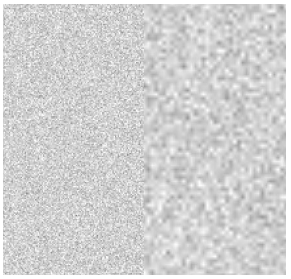

The computational complexity of the filter for an image of size $n \times n$ is $O(n^2)$ as the operations performed on the image are various types of point transforms. I also use OpenCV's Gaussian blur

to remove aliasing introduced in the mask by tilting the beam which has a runtime of $O(n^2k^2)$ for a $k \times k$ kernel. I use a fixed size (9*9) kernel, so the asymptotic runtime is $O(n^2)$.

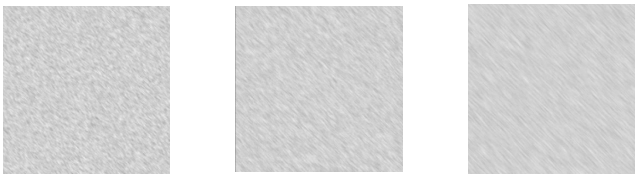
Problem 2 - Pencil / Charcoal Effect

I began by converting the image to grayscale using the formula 0.2989 * red channel + 0.587 * green channel + 0.114 * blue channel. Then blur is applied to the image using Gaussian blur with a 75*75 kernel and sigma X = sigma Y = 0. The grey image is then divided by the blurred image to obtain the sketch. Using a bigger kernel increases the computation complexity of the filter, but it provides a more detailed sketch, so I tried to find a balance between the two.

To generate the noise texture I begin by generating a 2D array of the same dimensions as the image and filling it with Gaussian noise. I then use a zoomed-in version of this image to increase the size of the grains and smooth it using bilinear interpolation. In the image on the left, the left half of the image is the original noise and the right half is the zoomed-in (x3) and smoothed noise using bilinear interpolation. The amount of zoom determines the size of the grains.



Finally, to generate the pencil stroke effect, I used OpenCV's filter2d function on the texture to create a motion blur effect where the kernel is an identity matrix of the user-specified size. The length of the pencil strokes is directly proportional to the size of the matrix as shown in the images below with kernel size 3*3, 5*5 and 9*9



I then combined the sketch with this texture to generate the final image for the monochrome mode using the same weighted addition formula as stated in problem 1.

To create the coloured sketch effect, I create 2 additional textures with motion blur applied in different directions and strokes of different size and length then combine them with the previously generated sketch effect so that I have 3 distinct sketch + texture images. I then multiplied each image by the red, blue or green value of a user chosen colour (FFA600 by default) and divided by 255 so that each image represents the red, blue or green channel of the colour sketch and merged them to form the colour sketch.

The following images show the final result for both colour modes

In terms of the computation complexity, the grayscale conversion is done in $O(n^2)$. To generate the sketch Gaussian blur is applied in $O(n^2k^2)$ time, in my implementation k is 75. Dividing the gray image by the blurred one takes $O(n^2)$ time. To generate the texture, bilinear interpolation is applied to smooth out the noise which is done in $O(n^2)$ time and motion blur that is applied to this in $O(n^2k^2)$ time where k is user specified stroke length.

Combining the sketch and the texture takes $O(n^2)$ time. For a coloured sketch this whole process is repeated 3 times, so, asymptotically it's still done in $O(n^2)$ time.

Problem 3 - Smoothing and Beautifying Filter

I began by smoothing the image using a median filter with a 3*3 kernel (kernel size can be changed by the user). I chose to use median filter as it is very effective against salt-pepper noise, speckle noise, and it is better at preserving edges compared to the Gaussian filter which makes it ideal for smoothing small irregularities on the subject's skin without making the blurring look too obvious.

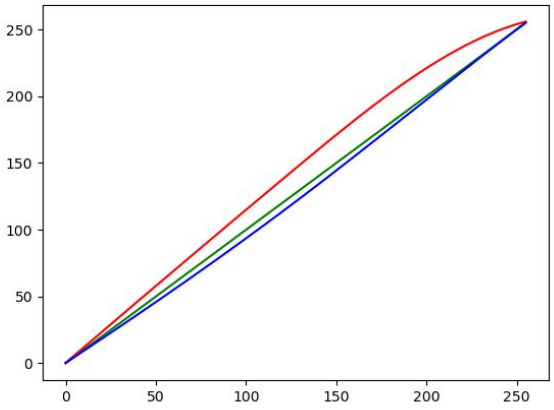
After blurring the images I applied contrast limited adaptive histogram equalization to improve the dynamic range of the image without making the image look overexposed to give them a "portrait mode" effect.

The images below show the original image, image blurred using median filter and the image after applying CLAHE respectively.



To beautify the image I wanted to create a filter that adds the atmosphere of autumn to the image i.e. the filter boosts the intensity of red and orange colours in the image, so, I applied the colour curve shown below to the image using lookup table transformation to boost the intensity of red colours and decrease the intensity of blue slightly to achieve that effect.

The colour curve applied to the images and the final image generated by the filter —




In terms of computational complexity, the median filter is applied to the image in $O(n^2k^2)$ time for an $n \times n$ image with a $k \times k$ kernel (k is the user specified smoothing factor, 3 by default) and the lookup table enhancement is applied in $O(n^2)$ time.

Problem 4 - Face Swirl

I begin by preprocessing the image using a 3*3 box filter which is a low pass filter. I used this filter because it is a low pass filter which can be used to remove aliasing from the input image, it is often used to approximate Gaussian blur, but it's not as costly as applying a Gaussian blur.

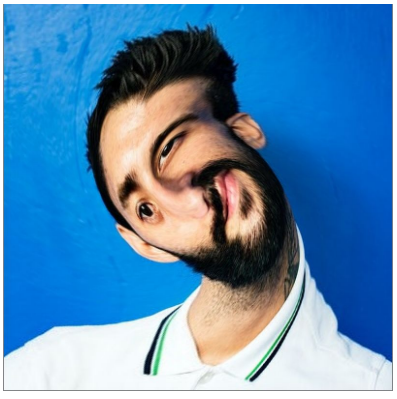
After blurring the image, the program iterates through each pixel in the image calculating its distance from the centre. If the distance is less than the radius of the swirl specified then, the amount of rotation that needs to be applied to the point is calculated using the following formula

Swirl amount = 0.1 * 10^(1 - (distance from centre/swirl radius)) - 0.1

After some experimentation, I found that using this exponential function above instead of a linear function to calculate the swirl amount produced an image that was very close to the reference image, and it allows for the edges to blend more naturally with the rest of the image. After that, the amount calculated using the following formula is subtracted from the argument of the current coordinate as such

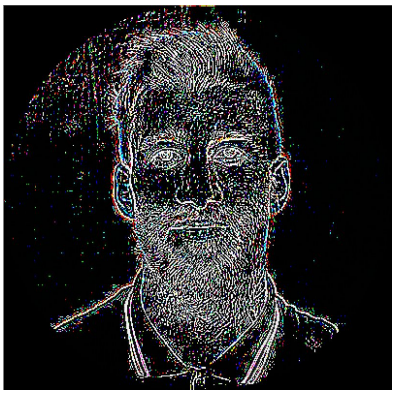
θ = θ - (swirl amount * π * swirl strength)

After calculating the new value of θ, the Cartesian coordinates of the pixel are recalculated. To translate these coordinates into pixel value I initially used the nearest neighbour interpolation as seen in the left image below, but, when compared with bilinear interpolation (centre) there was clear evidence of aliasing. After preprocessing using a low pass box filter and then using bilinear interpolation the resulting image as seen below on the right, has the least amount of aliasing.

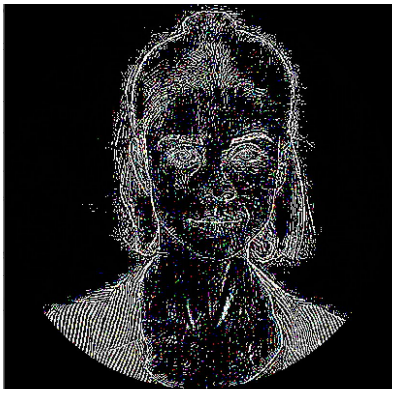

The original image, the unswirled image, and their difference is visualised on the next page respectively. As seen in the picture on right, the outline of the subject and the uneven texture on the wall within the swirl radius i.e. features in the image with edges or places with large contrast can't be unswirled to their original value. On the other hand, features with flat surfaces can be mapped to their original location very well. I think this is because the details of edges, areas of contrast, etc. are changed more in the swirled image when they are interpolated because the surrounding pixels have more variation so their interpolated value is quite different from their original value. Conversely, flat surfaces have less variation between neighbouring pixels so their interpolated value is very similar to their original value thus producing minimal difference.





This effect can be strongly seen in the difference between the original face1.jpg and swirled and unswirled face1.jpg included below with the original on the left, swirled and unswirled image in the middle and their difference on the right.





We can see that the background in face1.jpg is a solid colour with no irregularities and there's almost no difference between the background of the original and processed image. In contrast, from the difference we can see the stripes on the shirt which have sharp edges are very blurred compared to the original, there is also the same loss of information around the outline of the face and hair, shadows around the collarbone which are all areas of sharp edges or higher contrast.

Applying the box blur filter takes $O(n^2k^2)$ time where $k = 3$ in my implementation, so the asymptotic runtime is $O(n^2)$. In this geometric transform the new position of each pixel has to calculated which takes $O(n^2)$ time for an $n \times n$ image. The new coordinates of the point are interpolated in constant time.