E E 440 Final Project Report

Jenny Cho

University of Washington, Seattle
November 2019

1 Summary

This project summarizes the content learned in E E 440. This project involves building a graphical user interface (GUI) using Python and the `tkinter` package. Several effects and filters are implemented. Some are built using `OpenCV` and others are written with personal code (non-built-in functions).

2 GUI Description

2.1 GUI Implementation

Unlike MATLAB, Python does not have a convenient way to create a GUI. However, using packages such as `tkinter`, `cv2`, and PIL, building a GUI with image display capabilities became possible. The interface was mainly built with `tkinter`'s Frame and Canvas widgets. The adjustable parameter of each effect was implemented using the Scale widget [1]. Creating a visually intuitive and aesthetic was difficult, but it was rewarding to play around with `tkinter` and was a valuable experience designing a GUI with Python.

2.2 Instructions (How-to-use)

There are two methods to run the GUI.

1. Run `.py` file in conda environment, or

2. Run the first cell of the provided Jupyter Notebook.

Use the second method only if conda environment cannot be accessed. Figure 1 shows the GUI. The image on the left displays the original image. The image on the right displays the changing image that renders the effect of the filter selected. Each filter can be applied to the image one at a time. The original goal was to build each effect on top of each other, but due to the purpose and scope of the project this feature was not implemented.

Figure 1: Graphical user interface created by Jenny Cho

The *Reset* button allows user to reset all the sliders to their default value. The *Load* button allows the user to upload any image from their machine. This is shown in Figure 2.

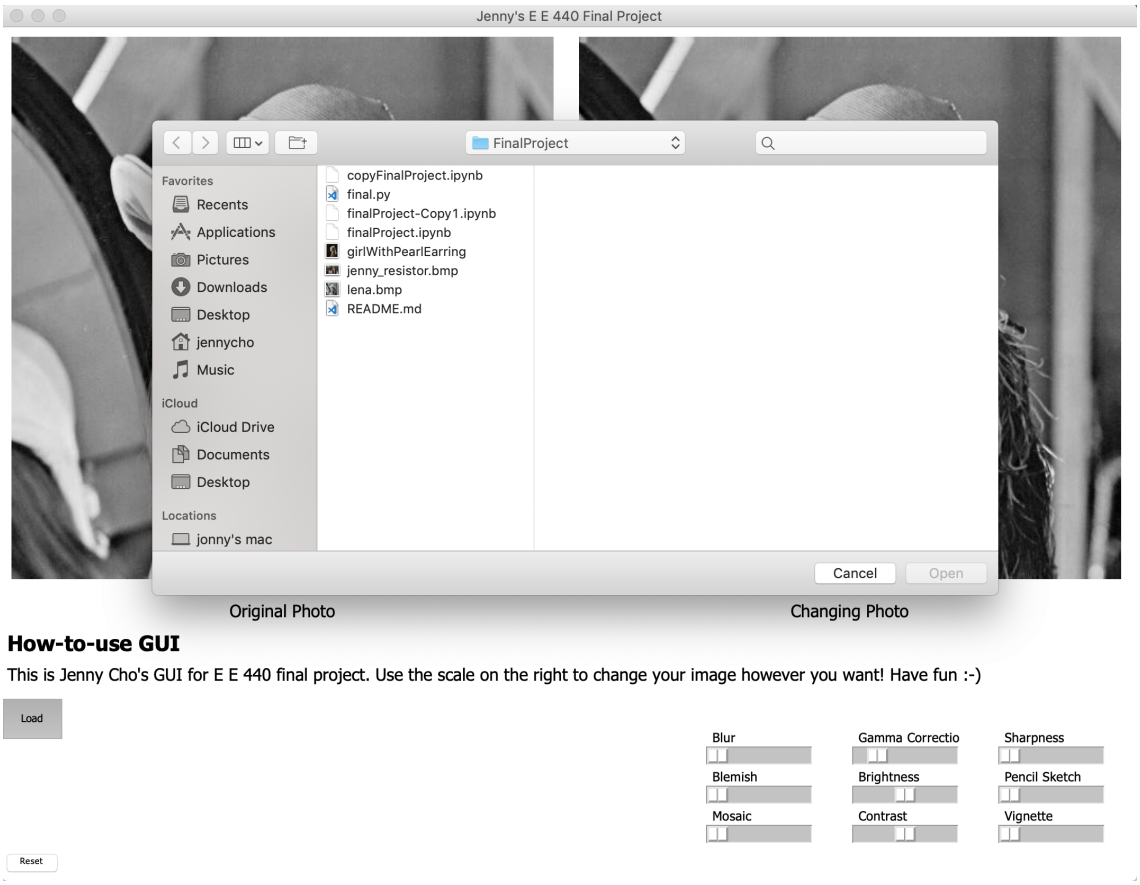

Figure 2: Load button

3 Effects

This project consists of nine different effects: blur, blemish, mosaic, gamma correction, brightness, contrast, sharpness, pencil sketch, and vignette. The project first started off with simple filters, such as blurring with a built-in average filter, then moved towards more advanced and self-implemented effects.

3.1 Blur

The blur effect was achieved by `cv2.blur()` (see Figure 3). This effect uses an averaging filter with dimensions determined by k, user input from the scale bar. Figure 4 shows an example of the 'blur' effect. This effect is one of the less interesting ones because it uses OpenCV's built-in function.

```
def blur_image(self, k):
    k = self.scl_blur.get()
    self.NEWcv_img = cv2.blur(self.cv_img, (k, k))
    self.photo = PIL.ImageTk.PhotoImage(image = PIL.Image.fromarray(self.NEWcv_img))
    self.canvas1.create_image(MAXDIM//2, MAXDIM//2, image=self.photo, anchor=tk.CENTER)
```

Figure 3: Blur filter code

Figure 4: Blur filter, k = 30

3.2 Blemish

```
def decBlemish_image(self, k):
    k = self.scl_blmsh.get()
    # cancel out the effect
    if k == 0:
        self.photo = PIL.ImageTk.PhotoImage(image = PIL.Image.fromarray(self.cv_img))
        self.canvas1.create_image(MAXDIM//2, MAXDIM//2, image=self.photo, anchor=tk.CENTER)
    else:
        sigmaColor = 75
        sigmaSpace = 10
        self.NEWcv_img = cv2.bilateralFilter(self.cv_img, k*4, sigmaColor, sigmaSpace)
        self.photo = PIL.ImageTk.PhotoImage(image = PIL.Image.fromarray(self.NEWcv_img))
        self.canvas1.create_image(MAXDIM//2, MAXDIM//2, image=self.photo, anchor=tk.CENTER)
```

Figure 5: Blemish filter code

The blemish effect was achieved by `cv2.bilateralFilter()` (see Figure 5). This effect uses a Gaussian filter with edge consideration to smooth out similar surfaces of the image. The greater the k value, the smoother the image. Figure 6 shows an example of the 'blemish' effect. This is the other less interesting effect.

Figure 6: Blemish filter, k = 10

3.3 Mosaic

Given an image and a parameter k, a mosaic effect is applied to the image. Figure 7 shows the full implementation. In summary, k is the number of pixels within a bigger pixel of the image. For example, if $k = 3$, then each bigger pixel will be a $3 * 3$ bigger "pixel" where the value is determined from sampling the original image, every $k = 3$ pixels in both x and y direction. Figure 8 shows an example of the mosaic effect. The effect works nicely. There are some values of k that cannot divide the original image perfectly. This result is negligible because it only applies to the few columns at the right-edge of the image. Thus, this effect can be improved by perfecting the effect on the edges for all values of k.

3.4 Gamma Correction

Gamma correction is also known as the Power Law Transform. First, our image pixel intensities must be scaled from the range [0, 255] to [0, 1.0]. From there, we obtain our output gamma corrected image by applying the following equation:

$$O = I^{1/G}$$

```python
# Mosaic Effect
def mosaic_effect(self, k):
    img_mosc = np.zeros_like(self.cv_img)
    for ch in range(3):  # all three bgr channels
        img = self.cv_img[:,:,ch]

        # save a "small photo" for every "k"
        img_small = img[0::k, 0::k]
        h, w = img_small.shape

        # new image frame
        x, y, ignore = self.cv_img.shape
        img_ch = np.zeros((x,y))

        # fill picture with mosaic-ed pixels
        for i in range(h):
            for j in range(w):
                if ((i*k) < img.shape[0]) or ((j*k) < img.shape[1]):
                    img_ch[(i*k):(i*k)+k, (j*k):(j*k)+k] = img_small[i][j]

        img_mosc[:,:,ch] = img_ch
    return img_mosc

# Callback for the "Mosaic" Scale
def mosaic_image(self, k):
    k = self.scl_mosc.get()
    if k == 1:
        self.photo = PIL.ImageTk.PhotoImage(image = PIL.Image.fromarray(self.cv_img))
        self.canvas1.create_image(MAXDIM//2, MAXDIM//2, image=self.photo, anchor=tk.CENTER)
    else:
        self.NEWcv_img = self.mosaic_effect(k)
        self.photo = PIL.ImageTk.PhotoImage(image = PIL.Image.fromarray(self.NEWcv_img))
        self.canvas1.create_image(MAXDIM//2, MAXDIM//2, image=self.photo, anchor=tk.CENTER)
```

Figure 7: Mosaic filter code

Figure 8: Mosaic filter, k = 15

where I is our input image and G is our gamma value. The output image O is then scaled back to the range [0, 255]. Gamma values < 1 will shift the image towards the darker end of the spectrum while gamma values > 1 will make the image appear lighter. A gamma value of $G = 1$ will have no affect on the input image.

```
# Gamma Correction / Power Law Transform
#   build a lookup table mapping the pixel values [0, 255] to their adjusted gamma values
def gamma_effect(self, gamma=2):
    invGamma = 1.0 / gamma
    table = np.array([((i / 255.0) ** invGamma) * 255 for i in np.arange(0, 256)]).astype(np.uint8)

    # apply gamma correction using the lookup table
    return cv2.LUT(self.cv_img, table)

# Callback for the "Gamma Correction" Scale
# reference  >> https://www.pyimagesearch.com/2015/10/05/opencv-gamma-correction/
def gamma_image(self, k):
    k = self.scl_gamma.get()
    self.NEWcv_img = self.gamma_effect(gamma=k)
    self.photo = PIL.ImageTk.PhotoImage(image=PIL.Image.fromarray(self.NEWcv_img))
    self.canvas1.create_image(MAXDIM//2, MAXDIM//2, image=self.photo, anchor=tk.CENTER)
```

Figure 9: Gamma correction code

Figure 10: Gamma correction filter, k = 2.00

3.5 Brightness

Figure 11 shows an example of the brightness effect. Given an image and a parameter k, the brightness is increased and decreased. For the purpose of showing effects, k was scaled

Figure 11: Brightness filter, k = 15

to be five times the given value. The operation was done on the value channel of the image (the value channel of gray scale images are the same as its original pixel value), increasing in brightness for $k > 0$ and decreasing for $k < 0$. A difficult part of this code came from controlling the wraparound since the pixels need to have a value $[0, 255]$. Since the image array data type is `uint8`, addition that yielded a value greater than 255 would overflow to $0, 1, 2, ..., N$. This was addressed by casting image to `uint16` then adding the parameter value. After the operation, the pixel values greater than 255 were floored to 255. Any values less than 0 was changed to 0. This is repeated for all three RGB channels of the image.

Initially, only the intensity channel of the image was changed (only applies to color images). This yielded an image where the color of the image would change as the brightness increased (see Figure 12). This was addressed by applying the brightness change on all three color channels (RGB) (see Figure 13). Figure 14 shows the difference between changing just the value channel (a) vs. changing every color channel (b). The second method yields a much better result.

```
# Brightness Effect
def brightness_effect(self, k):
    img = cv2.cvtColor(self.cv_img, cv2.COLOR_BGR2HSV)
    img_v = (img[:,:,2].astype(np.uint16))+(k*5)  # add 5*k value channel
    img_v[img_v < 0] = 0      # adjust wrap around pixel (bottom)
    img_v[img_v > 255] = 255  # adjust overflow (top)
    img[:,:,2] = img_v.astype(np.uint8)
    img = cv2.cvtColor(img, cv2.COLOR_HSV2BGR)
    return img
```

Figure 12: Brightness code – change value only

```
def brightness_effect(self, k):
    img_new = np.zeros_like(self.cv_img)
    if np.array_equal(self.cv_img[:,:,0],self.cv_img[:,:,1]):  # grayscale
        img_ch = (self.cv_img[:,:,0].astype(np.uint16))+(k*5)  # add 5*k value channel
        img_ch[img_ch < 0] = 0      # adjust wrap around pixel (bottom)
        img_ch[img_ch > 255] = 255  # adjust overflow (top)
        img_new[:,:,0] = img_ch
        img_new[:,:,1] = img_ch
        img_new[:,:,2] = img_ch
    else:
        for ch in range(3):  # all three bgr channels
            img_ch = (self.cv_img[:,:,ch].astype(np.uint16))+(k*5)  # add 5*k value channel
            img_ch[img_ch < 0] = 0      # adjust wrap around pixel (bottom)
            img_ch[img_ch > 255] = 255  # adjust overflow (top)
            img_new[:,:,ch] = img_ch
    return img_new
```

Figure 13: Brightness code – change all RGB

(a) Value channel (b) All RGB channels

Figure 14: Comparison of brightness implementation, k = 25

3.6 Contrast

Contrast was adjusted using a piece-wise curve. There were three lines that made up the transformation curve. Many potential values of (r_1, s_1) and (r_2, s_2) exist, and there is no "correct" point to choose as long as the scale bar is calibrated according to the chosen

```python
# Change contrast of image
#   scale [0, 127] starts at 27   (r1=s1 = 100; r2=s2 = 154)
#   scale [0, 127] starts at 63   (r1=s1 =  64; r2=s2 = 190)
def contrast_effect(self, k):
    r1, r2 = 64, 190  # fixed input image values
    s1, s2 = 127-k, 127+k

    img_new = self.cv_img.astype(np.uint16)

    slope1 = s1 / r1
    mask1 = (img_new >= 0) & (img_new <= r1)
    img_new[mask1] = slope1*img_new[mask1]

    slope2 = (s2-s1) / (r2-r1)
    mask2 = (img_new > r1) & (img_new <= r2)
    img_new[mask2] = slope2*(img_new[mask2]-r1) + s1

    slope3 = (255-s2) / (255-r2)
    mask3 = (img_new > r2) & (img_new < 256)
    img_new[mask3] = slope3*(img_new[mask3]-r2) + s2

    return img_new.astype(np.uint8)

# Callback for the "Contrast" Scale
def contrast_image(self, k):
    k = self.scl_contrast.get()
    self.NEWcv_img = self.contrast_effect(k)
    self.photo = PIL.ImageTk.PhotoImage(image=PIL.Image.fromarray(self.NEWcv_img))
    self.canvas1.create_image(MAXDIM//2, MAXDIM//2, image=self.photo, anchor=tk.CENTER)
```

Figure 15: Contrast filter code

Figure 16: Contrast filter, k = +20

points. r_1 and r_2 determine the x-axis location of the chosen points of diving up the three parts of the curve. The first line goes from the origin to (r_1, s_1), the second line goes from

(r_1, s_1) to (r_2, s_2), and the third line goes from (r_2, s_2) to $(255, 255)$. See Figure 15 for details on implementation. The user-given k value determines how steep the slope of the middle line is. Initial k value is set to make the slope $= 1$. The greater the k-value, the greater the slope, and thus greater contrast and vice versa.

3.7 Sharpen

```
def sharpen_image(self, k):
    k = self.scl_sharpen.get()
    hsv = cv2.cvtColor(self.cv_img, cv2.COLOR_BGR2HSV)
    img_v = hsv[:,:,2]  # value channel

    mask = np.array([[-k,    -k, -k],
                     [-k, 1+8*k, -k],
                     [-k,    -k, -k]]) # og + k*mask
    hsv[:,:,2] = cv2.filter2D(img_v, 64, mask)
    self.NEWcv_img = cv2.cvtColor(hsv, cv2.COLOR_HSV2BGR)
    self.photo = PIL.ImageTk.PhotoImage(image=PIL.Image.fromarray(self.NEWcv_img))
    self.canvas1.create_image(MAXDIM//2, MAXDIM//2, image=self.photo, anchor=tk.CENTER)
```

Figure 17: Sharpen filter code

Figure 18: Sharpen filter, k = 0.750

Sharpening algorithm consists of a 2-D convolution with a high boost kernel [2]. A high boost kernel is the image itself (1 in the middle of a matrix and all else) plus a high pass filter multiplied by some constant [2], which is determined by user input from the corre-

sponding scale bar on GUI.

$$W_{hb} = W_{original} + W_{HPF} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} + k \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix} = \begin{bmatrix} -k & -k & -k \\ -k & 1+8*k & -k \\ -k & -k & -k \end{bmatrix}$$

This effect emphasizes the edges of an image and creates a sharpening effect. See Figure 18 for an example of the sharpening effect.

3.8 Pencil Sketch

```
def dodge(self, top, bottom):
    return cv2.divide(top, 255-bottom, scale=256)

# k = [4, 30] --> [3, 29]
# k == 2: "reset"
def sketch_image(self, k):
    # Invert it original image
    # Blur the inverted image
    k = self.scl_sketch.get()-1  # -1 to make k odd
    if k == 1:
        self.photo = PIL.ImageTk.PhotoImage(image = PIL.Image.fromarray(self.cv_img))
        self.canvas1.create_image(MAXDIM//2, MAXDIM//2, image=self.photo, anchor=tk.CENTER)
    else:
        self.gray_img = cv2.cvtColor(self.cv_img, cv2.COLOR_BGR2GRAY)
        self.inv_img = 255-self.gray_img
        self.blur_img = cv2.GaussianBlur(self.inv_img, (k, k), sigmaX=k)
        # Dodge blend the blurred and grayscale image.
        self.NEWcv_img = self.dodge(self.gray_img, self.blur_img)
        self.photo = PIL.ImageTk.PhotoImage(image=PIL.Image.fromarray(self.NEWcv_img))
        self.canvas1.create_image(MAXDIM//2, MAXDIM//2, image=self.photo, anchor=tk.CENTER)
```

Figure 19: Pencil sketch filter code

Pencil sketch effect was initially inspired by a TA in a different course. There are published work [3] online but the difficulty of the implementation was too high. A simpler version of a "sketch" effect is used for this program. A pencil sketch mostly requires dark spots where there are edges and other darker areas of the image. The implemented method uses color dodge blending mode on an inverted image in order to bring out the edges. Since the effect is *pencil* sketch, all images for this filter is first converted to gray scale. Then an inverted and blurred copy is saved. Color dodge occurs with the gray scale version of the original image as the top layer and the blurred inverted image as the bottom layer. Color dodging is just dividing the top layer by the inverted bottom layer [4]. The

brighter the bottom layer, the more its color affects the top layer, thus by inverting the bottom layer (and making the dark spots and the edges whiter), the resulting image will have dark spots at the dark spots and the edges. The k value changed the kernel size of blurring, which was used to emphasize the stronger edges.

Figure 20: Pencil sketch filter, k = 20

3.9 Vignette

```
# k = [0, 100-1]
def vignette_image(self, k):
    k = self.scl_vignette.get()
    row, col, ignore = self.cv_img.shape
    # mask (oval)
    black = np.zeros_like(self.cv_img)
    center_coordinates = (col//2, row//2)
    radius = (max(row,col)//2) + (100-k)
    color = (255, 255, 255)  # white
    thickness = -1
    mask = cv2.circle(black, center_coordinates, radius, color, thickness)
    mask = cv2.blur(mask, (200, 200)).astype(float)/255
    img = np.copy(self.cv_img).astype(float)/255
    self.NEWcv_img = (self.cv_img * mask).astype(np.uint8)
    self.photo = PIL.ImageTk.PhotoImage(image=PIL.Image.fromarray(self.NEWcv_img))
    self.canvas1.create_image(MAXDIM//2, MAXDIM//2, image=self.photo, anchor=tk.CENTER)
```

Figure 21: Vignette filter code

Vignette effect darkens the corners of an image to make it look more dramatic. The multiply blend mode is used to achieve this effect. A mask, white circle on a black background)

is multiplied with the image. The pixel values are converted to $[0, 1]$ (instead of $[0, 255]$) and then multiplied. This is why the white pixels of the mask do not affect the original, but the darker pixels will decrease the pixel value of the original pixel. The parameter k is used to determine the radius of the circle in the mask. The greater the k, the smaller the circle and therefore applying a heavier darkening effect.

$$f(a, b) = ab \quad [4]$$

Figure 22: Vignette filter, k = 45

4 Personal Takeaways

This project was very fun for me. Playing with `tkinter` was interesting. I think it was a good experience to explore GUI design and implementation in python. It was cool to have the flexibility to implement what I wanted to do. I started out with the "easy" effects like blurring of different kinds then moved onto cooler effects.

One of the things I would like to improve on is the way the effects interact with each other. Currently none of the filters can build on top of another and it would be cool to apply

more than one effect on an image at the same time. There are couple filters I would also like to improve: pencil sketch and vignette. For the pencil sketch effect, it would be cool if I could implement what was written in the paper that I read. This paper's algorithm can detect the direction of shade using the gradient and also shade in different parts of the image. It was really long and difficult to understand at first few glances so I stopped exploring that paper. For the vignette effect, as the radius decreases, the darkening appears to be four dots on the corner of the image. Instead, I want the effect to darken the image on all corners *and* sides so the dramatic focus is on the middle of the image.

In the future, it would very cool to explore edge detecting algorithms to do object detection and carrying to video analysis. I was joking about it with a friend to implement a program that can detect "bad dancing". This is nearly impossible with many limitations but maybe one day my memes will come to life and be useful in the world.

5 Referred Material

1. http://effbot.org/tkinterbook/tkinter-index.htm#class-reference

High Boost

2. http://fourier.eng.hmc.edu/e161/lectures/gradient/node2.html

Pencil Sketch

3. https://pdfs.semanticscholar.org/6317/a749fe29467cdb36b0b3cef492940ee6beb9.pdf
4. https://en.wikipedia.org/wiki/Blend_modes
5. http://www.askaswiss.com/2016/01/how-to-create-pencil-sketch-opencv-python.html
6. https://www.freecodecamp.org/news/sketchify-turn-any-image-into-a-pencil-sketch-with-10-lines-of-code-cf67fa4f68ce/