REPORT

PROJECT 01

(Video-Special Effects)

By – Veditha Gudapati
&
Tejasri Kasturi

Introduction

We have implemented a project with C++ language. It primarily uses the OpenCV library to interact with video input, carry out various image processing tasks, and create creative visual effects. The first task is to write a program to record live video and show visuals. The project moves forward by adding greyscale transformations, to switch between color and greyscale video streams using a mathematical approach. To improve code structure and modularity, image manipulation functions are moved into a separate file called filter.cpp. Additionally, the project includes more advanced filters, like a sepia tone filter, blur filter and vignette filter. The Blur filter is implemented in two methods. The first method is to construct a 5x5 blur filter: a simple implementation and a second implementation which uses separable 1x5 filters. The latter method presents methods for performance improvement through optimization. Edge identification is made possible by the integration of the Sobel X and Sobel Y filters, which gives an important new perspective on gradient-based image processing methods. The next step is to calculate the gradient magnitude, which explains how to combine several filters to get a meaningful outcome. The project also consists of the implementation of face recognition in a video stream.

TASK – 1:

The task required us to write a program in "imgDisplay.cpp" to enter a loop, checking for a keypress. If the user types 'q', the program should quit.
Additionally, we have added:
"s" which saves the image.
"r" which rotates the image clockwise
"l" which rotates the image anti-clockwise.

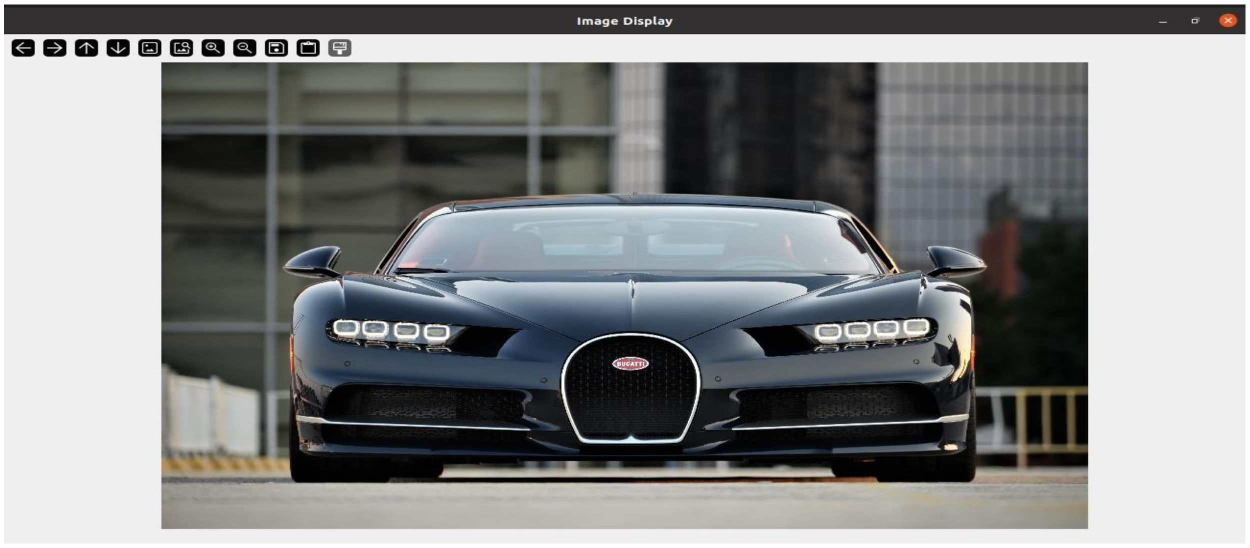
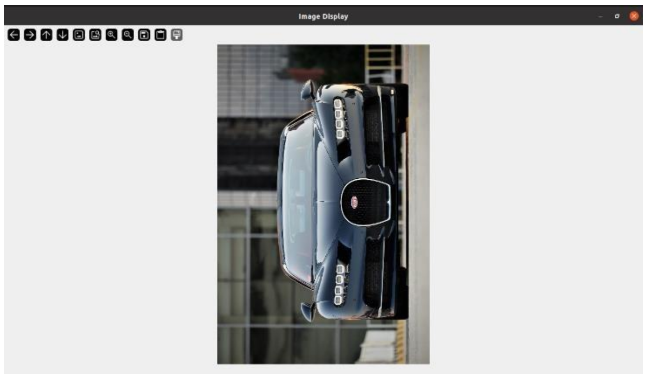


Fig 1: Original Image



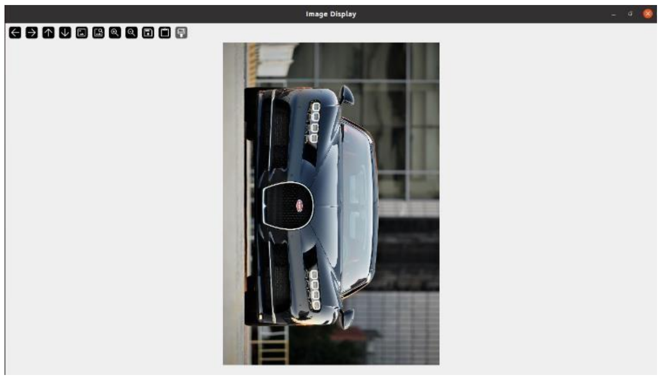


Fig 2: Rotate Anti-Clockwise on "l" ***Fig 3: Rotate Clockwise on "r"***

TASK – 2:

The task requires us to write a program "vidDisplay.cpp" to open a video channel, create a window, and then loop, capturing a new frame and displaying it each time through the loop. In addition, the program must include key presses. When "q" is pressed, it must quit; when "s" is pressed, it must save the video.

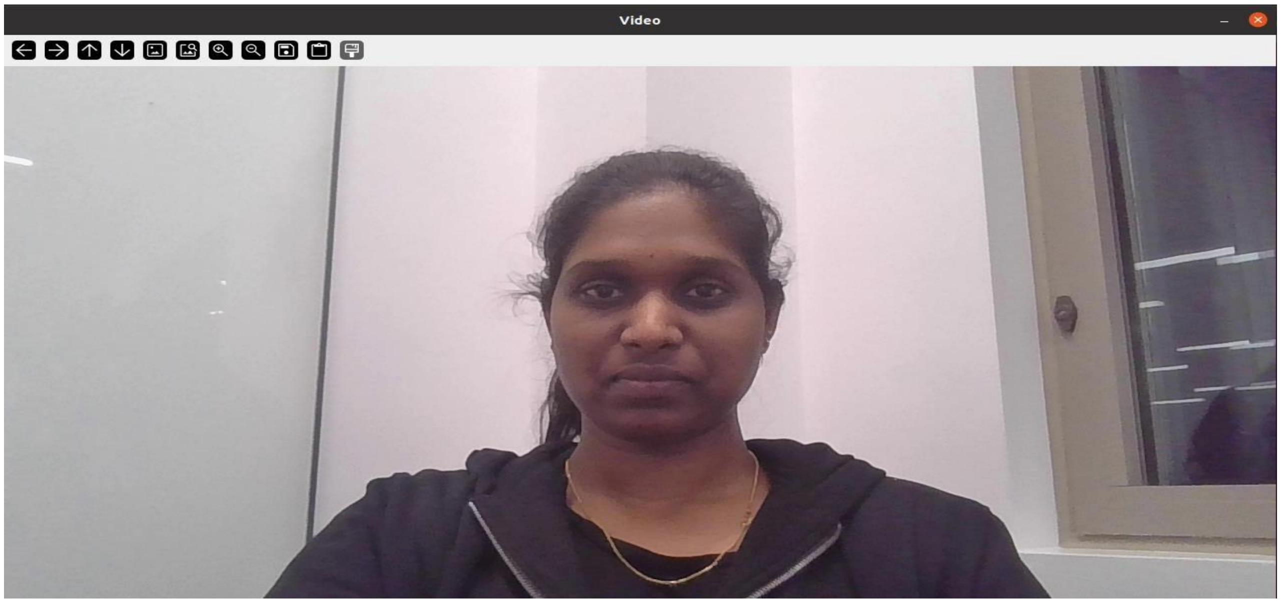


Fig 4: Video Stream

TASK – 3:

The task requires us to display the greyscale version of the live video stream when the “g” key is pressed. The “cvtColor” function from OpenCV is used to convert the video stream to greyscale. In the process of converting RGB to greyscale in the video stream, each RGB pixel in the RGB image is converted into a single greyscale intensity value in this process. The weighted method computes the greyscale intensity by assigning a distinct weight to each color channel (R, G, and B).

The weighted greyscale conversion formula in OpenCV is:

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

Red Channel (R): Weight: 0.30

- Explanation: The human eye is more sensitive to variations in the red spectrum, and this weight is chosen to represent its importance in the greyscale representation.

Green Channel (G): Weight: 0.587

- Explanation: Green contributes significantly to the overall brightness of an image, and the higher weight reflects its stronger impact on luminance. The human eye is most sensitive to green light.

Blue Channel (B): Weight: 0.114

- Explanation: Blue has the least impact on perceived luminance, and this weight reflects its lesser importance in greyscale representation.

The weights 0.299, 0.587, and 0.114 are chosen because the human eye is more sensitive to green than to red or blue when it comes to color perception. These weights are selected to both visually depict the original image in greyscale and to reflect this sensitivity.

The cvtColor function converts each pixel from its RGB representation to a single greyscale intensity value by applying these weights; the luminance information is retained while the color information is discarded.

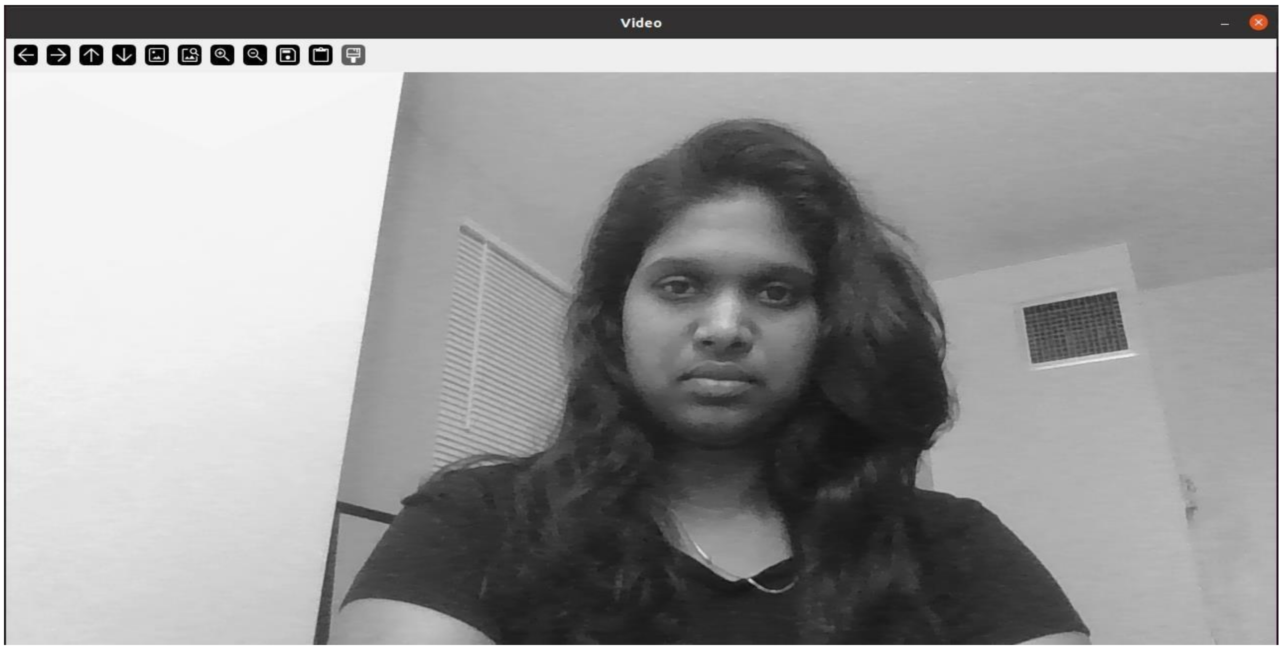


Fig 5: Greyscale Filter on Video Stream

TASK – 4:

This task requires us to add a new keypress “h”, which displays an alternative greyscale version of the video stream. Additionally, a new filter.cpp file must be created. This file contains all the image manipulation functions.
For the alternate greyscale, we subtracted the red channel from 255 to be able to attain the desired output. Unlike the conventional greyscale conversion formulas, which consider the weighted sum of all color channels, the approach in the project only concentrates on the red channel. The red channel value is subtracted from 255. This operation inverts the values in the red channel. The resulting greyscale intensity will be low (close to 0) while the red channel value is high (closer to 255), and vice versa.

Once the greyscale intensity is calculated, it is assigned to all three color channels of the output image. This produces a greyscale effect by ensuring that the greyscale image maintains the same intensity value over all color channels. The final image will be a unique greyscale representation of the input image with each pixel's intensity inversely proportionate to its red channel value once all pixels have been processed. In the greyscale image, areas with high red intensity will seem darker, and areas with low red intensity will appear lighter.
The difference between the default greyscale and the custom greyscale is that in the default greyscale image the original image's luminance information is retained, while discarding its color information. This method generates images in greyscale, in which various shades of grey stand in for different colors.

However, in the custom greyscale image, color information other than red is completely ignored. The intensity of the red channel in the source image will be the only factor influencing the greyscale image's shades of grey.

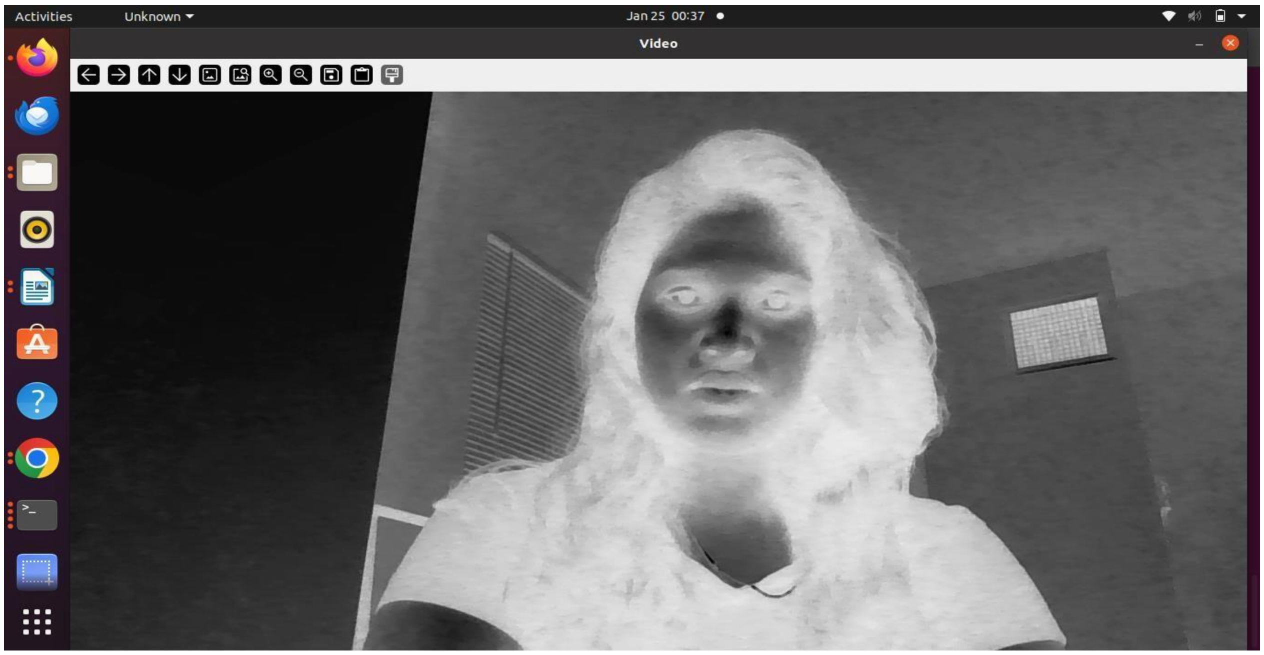


Fig 6: Alternative Greyscale Filter on Video Stream

TASK – 5:

This task requires us to implement a sepia filter on the video stream, with the new blue value should be 0.272*R + 0.534*G + 0.131*B, where R, G, and B are the original color values. The video stream turns into the sepia effect on pressing the “p” key.
The program makes use of the “at<cv::Vec3b>(y, x)” function to retrieve the RGB values for each pixel. The blue, green, and red channel intensities are represented, respectively, by a vector of three “uchar” values that this function returns.

A linear combination of the original RGB values for each channel (red, green, and blue) is calculated to determine the sepia effect. The original red (pixel[2]), green (pixel[1]), and blue (pixel[0]) values for the red channel in the destination image (sepiaR) are multiplied by coefficients (0.272f, 0.534f, and 0.131f, respectively), and then added together. The green and blue channels' sepia levels are also determined in the same way.

Each channel's sepia values are computed and then allowed to be in the range of [0, 255]. Any value that is greater than 255 is decreased to 255. “at<cv::Vec3b>(y, x)” is used to assign the computed sepia values for each channel to the appropriate channels of the destination picture (dst).

By doing this, the sepia function makes sure that each pixel in the source image's original RGB value is used in the computation of the sepia effect, preserving the original colors in the final image while adding the appropriate sepia tone.

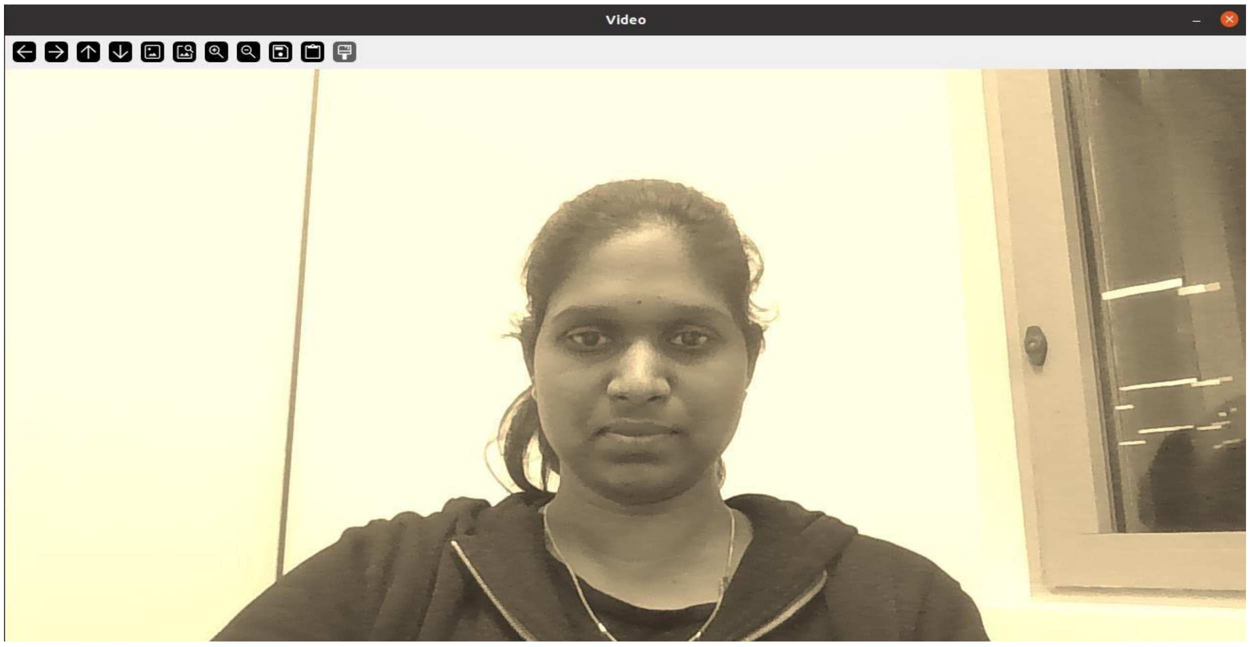


Fig 7: Sepia tone filter on video stream

EXTENSION: Vignetting Filter

As an extension we have added the vignetting filter to the video stream. In The vignetting effect the corners or edges of an image appear darker than the center, creating a subtle darkening effect towards the edges. This effect is obtained in the “sepiaWithVignetting” function by determining the vignette factor for every pixel depending on how far it is from the image's center. The middle point of the image's dimensions is usually chosen as the center. The Euclidean distance formula is used to calculate each pixel's distance from the center. The vignette factor is then calculated using this distance; it inversely correlates with the distance, producing more vignetting in the direction of the edges and less vignetting in the direction of the center. The sepia tones are then calculated using the original pixel values modified by the vignette factor. Finally, the sepia values are reduced to ensure they fall within the valid range of [0, 255], and the resulting image shows the vignetting effect.

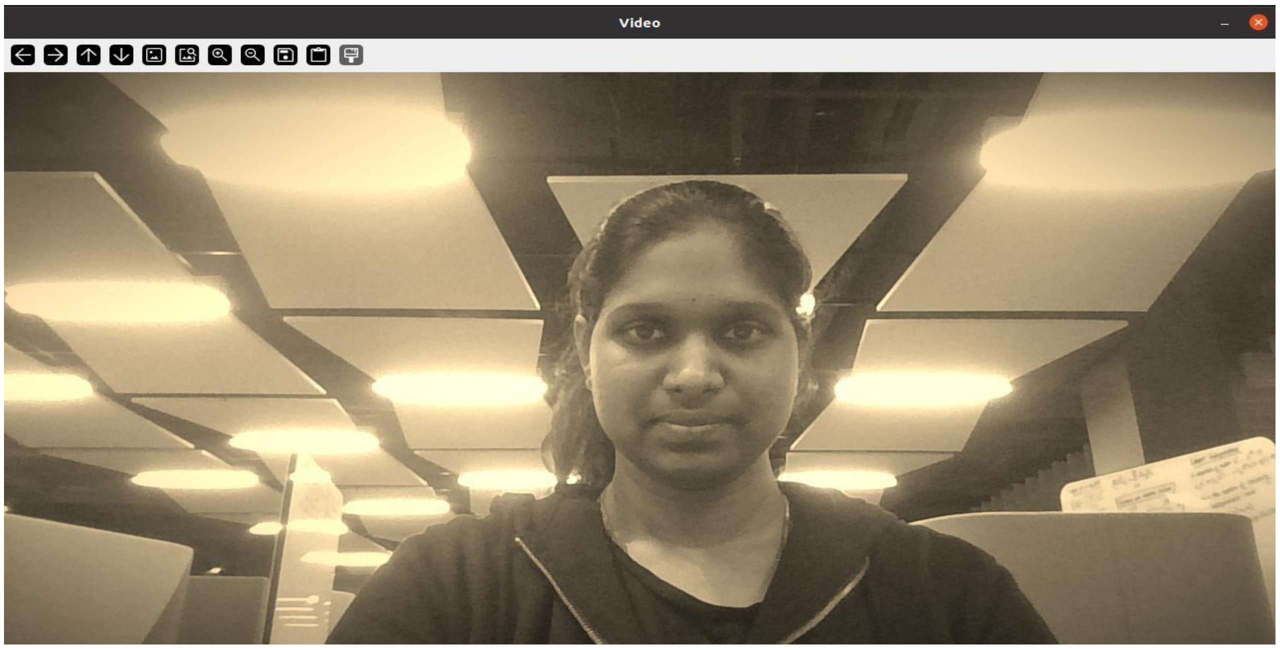


Fig 8: Vignette with Sepia tone filter on video stream

TASK – 6 (a):

This task requires us to implement a 5x5 blur filter from scratch and time the implementation.

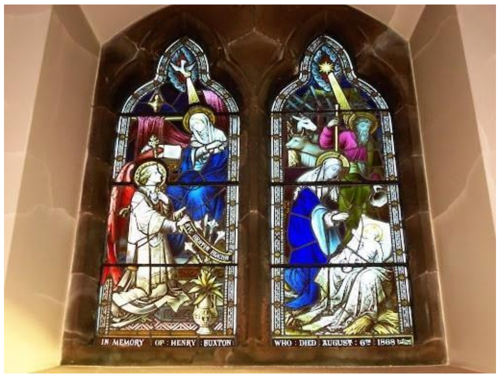


Fig 9: Original Cathedral Image

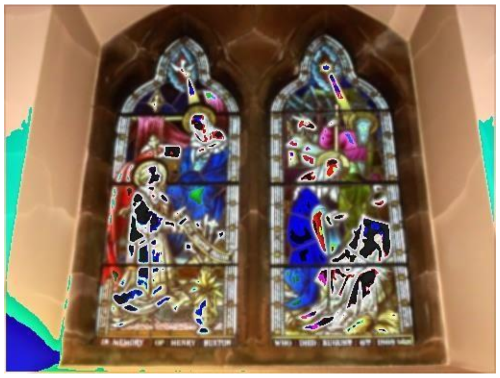


Fig 10: First Implementation of Blur 5x5

TASK – 6 (b):

This task requires us to perform a second implementation of a 5x5 blur filter from scratch but make it faster than the first implementation. The video stream must change into a blurred video stream, when the "b" key is pressed.

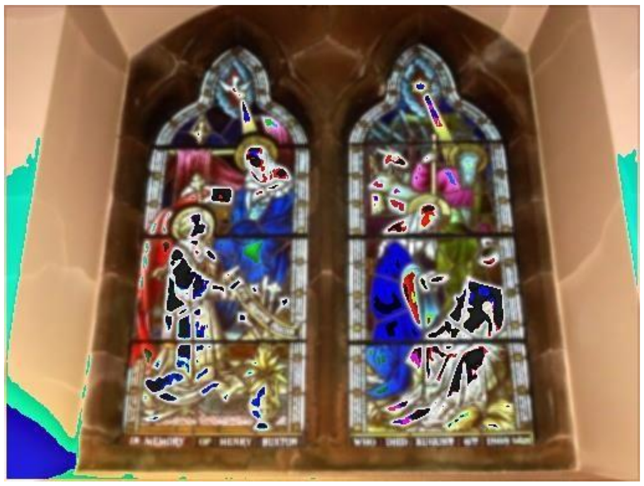
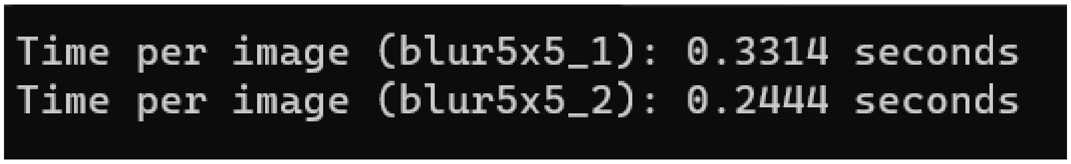


Fig 11: Second Implementation of Blur 5x5



```
Time per image (blur5x5_1): 0.3314 seconds
Time per image (blur5x5_2): 0.2444 seconds
```

Fig 12: Time Stamp for Blur Implementation

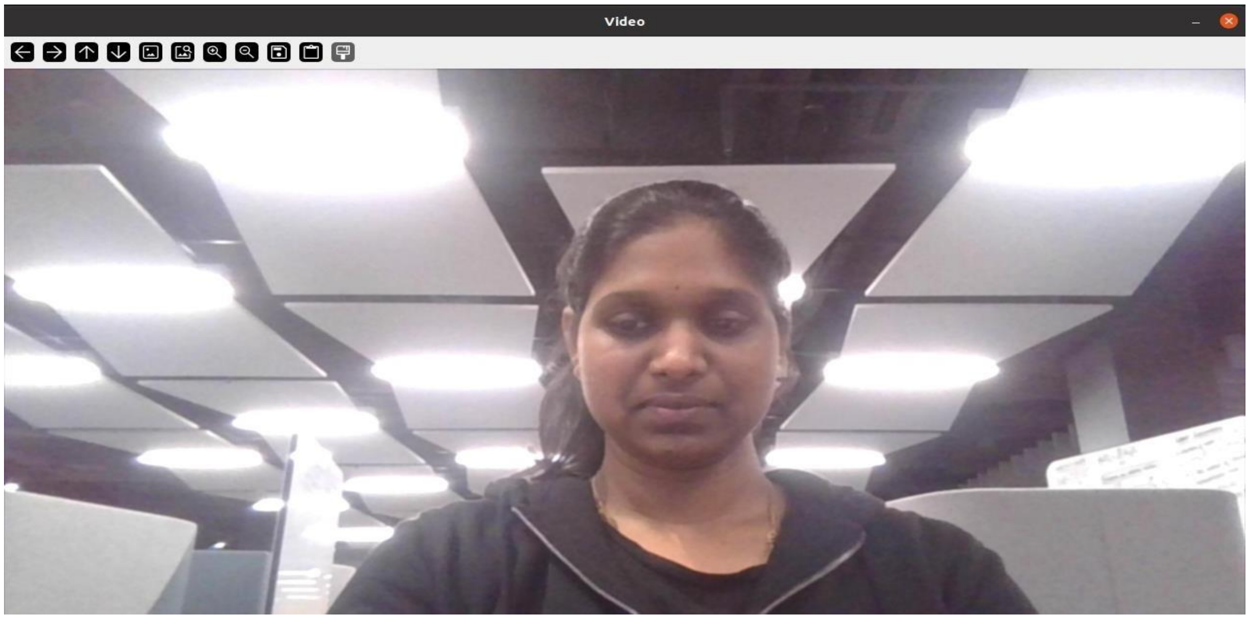


Fig 13: First Implementation of Blur Filter on Video Stream

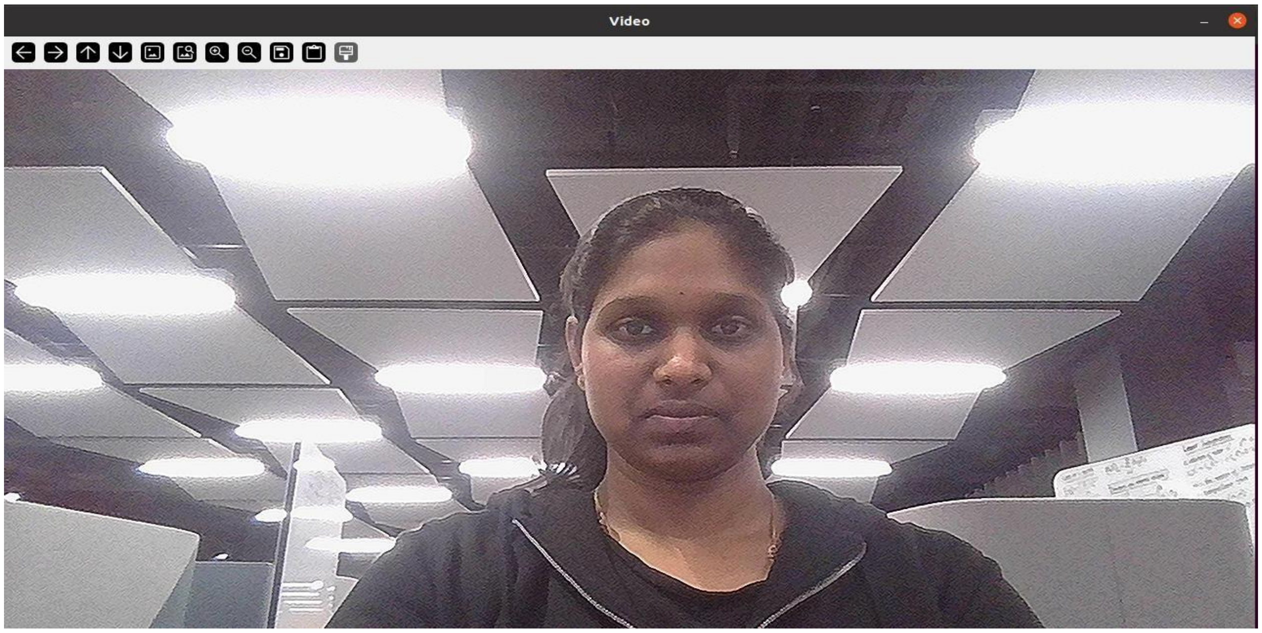


Fig 14: Second Implementation of Blur Filter on Video Stream

Blur5x5_1 directly convolves a 5x5 kernel, iterating over each pixel with nested loops and carrying out operations per channel. This simple method requires a lot of calculation.
Blur_5x5_2 applies two 1x5 kernels in a vertical and horizontal sequence. Computes per pixel are decreased as a result. The normalization factor of the 5x5 kernel in the original blur5x5 implementation is 88. In the second blur_5x5 version, each of the separable kernels has a normalization of 10.
If the second method is quicker, it is because the separable approach's computations have been reduced into fewer operations. Variations in performance could be the result of hardware optimizations, caching, memory access, etc. The initial blur5x5 implementation's direct convolution has a greater processing load, which could affect time.

TASK – 7:

This task required us to apply a “3x3 Sobel X” and “3x3 Sobel Y” as “separable 1x3” filters to the video stream using manual functions and not the OpenCV functions. The Sobel X needs to be applied to the video stream on pressing the “x” key, and the “Sobel Y” filter needs to be applied to the video stream on pressing the “y” key.

The Sobel filter is applied individually in the horizontal and vertical axes, using 3x3 kernels. These kernels are matrices that are convolved with the image to provide the horizontal and vertical gradients.
The program employs a 3x3 kernel for each direction as the Sobel operator; the surrounding pixels have positive and negative weights, while the central pixel has a weight of 0. These weights are made to reduce noise and moderate intensity variations while emphasizing changes in intensity along the designated direction.
The Sobel filter computes the picture's gradients, both vertically and horizontally, by convolving the image with these Sobel kernels. For edge recognition and image analysis applications, these gradient images provide important information by emphasizing edges and transitions in the image.

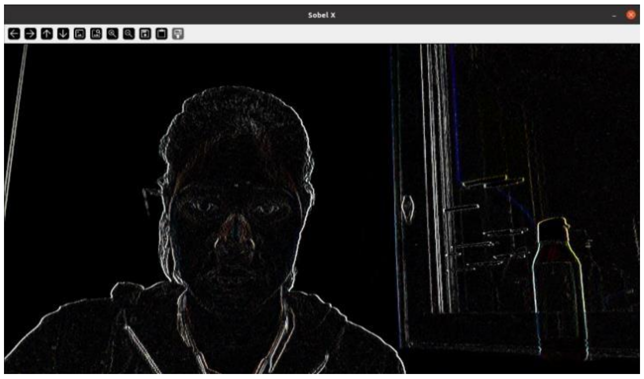


Fig 15: Sobel X filter on Video Stream

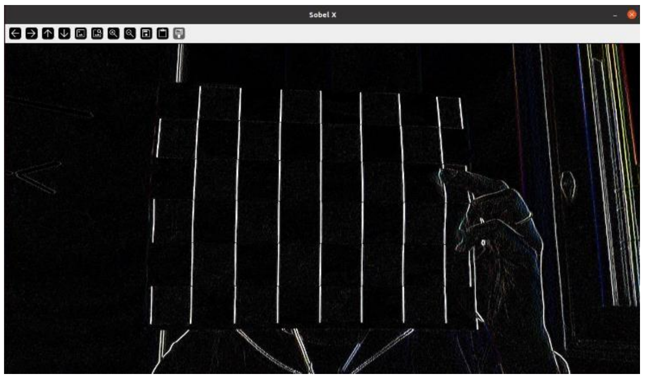
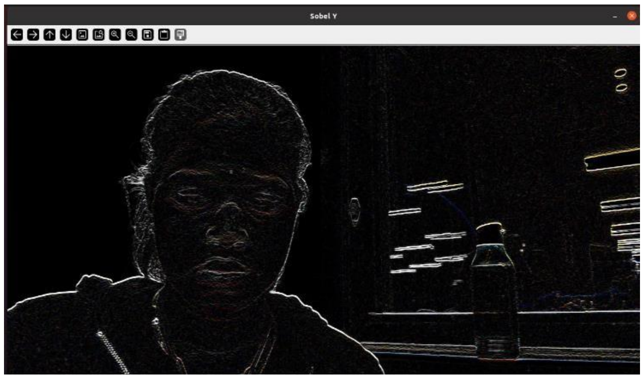


Fig16: Sobel X filter with checkerboard



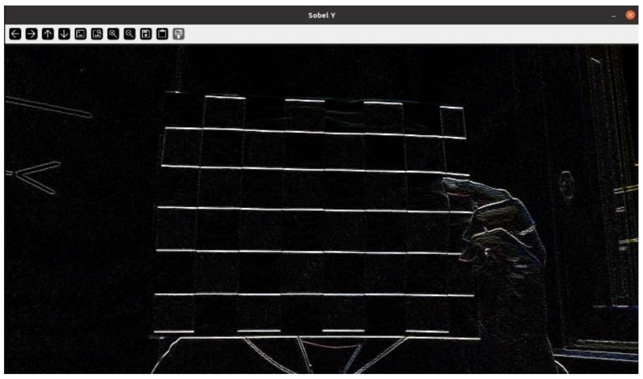


Fig 17: Sobel Y filter on Video Stream ***Fig 18: Sobel Y filter with checkerboard***

TASK – 8:

This task requires us to implement a function that generates a gradient magnitude image from the X and Y Sobel images and incorporate it in the vidDisplay.cpp file so that the gradient magnitude of the video stream is generated on pressing the “m” key.
The implementation is done by determining the magnitude for every channel independently by taking the square root of the total squared gradients in that channel. The magnitude of each channel is then separately saturated to fit into the “uchar” range. This makes it possible for each channel to have a different value depending on the gradient magnitudes in that color channel.

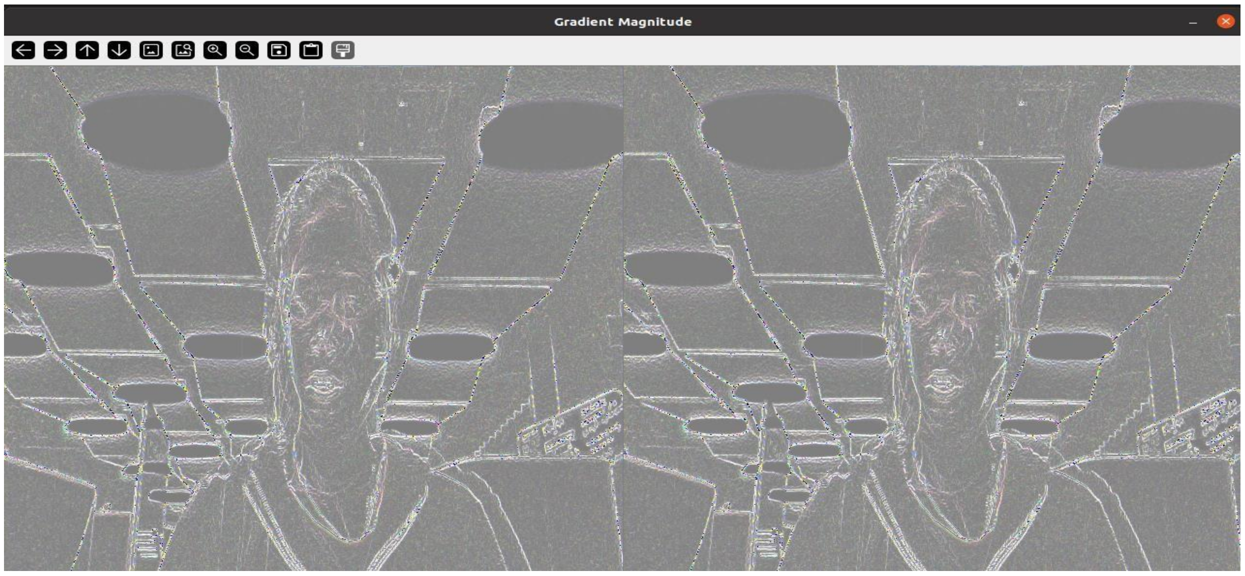


Fig 19: Gradient Magnitude on Video Stream

TASK – 9:

This task requires us to implement a function that blurs and quantizes a color image in filter.cpp. The displayed video stream should be quantized on pressing the “l” key. First, the image is blurred, which helps to remove noise and details from the picture. The intensity range of each channel is then divided into "buckets" according to the chosen number of levels, which quantizes the image. The original intensity value is then converted to the closest bucket for each pixel and channel, rounding it to the closest quantized value. As a result, the image's total number of unique intensity levels decreases. An image with a reduced intensity range is created by assigning the quantized intensity values to the appropriate channels of the destination Image.

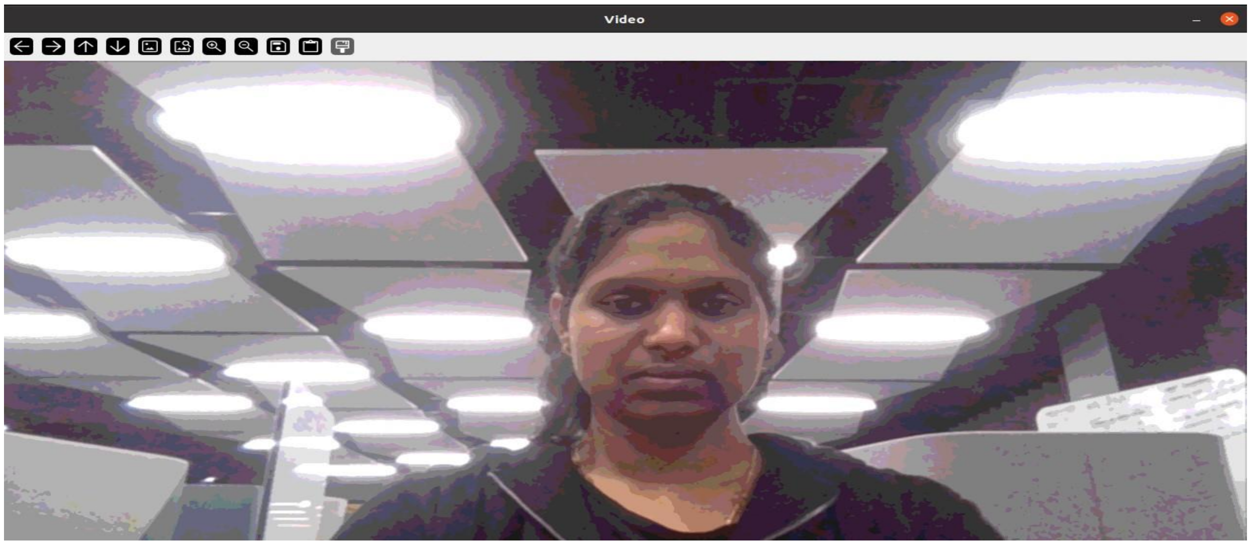


Fig 20: Blur And Quantize Video Stream

TASK – 10:

This task requires us to detect faces in an image. We need to link the faceDetect.cpp file to the vidDisplay.cpp. The face detection must activate when the user presses the 'f' key.

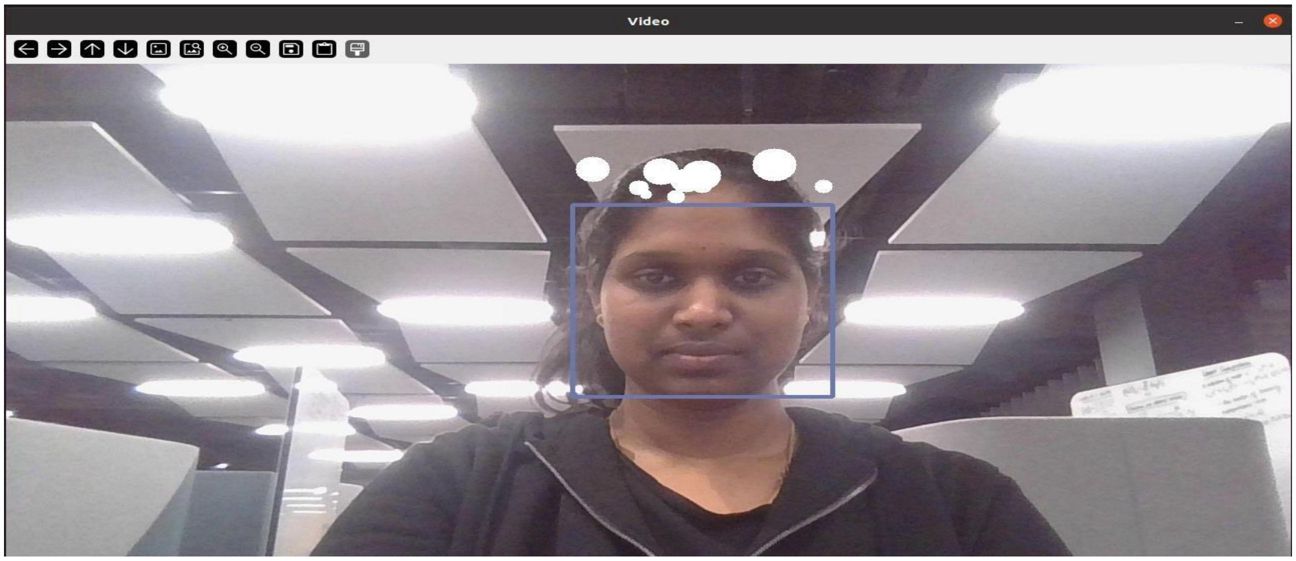


Fig 21: Face Recognition on Video Stream

TASK – 11:

This task requires us to implement any three filters of our choice. The three filters we implemented on the video stream are:

1. **Embossing Filter:**
   It begins by computing horizontal and vertical gradients using Sobel operators, which highlight edges and transitions in the image. To maximize the visual effect, a direction vector is usually set at an angle to indicate the intended embossing direction. The horizontal and vertical gradient images are then multiplied elementwise by the components of the direction vector, resulting in the dot product of the gradient vectors in the chosen direction. Gradients aligned with the direction vector are emphasized while gradients orthogonal to it get reduced in this process. After that, the embossing image is adjusted to make sure that all the values are positive before it is normalized to fall between 0 and 255. The embossing effect adds depth and texture to the image.

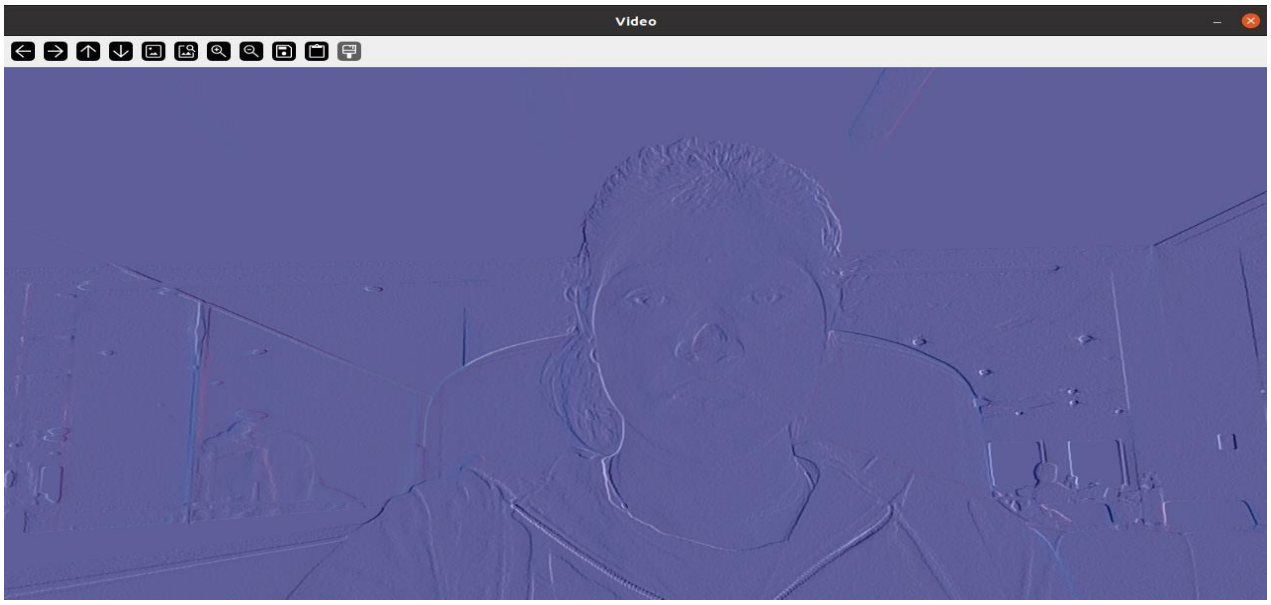


Fig 22: Embossing Filter on Video Stream

2. **Halo Filter:**
   Within the “drawBoxes” function, after drawing rectangles around the detected faces, the code proceeds to generate sparkling halos above each face. This is achieved by randomly placing a predefined number of sparkles, each represented as filled circles, above the face region. The radius of each sparkle is randomly selected within a specified range, ensuring variability in size. To ensure that the sparkles are positioned above the face, their coordinates are calculated relative to the top boundary of the face rectangle.

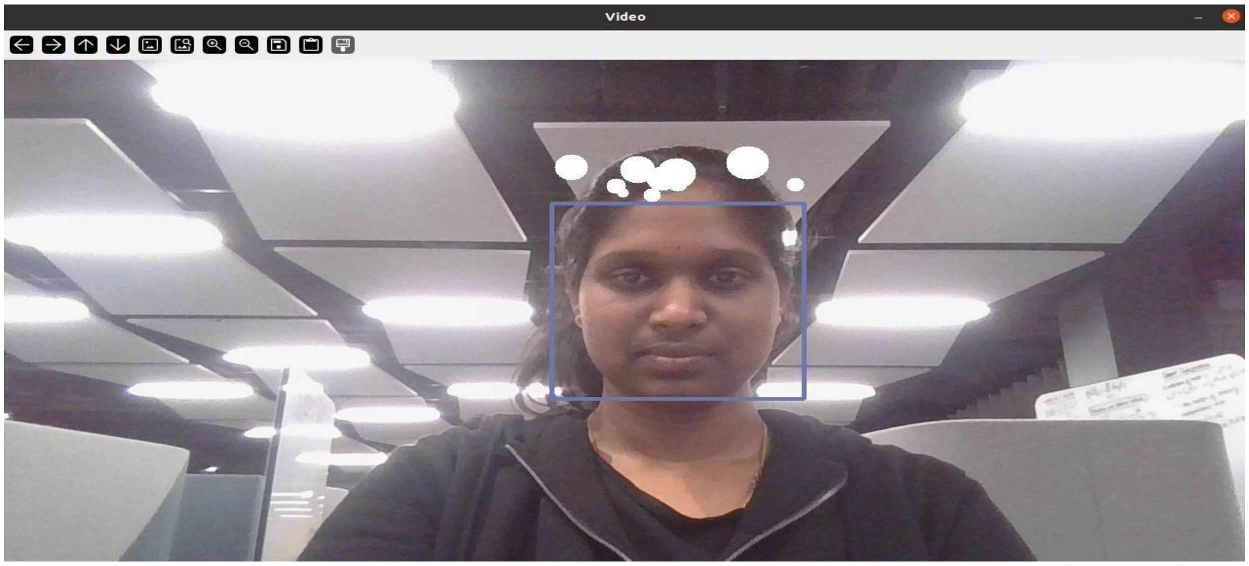


Fig 23: Halo Filter on Video Stream

3. **Adjust Brightness and Contrast:**
   UpdateBrightnessContrast() is a function in the program that adjusts brightness and contrast and makes it easier to precisely manipulate the level of contrast and brightness inside an image frame. This function dynamically modifies the intensity distribution of pixel values across the frame by receiving brightness and contrast parameters. It uses a linear transformation, scaling each pixel's value by the given contrast factor and altering its intensity based on the brightness factor, by utilizing OpenCV's convertTo() function. Through the effective redistribution of pixel intensities, this technique modifies the image's overall brightness and contrast. As a result, users can adjust the frame's visual appearance to highlight specifics, increase visibility, or produce desired aesthetic effects.

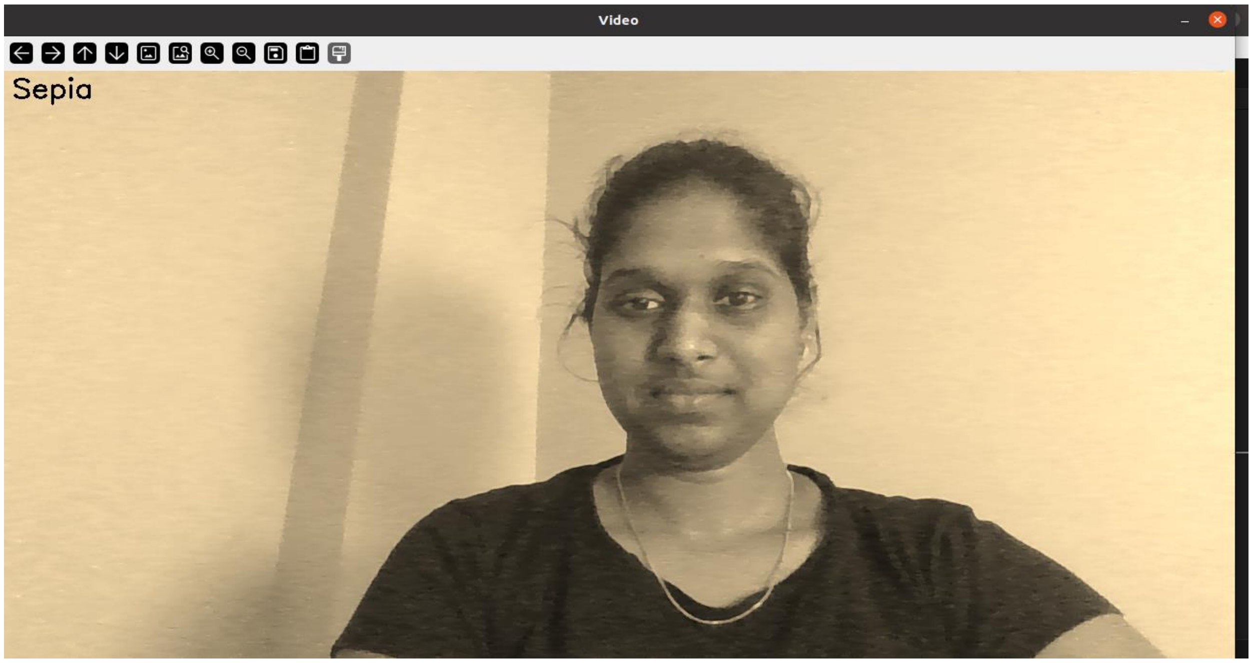


Fig 24: Brightness Before Image

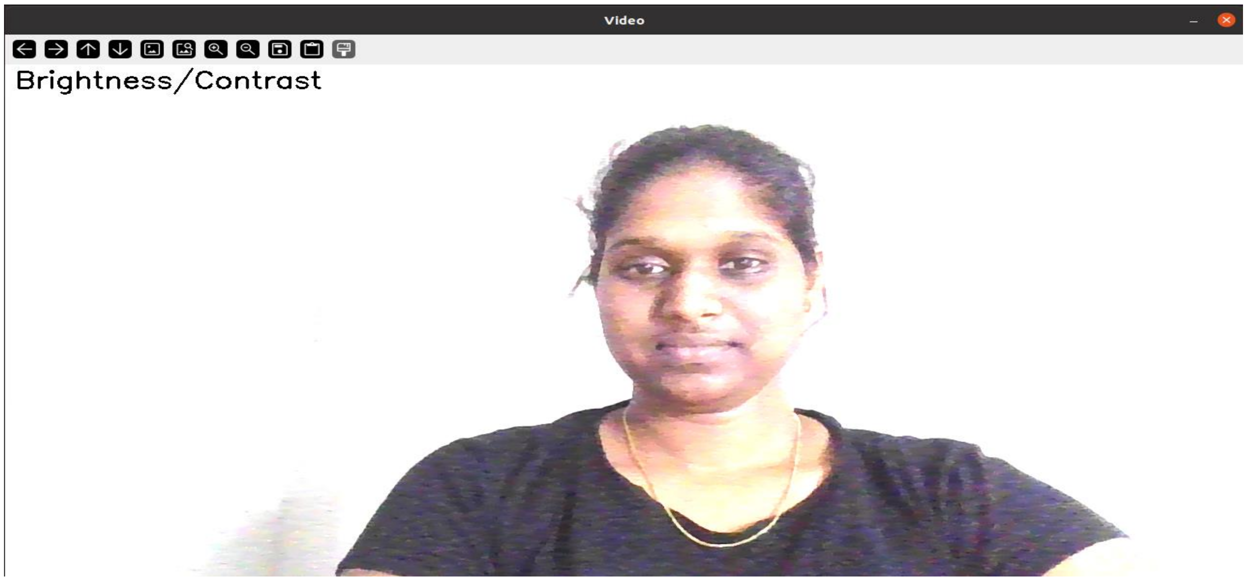


Fig 24: Brightness After Image

EXTENSION:

1. **Captions for Images:**
   The program enables the users to dynamically add captions to video frames and subsequently save them with the added annotations. Upon triggering the 's' key, the program, if in caption mode, prompts users to input a caption through the console. The “j” keypress acts as a toggle for the “addCaptionMode”. When the flag is set, it means the user wants to add a caption. Following this, users are prompted to specify coordinates for positioning the caption within the frame. Using OpenCV's drawing functions, the program joins the caption onto the frame with a filled rectangle background. Upon completion, the frame, now containing the captions, is saved as an image file named "caption_image.jpg" using OpenCV's **`cv::imwrite()`** function. The management of the **`addCaptionMode`** function ensures the conditional prompting of users for captions when saving the frame.

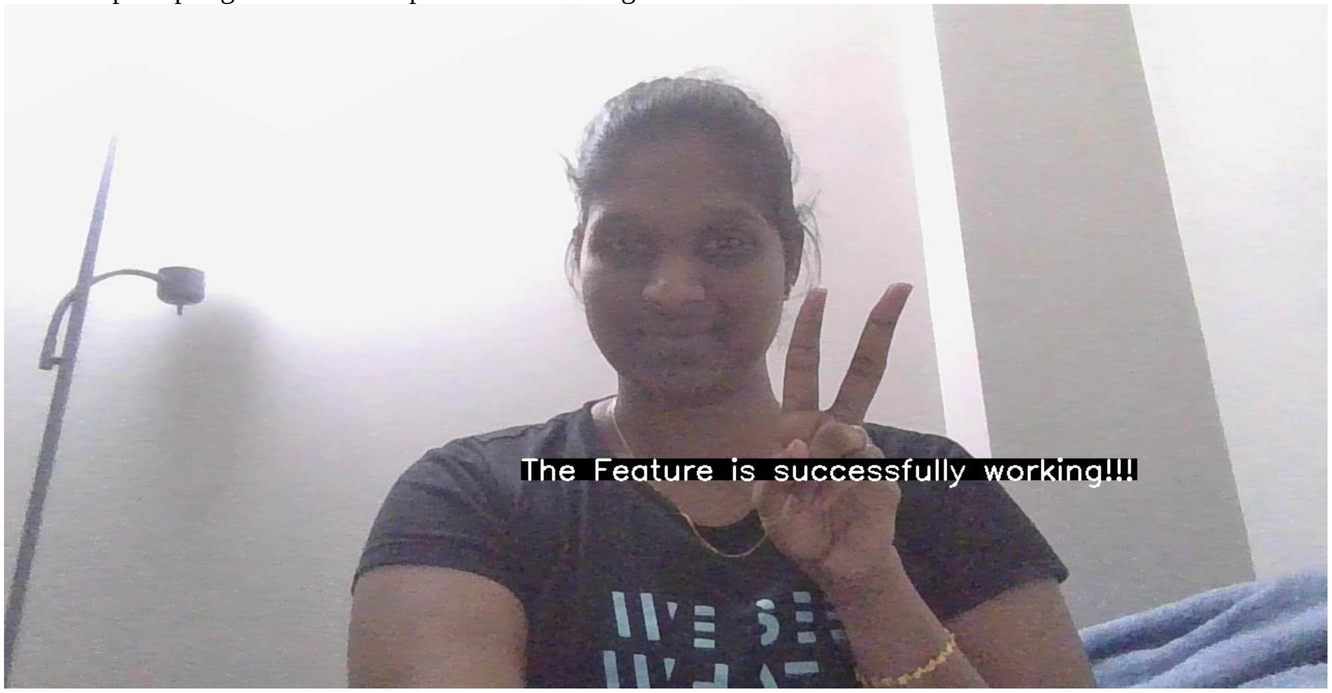


Fig 26: Caption Image Function

REFLECTION ON WHAT WE LEARNT:

1. This project provided firsthand experience, allowing us to learn by implementing various tasks and experimenting with different techniques. This practical approach deepened my understanding of concepts and techniques in computer vision.
2. The project introduced us to the OpenCV library, solving the tasks aided in enabling a deeper understanding of its functionalities and enhancing proficiency in leveraging its capabilities for video processing and image manipulation.
3. Working on tasks that involve pixel-level manipulation gave a better understanding of the intricacies of image processing algorithms. We understood how crucial it is for individual pixels to contribute to the overall transformation of an image.
4. Through implementing various filters and effects from scratch, such as the blur filter and the embossing filter, we could better understand the underlying algorithms and techniques used in image processing.
5. The project's implementation of a 5x5 blur filter introduced us to the concept of optimization techniques. Exploring both a simple approach and a faster version using separable 1x5 filters highlighted the significance of optimizing algorithms for real-time video processing.
6. The incorporation of various image filters, including greyscale transformations, Sobel filters, sepia tone filters, and blur filters, gave us a better understanding of their functionalities and applications in computer vision.

ACKNOWLEDGEMENTS:

I am grateful to the OpenCV documentation, which has been a vital resource during this project. Understanding and putting different computer vision algorithms and techniques into practice has been made possible by the comprehensive explanations given in the OpenCV documentation.

I would also like to thank my peers for their help and feedback during this project. Throughout this process, exchanging knowledge, working together to troubleshoot challenges, and sharing ideas have all been extremely beneficial.