Cartoonization Image Processing Demo GUI

By: Aravind Narayanan

Introduction and Objective:

I have created a GUI using Python, which uses concepts from EE 440 to achieve a cartoonization effect. This report describes the effect, outlines how to use my program, and explains how I implemented the effects.

Cartoonization:

Cartoonization is an effect which makes an image look like a cartoon. This effect is achieved using bilateral filtering followed by adaptive thresholding. Bilateral filtering is form of low-pass filtering which denoises the original image without blurring the edges. This is done so the fine details of the image are filtered out – to make the image seem more like a drawing. The next step is adaptive thresholding. This step detects the edges in the original image and enhances them. This is done because edges are usually thick in cartoons. Finally, the images from the two steps are combined together, to get the final result.

Program Instructions:

From the "cartoonization" GitHub repository, download "Cartoonization GUI_Aravind Narayanan.ipynb," as well as the three image files. **Make sure all of these downloads are in the same folder.** To start the GUI, open the ipynb file press the "Run" button on whatever IDE you are using, to run the code. The GUI will first load up with the image of the rose ("rose.jpeg").

Cartoonizing an Image:

To cartoonize an image, use the slider in the bottom middle of the widget. The further right you move the slider, the more cartoonized the image will get. The further left you move the slider, the less cartoonized the image will get. The number on the slider specifies the sigma value used for bilateral filtering (σ_s and σ_r). The original image will show on the top left of the widget, and the modified image will show on the top right of the widget. When the slider value is at 0, the image won't be cartoonized at all (the modified image will match the original image).

Resetting an Image:

Whenever you want to reset the modified image back to original, click the "Reset" button on the bottom left of the widget. This will reset the modified image, and also reset the slider to 0.

Picking a New Image:

Whenever you want to pick a new image, click the "Load New Image" button on the bottom right of the widget. This will take you to your file directory where you can pick any image. **Make sure the image file is in the same folder as the Python program!** This button also resets the slider to 0.

Implementation Description:

GUI Implementation:

The GUI is implemented using *Tkinter*. There are three frames in the widget. Frame 1 is for the images, Frame 2 is for the labels, and Frame 3 is for the features. Frame 1 has two canvasses, Canvas 0 and Canvas 1. Canvas 0 is for the original image, and Canvas 1 is for the modified image. Frame 2 has two cavasses, Canvas 2 and Canvas 3. Canvas 2 is for the "Original Photo" label, and Canvas 3 is for the "Modified Photo" label. Using PIL, the images are converted from the *NumPy ndarray* format to the *PhotoImage* format, in order to be displayed on the canvasses.

Cartoonization Implementation:

The slider is saved in a variable called *self.scl_car*, implemented using *tk.Scale()*. The *command* argument is set to *self.cartoonize*. The *cartoonize()* callback function takes in the value from *self.scl_car* and saves it in the variable *sigma_value*. If this value is 0, the modified image is just set to the original image. For a nonzero value, there are a lot of steps. First, the original image is filtered with a bilateral filter, using *cv2.bilateralFilter()*. The result is saved in *bil_filter*. The arguments *SigmaColor* and *SigmaSpace* are both set to *sigma_value*. According to the documentation for *cv2.bilateralFilter()*, when the sigma values are at 150, there will be a heavy cartoonish effect on the image [1]. This is why I set the slider's maximum value to 150. The next step is to do adaptive thresholding on the original image. First, the original image is converted to grayscale using *cv2.cvtColor()*, and the result is saved in *cv_img_grayscale*. Then, the image is filtered with a median filter using *cv2.medianBlur()*, in order to get rid of salt-and-pepper noise, and the result is saved in *cv_img_grayscale_blurred*. Now we have a noise-free grayscale image, so we can do edge detection. This is done with adaptive thresholding, using *cv2.adaptiveThreshold()*. In this function, the argument *C* is what determines how thick the edges will be; the lower the *C* value, the thicker the edges. We want the edges to get thicker as the image gets more cartoonized, so *C* should be inversely proportional to *sigma_value*. I use the function *c_value = int(-0.08*sigma_value + 15)*, to determine *C*. This outputs *C* values between 3 and 15, which I found was a perfect range to use. The result after adaptive

thresholding is saved in *cv_img_grayscale_edges*. This is then converted back to color using *cv2.cvtColor()*, and the result is saved in *cv_img_color_edges*. Finally, the bilaterally filtered image (*bil_filter*) is combined with the edge mask (*cv_img_color_edges*) using *cv2.bitwise_and()*, and saved in *self.NEWcv_img*. This is the final result, which is converted to the *PhotoImage* format and displayed in Canvas 1.

Reset Implementation:

The button is saved in a variable called *self.btn_reset*, implemented using *tk.Button()*. The command argument is set to *self.reset*. The *reset()* callback function first sets the modified image (*self.NEWcv_img*) equal to the original image (*self.cv_img*). *self.NEWcv_img* is then converted to the *PhotoImage* format and displayed in Canvas 1. Finally, the slider is reset to zero, by doing *self.scl_car.set(0)*.

Load New Image Implementation:

The button is saved in a variable called *self.btn_load*, implemented using *tk.Button()*. The command argument is set to *self.load*. The *load()* callback function asks for a file name using *filedialog.askopenfilename()*. The file name is saved in *self.cv_img*. The image is read in using *cv2.imread()* and saved in *img*. Next, it is converted from BGR format to RGB format using *cv2.cvtColor()*, and saved in *self.cv_img*. *self.cv_img* is then converted to the *PhotoImage* format and displayed in Canvas 0. Finally, the *reset()* function is called to reset the modified image and set the slider to zero.

Results discussion and output images:

The parameters that the user directly controls are the *SigmaColor* and *SigmaSpace* arguments in *cv2.bilateralFilter()*. These two parameters are both set to the value from the slider. As sigma increases, more and more high frequency elements from the original image get filtered out, smoothing out the image. The argument *C* in the function *cv2.adaptiveThreshold()* is also tied to sigma. The lower the *C*, the thicker the edges, so I set *C* to decrease as sigma increases. To summarize, the image gets more cartoonized as sigma increases and *C* decreases. Below are some examples of my program in action.

From left to right, here you can see the original images, the images modified with a slider value of 80, and the images modified with a slider value of 150, respectively. As you can see, the further right you move the slider, the more the fine details get smoothed out, and the thicker the edges get.

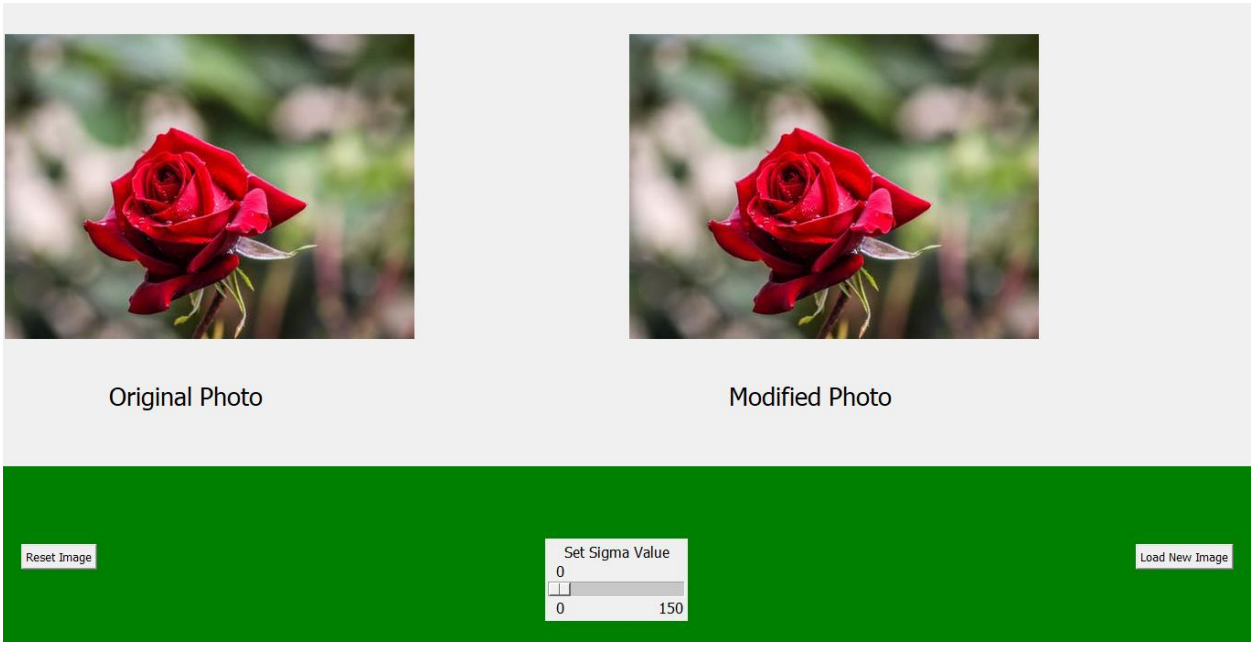
Future work/Things I noticed:

Large images:

I noticed that when moving the slider for the "city.jpg," the program lagged a bit. I think that is because cv2.bilateralFilter() is slow for large images. I've seen some sites that said that it is quicker to run smaller bilateral filters repeatedly, rather than run a large bilateral filter once [2]. In a future version of this program, I can working on changing the code to run smaller filters and lessen the runtime.

GUI Issue:

I noticed that the GUI loses symmetry when the widget is expanded:



This is because the image and label positions are based on the constant "MAXDIM," rather than a variable. I would have to base the image and label positions on the size of the widget in order for the GUI to maintain symmetry no matter the widget size. In a future version of this program, I can figure out how to keep track of the widget size, and alter arguments accordingly.

Giving the User More Freedom:

In this version, the user only controls one slider, and three arguments depend on this slider: *SigmaColor*, *SigmaSpace*, and *C*. In a future version, I can add two more sliders, so the user can control all three parameters independently. After all, there is no rule that SigmaColor and SigmaSpace need to be the same; I just made it that way for simplicity's sake. It would also be fun to make the edge thickness independent from how bilaterally filtered the image is.

References:

[1] "Image Filtering," *OpenCV*. [Online]. Available: https://docs.opencv.org/master/d4/d86/group__imgproc__filter.html. [Accessed: 10-Dec-2020].

[2] Unknown, "How to create a cool cartoon effect with OpenCV and Python," *Ask a Swiss*, 01-Jan-1970. [Online]. Available: https://www.askaswiss.com/2016/01/how-to-create-cartoon-effect-opencv-python.html. [Accessed: 10-Dec-2020].