Username: lzhc88

Image Processing coursework

In the following report I will explain my code for the coursework for every exercise in the following order. First, I will summarize my initial idea and describe my actual implementation. Then, I will justify my code, briefly discuss the computational complexity and then discuss any specifics that were requested in a problem.

Simple light leak effect; darkening coefficient: 0.8, blending coefficient: 0.7

Rainbow light leak effect; darkening coefficient: 0.8, blending coefficient: 0.7

For problem 1 I created the mask to create the light leak effect with code. After observing a ray of light in a picture, I noticed a few things that would make it look as real as possible. First, a ray looks more natural if it goes diagonally down. Second, light bends at surface corners. Third, if light shines down, the ray gets slimmer. And lastly, dark surfaces do not reflect light well.

So, the basis of my mask are black pixels, with white pixels in a width of 25 pixels going from about the middle of the image, down towards the right corner. 100 pixels are used as the range for the ray to move diagonally right. Next, bending the light was a difficult process, since I could not use any face recognition. For this purpose, I used a lot of different variables to keep track of when to bend and when to make the ray slimmer. These variables change after x many rows, signalling to my program for the ray to bend more or for it to get slimmer. They also depend on how dark the pixel area is, i.e., if an area at about 80% the length of the image is darker, then I guessed that might be the end of the face, so the ray should bend there. Admittedly, the coding has been customised to bend perfectly for the two images provided, but it might not perform as well on other images, because it does not use actual face recognition. To recognize the darker areas of the image, I used thresholding on the grey copy of the image with three different thresholds of 20, 50 and 80. These were the most promising thresholds after lots of testing to obtain the most important contours of the face. These thresholded copies of the image were not only useful to find the contours of the face for bending but were most important to determine which area of the face would not reflect the ray of light as well. The ray should be darker in those areas. Lastly, I blurred the ray of light, so that the ray would not look so pixeled.

To implement the rainbow leak, I used the same algorithm as for the simple leak but changed the ray width to be broader and made a variable to change colour after a predetermined number of columns in the ray for every row. Then I added the simple mask to this new mask, weighted 3:2.

For me creating the light leak with code seemed like a better idea to be able to use it for other images as well. The computational complexity is O(n*m), where n is the horizontal

length and m is the vertical length of the image. This complexity results from running the for loops in range rows and columns to darken the image, to add the image and mask weighted and to make the mask.

Monochrome pencil effect; blending coefficient: 0.7

Coloured pencil effect; blending coefficient: 0.7

For problem 2 I created a salt and pepper noise texture for the monochrome pencil effect. I thought that this would be a suitable noise texture because to recreate strokes of the pencil there must be black pixels randomly in some order from which these strokes originate. Then I applied a horizontal motion blur to this mask to create pencil stroke like effects. For this I originally used the following method: For a pixel (x,y) in the mask, calculate the mean of this pixel and the horizontally neighbouring 4 pixels to the right and left and set this as the new value of pixel (x,y). But then, it was mentioned in the FAQ that we were allowed to use the function cv2.filter2D(), which makes convolution much faster than hard-coding it. Hence, I made a kernel to do the same operations as stated above and extended it to take the mean from the 10 neighbouring pixels to the right, left, above and under the pixel (x,y). In this way I could simulate not only horizontal pencil strokes, but also vertical pencil strokes that were longer than in my original implementation.

To extend my function for the coloured pencil effect, I created a new noise texture that assigns each pixel in the texture a random integer in the range (0,255). Then I applied a diagonal motion blur on this texture. Instead of calculating the new value of each pixel with the mean of the horizontally neighbouring 4 pixels on each side, the diagonal blur calculates the mean with the diagonally neighbouring 4 pixels from the upper left side to the lower right side. I later changed this to work with cv2.filter2D() as well, to shorten the runtime of my program. Then, I blended both the first and the second noise with the grey copy of the original image and created a BGR image by merging these two images with the grey copy of the original image. It was crucial to use a different noise texture and motion blur effect on the two channels, so that one effect would not cover the other. I noticed that the pixels in the second noisy image were lighter than in the first noisy image, so I multiplied the second noise texture by 0.95.

The running time of problem 2 is O(k*n*m). As in problem 1, when blending the images weighted, multiplying each pixel by a percentage of its own value takes n*m time. Moreover, creating the noise takes n*m time. The convolution for the motion blur is what could take the most time, since for each pixel it takes kernel size (k) time, which in the case of a 21*21 kernel would be 441*n*m, but using filter2D() shortens the running time, because it says in

Username: lzhc88

the description that the function uses the DFT-based algorithm for all kernels larger than 11*11.

"Summer Blues"; blur amount: 30

In problem 3 I first implemented the gaussian blur function to smooth the features of the face, but then I noticed that this blurred too much of the face. For example, the eyes, which are one of the most important features of the face where blurred and all in all the original image seemed to look better. Hence, I implemented the bilinear filter instead. This has the advantage of smoothing areas of the image where the pixels are similar in brightness, but stopping at edges, i.e., pixel areas where the pixels have vastly different brightness. Only the brightness here is considered, so the input image must be split into its three colour channels to perform this operation.

For the beautification with lookup table transformation, I hard coded a few lookup tables and applied them to each bgr channel. I aimed to make a filter that made the image look refreshing, but most importantly also beautified the skin-tone. After testing with different values in the lookup table, applying a lookup-table that mapped lower pixel values slightly lower than the original pixel value and the higher two-third of the values until 200 to a slightly higher value, to the r-channel worked well for the skin-tone. The skin-tone is transformed to a lighter and less reddish tone, which I appreciate in a filter from an Asian-beauty point of view. Next, I tried applying other lookup-tables on the b- and g-channels, but some of them just made the skin look reddish again. In the end, I used the same look-up table for the b- and g-channel, which maps the original pixel values to higher pixel values. This gives the image a refreshing, but subtle mint-tinge. The subtlety of this filter goes with the trend of beautification filters where people who have not seen the original image cannot see that the image was beautified. Since the image after applying these transformations reminds me of cool summer breeze, I have named this filter “Summer Blues”.

Regarding the complexity of problem 3, I will first explain the complexity of the bilinear filtering. For every pixel, a new kernel is computed, which takes O(k=d**2) time. Then to compute the new value, it takes another O(k) time. All in all, the bilinear filter takes O(k*n*m) time. In comparison to the run time of a gaussian blur, one must consider that the kernel must be recomputed for every pixel and then applied to the pixel, hence there is not only one operation that runs in O(k) time. The constants that disappear with the big-O notation turn out not to be trivial, because the run-time with the bilinear filtering with a kernel size of 3 is about two to three minutes on my laptop. The gaussian filter in comparison does not take half this time, but performs worse for the beautification.

Username: lzhc88

Swirl image nearest neighbour interpolation; strength of swirl: -0.4, radius of swirl: 150

Swirl image bilinear interpolation; strength of swirl: -0.4, radius of swirl: 150

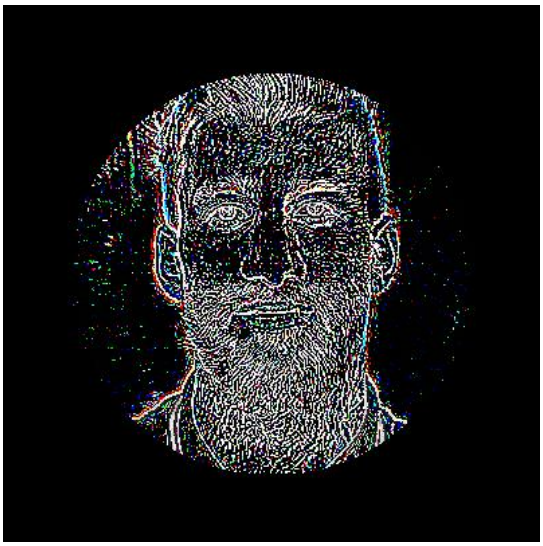
Swirl image with prefiltering; strength of swirl: -0.4, radius of swirl: 150

De-swirled image, strength of swirl: -0.4, radius of swirl: 150

For problem 4, to create a swirl in the middle of the image, I first changed the x,y coordinates in my nested for-loop to be centred in the middle of the image. Then I changed the coordinates into polar coordinates. Changing the cartesian coordinates into polar coordinates is helpful to do operations that involve the distance to the middle of the image and rotation of pixels around the centre. Now, to implement a swirl and not a rotation around a specific radius from the centre, the rotation of each pixel was calculated considering its distance from the centre. In case of the nearest neighbour interpolation, the pixel values from the nearest coordinates to the float values obtained by the swirl function are taken for the new pixel value at coordinates x,y. The bilinear interpolation technique in comparison, takes the four coordinates in consideration that are closest to the float values obtained by the swirl function and calculates the new pixel value for x,y by weighting each of the four cells by distance and then averaging their pixel values. Comparing the two implementations, the bilinear interpolation makes the resulting image have less artifacts and produces a smoother result. Hence, I have commented out the nearest neighbour interpolation in my code.

Username: lzhc88

Subtraction image



The runtime for problem 4 is O(n*m), because the swirling only involves a constant number of operations for each pixel in the image. I used a gaussian filter for prefiltering in part 2 of problem 4, which runs in O(k*n*m) steps. For face 2 this did not supress aliasing artifacts as much as I had hoped. The swirl can still be seen on the blue background wall. For part 3, I made my function run the swirl again, this time over the original image without prefiltering and then inverted this transformation by running the same algorithm to swirl the image but using the negative value for the angle. Subtracting the original image from the de-swirled image showed that the error from the swirling and de-swirling is bigger than it seemed from the output image. Especially for regions with many big differences in colour between neighbouring pixels such as the lines of the shirt, hair and the eye there are many errors in pixel values.

My code can be run from the command line, but also in the python file. Please look at the example_commands.txt file to see how to run the commands. I have tried to value check for the parameters and if the commands are run from the command line, I have set default parameters for when the given parameter values are not valid. If the commands are given in the python file, my code only value checks and returns with an error message, if the values are not in the value format specified.