PROJECT 1: VIDEO-SPECIAL EFFECTS

This project is about learning and applying real-time image and video processing using C++ and OpenCV, focusing on performance, hands-on coding, and creativity. The goal was to start with basic task and build up to more advanced techniques while experimenting with custom effects and deep learning models.

The first task of the project was to implement simple tasks like displaying an image and saving it as a response to a specific key press. The project also involved working with live video stream and responding to key presses to implement filters like grayscale, sepia tone, and custom 5x5 blur. The filters were coded from scratch to better understand how image processing works. Filters for edge detection like Sobel filters were implemented as well.

The project also had a task to implement a Depth Analysis V2 deep learning model with ONNX Runtime to estimate depth from video frames. Another feature of the project was Face Detection, where a rectangular box is formed around the face when it is detected.

In the last task, three custom effects were developed i.e., Colorful Face, Blur Outside the Face and Embossing effect from the concepts learned. The main aim of the project was to understand OpenCV, and apply real-time computer vision techniques effectively.

Task 1: Read an image from a file and display it.

The program reads an image from a file and displays it in a window using OpenCV. The user can interact with the program by pressing keys: ‘q’ to quit and ‘s’ to save the image. The program ensures smooth execution by checking if the image loads successfully.

Task 2: Display live video

In this task, the program vidDisplay.cpp opens a video channel to capture a live video and display it in a window. It also allows the user to interact through keypresses. Pressing ‘q’ quits the program and ‘s’ saves the current frame as an image file. The program continuously captures and displays video frames in a loop, ensuring smooth real-time video playback.

Task 3: Display greyscale live video

In this task, the user is able to toggle between color and greyscale video streams. When the user presses ‘g’, the filtered frame is displayed in greyscale.

The greyscale conversion is done using the inbuilt OpenCV function i.e., cv::cvtColor with cv::COLOR_BGR2GRAY flag. This function applies a weighted formula to the RGB channels to calculate brightness.

Gray = 0.299 * R + 0.587 * G + 0.114 * B

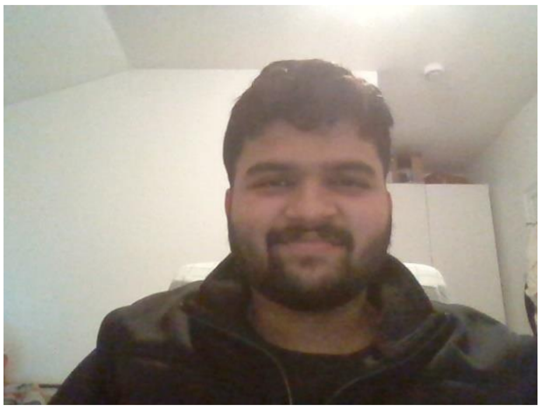


Fig 1: Original Frame

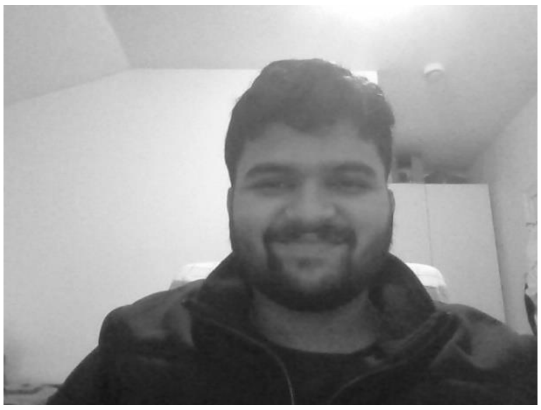


Fig 2: Grayscale Frame

The Fig 1 is the frame with the default color before conversion and Fig 2 is the converted greyscale image.

Task 4: **Display alternative greyscale live video**

In this task, the user can toggle to an alternate custom greyscale version of the video when the user presses ‘h’. The custom greyscale filter is implemented in a new file filter.cpp.

The custom greyscale conversion is done by applying a unique transformation to each pixel. Instead of using the weighted RGB formula, you can subtract from 255 and assign the result as a new value. The transformation is applied pixel by pixel using OpenCV’s cv::Mat structure.

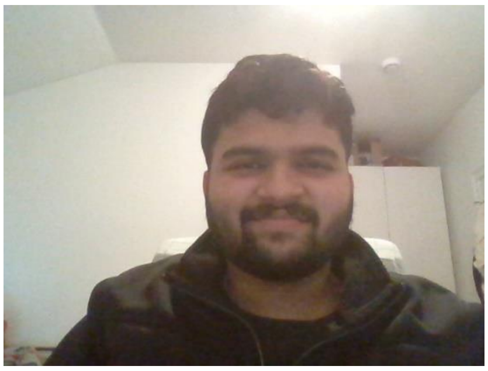


Fig 3: Original Frame

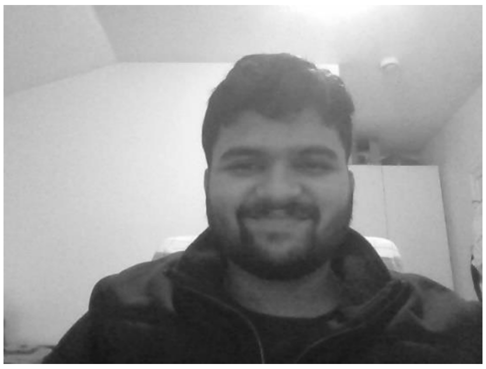


Fig 4: OpenCV Grayscale Frame

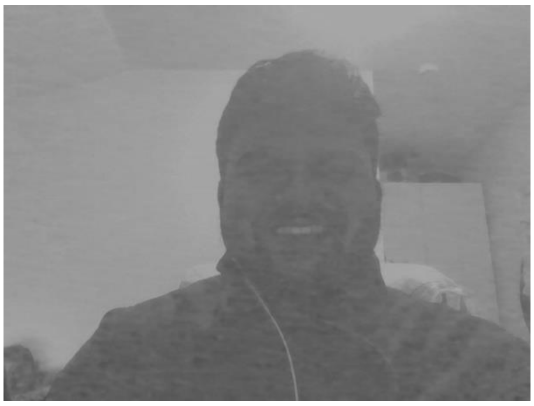


Fig 5: Custom Grayscale Frame

Fig 3 is the original frame without greyscale transformation. Fig4 and Fig 5 are the OpenCV function conversion and custom greyscale conversion respectively. The OpenCV function looks more realistic because it uses weights for colors that match how our eyes see brightness. The custom greyscale, like subtracting the red channel from 255, looks more like a negative image because it inverts certain colors and doesn't follow natural brightness levels.

Task 5: Implement a Sepia tone filter

A Sepia tone filter gives images a warm and antique look by modifying each pixel's RGB values using specific coefficients. The filter calculates new red, green and the blue values as weighted combinations of the original RGB channels as old values are used for calculations. It also ensures the output values stay within the range of 0-255.

The new RBG values are calculated using weighted combinations of original RGB values with specific coefficients:

Blue = 0.272*R + 0.535*G + 0.131*B

Green = 0.349*R + 0.686*G + 0.168*B

Red = 0.393*R + 0.769*G + 0.189*B

The values are maintained within the range of 0-255 using the std::min(255, newVal) function.

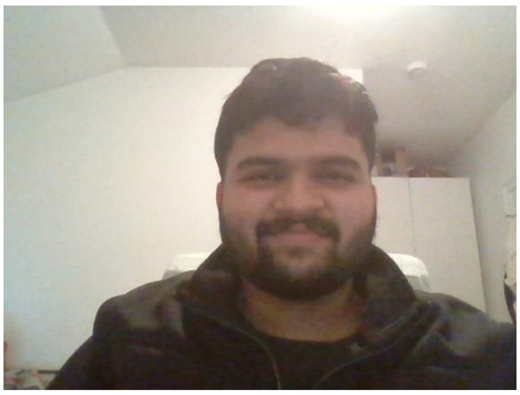


Fig 6: Original Frame

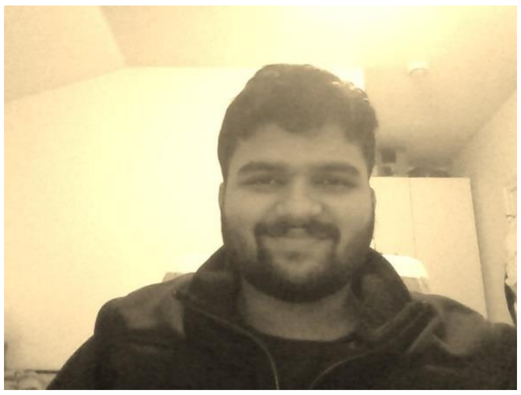


Fig 7: Sepia Tone Frame

Fig 6 is the original frame while Fig 7 is the Sepia tone-filtered frame.

Task 6: Implement a 5x5 blur filter

This task involves creating a custom 5x5 Gaussian blur filter for an image. The first implementation blur5x5_1 uses a straightforward nested loop to apply the filter to each pixel. The second implementation blur5x5_2 optimizes the process by separating the 5x5 filter into two 1x5 filters (horizontal and vertical passes) and avoiding the costly 'at' method for better performance. The function is then integrated into the program to blur the live video stream when the user presses 'b'.

Using ‘ptr’ in OpenCV is faster than ‘at’ because it directly accesses the image in memory, avoiding the overhead of bounds checking and type validation that ‘at’ performs.

Fig 8: Original Frame

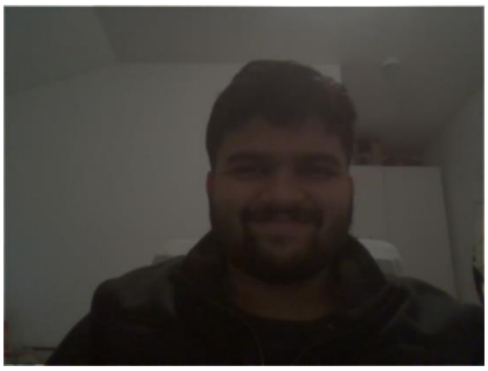
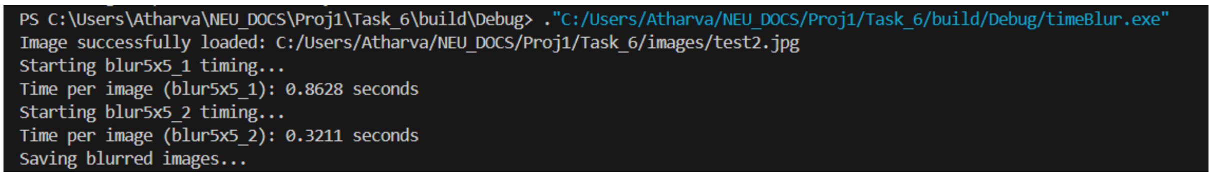


Fig 9: 5x5_2 Blur Frame



```
PS C:\Users\Atharva\NEU_DOCS\Proj1\Task_6\build\Debug> ."C:/Users/Atharva/NEU_DOCS/Proj1/Task_6/build/Debug/timeBlur.exe"
Image successfully loaded: C:/Users/Atharva/NEU_DOCS/Proj1/Task_6/images/test2.jpg
Starting blur5x5_1 timing...
Time per image (blur5x5_1): 0.8628 seconds
Starting blur5x5_2 timing...
Time per image (blur5x5_2): 0.3211 seconds
Saving blurred images...
```

Fig 10: Timing Blur

Fig 8 is the original color image. Fig 9 is the implementation of blur5x5_2 and Fig 10 is the time taken by both functions to implement blur filter. The time taken by blur5x5_1 is 0.8628 seconds and the time taken by 0.3211 seconds.

Task 7: Implement a 3x3 Sobel X and 3x3 Sobel Y filter as separable 1x3 filters

The Sobel filters detect edges in an image by calculating intensity gradients. Sobel X detects the vertical edges while Sobel Y detects the horizontal edges. Both outputs are signed short values because the values can be in the range of [-255,255].

The filter uses a 3x3 kernel for convolution. The kernel used to detect horizontal edges(Sobel Y) is [-1,-2,-1; 0,0,0; 1,2,1]. The kernel used for vertical edge detection (Sobel X) is [-1,0,1; -2,0,2; -1,0,1]. The edges are highlighted where intensity changes sharply. The output is converted to absolute values for visualization as ‘imshow’ only displays unsigned char images.

Fig 11: Original Frame

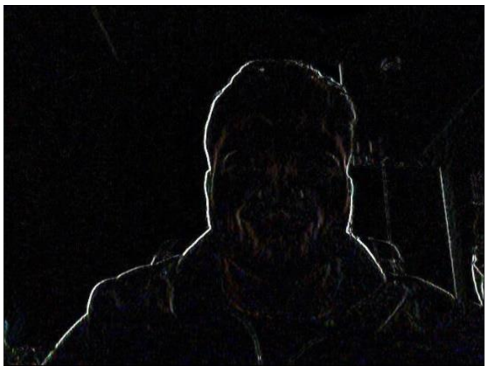


Fig 12: Sobel X Frame

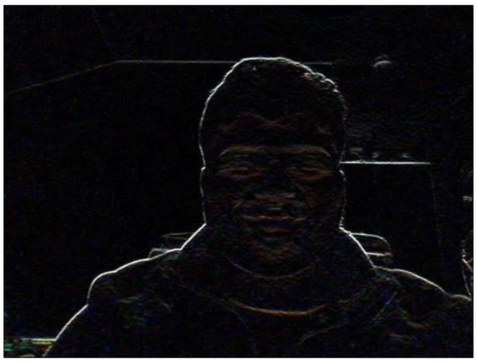


Fig 13: Sobel Y Frame

Fig 11 is the original color image. Fig 12 is the implementation of Sobel X and Fig 13 is the implementation of Sobel Y.

Task 8: Implement a function that generates a gradient magnitude image from the X and Y Sobel images

The magnitude function computes the gradient magnitude from the X and Y Sobel images using Euclidean distance formula. The output is a color image, where the intensity of each pixel reflects the edge strength.

Euclidean Distance = $\sqrt{(sx)^2 + (sy)^2}$

Fig 14: Original Frame

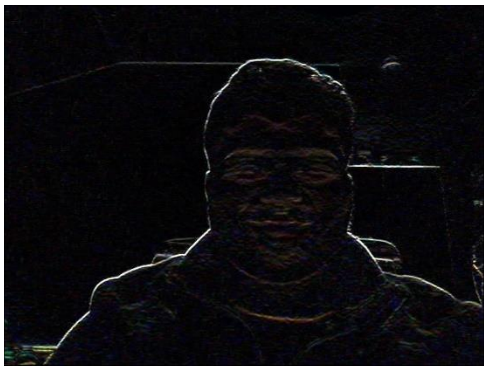


Fig 15: Gradient Magnitude Frame

Fig 14 is the original image while Fig 15 is the gradient magnitude image.

Task 9: Implement a function that blurs and quantizes a color image

The blur quantized function works by blurring the image to smooth out noise, creating a more uniform base for optimization. It then reduces the color range of each pixel by dividing the color value of each channel into discrete levels using a bucket size i.e., b = 255/levels. Each color is then scaled to the nearest bucket multiple which will group the similar colors together and reduce the total number of unique colors. The levels here is set to 10. Therefore the number of uniqe colors is $(\text{levels})^3$ i.e., $10^3 = 1000$.

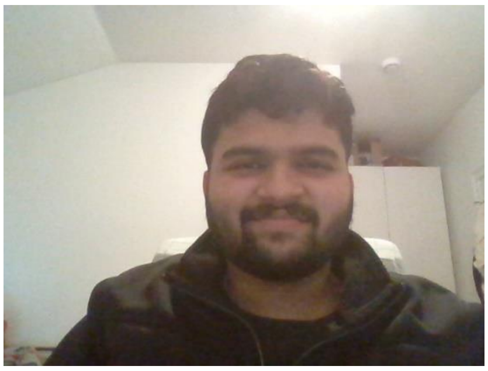


Fig 16: Original Frame

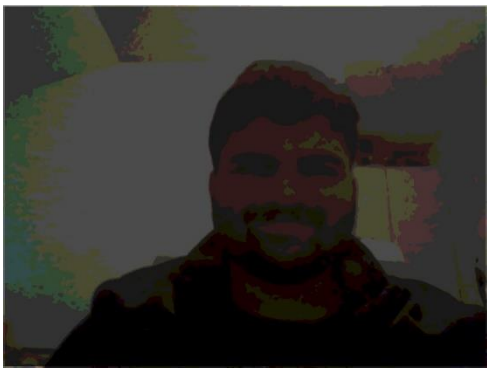


Fig 17: Blur Quantized Frame

Fig 16 is the original image while Fig 17 is the blur quantized image.

Task 10: Detect faces in an image

This task integrates face detection into the video stream program using the provided ‘faceDetect.cpp’, ‘faceDetect.h’, and a haarcascade XML file. When the user presses ‘f’, the program detects faces in the video feed and highlights them with bounding boxes.

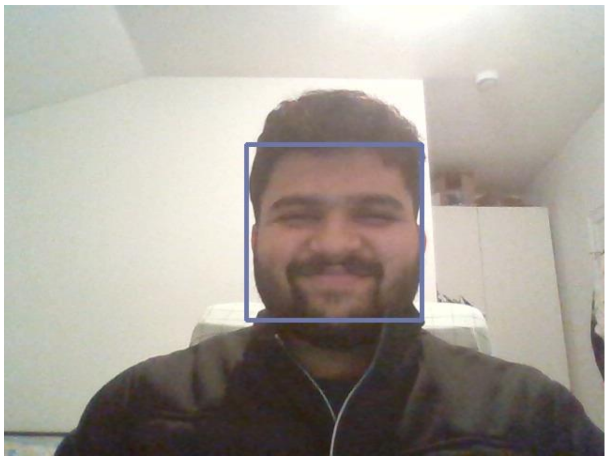


Fig 18: Face Detection Frame

Fig 18 shows bounding boxes around detected faces.

Task 11: Use the Depth Anything V2 network to get depth estimates on your video feed

In this task, the Depth Anything V2(DA2) network was used to estimate depth in a video stream. Provided example initially scaled the video stream to 0.5x, which caused noticeable lag during depth video display. When the scale was reduced there was an increase in the lag. The scaling factor was removed to process the video at its original resolution and reduced lag. The Depth Anything V2 network provides both high quality and reasonably fast depth estimates from a single image.

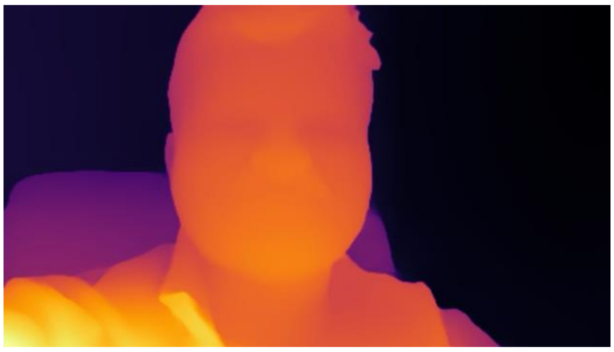


Fig 19: Depth Image from Video Stream

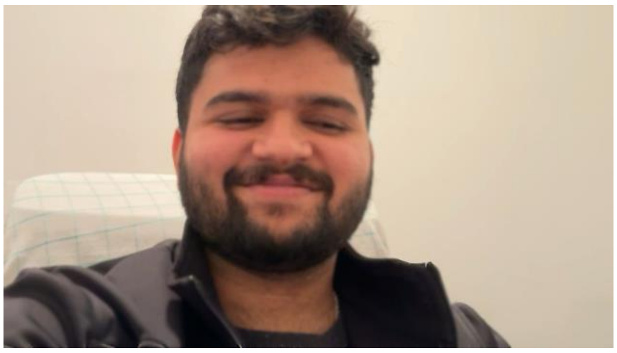


Fig 20: Filter using Depth Values

Task 12: Implement three more effects on your video

1. **Emboss Effect:** This function applies an embossing effect to an image using the Sobel X and Y gradients. It combines the horizontal and vertical gradients using dot product in a normalized direction (0.7071, 0.7071) to create a raised or embossed appearance. The gradient values are adjusted by adding 128 to bring the output into 0-255 range for visualization.

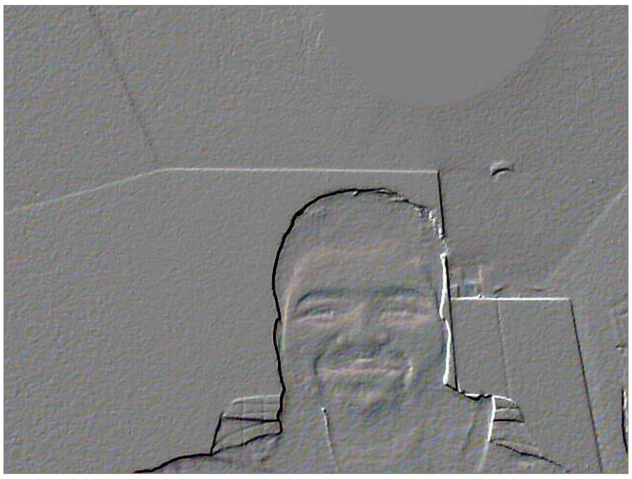


Fig 20: Emboss Effect Frame

2. Colorful Face with Grayscale Background

This function creates an effect where detected faces in an image remain colorful, while the rest of the image is converted to grayscale. The function uses Haarcascade classifier to detect faces and converts the entire image to grayscale. Then, it copies the original color face region back into the grayscale image, creating a contrast between colorful faces and the grayscale background.

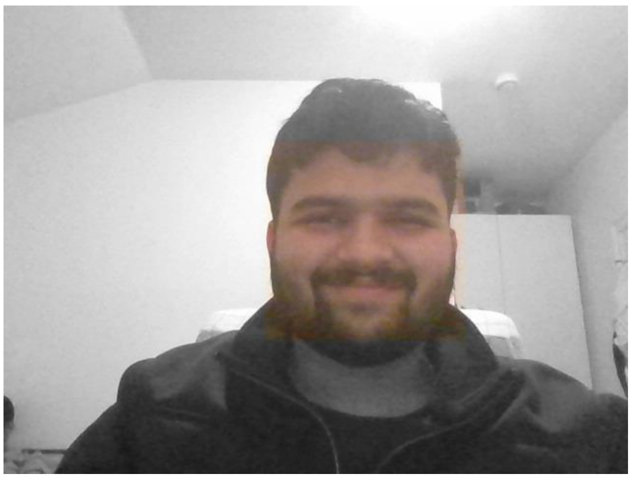


Fig 21: Colorful Face with Grayscale Background Frame

3. Blur Outside Face

This function blurs everything in an image except for detected face regions. It uses Haarcascade classifier to detect faces and create a blurred version of the entire image using the custom blur function discussed previously. The detected face regions are then copied from the original image onto the blurred image, leaving the faces sharp and clear while the rest of the image is blurred.

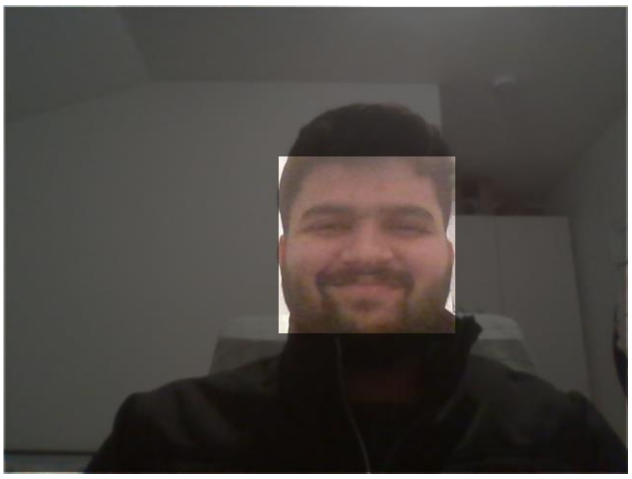


Fig 22: Blur Outside Face Frame

Extensions:

1. Add Captions:

The Add Captions function overlays text captions on the top and bottom of an image, making it ideal for creating memes. It uses OpenCV's 'cv::getTextSize' to calculate the dimensions of the text, ensuring that the captions are properly centered. The text is inserted using 'cv::putText' function. The color of the text is kept white and anti aliasing is applied using 'cv::LINE_AA' to get smooth edges.

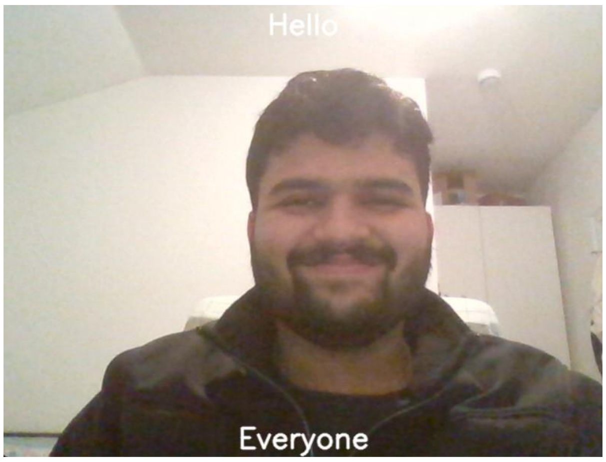


Fig 23: Add Caption (Meme) Frame

2. Apply Frame Mask

The Apply Frame Mask function adds an image as a mask on top of the live video stream where the central part of the video frame remains unmasked making it look like a photo frame. The mask is resized to match the dimensions of the frame using 'cv::resize'. A visible region is defined, and the corresponding section of the original frame is copied to ensure the region remains unchanged. OpenCV's cv::addWeighted function is used and transparency is controlled by alpha parameter.

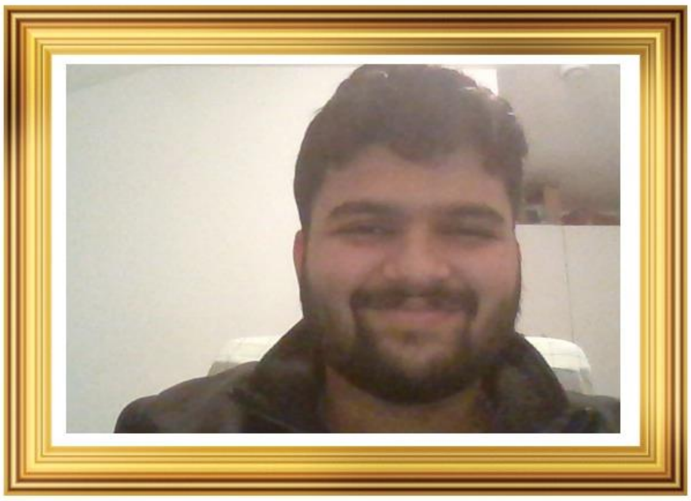


Fig 24: Add Frame Mask

Reflection

Through this assignment I got hands-on experience with computer vision concepts and OpenCV functions. I learned how to implement image and video processing techniques such as filtering, depth estimation and feature detection and also performed the same on live video processing. It was a great assignment to start with as I got familiar with C++ and OpenCV while completing it. It also helped me look at the problems in a different way and troubleshoot technical issue.

Acknowledgment

I would like to thank my professor for sending the OpenCV documentation because it was very helpful material during the assignment. I also used the example codes and materials provided by the professor, which helped execute different tasks. I would also like to thank my teammate, Akshaj Raut, for helping with Task 11 of the Depth Anything V2 network because I had difficulties setting up the environment on my machine. His input was very important for the successful completion of this task.