ONTARIO TECH UNIVERSITY
COMPUTER SCIENCE

Group 18 December 8, 2023

Tech Report
Computational Photography FinalProject

by

Abdulwasay Mansoor Saenthuran Vignarajah

Abstract: Our Project was based on implementing Analog Film Effect and refining the GUI of our labs

Keywords: Analog Film Effects, Grain, Scratches, Dust, Double Exposure

1 Introduction

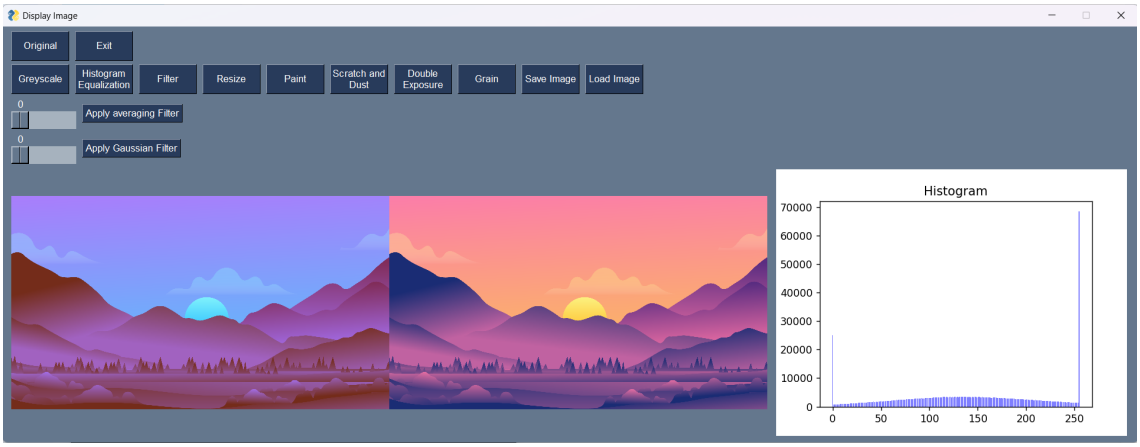


Photo editing has become a vital part of current society, from consumers using it for their personal photos to professionals using it in the cinematic world. This report was made to introduce our interactive photo editing software that was developed using Python, PySimpleGUI, Open CV, and Numpy libraries.

The project was created to produce a user-friendly program that provides the most essential features that someone would use to edit photos. OpenCV and Numpy allow us to load and apply edits to the photos, while PySimpleGUI allows us to create a clean-looking GUI.

We have been working on a photo editing software throughout the course and for the final project we based it on our labs and refined the GUI, some changes to how it works, and implemented 3 different film effects as per our proposal. This technical report was made to highlight the film effects that we made, as those are the main points of our project, rather than focus on the program as a whole.

After using PySimpleGUI to create an intuitive GUI, we started working on our film effects using OpenCV and Numpy.

The first film effect is Emulsion Scratches and Dust, analog films usually have imperfections and by attempting to mimic the imperfections, we can have an image that looks similar to analog films. Implementing this effect required us to overlay texture images onto the image we were editing. We also implemented the ability to change the opacity and rotation of the overlay image to change how the resulting image would look.

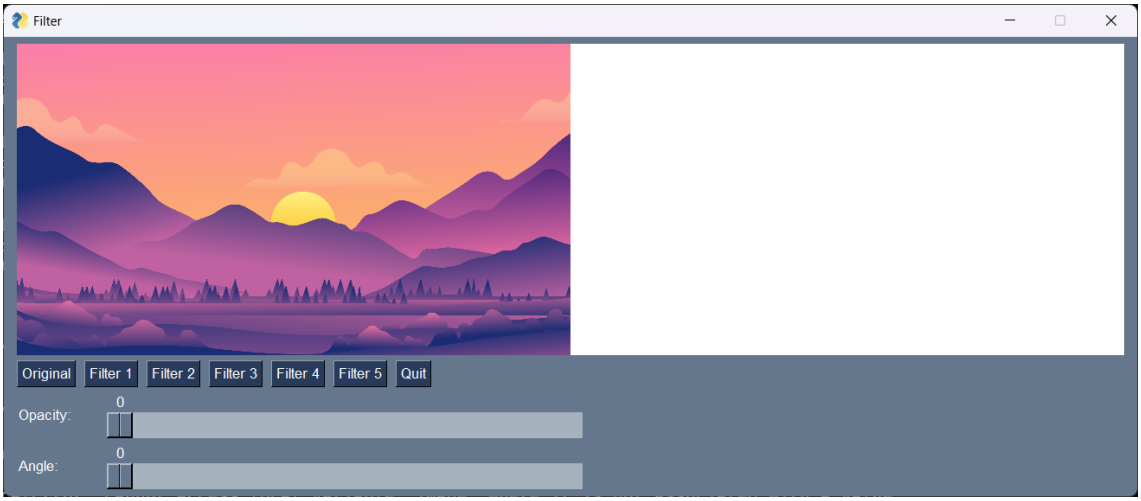
The second film effect is the Grain Effect. The grain effect refers to the random varia-

tions in brightness and colour pixel due to the presence of silver halide crystals in the film emulsion. Artificial Grain is used in photography to give the effect of a film grain. To achieve this, we were required to create noise and add it to the photo.

The third film effect is Double Exposure. The double exposure effect refers to combining two different images into a single frame, we've modified it so we can take any person and cut them out of it to create a mask and apply that mask on top of the background to create a double exposure effect.

2 Scratches and Dust

2.1 GUI



The image above shows how the GUI initially looks when the dust and scratches option is selected. On the left, we have a display of the original picture and on the right which is currently blank is where the resulting image will be shown, this will be showcased in the results subsection. Under the image we have several buttons, one is to reset the picture on the right to the original image. The filter buttons are for changing which overlay image is being blended allowing different resulting images and allowing for multiple overlay images to be blended with the image. Then quit button quits the sub-window and goes back to the main menu. Then we have 2 sliders; the opacity slider that sets the opacity of the overlay image and the angle slider can be tweaked to rotate the overlay image. The sliders together with the different blenders allow us to get multiple different-looking images.

2.2 GUI code

```
def resetLayout4(width, height):
    layout4 = [
        [sg.Graph(
            canvas_size=(width*2, height),
            graph_bottom_left=(0, 0),
            graph_top_right=(width*2, height),
            key='-IMAGE-',
            background_color='white',
            change_submits=True,
            drag_submits=True
        )],
        [sg.Button('Original', key='original'),
         sg.Button('Filter 1', key='Filter1'),
         sg.Button('Filter 2', key='Filter2'),
         sg.Button('Filter 3', key='Filter3'),
         sg.Button('Filter 4', key='Filter4'),
         sg.Button('Filter 5', key='Filter5'),
         sg.Button('Quit')
         ],
        [sg.Text('Opacity:', size=(10, 1)), sg.Slider(range=(0,100), default_value=0, orientation='h',size=(50,30), key='-Opacity-')],
        [sg.Text('Angle:', size=(10, 1)), sg.Slider(range=(0,360), default_value=0, orientation='h',size=(50,30), key='-Angle-')]
    ]
    return layout4
```

The following image shows the code to produce the layout of the GUI for this effect. We first have sg.Graph which creates the canvas used to display the images. Next, we have all the buttons; the button to reset the image, all the filter buttons, and the Quit button. Lastly, we have the 2 sliders, opacity ranges from 0 to 100 and the angle ranges from 0 to 360.

```
if event == '-scratch-':
    window.close()
    initialImage = np_image.copy()
    initialImageData = np_im_to_data(initialImage)
    FilterPopUp = sg.Window('Dust&Scratches', layout4, finalize=True)
    FilterPopUp['-IMAGE-'].draw_image(data=initialImageData, location=(0, height))
```

Code awaits for the scratch event which is called when the Dust And Scratches button is clicked. When the event is activated the main window is closed and then a new window is opened using the layout we previously created. We also created a copy of the image so we can reset the image using the original button. Lastly, we display the new image, we created from copying, on the left which shows the image before any overlay images are blended.

```
paths =['dust&scratches/overlay1.png',
        'dust&scratches/overlay2.png',
        'dust&scratches/overlay3.png',
        'dust&scratches/overlay4.png',
        'dust&scratches/overlay5.png']
```

We created an array that stores the directories of the overlay images, this allows us to add or change overlay images easily.

```
while True:
    event, values = FilterPopUp.read()
    opacity = int(values['-Opacity-'])
    angle = int(values['-Angle-'])
    if event == 'original':
        np_image = initialImage.copy()
        imageData = np_im_to_data(np_image)
        FilterPopUp['-IMAGE-'].draw_image(data=imageData, location=(width, height))
    if event == 'Quit':
        FilterPopUp.close()
        display_image(width, height, np_image, beforeImage, originalImage)
    filter_events = ['Filter1', 'Filter2', 'Filter3', 'Filter4', 'Filter5']
    for i, filter_event in enumerate(filter_events):
        if event == filter_event:
            np_image = add_scratches_and_dust(np_image, paths[i], opacity/100, angle)
            imageData = np_im_to_data(np_image)
            FilterPopUp['-IMAGE-'].draw_image(data=imageData, location=(width, height))
```

In the last part of this event, we just read the change in sliders and stored the opacity and the rotation angle. then we just await for a button press. If the original button is clicked then the image on the right is reset by setting the image to the copy we created in the beginning. If the quit button is clicked then goes back to the previous window by calling the main function in the whole program which is the display function. And lastly, we have the filter button in a list and we just check to see if any of them have been pressed, if pressed we pass the corresponding path to the function.

2.3 Function

```
def add_scratches_and_dust(original_image, overlay_image_path, opacity, angle):
```

Firstly we start with the function for overlaying the texture, the arguments taken are the original image, the path to the image, opacity, and rotation angle. This function will transform the overlay image using the rotation angle, and then blend it using the alpha values of the overlay image and the opacity.

```
overlay_image = cv2.imread(overlay_image_path, cv2.IMREAD_UNCHANGED)
overlay_image = cv2.resize(overlay_image, (original_image.shape[1], original_image.shape[0]))
```

The following code loads in the overlay image and resizes it to match the image we are editing.

```
angle_rad = np.radians(angle)
cos_theta = np.cos(angle_rad)
sin_theta = np.sin(angle_rad)
center_x, center_y = overlay_image.shape[1] // 2, overlay_image.shape[0] // 2
rotation_matrix = np.array([
    [cos_theta, -sin_theta, (1 - cos_theta) * center_x + sin_theta * center_y],
    [sin_theta, cos_theta, -sin_theta * center_x + (1 - cos_theta) * center_y]
])
```

The following code first converts the rotation angle from degrees to radians. Then save the values of cos and sin angles. Next, we get the center of the x-axis and y-axis of the image. Next, let's look at the derivation for the rotation matrix we have in the code.

A rotation matrix is denoted by;

$$\begin{bmatrix} \cos(\theta) & -\sin(\theta) & t_x \\ \sin(\theta) & \cos(\theta) & t_y \end{bmatrix}$$

Therefore we have the following equations;

$$x' = \cos(\theta) \cdot x - \sin(\theta) \cdot y + t_x$$
$$y' = \sin(\theta) \cdot x + \cos(\theta) \cdot y + t_y$$

As we are doing a rotation at the center for our case, we change those equations to;

$$\text{center}_x = \cos(\theta) \cdot \text{center}_x - \sin(\theta) \cdot \text{center}_y + t_x$$
$$\text{center}_y = \sin(\theta) \cdot \text{center}_x + \cos(\theta) \cdot \text{center}_y + t_y$$

Next we solve for t_x;

$$\begin{aligned} \text{center}_x &= \cos(\theta) \cdot \text{center}_x - \sin(\theta) \cdot \text{center}_y + t_x \\ t_x &= \text{center}_x - \cos(\theta) \cdot \text{center}_x + \sin(\theta) \cdot \text{center}_y \\ &= (1 - \cos(\theta)) \cdot \text{center}_x + \sin(\theta) \cdot \text{center}_y \end{aligned}$$

Therefore;

$$t_x = (1 - \cos(\theta)) \cdot \text{center}_x + \sin(\theta) \cdot \text{center}_y$$

Following the same steps we can get;

$$t_y = -\sin(\theta) \cdot \text{center}_x + (1 - \cos(\theta)) \cdot \text{center}_y$$

Using those derivations we can form a 2x3 matrix and use it in our code.

$$\text{Rotation Matrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & (1 - \cos(\theta)) \cdot \text{center\_x} + \sin(\theta) \cdot \text{center\_y} \\ \sin(\theta) & \cos(\theta) & -\sin(\theta) \cdot \text{center\_x} + (1 - \cos(\theta)) \cdot \text{center\_y} \end{bmatrix}$$

```
overlay_image = cv2.warpAffine(overlay_image, rotation_matrix, (overlay_image.shape[1], overlay_image.shape[0]))
```

WarpAffine is a function from the cv2 library that allows easy transformations on images, we are using the function for rotation if given by the user. The first argument would be the image that we are rotating, in this case, the overlay image. The next argument would be a 2x3 transformation matrix, this matrix determines translation, rotation, scaling, and shearing. In this case, the second argument will be the rotation matrix we previously calculated. The last argument is supposed to be a tuple specifying the size of the output image (width, height), in this case, we will just set it to be the initial size of the overlay images.

```
alpha_channel = overlay_image[:, :, 3] / 255.0
alpha_mask = cv2.merge([alpha_channel, alpha_channel, alpha_channel])
```

In the first line, the alpha channel is first extracted, the alpha channel represents the transparency of each pixel. Then is divided by 255 to normalize the value to 0 or 1 as the alpha channel value can range between 0 to 255, 0 meaning transparent and 255 meaning opaque. Then in the second line, a 3-channel alpha mask is created with the same value for the 3 channels (R,G,B).

```
original_image = original_image.astype(float)
overlay_image = overlay_image[:, :, :3].astype(float)
```

In this block of code, we are turning pixel values that are usually represented as integers into to floats avoid any issues that may occur during the blending process. The only difference during the conversion for the overlay image is that we remove the alpha channel as we want to blend the RGB and alpha channel separately.

```
blended_image = original_image * (1 - opacity) + overlay_image[:, :, :3] * opacity
result = blended_image * alpha_mask + original_image * (1 - alpha_mask)
```

These 2 lines are based on the equation used for blending:

$$Blended_p = (1 - A_p) \cdot O_p + A_p((1 - \alpha) \cdot O_p + \alpha \cdot B_p)$$

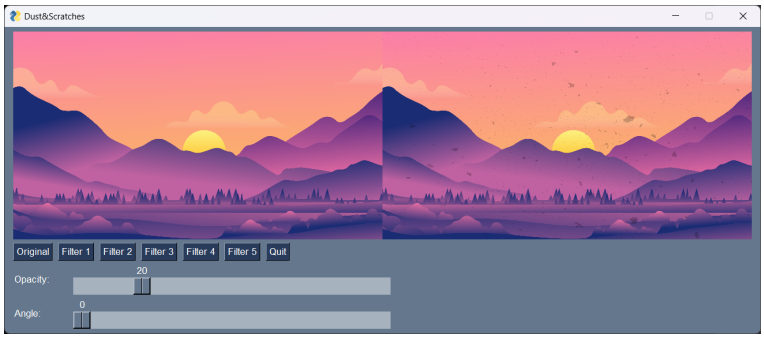
O represents the original picture
A_p represents the pixel value of the alpha mask
O_p represents the pixel value of the original image
B_p represents the pixel value of the overlay image
α represents the opacity
The first line combines the original image and the overlay image depending on their opacities. Then in the second line the contribution of the original image, scaled by its transparency, and the contribution of the RGB channels of the overlay image, scaled by its opacity, is combined.

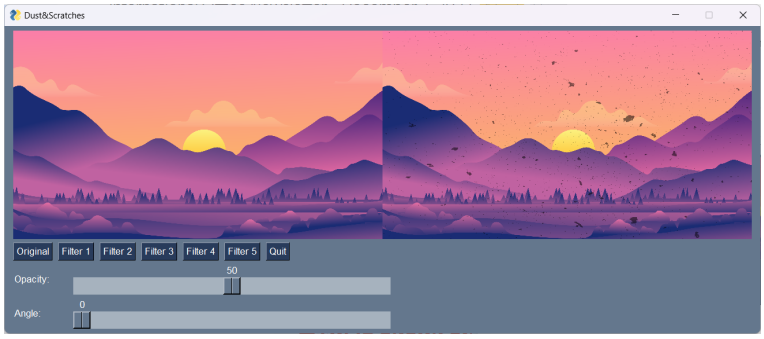
```
return result.astype(np.uint8)
```

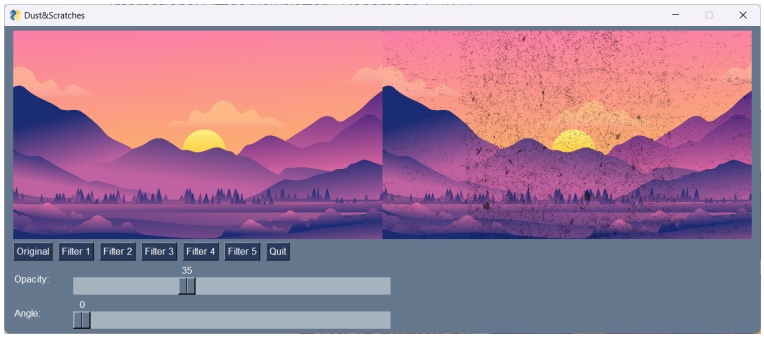
The last line of the function converts the float values back to integers and returns those values.

2.4 Results

Here are some of the example results you can achieve using this effect

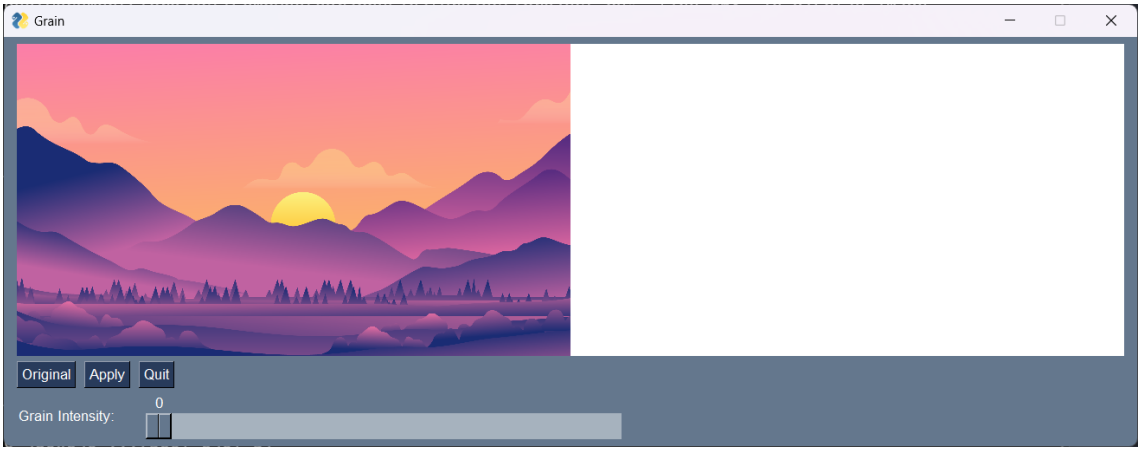






3 Grain

3.1 GUI



The image above shows how the GUI initially looks when the grain option is selected. On the left, we have a display of the original picture and on the right which is currently blank is where the resulting image will be shown, this will be showcased in the results subsection. Under the image, we have three buttons, the original button resets the image back to the non-grain version. Apply will pass the intensity to the grain function and apply the filter. The quit button quits the sub-window and goes back to the main menu. And lastly, we have the slider for the grain intensity.

3.2 GUI Code

```
def resetLayout5(width, height):
    layout5 = [
        [sg.Graph(
            canvas_size=(width*2, height),
            graph_bottom_left=(0, 0),
            graph_top_right=(width*2, height),
            key='-IMAGE-',
            background_color='white',
            change_submits=True,
            drag_submits=True
        )],
        [
         sg.Button('Original'),
         sg.Button('Apply'),
         sg.Button('Quit')
        ],
        [sg.Text('Grain Intensity:', size=(10, 1)), sg.Slider(range=(0,100), default_value=0, orientation='h',size=(50,30), key='-GIntensity-')],
    ]
    return layout5
```

The image above shows the code to produce the layout of the GUI for this effect. We first have sg.Graph which creates the canvas used to display the images. Next, we have the 3 buttons; the button to reset the image, the apply button, and the quit button. Then we create a slider for the intensity that ranges from 0 to 100.

```
if event == '-noise-':
    initialImage = np_image.copy()
    initialImageData = np_im_to_data(initialImage)
    window.close()
    FilterPopUp = sg.Window('Grain', layout5, finalize=True)
    FilterPopUp['-IMAGE-'].draw_image(data=initialImageData, location=(0, height))
```

Code awaits for the noise event which occurs when the grain button is pressed. When the event is activated the main window is shut down and a copy of the image is made that can be used to reset the image. Then using the layout we created earlier we open a new window. Lastly, we display the image we copied on the left.

```
while True:
    event, values = FilterPopUp.read()
    gIntensity = int(values['-GIntensity-'])
    if event == 'Apply':
        np_image = initialImage.copy()
        np_image = grain(np_image, gIntensity)
        imageData = np_im_to_data(np_image)
        FilterPopUp['-IMAGE-'].draw_image(data=imageData, location=(width, height))
    if event == 'Original':
        np_image = initialImage.copy()
        imageData = np_im_to_data(np_image)
        FilterPopUp['-IMAGE-'].draw_image(data=imageData, location=(width, height))
    if event == 'Quit':
        FilterPopUp.close()
        display_image(width, height, np_image, beforeImage, originalImage)
```

This part of the code reads the slider value for grain intensity and once the apply button is clicked we create a copy of the initial, we create the copy to ensure that the grain effect is not being applied on top of itself, then send the copy and the stored grain intensity onto the grain function. The other events check for other button presses, if the original button is pressed then just set the image to the initial image copy we made at the start, and if the quit button is pressed, we just call the display function, which is the main function of the program

3.3 Function

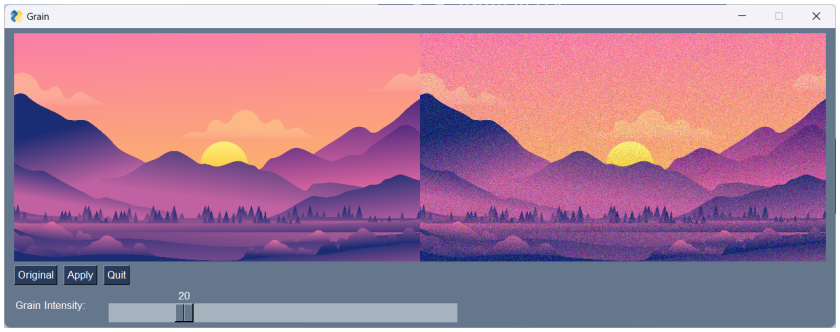
We first take the height and width of the image and store it as h and w. we then create an array of shape (h,w,3) that is the height, width and the last value used is 3 corresponding to RGB. That shape is then filled with random values from a standard normal distribution, which then all values are multiplied by the intensity and stored in

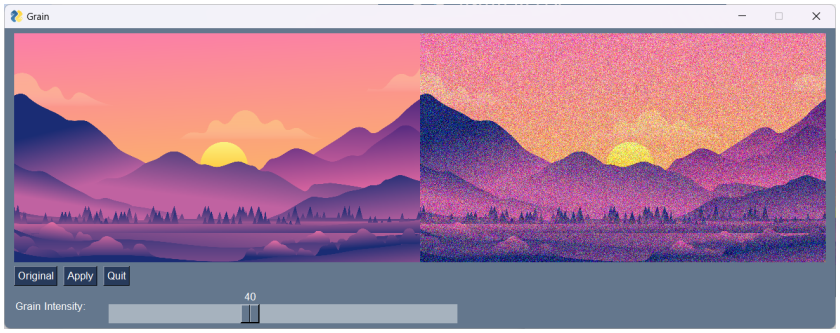
```
if event == '-noise-':
    initialImage = np_image.copy()
    initialImageData = np_im_to_data(initialImage)
    window.close()
    FilterPopUp = sg.Window('Grain', layout5, finalize=True)
    FilterPopUp['-IMAGE-'].draw_image(data=initialImageData, location=(0, height))
```

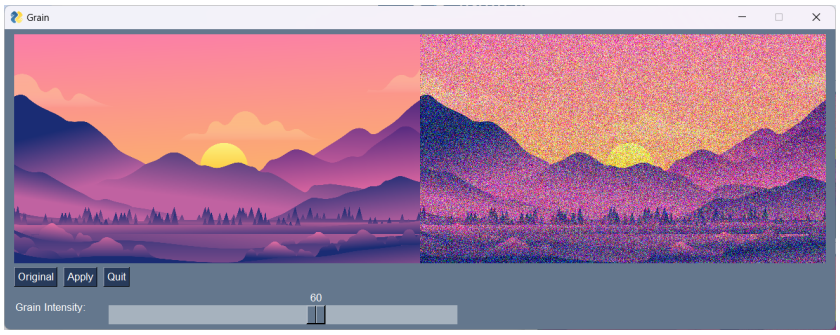
the noise variable. Then np.clip we can add the image pixel values with the noise values and ensure that resulting pixel values stay within the range of 0 to 255. It is then converted to an 8-bit integer type then returned.

3.4 Results

Here are some of the example results you can achieve using this effect







4 Double Exposure

4.1 GUI

The image above shows how the GUI initially looks like when we first hit the Double Exposure option. We can see the person image on the left and the background on the right, these are placeholder images to give the user a reference idea. There are 3 buttons at the top Load Person to change the person and load background to change the background image. We can use any we like and then the last button is where we apply the double exposure effect. Please take note that if you would like to use a picture of yourself make sure you are in front of a solid background and no shadows appear the less in your background the easier it is for the code to work.

4.2 GUI Code

```
if event == '-dexposure-':

    window.close()
    # Define the layout
    #sg.Image('path/to/placeholder.png', key='-PERSON_IMAGE-', size=(200, 200))
    #sg.Image('path/to/placeholder.png', key='-PERSON_IMAGE-', size=(200, 200))
    # Attempt to load the placeholder images
    placeholder_size = (200, 200)  # Width, Height
    # Use the function for both person and background images
    person = cv2.imread('dexposure/place_holder_person.jpg', cv2.IMREAD_COLOR)
    background = cv2.imread('dexposure/place_holder_background.jpg', cv2.IMREAD_COLOR)
    person = cv2.cvtColor(person, cv2.COLOR_BGR2RGB)
    background = cv2.cvtColor(background, cv2.COLOR_BGR2RGB)
    resized_image_person = resize_image_aspect_ratio(person,height=250,width=250)  # Resize by width, maintaining aspect ratio
    resized_image_background = resize_image_aspect_ratio(background,height=250,width=250)  # Resize by width, maintaining aspect ratio

    person_image_data = np_im_to_data(resized_image_person)
    background_image_data = np_im_to_data(resized_image_background)




        # Update the layout with the image data
    layout = [
            [sg.Button('Load Person'), sg.Button('Load Background'), sg.Button('Double Exposure')],
            [sg.Image(data=person_image_data, key='-PERSON_IMAGE-'),
             sg.Image(data=background_image_data, key='-BACKGROUND_IMAGE-')]
            ]


    # Create the window
    #window['-PERSON_IMAGE-'].update(data=person_image_data)
    #window['-BACKGROUND_IMAGE-'].update(data=background_image_data)

    window = sg.Window('Double Exposure App', layout,resizable=True)
```

In this code block, we read the 2 placeholder images for the person and background and then make sure they are converted to RGB and we resize the images so we can have them fit onto the screen. We then have the layout with the buttons and below the buttons are the placeholder images

```
# Event loop9
while True:
    event, values = window.read()

    if event == sg.WIN_CLOSED:
        break
    elif event == 'Load Person':
        # Open a file dialog to choose an image
        file_path = sg.popup_get_file('Select Person Image', file_types=(("Image Files", "*.png;*.jpg;*.jpeg;*.bmp"),))

        # Check if a file was selected
        if file_path:
            # Load the image
            person = cv2.imread(file_path)

            # Check if the image was loaded successfully
            if person is not None:
                # Convert the image to a format suitable for displaying in PySimpleGUI
                resized_image_person = resize_image_aspect_ratio(person,height=250,width=250)  # Resize by width, maintaining aspect ratio
                person_image_rgb = cv2.cvtColor(resized_image_person, cv2.COLOR_BGR2RGB)
                person_image_data = np_im_to_data(person_image_rgb)

                # Update the GUI element to display the loaded image
                window['-PERSON_IMAGE-'].update(data=person_image_data)
            else:
                sg.popup_error('Unable to load image.', title='Error')
```

In this code block, we have a while loop to constantly check on the events, if a user hits the load person button it will open up a separate box where they can put in a file path or just browse and select the person image they like, once one has been selected we resize, convert to RGB and then apply the np im to data function on it so we can display it

```
elif event == 'Load Background':
    # Open a file dialog to choose an image
    file_path = sg.popup_get_file('Select Background Image', file_types=(("Image Files", "*.png;*.jpg;*.jpeg;*.bmp"),))

    # Check if a file was selected
    if file_path:
        # Load the image
        background = cv2.imread(file_path)

        # Check if the image was loaded successfully
        if background is not None:
            # Convert the image to a format suitable for displaying in PySimpleGUI
            resized_image_background = resize_image_aspect_ratio(background,height=250,width=250)  # Resize by width, maintaining aspect ratio
            background_image_rgb = cv2.cvtColor(resized_image_background, cv2.COLOR_BGR2RGB)
            background_image_data = np_im_to_data(background_image_rgb)

            # Update the GUI element to display the loaded image
            window['-BACKGROUND_IMAGE-'].update(data=background_image_data)
        else:
            sg.popup_error('Unable to load image.', title='Error')
```

This code block is similar to the one above only difference is this is for loading up a background image

```
elif event == 'Double Exposure':

    # Implement the double exposure effect
    # Step 1: Perform GrabCut
    cut_image, cut_mask = perform_grabcut(person, 80, 150)

    # Step 2: Resize the Mask
    target_size = (background.shape[1], background.shape[0])
    resized_mask = resize_with_padding(cut_mask * 255, target_size[0], target_size[1])


    # Step 3: Invert the mask
    print("Shape of double_exposure:", background.shape)
    print("Shape of double_exposure:", resized_mask.shape)

    double_exposure= invert(background,resized_mask)
    double_exposure = resize_image_aspect_ratio(double_exposure,height=250,width=250)  # Resize by width, maintaining aspect ratio
    print("Shape of double_exposure:", double_exposure.shape)
    print("hi")
    double_exposure = np_im_to_data(double_exposure)

    #double_exposure_image_data = np_im_to_data(double_exposure)

    layout = [
            [sg.Button('Load Person'), sg.Button('Load Background'), sg.Button('Double Exposure')],
            [sg.Image(data=person_image_data, key='-PERSON_IMAGE-'),
             sg.Image(data=background_image_data, key='-BACKGROUND_IMAGE-'),
             sg.Image(data=double_exposure, key='-DOUBLE_EXPOSURE_IMAGE-')],


            ]
    window.close()

    # Update the layout with the new image
    window = sg.Window('Double Exposure App', layout, finalize=True)
```

In this code block when the ”Double Exposure” button is hit we apply perform grabcut on the person where we have the parameters person, threshold1, and threshold 2. Threshold 1 is the lower threshold and threshold 2 is the upper. The values 80 and 150 are chosen based on experimentation we found to create the best results. After that, we get the size of the background shape and then use resized with padding. Essentially what this code does is it takes the cut mask we generated around the person and then it will retain its aspect ratio and then change its dimensions to fit better with the background so we can combine it later. The print statements are meant for debugging. now we apply the invert function. The point of the invert function is to invert the mask and then apply it over the background this will be further explained in the function section. After that, we reapply the resize image aspect ratio and then we update the layout to contain the double exposure.

4.3 Function

```python
def perform_grabcut(img, threshold1, threshold2):
    # Load the image

    # Apply Canny edge detection
    edges = cv2.Canny(img, threshold1, threshold2)

    # Define a kernel for morphological operations
    kernel = np.ones((5,5), np.uint8)

    # Apply morphological transformations
    closing = cv2.morphologyEx(edges, cv2.MORPH_CLOSE, kernel, iterations=3)
    erosion = cv2.morphologyEx(closing, cv2.MORPH_ERODE, kernel, iterations=1)

    # Initialize mask for GrabCut
    mask = np.zeros(img.shape[:2], np.uint8)
    mask[:] = 2  # Possible background areas
    mask[erosion == 255] = 1  # Probable foreground areas

    # Background and foreground models for GrabCut
    bgdmodel = np.zeros((1, 65), np.float64)
    fgdmodel = np.zeros((1, 65), np.float64)

    # Apply GrabCut with the mask
    cv2.grabCut(img, mask, None, bgdmodel, fgdmodel, 5, cv2.GC_INIT_WITH_MASK)

    # Final mask where 0 and 2 are background, 1 and 3 are foreground
    mask2 = np.where((mask == 2)|(mask == 0), 0, 1).astype('uint8')
    # Apply the mask to the image to get the result
    result_img = img * mask2[:, :, np.newaxis]
    return result_img, mask2
```

So we first use canny edge detection to get the edges and we define a 5x5 kernel for morphological transformations. We then apply morphology transformation for closing and then we redo it again but with erosion instead. This is meant to dilate and erode the image it helps with closing small holes or objects and erosion helps with removing noise. We then create a mask. Mask = 2 indicates areas of the background and erosion == 255 = 1 is for foreground areas. after that bdmodel is the background and the fdmodel is the foreground. we then apply a grabcut with the previous results. The final mask is then created where we set the original mask values 0 and 2 to 0 and foreground values of 1 and 3 to 1. After we do img*mask2 this separates the mask so we only keep the foreground and remove the background that's how we can cut the person out of the original image.

```python
def invert(background,mask):
    updated_mask = cv2.cvtColor(mask, cv2.COLOR_BGR2GRAY)  # Read the mask in grayscale


    # Create an alpha channel based on the updated mask
    alpha_channel = np.where(updated_mask == 0, 255, 0).astype(np.uint8)  # Opaque where mask is black

    # Merge the alpha channel with the background image
    rgba_background = cv2.merge((*cv2.split(background)[:3], alpha_channel))

    # Extract the alpha channel from the rgba image
    alpha_channel = rgba_background[:, :, 3]

    # Invert the alpha channel to flip transparency and opacity
    inverted_alpha = 255 - alpha_channel

    # Replace the alpha channel in the image with the inverted alpha
    rgba_background[:, :, 3] = inverted_alpha

    # Create a background image of the same size -- fill it with white color for example
    background = np.ones_like(rgba_background) * 255

    # Extract the alpha channel from the foreground image
    alpha = rgba_background[:, :, 3] / 255.0

    # Blend the foreground with the background based on the alpha channel
    for c in range(0, 3):
        background[:, :, c] = alpha * rgba_background[:, :, c] + (1 - alpha) * background[:, :, c]
    return background
```

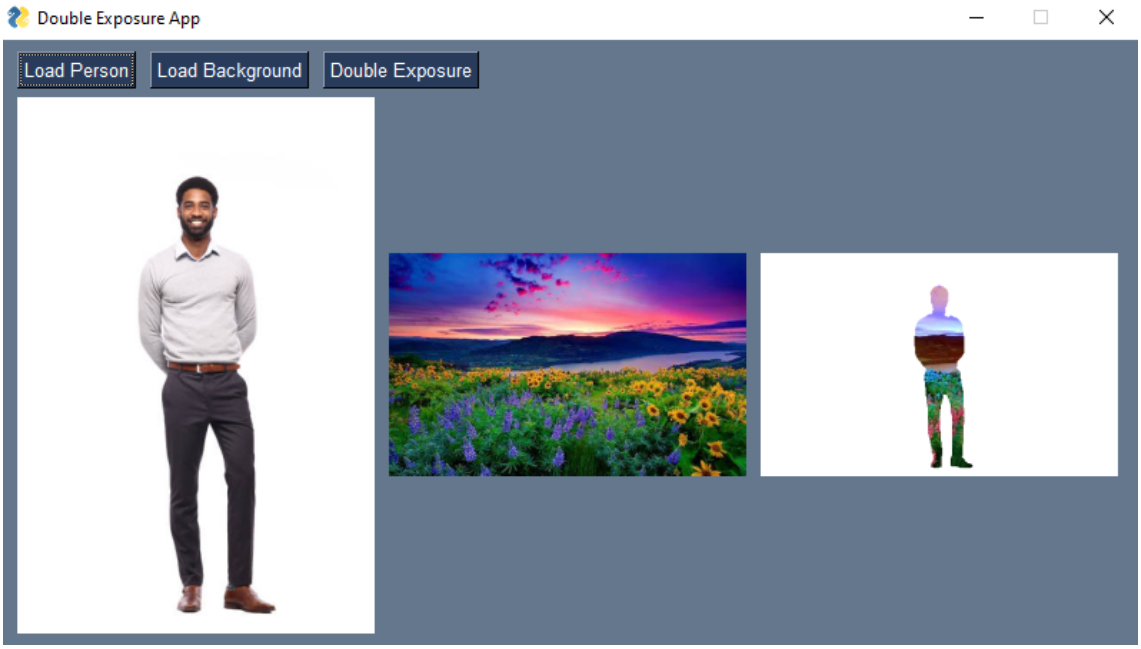
We convert the mask to grayscale. After we create an alpha channel based on the updated mask, we merge the alpha channel with the background image. We then invert the alpha channel and then replace the alpha channel in the image and we create a white background canvas. Then we blend the foreground with the background based on the alpha channel. So we get the original height and width of the image. We

```python
def resize_with_padding(img, background_width, background_height):
    # Get the dimensions of the original image
    original_height, original_width = img.shape[:2]

    # Calculate the aspect ratio of the original image
    aspect_ratio = original_width / original_height

    # Calculate the new dimensions while maintaining the aspect ratio
    if aspect_ratio >= 1:
        new_width = background_width
        new_height = int(new_width / aspect_ratio)
    else:
        new_height = background_height
        new_width = int(new_height * aspect_ratio)

    # Resize the image using the calculated dimensions
    resized_img = cv2.resize(img, (new_width, new_height), interpolation=cv2.INTER_LINEAR)

    # Create a blank canvas with the target size and 3 channels (RGB)
    img_with_padding = np.zeros((background_height, background_width, 3), dtype=np.uint8)

    # Calculate the position to paste the resized image
    x_offset = (background_width - new_width) // 2
    y_offset = (background_height - new_height) // 2

    # Paste the resized image onto the canvas
    img_with_padding[y_offset:y_offset + new_height, x_offset:x_offset + new_width] = cv2.cvtColor(resized_img, cv2.COLOR_GRAY2BGR)

    return img_with_padding
```

then calculate the aspect ratio with width/height. Then depending on the if statement we calculate the new width by height*aspect ratio or the new height by width/aspect ratio. Then we resize the image. We create a padded image using np.zeros. The whole point of this is to update the silhouette so that the width and height match with

the background for overlaying it. We then calculate the positions x offset and y offset. img with padding we are placing the resized image onto the canvas of the calculated position before. Also if the image is gray scale we convert to BGR.

4.4 Results



5 Conclusions

Our journey in developing interactive photo editing software has created a tool that blends simplicity with functionality. By utilizing Python, PySimpleGUI, OpenCV, and Numpy we've created a user-friendly interface that makes photo editing accessible to anyone. We created three film effects, Emulsion Scratches and Dust, Grain Effect, and Double Exposure. The Emulsion Scratches and Dust effect brings the nostalgic charm of analog photography to digital images. The grain effect adds a layer of authenticity and texture, reminiscent of traditional film photography. The Double Exposure effect, updated for modern use, allows for imaginative overlays that can transform the mundane into the extraordinary.

YouTube Link: https://youtu.be/AXcDT7pd70I
Github Link: https://github.com/wasayMansoor/CompPhotographyFinalProject