Rucha Pendharkar 31st January 2024

Project I
Video Special Effects

I. Description

This project is to get familiarized with C++, the OpenCV package, and the mechanics of opening, capturing, manipulating, and writing images and live video data. It also involved writing custom filters

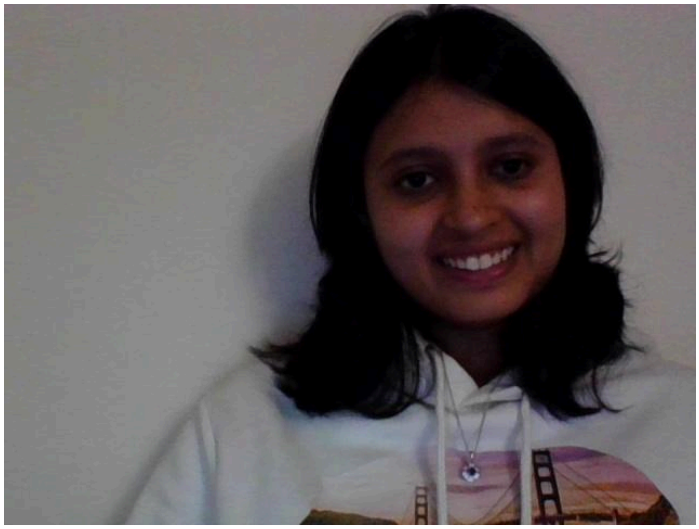
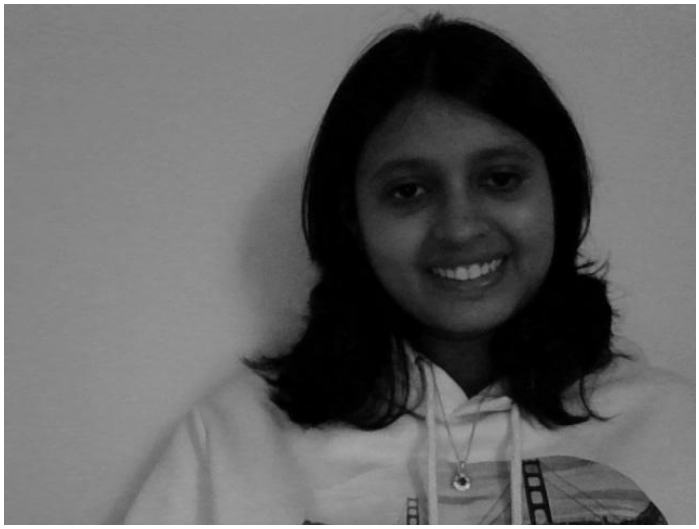
TThe scripts were authored using VS Code, and code compilation took place in the Ubuntu 20.04.06 LTS environment, utilizing CMake through the terminal

The following tasks have been defined:

1. Read an image from a file and display it
2. Display live video
3. Display grayscale live video
4. Display alternative greyscale live video
5. Implement a Sepia tone filter
6. Implement a 5x5 blur filter
7. Implement a 3x3 Sobel X and 3x3 Sobel Y filter as separable 1x3 filters
8. Implement a function that generates a gradient magnitude image from the X and Y Sobel images
9. Implement a function that blurs and quantizes a color image
10. Detect faces in an image
11. Implement three more effects on your video
    a. Single-step pixel-wise modification
    b. Area effect
    c. Building on the face-detector

II. Tasks

1. Greyscale Live Video

Original Image (Fig 1)	Greyscale Image (Fig 2)
	

This filter is implemented when the ‘g’ key is pressed.
The original figure (Fig 1) was converted to greyscale figure (Fig2) using the OpenCV function from cvtColor, **`COLOR_BGR2GRAY`**
.

$$Y \leftarrow 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B$$

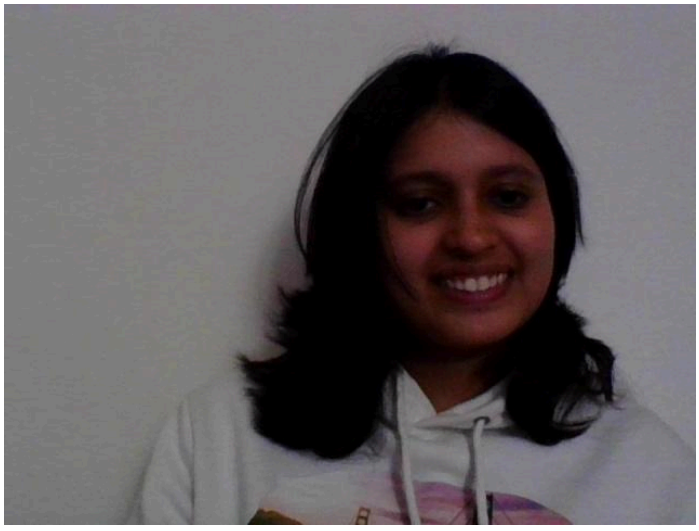
The above formula is used by applying these weights to color channels R,G and B.

2. Custom Greyscale Live Video

This filter is implemented when the ‘h’ key is pressed

The original figure (Fig 3) was converted to this custom greyscale figure (Fig 4) by individually setting each pixel frame to the average of the R,G and B color channel values for that particular pixel. This yields a slightly darker image as compared to the OpenCV grey conversion

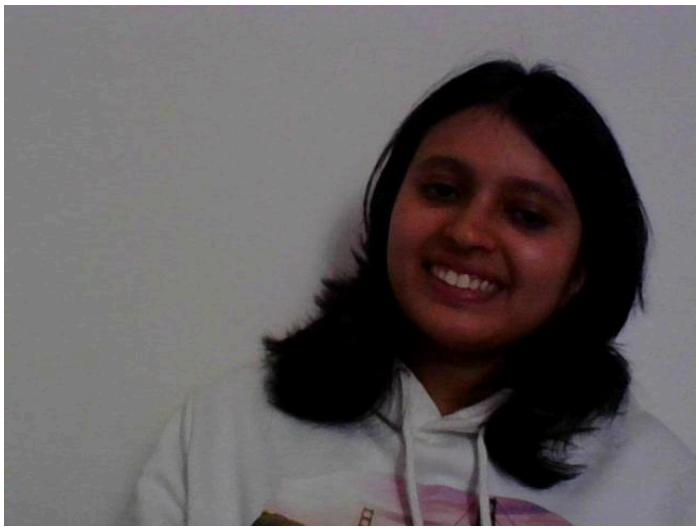
$$Y \leftarrow (R+G+B)/3$$

Original Image (Fig 3)	Custom Greyscale Image (Fig 4)
	

3. Sepia Tone Filter

This filter is implemented when the ‘p’ key is pressed.
Sepia filter operates on the original RGB values by first converting the image to the RGB color space, applying a transformation matrix with required coefficients, and then converting it back to BGR ensuring that the sepia effect is computed based on the original RGB color information of each pixel in the input image (fig5)

Original Image (Fig 5)	Sepia Image (Fig 6)



4. 5x5 Blur Filter

This filter is implemented when the ‘b’ key is pressed.

Two functions were written for generating the blur effect as needed and both of their performances were compared using the timeBlur.cpp file

Time per image (1): 0.2890 seconds
Time per image (2): 0.1415 seconds

The first image took almost twice as long as the second image. The reason for the second implementation being faster was:

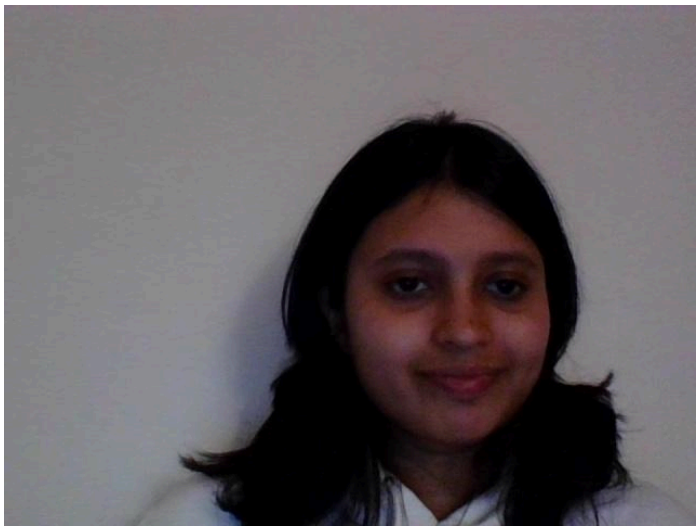
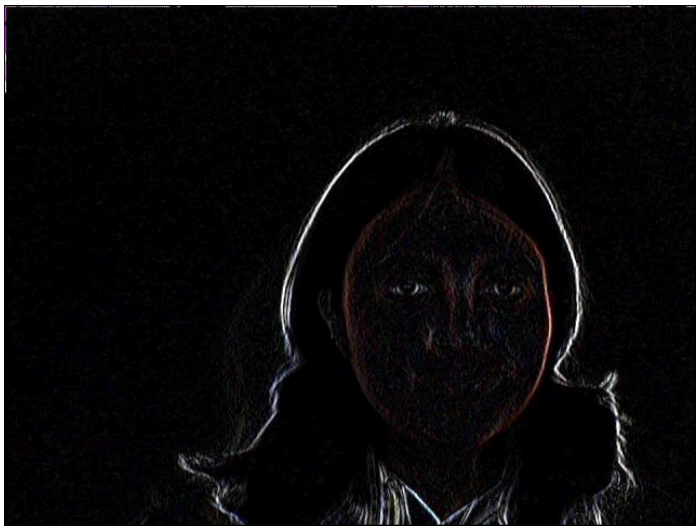
a. not using the ‘at’ method and instead using pointers,
b. the 5x5 filter was implemented as a 5x1 and 1x5 separable horizontal and vertical filters.

Original Image (Fig 7)	5x5 Image (Fig 8)

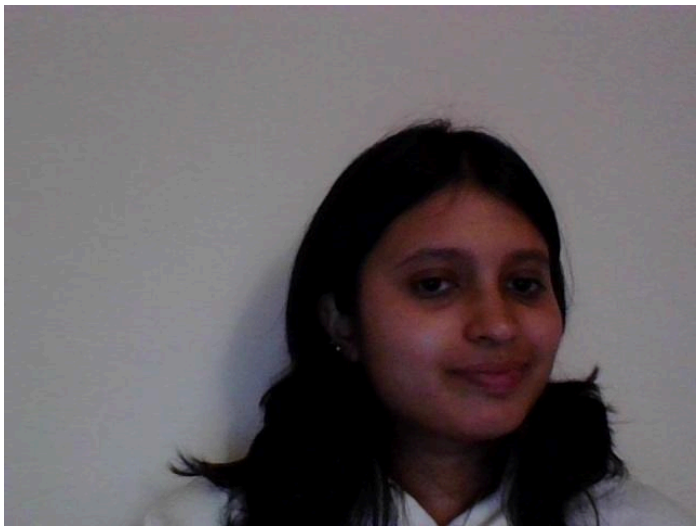
5. 3x3 Sobel X and 3x3 Sobel Y

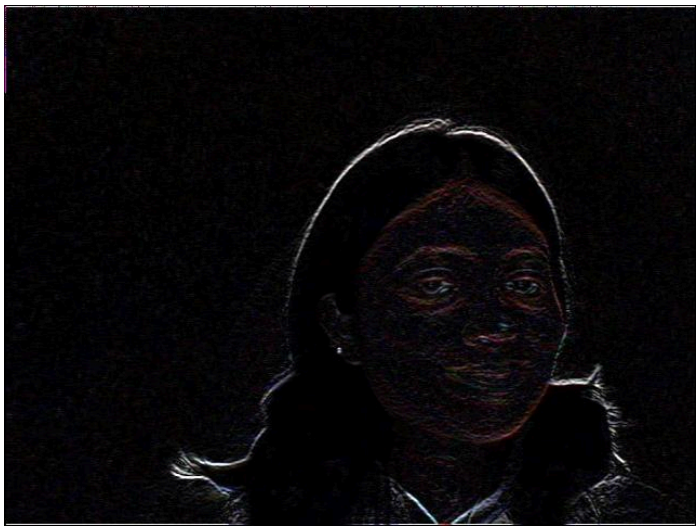
The Sobel X filter is implemented when the ‘x’ key is pressed, and Sobel Y filter is implemented when the ‘y’ key is pressed

The Sobel X filter shows vertical edges (fig10) and Sobel Y shows horizontal edges (fig12) They were implemented as separable filters (1x3 and 3x1)

Original Image (Fig 9)	Sobel X Image (Fig 10)
	

Original Image (Fig 11)	Sobel Y Image (Fig 12)





6. Gradient Magnitude Image

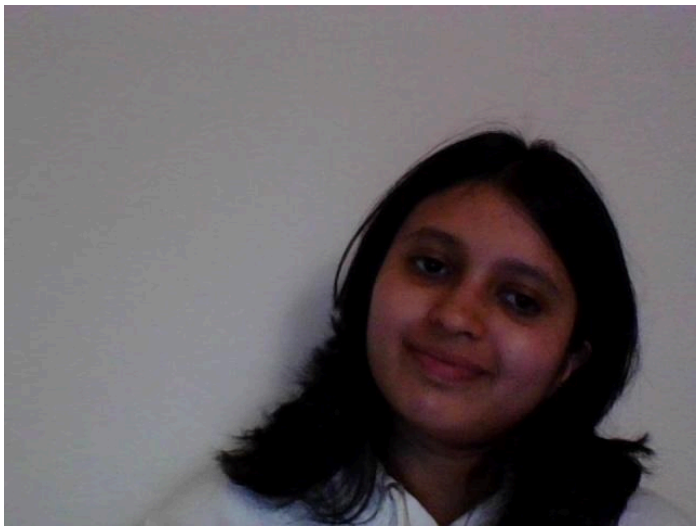
This filter is implemented when the ‘m’ key is pressed.

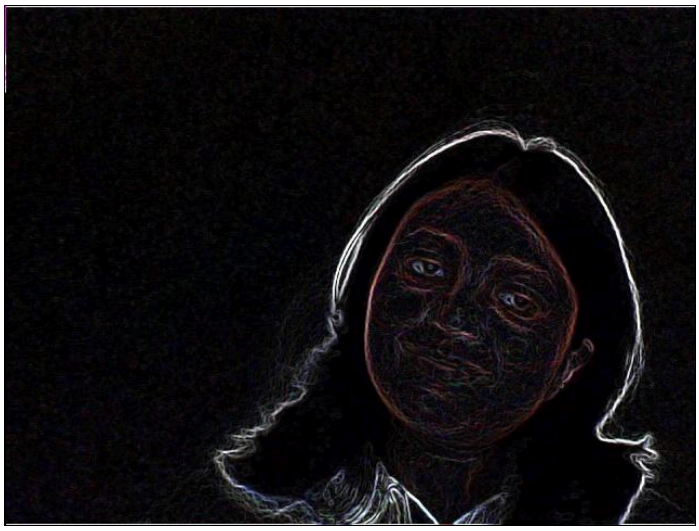
The filter generates a gradient magnitude image using Euclidean distance for magnitude:

$$l = \sqrt{sx^2 + sy^2}$$

Here sx and sy are outputs from the Sobel X and Sobel Y filters respectively.

Original Image (Fig 13)	Magnitude Image (Fig 14)

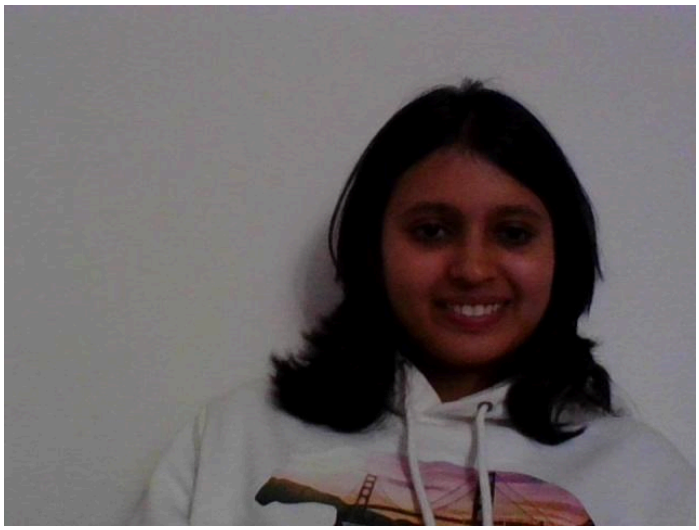




7. Blurred and Quantized Image

This filter is implemented when the “l” key is pressed.

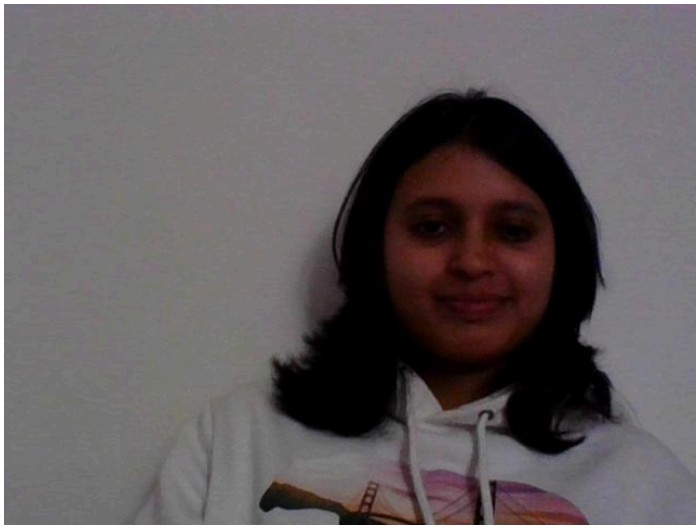
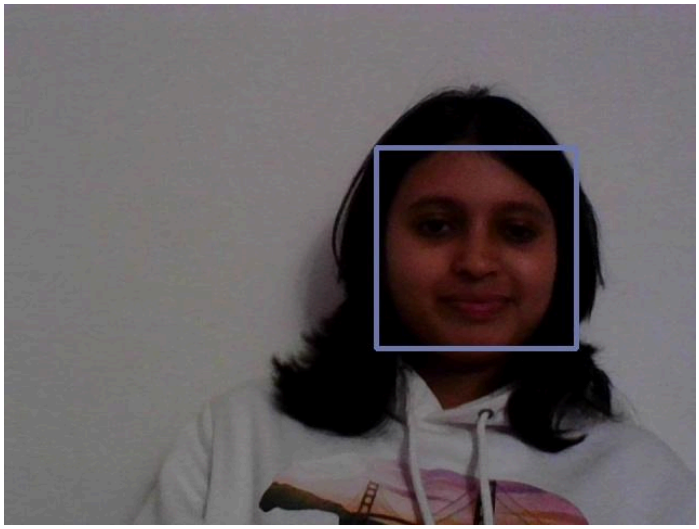
The function takes in a color image, blurs the image, and then quantizes the image (fig16) into a fixed number of levels as specified by a parameter. Calculate number of bucket, $b = 255 / levels$
Then execute $xt = x/b$, followed by $xf = xt * b$.

Original Image (Fig 15)	Blur Quantized Image (Fig 16)
	

8. Face Detection

This filter is implemented when the “f” key is pressed.

It detects a face using the faceDetect.cpp function and draws a box around it (fig18)

Original Image (Fig 17)	Face Detection Image (Fig 18)
	

9. Negative Image

This filter is implemented when the “n” key is pressed.

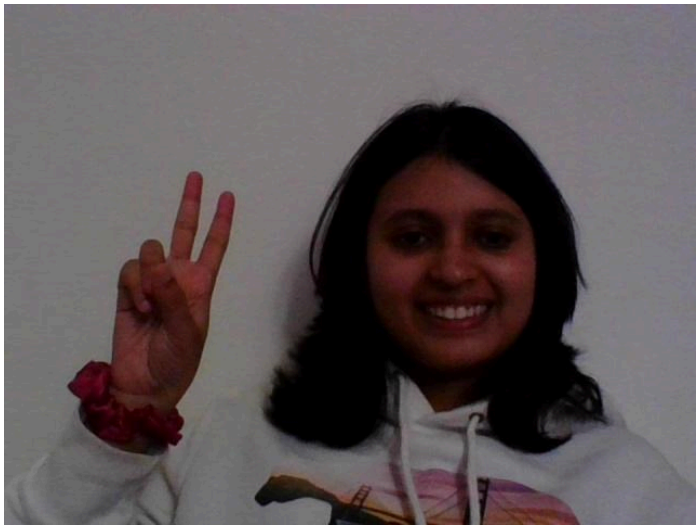
This is achieved by subtracting input (fig 19) pixel values from 255.

Original Image (Fig 19)	Negative Image (Fig 20)
	

10. Sparkles Around Edges

This filter is implemented when the “w” key is pressed.

This function uses the Canny Edge filter to detect edges and by overlaying white sparkles on the detected edges

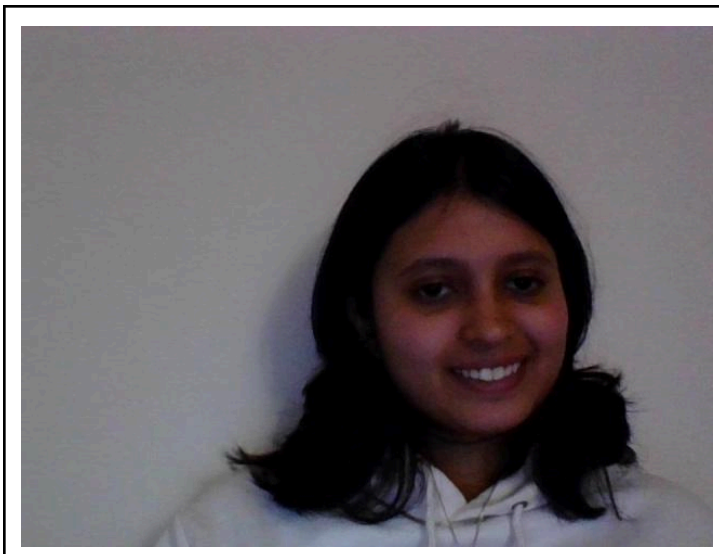
Original Image (Fig 21)	Sparkly Edge Image (Fig 22)
	

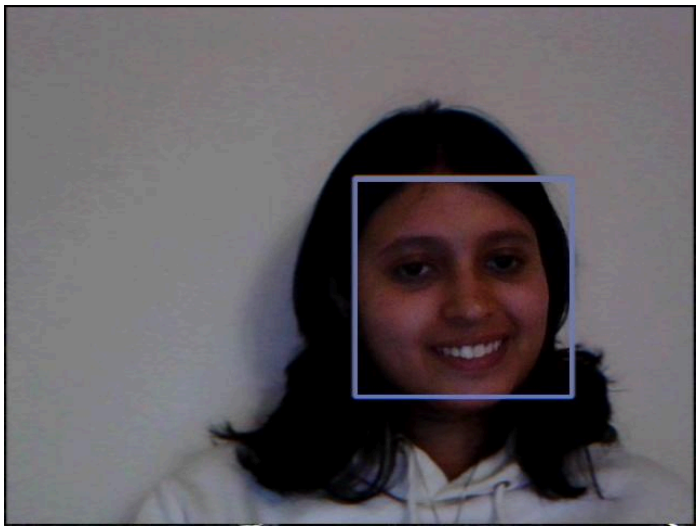
11. Blurs Area Around Face

This filter is implemented when the “r” key is pressed.

This function creates a mask to represent region outside of detected faces, applies blur to the original image, sets pixel values outside of the face to the corresponding pixel values of the blurred image (fig 24)

Original Image (Fig 23)	Blurred Face Image (Fig 24)



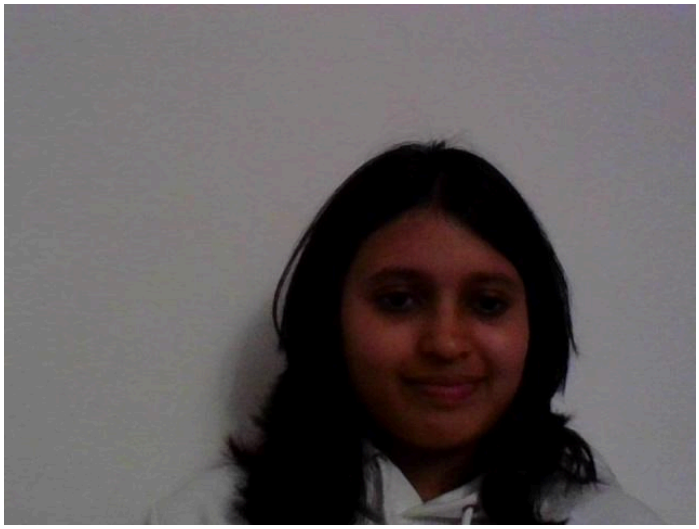
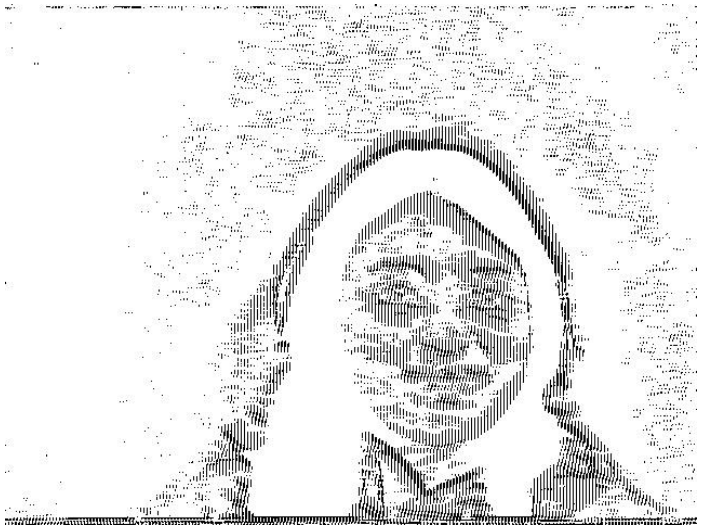


III. Extensions

For the extension, a new sketch filter was defined (inspired by a filter I saw on Snapchat). This filter has the following steps that need to be implemented in order to achieve the desired effect:

1. Convert the original frame to greyscale (used from task 1)
2. Apply the blur function (5x5_2 from task 6 was used)
3. Laplachian filter from OpenCV to to enhance the edges and details in the image
4. Invert colors to enhance dark lines and sketch like appearance
5. A binary threshold is applied to create a high-contrast image using the threshold function from OpenCV. Pixels with values greater than 150 become white, and those below 150 become black. This helps emphasize the edges and finalizes the pencil sketch effect (fig 26)

This filter is implemented when the “w” key is pressed.

Original Image (Fig 25)	Sketch Image (Fig 26)
	

IV. Reflection

I particularly enjoyed this project. I felt it was a comprehensive base to get familiar with OpenCV functions and coding in C++. Playing around with different combinations of filters to achieve completely new filters was also fun. Once I had the basic pipeline set for my code, implementing newer functions became easier. It also took me some time to understand the use cases for the different data types in OpenCV.

V. Acknowledgments

I would like to thank Prof Bruce Maxwell for helping me grasp these new concepts quickly, and for providing various course materials that were useful in completing this project. I would like to thank the Teaching Assistants for their timely help in debugging small errors in my code during implementation of these tasks.

Materials referred -

a. CS 5330 Lecture notes and materials
b. OpenCV documentation
c. https://learnopencv.com/image-filtering-using-convolution-in-opencv/
d. https://linuxize.com/post/how-to-install-opencv-on-ubuntu-20-04/
e. Stackoverflow