Video-Special effects

Chirag Dhoka Jain, Keval Visaria
Project 1 Submission
course - ***EECE5330***
{dhokajain.c, visaria.k}@northeastern.edu
January 26, 2023
Northeastern University

I. Introduction

This project delves into real-time video processing using the OpenCV library, employing C++ for a versatile and dynamic approach. Beginning with a basic image display program, the project progresses through live video processing, grayscale transformations, and custom filters. Fundamental image processing techniques, including a sepia tone filter and an optimized 5x5 blur filter, lay the groundwork. To address practical issues, basic mathematical calculations are integrated, refining the implemented filters.

The inclusion of Sobel filters for edge detection, blurring, and quantization enhances versatility, while face detection demonstrates adaptability. Beyond utility, the project explores creative dimensions with three additional effects. Emphasizing real-world application, basic mathematical calculations are employed to sort issues and optimize performance.

A. *Task 1: Reading an Image*

In the imgDisplay.cpp program, we implemented image display functionality using C++ and an image processing library. The main function reads an image file from the laptop drive, displays it in a window, and enters a loop to check for user input. The program allows the user to interact with the displayed image by detecting keypresses. If the user types 'q', the program gracefully exits the loop and terminates.

The input image serves as the source file for the program, and the output image demonstrates the successful display in the application window.

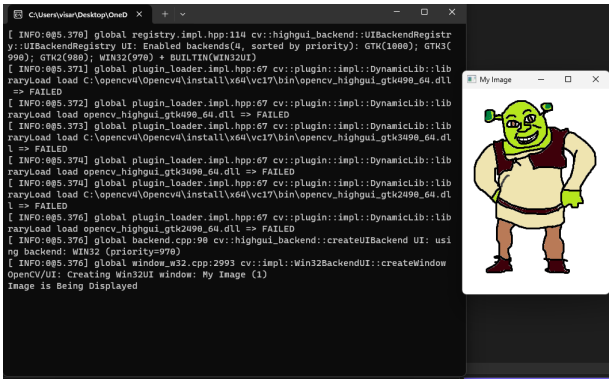

Fig. 1. Output of image from file

B. *Task 2: Displaying live video*

In this task we have created vidDisplay.cpp program, the main function establishes a video channel, creates a display window, and continuously captures and displays video frames in a loop. The program allows users to interact with the video stream by typing 'q' to exit gracefully. Additionally, users can save a current frame to a file by typing 's', enhancing the functionality for practical applications. The seamless integration of video capture, display, and user interaction makes this program a versatile tool for real-time video processing.

To illustrate the program's functionality, we have attached snapshots showcasing the output image representing a frame saved upon user request.

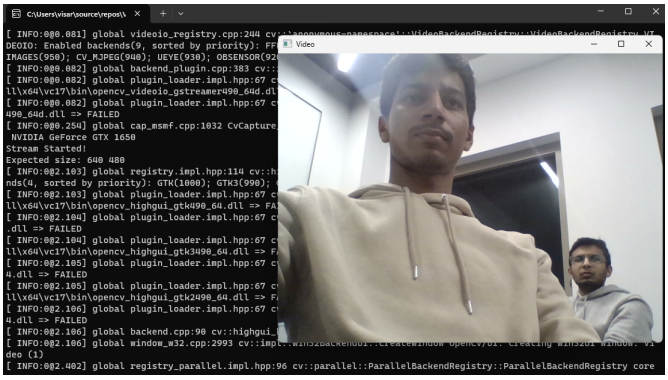

Fig. 2. Output of image in the application window

C. *Task 3: Conversion to Greyscale*

In this task, we enhanced the vidDisplay.cpp program so that the user can toggle between color and grayscale live video displays by typing 'g'. The implementation utilizes the OpenCV cvtColor function to convert each video frame accordingly, and the program remembers the last keypress to maintain the selected mode. This feature adds flexibility to the application, allowing users to dynamically switch between color and grayscale video output using key 's' and 'g'.

To illustrate the program's functionality, we have attached snapshots showcasing the output images representing a frame saved upon user request.

Fig. 3. Output of image saved 's' in video stream

Fig. 5. Output of image saved in greyscale

Fig. 4. Output of image saved in grayscale

Fig. 6. Output of image saved in custom greyscale

D. Task 4: Conversion to custom greyscale

In this task, we modified vidDisplay.cpp programme allows users to switch between multiple greyscale video modes by inputting 'g' for the OpenCV greyscale version and 'h' for the alternate greyscale version. The alternate greyscale transformation is included in the newly built filter.cpp file. The core of the function involves a pixel-wise transformation of the source image. The nested loops iterate through each pixel of the source image. For each pixel, the red channel value is extracted, and a custom transformation is applied by subtracting it from 255. This unique transformation creates a distinct greyscale effect. The resulting greyscale value is then set for all three color channels (Blue, Green, and Red) in the destination image. This process is repeated for each pixel in the image.This feature adds flexibility to the application, allowing users to dynamically switch between color and custom grayscale video output using key 'h'.

```
vid: vidDisplay.o filters.o
    $(CC) $^ -o $(BINDIR)/$@ $(LDFLAGS) $(LDLIBS)

int greyscale( cv::Mat &src, cv::Mat &dst );
```

Listing 1. C++ code for vidDisplay.o and filters.o

We used the above function to create customization on greyscale program. To illustrate the program's functionality, we have attached snapshots showcasing the output images representing a frame saved upon user request.

E. Task 5: Sepia tone filter

In this task, we have successfully implemented the sepia tone filter in the image processing function sepiaToneFilter, incorporating the specified matrix for color channel modification. The values within the matrix shown below were chosen to evoke the antique camera aesthetic, imparting a nostalgic and warm tone to the images. The program's user-friendly interface allows toggling between the original and sepia-toned versions using the 't' key.

$$0.272 + 0.534 + 0.131 \quad \text{// Red coefficients}$$
$$0.349 + 0.686 + 0.168 \quad \text{// Green coefficients}$$
$$0.393 + 0.769 + 0.189 \quad \text{// Blue coefficients}$$

The core of the function involves iterating through each pixel of the source image using nested loops. For each pixel, the RGB values are accessed through the $cv :: Vec3b$ structure, representing the Blue, Green, and Red channels. The sepia tone transformation is then applied using a predefined $3x3$ matrix $sepiaMatrix$. This matrix contains specific coefficients for each channel to create the sepia effect. The mathematical transformation involves multiplying each channel's original intensity by the corresponding coefficient in the $sepiaMatrix$ and summing up the results. This produces new intensity values for the Red, Green, and Blue channels in the sepia-toned image. It's important to note that these calculations may result in values outside the valid color range of [0, 255]. To

address this, the calculated sepia values are clipped to ensure they fall within the valid range. This is achieved using the $std :: min$ and $std :: max$ functions, constraining the values between 0 and 255. The clipped sepia values are then assigned to the destination image dst using the $cv :: Vec3b$ structure. In summary, the function performs a pixel-wise sepia tone transformation by applying a matrix operation to the RGB channels of each pixel in the source image

To illustrate the program's functionality, we have attached snapshots showcasing the output images representing a frame saved upon user request.

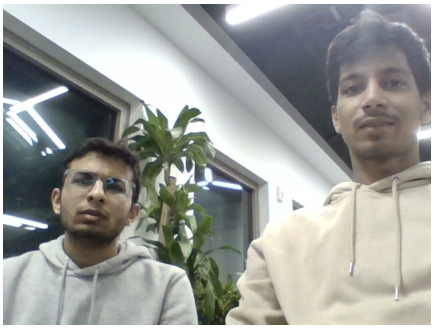

Fig. 7. Output of image saved in live video stream

Fig. 8. Output of image saved in SepiaTone filter mode

An nice extension to the filter we have included the addition of vignetting, where the image gradually darkens towards the edges. Within the function, each pixel of the source image is processed in a nested loop. For every pixel, the distance from the pixel to the center of the image is calculated, normalized by the distance to the center. This normalized distance $dist$ ranges from 0 at the center to 1 at the corners.

The vignette effect is then determined using a mathematical formula:

$$\text{vignette} = 1.0 - \text{vignetteStrength} \cdot \left(1.0 - \exp\left(-0.5 \cdot \left(\frac{\text{dist}}{\text{vignetteRadius}}\right)^2\right)\right)$$

Here, $vignetteStrength$ influences the overall darkness of the vignette, and $vignetteRadius$ controls how quickly the effect fades towards the image corners. The computed vignette value is then used to adjust the intensity of each color channel (Red, Green, Blue) of the pixel. Multiplying each channel by the vignette value ensures that the vignetting effect is applied uniformly across all color components. This fucntion can be used using key 'v'.

Fig. 9. Original image

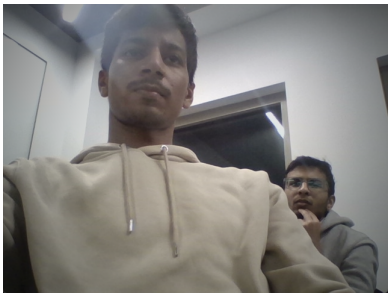

Fig. 10. Output of image saved in Vignetting mode

F. Task 6: Blur effect

1) 6A: Blur effect

In this task, we have created a $5x5$ blur filter, using a simple technique with a single nested loop with the given below Gaussian approximation matrix. The function works separately on each colour channel of the picture, retaining the original size but omitting the outside two rows and columns from change. The implementation preserves the src picture and stores the result in the dst image. Timing study was done to determine the first implementation's execution speed. $blur5x5_1$ employs a direct convolution technique, traversing the entire image with a 5x5 kernel. This method involves nested loops to iterate over each pixel, executing multiplications and additions for each color channel. The implementation, while straightforward, carries a significant computational load, particularly within the nested loops. Numerically, the $5x5$ convolution kernel in $blur5x5_1$ has coefficients that sum to 88, representing the normalization factor for the convolution operation.

The C++ function signature for the blur operation is:

```
int blur5x5_1(cv::Mat &src, cv::Mat &dst);
```

Listing 2. C++ Function Signature for blur5x5_1

The Gaussian kernel matrix is represented as:

$$\begin{bmatrix} 1,2,4,2,1 \\ 2,4,8,4,2 \\ 4,8,16,8,4 \\ 2,4,8,4,2 \\ 1,2,4,2,1 \end{bmatrix}$$

The blur operation can be expressed as:

$$\text{dst}(i,j) = \sum_{m=-2}^{2} \sum_{n=-2}^{2} \text{src}(i+m, j+n) \times \text{Gaussian}(m,n)$$

To illustrate the program's functionality, we have attached snapshots showcasing the output images representing a frame saved upon user request.

Fig. 11. Output image

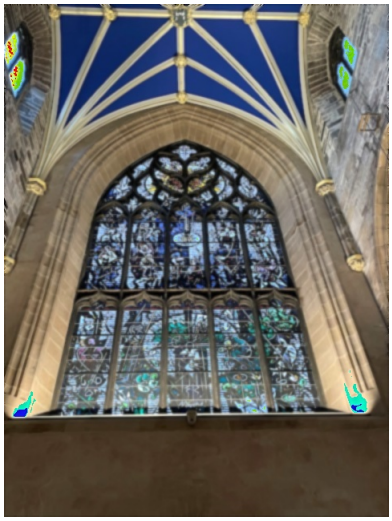

Fig. 12. blur effect on the image

2) 6B: Blur effect- separable filter

In this task, we have created a second version of the $5x5$ blur filter, which aims to improve speed by utilising the separable filters described below both vertically and horizontally. The optimised method, which is activated by the 'b' key in vidDisplay.cpp, improves the real-time video processing experience by producing a quicker and smoother blurred image.

On the other hand, $blur_5x5_2$ adopts a more computationally efficient strategy by applying two 1x5 kernels sequentially in a separable manner. The separation into horizontal and vertical operations reduces the number of computations per pixel. This approach aims to achieve a Gaussian blur with fewer multiplications and additions, potentially enhancing overall performance. In $blur_5x5_2$, the separable kernels are $[-5, 0, 20, 0, -5]$, each with a normalization factor of 10.

The C++ function signature used for the blur operation:

```
int blur5x5_2(cv::Mat &src, cv::Mat &dst);
```

Listing 3. C++ Function Signature for blur5x5_2

Fig. 13. Output image

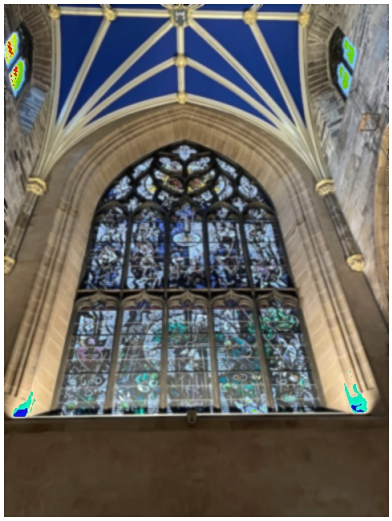

Fig. 14. blur effect on the image

```
Time per image (blur5x5_1): 0.4644 seconds
Time per image (blur5x5_2): 0.3794 seconds

D:\My source\Spring2024\PRCV\c++\blur\out\build\x64-debug\blur.exe
 (process 30080) exited with code 0.
To automatically close the console when debugging stops, enable To
ols->Options->Debugging->Automatically close the console when debu
gging stops.
Press any key to close this window . . .
```

Fig. 15. Calculated time for both blur filters

When we executed we had two time difference $fig.15$, First $blur5x5_1$ blurred it took around 0.464s. Second time using $blur_5x5_2$ it took around 0.3764 . A bit faster than before image generation.

During performance evaluation, if $blur_5x5_2$ exhibits faster timing than $blur5x5_1$, the observed efficiency is likely attributed to the reduced computational complexity of the separable kernel approach. The advantages of this strategy include a more streamlined execution, benefiting from fewer operations required to achieve the Gaussian blur effect. The observed performance difference could be influenced by various factors, such as memory access patterns, cache efficiency, and the specifics of hardware optimizations. The direct convolution in $blur5x5_1$ may lead to a higher computational load, potentially impacting its overall execution time. In conclusion, the comparison underscores the trade-offs between direct convolution and separable kernel strategies for Gaussian blurring. The efficiency gains of the separable approach in $blur_5x5_2$ highlight its potential benefits in scenarios where computational performance is a critical consideration.

G. *Task 7: Sobel X and Sobel Y*

In this task the Filter.cpp has two functions, sobelX and sobelY, each of which applies a three-cross-three Sobel filter as separable filters. The X filter recognises horizontal edges with positive values to the right, whereas the Y filter recognises vertical edges with positive values upwards. When testing these functions with vidDisplay, the 'x' key displays the X Sobel output (the absolute value), while the 'y' key displays the Y Sobel output. The use of Sobel filters improves real-time video processing by emphasising edges in both horizontal and vertical dimensions.

The C++ function used for Sobel X and Sobel Y operations are:

```
1 int sobelX3x3(cv::Mat &src, cv::Mat &dst);
```

Listing 4. C++ Function Signature for sobelX3x3

```
1 int sobelY3x3(cv::Mat &src, cv::Mat &dst);
```

Listing 5. C++ Function Signature for sobelY3x3

To illustrate the program's functionality, we have attached snapshots showcasing the output images representing a frame saved upon user request.

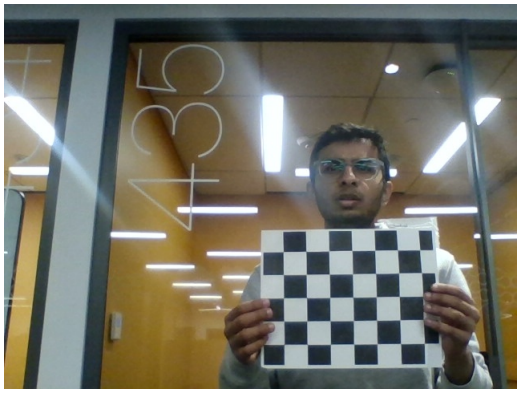

Fig. 16. Output image on live video stream

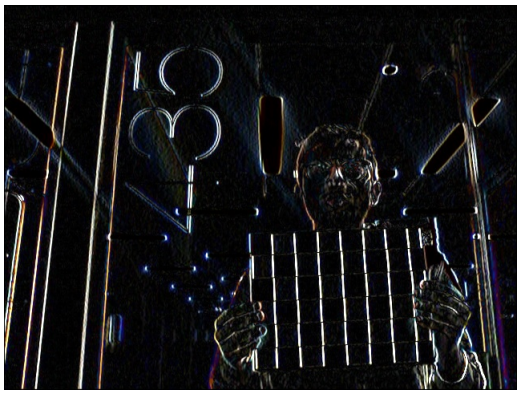

Fig. 17. Output image saved on Sobel X

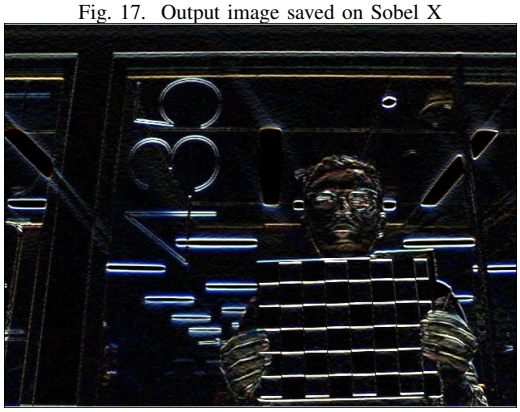

Fig. 18. Output image saved on Sobel Y

H. *Task 8: Gradient magnitude on Sobel X Y image*

In this task, in filter.cpp file the function 'magnitude' generates a gradient magnitude image using the Euclidean distance formula shown below. This color image function takes two 3-channel signed short images (sx and sy) as input and produces a uchar color image suitable for display (dst). The 'm' key in vidDisplay can now be used to showcase the color gradient magnitude image, providing a visual representation of the combined effects of the X and Y Sobel filters.

The Euclidean distance for magnitude is given by the formula:

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

where I is the magnitude and sx and sy are the components along the x and y axes, respectively.

The C++ function for calculating the magnitude is:

```
int magnitude(cv::Mat &sx, cv::Mat &sy, cv::Mat
    &dst);
```

Listing 6. C++ Function Signature for Magnitude Calculation

To illustrate the program's functionality, we have attached snapshots showcasing the output images representing a frame saved upon user request.

Fig. 19. Output image

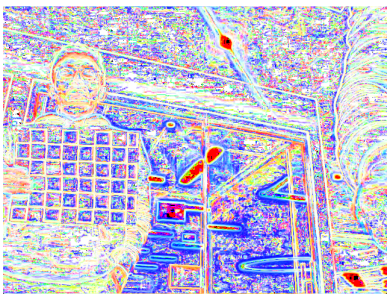

Fig. 20. Gradient effect

I. *Task 9: Blur and Quantize a color image*

In this task, the method blurQuantize, which is implemented in filter.cpp, applies a two-step procedure on a colour picture. It first blurs the image before quantizing it into a defined number of levels. The quantization is accomplished by dividing each colour channel value by the bucket size and multiplying it back. The parameter used in our situation is 10. When testing the functions with vidDisplay, the 'l' key displays the X BlurQuantize image output. The key is used in 'l'.

The C++ function for the blurQuantize operation is:

```
int blurQuantize(cv::Mat &src, cv::Mat &dst, int
    levels);
```

Listing 7. C++ Function for blurQuantize

To illustrate the program's functionality, we have attached snapshots showcasing the output images representing a frame saved upon user request.

Fig. 21. Output image

Fig. 22. Blur and quantize effect on the image

J. *Task 10: Face Detection*

In this task, the face detection feature has been seamlessly integrated into the video stream program, enhancing it with the ability to locate and highlight faces when the user presses the 'f' key.

To illustrate the program's functionality, we have attached snapshots showcasing the output images representing a frame saved upon user request.

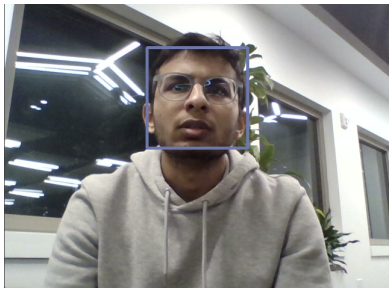

Fig. 23. Detected face on video stream

K. *Task 11: Effects on video*

In this task, we considered using four ideas as per given;

1) Strong color to remain and else to greyscale

Here we made a function named pickStrongColor that processes an input color image (src) to emphasize pixels with an intensity greater than a specified threshold. The resulting processed image is stored in the output matrix (dst). This function aims to accentuate pixels with strong colors while converting others to greyscale. the function iterates over each pixel in the input image. For each pixel, the intensity, representing the overall brightness, is calculated by averaging the values of the three color channels (blue, green, and red). This intensity is then compared to a specified threshold. If the intensity is greater than the threshold, the pixel is considered strong, and its original color is preserved in the output image (dst). If the intensity is below the threshold, the pixel is considered weaker, and the function converts it to greyscale by assigning a grey value equivalent to its intensity. The resulting greyscale pixel is then stored in the output image. In essence, the pickStrongColor function performs a selective colorization, preserving strong colors in the output while converting weaker colors to greyscale. This can be useful for highlighting specific elements in an image based on their color intensity. To illustrate the program's functionality, we have attached snapshots showcasing the output images representing a frame saved upon user request. The key used for this fucntion is 'n'.

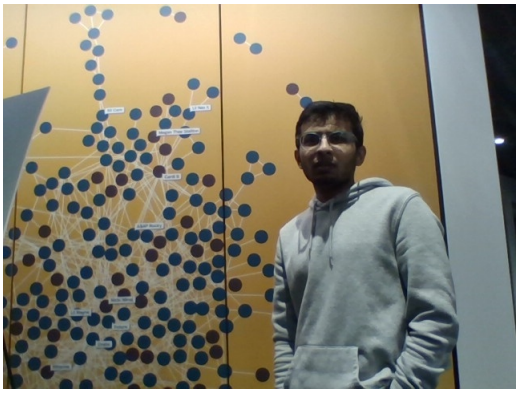

Fig. 24. Output image

Fig. 25. Detection of strongest color, rest grey

2) Sparkles in a halo above face

In this task Above each detected face, the function adds a whimsical touch by drawing a random number (default is 5) of hearts. Each heart is uniquely sized, with its dimensions

randomly selected between 20 and 50 pixels. The hearts are strategically positioned above the face, avoiding overlap with the facial region. The positions are determined by randomization using the $std :: rand$ function, ensuring a diverse and dynamic arrangement of hearts for each face. The hearts themselves are drawn using the $drawHeart$ function, which combines elliptical and polygonal primitives to create a heart shape. The hearts are filled with a pink color (RGB: 0, 0, 255), and the size and position of each heart contribute to the overall aesthetic appeal. The key used is 'c'.

To illustrate the program's functionality, we have attached snapshots showcasing the output images representing a frame saved upon user request.

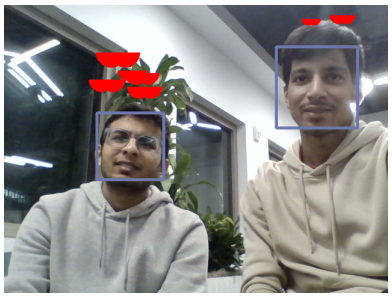

Fig. 26. Halo above face

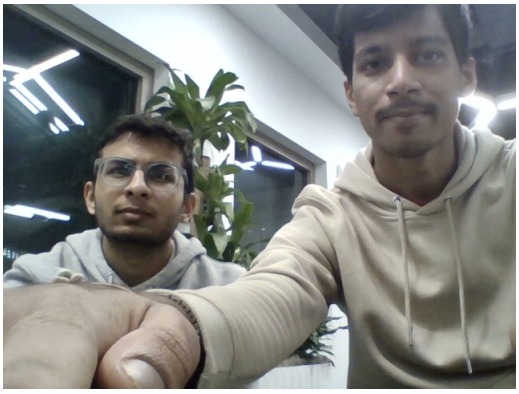

Fig. 27. Output image

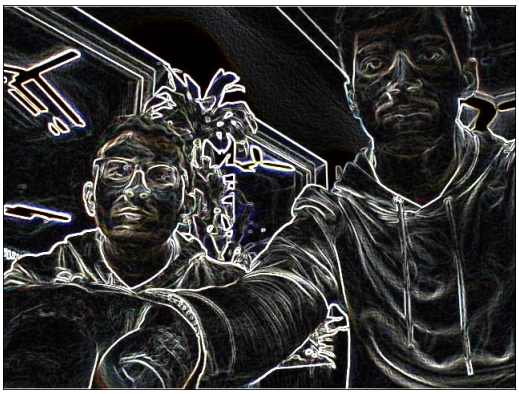

Fig. 28. Embosed effect on Sobel X Y image

3) *Embossing effect on Sobel X Y image*

Here we provided C++ code that defines the embossingEffect function, which applies an embossing effect to a given color image using Sobel X and Sobel Y filters. This effect enhances the prominence of edges in the image by emphasizing changes in intensity. The Sobel X filter, implemented by the $sobelX3x3$ function, computes the horizontal gradient of the image, while the Sobel Y filter, implemented by $sobelY3x3$, computes the vertical gradient. These gradients capture intensity variations in the horizontal and vertical directions, respectively.

The embossing effect is achieved by combining the Sobel X and Sobel Y results. The function iterates over each pixel in the source image, calculating the combined embossing value for each color channel. The embossing value is the absolute sum of the corresponding Sobel X and Sobel Y values at that pixel position.Numerically, the embossing value at each pixel is the sum of absolute gradient values in both the horizontal and vertical directions, contributing to the enhanced representation of edges in the image. The Key used is 'p'. To illustrate the program's functionality, we have attached snapshots showcasing the output images representing a frame saved upon user request.

4) *User control Contrast*

Here we made a function named $adjustBrightnessContrast$ designed to modify the brightness and contrast of an input image $inputFrame$. The adjusted image is stored in the output matrix $outputFrame$. The function utilizes the OpenCV library to perform these adjustments efficiently. The $convertTo$ function from OpenCV is employed to transform the input image's pixel values based on specified brightness and contrast parameters. The function takes the input frame and computes the corresponding output frame by applying a linear transformation to each pixel's intensity values. The transformation is controlled by the provided contrast and brightness factors. To increase key 'd' is used and to decrease key used is 'a'

To illustrate the program's functionality, we have attached snapshots showcasing the output images representing a frame saved upon user request.

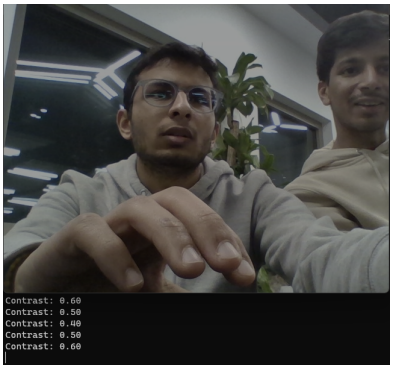

Fig. 29. Output image

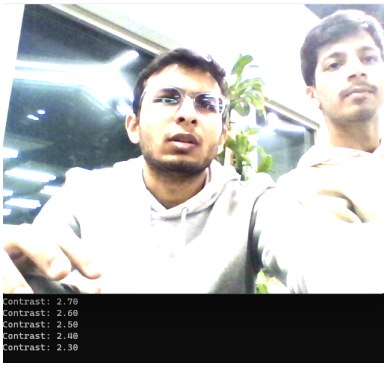

Fig. 30. Detected face on video stream

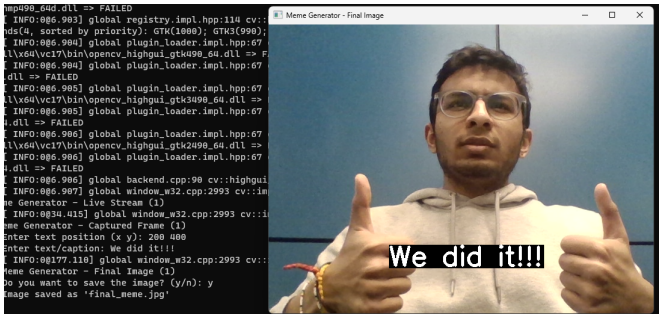

Fig. 31. Output image after adding up the desired text

L. *Extensions*

We tried to implement the extension which we felt could be :

1) Meme Generator

In this task , we tried Meme Generator using the OpenCV library. It captures frames from a webcam, enabling users to select a frame, and prompts them to input a caption. The entered caption is then overlaid onto the selected frame, forming a meme. Users are given the option to save the final meme image. Within the main function, the code establishes a connection to the webcam using the VideoCapture class, captures an initial frame from the webcam, and saves it as "$initial_image.jpg$". It then enters a loop for continuous frame processing, displaying the live stream from the webcam and awaiting user input.

Upon the user pressing 's', the code displays the captured frame, prompts the user to enter the text position (x, y), and subsequently requests the user to input the desired text or caption. The text is then drawn on the image with a white color and a black background, and the final meme image is displayed. Users are given the option to save the image as "$final_meme.jpg$" if desired.g".

2) Green Screen

In this task we took green color manipulation because our eyes are very feasible to green color. Program utilizing the OpenCV library to implement a basic green screen effect in real-time video capture from a webcam. The program is structured around a main loop that continuously captures frames from the webcam, applies a green screen effect when activated, and displays the result. The user can start and stop the green screen effect The main components of the code include the initialization of the webcam using OpenCV's VideoCapture class, the definition of the HSV color range corresponding to the green screen, and the manipulation of the captured frames. The green screen effect is achieved by converting each frame from the default BGR color space to the HSV color space, creating a mask based on the defined green color range, inverting the mask, and then applying it to the frame using bitwise operations. Overall, this code provides a simple and interactive implementation of a green screen effect using OpenCV

Fig. 32. Output image

Fig. 33. Output image after adding up the desired green screen

II. What we learned!!!

1) This project offered hands-on experience, enabling us to learn through practical tasks and experimentation with different techniques in computer vision. As we familiarized ourselves with the OpenCV library, solving tasks enhanced our understanding of its functionalities and improved our skills in video processing and image manipulation.

2) Dealing with pixel-level manipulation in tasks provided insights into the complexities of image processing algorithms and how individual pixels contribute to transforming an entire image.

3) By implementing filters and effects like blur and embossing from scratch, we gained a deeper understanding of the algorithms and techniques used in image processing.

4) Working with various image filters, including greyscale transformations, Sobel filters, sepia tone filters, and blur filters, enhanced our understanding of their functionalities and applications in computer vision.

III. Acknowledgement

Acknowledging the wealth of knowledge available online, particularly in the 'OpenCV' documentation and tutorials, played a crucial role. This project's success is a testament to the collective effort of the educational community, emphasizing the shared wealth of expertise. Interactions with ChatGPT proved invaluable for exploring concepts and troubleshooting, enhancing the overall learning experience of the project. This acknowledgment is a tribute to the collaborative and supportive environment that significantly facilitated the learning journey in this project.

For the code related to this project, please refer to: https://shorturl.at/CHRS8[1]

[1]Code for the project.