Pattern Recognition and Computer Vision

CS 5330 – Spring 2026

Project 1

“Video-special Effects”

Name: Harsh Vijay Mamania

NUID: 002036770

Overall Project Description:

This project implements a real-time video processing interface using C++ and OpenCV. The system captures live video from a webcam and applies various image filters and computer vision techniques activated by keyboard inputs. The implementation includes classical image processing operations (grayscale conversion, blur, edge detection), face detection using Haar cascades, and modern deep learning-based depth estimation using the Depth Anything V2 network. The project demonstrates both fundamental computer vision concepts and practical optimization techniques, including the use of separable filters for improved performance and direct pixel manipulation for efficient processing.

---

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

Summary:

This task implements a basic image processing application that reads an image from disk using a command-line argument and displays it using OpenCV. The program loads the image as a cv::Mat object in BGR format, resizes it to 50% for display purposes, and prints diagnostic information including dimensions, color channels, and bytes per pixel. As additional functionality, the program demonstrates pixel-level manipulation by swapping the red and blue color channels through direct pointer access, iterating through each pixel and exchanging the values at indices 0 (blue) and 2 (red). Both the original and color-swapped images are displayed in separate windows, and the program waits for 'q' or Enter key input before terminating.

Demo Images:

Figure 1: Original Image

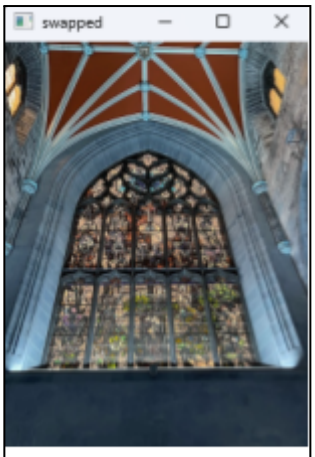


Figure 2: Colors Swapped

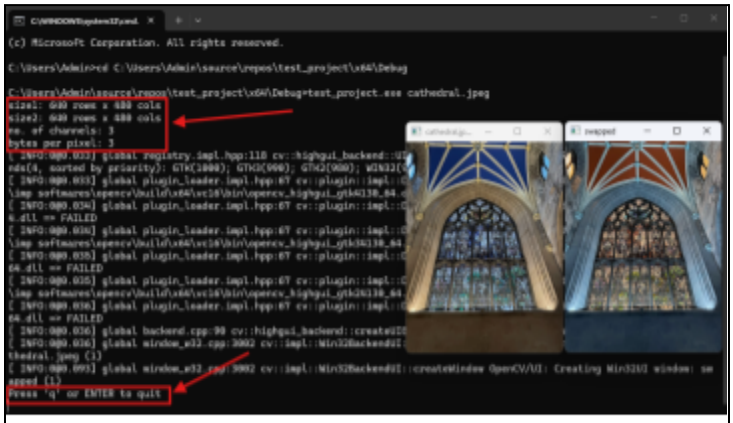


Figure 3: Addition Image Information; 'q' to close.

Observations:

- Figure 1 shows the original cathedral image with natural blue tones in the ceiling and warm lighting, displaying proper BGR color channel ordering
- Figure 2 demonstrates the color-swapped version where blue and red channels are exchanged, resulting in orange/brown ceiling tones and cooler overall color temperature
- Figure 3 displays the console output showing image properties: 640 rows by 480 columns, 3 color channels (BGR), and 3 bytes per pixel (1 byte per channel)
- The color swap produces a noticeably different aesthetic while maintaining all structural details and contrast relationships
- The resizing to 50% ensures both windows fit comfortably on screen without losing visual clarity of the architectural details

---

2. Display live video

Summary:

This task implements a real-time video capture and display system using OpenCV's VideoCapture class. The program opens an external webcam using the DirectShow backend (cv::CAP_DSHOW), retrieves its resolution properties, and enters an infinite loop that continuously captures and displays frames. The cv::waitKey(10) function pauses for 10 milliseconds between frames while checking for keyboard input. The 'q' key terminates the program, while the 's' key saves the current frame with an auto-incrementing filename (screenshot1.jpg, screenshot2.jpg, etc.). This forms the foundation for all subsequent filter implementations activated through different keystroke commands.

Demo Images:

Figure 4: Screenshot from live video stream

Observations:

- Figure 4 shows a successfully captured live video frame with clear image quality and proper color representation
- The video stream displays in real-time with no visible lag or frame drops, demonstrating efficient capture and display loop implementation

---

3. Display greyscale live video

Summary:

This task implements grayscale conversion using OpenCV's built-in cvtColor function with the COLOR_BGR2GRAY flag. When the user presses 'g', the filter mode is set to "grayscale" and each subsequent frame is converted from the three-channel BGR color space to a single-channel grayscale image before display.

OpenCV uses a perceptually weighted formula for this conversion: **Y = 0.299×R + 0.587×G + 0.114×B**, where the green channel receives the highest weight (approximately 59%), red receives medium weight (30%), and blue receives the lowest weight (11%). These weights are based on human visual perception, as our eyes are most sensitive to green light and least sensitive to blue. The conversion happens in real-time for each captured frame, and the program maintains this filter state until another key is pressed to switch modes.

Demo Images:

Figure 5: Original Image

Figure 6: using 'cv::cvtColor' grayscale

Observations:

- (I proctored the scene to include one item of each B, G, R color to see the transition)
- Figure 5 shows the original color image with distinct Blue shirt, Green bag (in hand), and Red/Orange bag (background), each displaying their natural hue and saturation
- Figure 6 demonstrates the grayscale conversion where the green bag appears brightest due to the 0.587 weighting factor giving it maximum contribution to luminance
- The blue shirt appears darker in grayscale compared to its vibrant appearance in color, reflecting the low 0.114 weighting assigned to the blue channel

- The red/orange bag in the background maintains moderate brightness in grayscale due to red's 0.299 weight, falling between the bright green and dark blue
- The perceptually weighted conversion preserves contrast relationships that match human perception, making the image appear natural despite losing color information

---

4. Display alternative greyscale live video

Summary:

This task implements a custom grayscale conversion algorithm as an alternative to OpenCV's standard method. The function is implemented in a separate filter.cpp file using direct pixel manipulation through pointer access. Instead of using OpenCV's perceptually weighted formula (0.299R + 0.587G + 0.114B), this implementation **reverses** the weighting to emphasize blue over green: **Y = 0.114×R + 0.299×G + 0.587×B**. This inverted weighting produces noticeably different luminance values, particularly for objects with strong blue or green components.

The function iterates through each row using the ptr method to access pixels directly, extracts the BGR values for each pixel, applies the reversed weighting formula, and assigns the resulting grayscale value to all three channels. This approach demonstrates custom filter development and the impact of different weighting schemes on perceptual brightness.

Demo Images:

Figure 5: Original Image

Figure 6: using 'cv::cvtColor' grayscale

Figure 7: using my alternative grayscale

Observations:

- Figure 6 shows OpenCV's standard grayscale where the green bag appears brightest due to the high green weighting (0.587), reflecting natural human perception
- Figure 7 demonstrates the alternative grayscale with reversed weights.
- The blue shirt shows increased luminance in the alternative method compared to the standard grayscale, as it now receives the highest weight (0.587) instead of the lowest (0.114)
- The red/orange bag shows visible darkening in the alternative method as red's weight decreases from 0.299 to 0.114, resulting in lower overall luminance for red-dominant regions.
- The perceptual difference between the two methods is most pronounced in regions with strong blue or green color components, demonstrating how weighting choices significantly affect grayscale appearance

---

5. Implement a Sepia tone filter

Summary:

This task implements a sepia tone filter that creates an antique photograph effect by applying a transformation matrix to each pixel's color values. Each output channel (BGR) is calculated as a weighted combination of all three input channels using predefined coefficients. The implementation stores original BGR values in temporary variables before modifications to ensure all calculations use the original colors. Since the coefficient sums exceed 1.0, the std::min function clamps results to 255 to prevent overflow. The filter applies transformations through direct pointer-based pixel manipulation, processing each frame in real-time.

Demo Images:

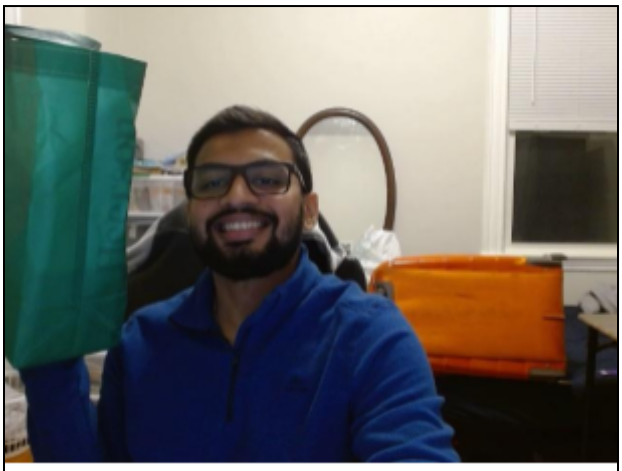


Figure 8: original image

Figure 9: using ‘sepia’ tone filter

Observations:

- Figure 8 shows the original image with vibrant blue shirt, green bag, and orange bag, displaying natural modern color saturation and cool white balance
- Figure 9 demonstrates the sepia effect creating a warm, vintage aesthetic with brown and amber tones replacing the original blues and greens, mimicking aged photographs
- The sepia transformation eliminates the cool blue tones entirely, converting the blue shirt to warm brown hues that evoke nostalgic, old-photograph characteristics

---

6. Implement a 5x5 blur filter

Summary:

This task implements two versions of a 5x5 Gaussian blur filter using the integer kernel [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]. The naive implementation uses the at method to apply the full 25-element kernel in one pass. The optimized version exploits separability by decomposing the blur into sequential horizontal and vertical [1 2 4 2 1] passes, reducing operations from 25 to 10 per pixel, and uses direct pointer access instead of at method to eliminate bounds-checking overhead. Timing was conducted using the professor's program modified for Windows by replacing sys/time.h with chrono.

Demo Images:

Figure 10: original image

Figure 11: using 5x5 blur filter (naive)

Figure 12: using 5x5 blur filter (separable filters)

Timing Result:

- Naive implementation (blur5x5_1): **0.3945** seconds per image on cathedral.jpeg
- Optimized implementation (blur5x5_2): **0.0496** seconds per image on cathedral.jpeg
- Speedup achieved: approximately 8x faster
- Performance gain from separable filters (25 to 10 operations per pixel) and pointer-based access

Observations:

- Figure 10 shows the original sharp image with clearly readable text on the blue book cover, demonstrating fine detail preservation
- Figures 11 and 12 both display equivalent blur effects, with text at the bottom of the book becoming illegible due to smoothing, confirming both implementations produce identical visual results
- The naive blur exhibited noticeable lag during live video processing, causing frame rate drops that impacted real-time performance
- The optimized separable filter maintained smooth real-time playback without perceptible lag, validating the 8x performance improvement

---

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

Summary:

This task implements 3x3 Sobel edge detection filters for both horizontal (X) and vertical (Y) gradients using separable 1x3 kernels. The Sobel X filter applies a vertical smoothing pass [1 2 1] followed by a horizontal derivative [-1 0 1], detecting vertical edges where intensity changes horizontally. The Sobel Y filter applies horizontal smoothing [1 2 1] followed by a vertical derivative [1 0 -1] with positive values upward, detecting horizontal edges. Both filters output 16-bit signed short (CV_16SC3) images to accommodate gradient values ranging from -255 to +255. The implementation uses direct pointer access with Vec3b for reading input and Vec3s for writing signed output. For visualization, cv::convertScaleAbs converts the signed values to absolute unsigned char format suitable for display, keeping the raw Sobel output separate from the visualization.

Demo Images:

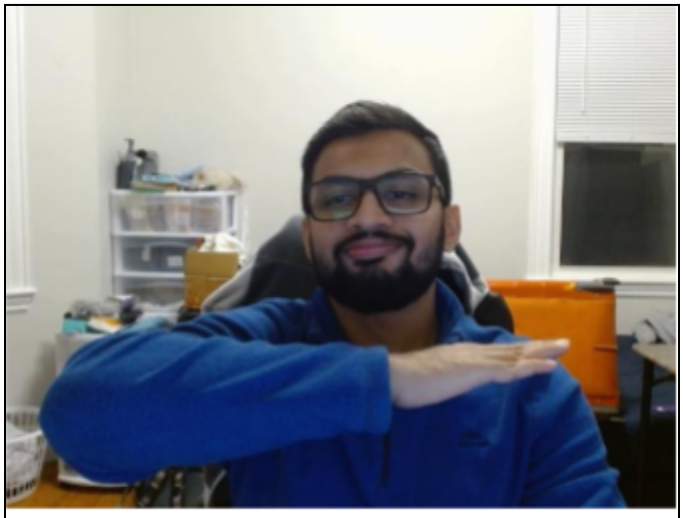


Figure 13: original image

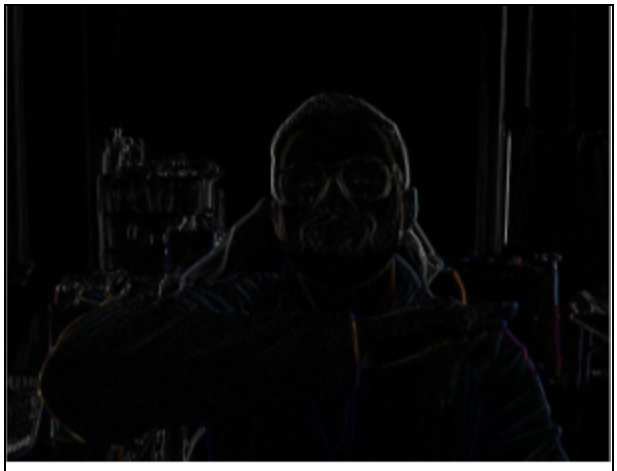


Figure 14: Sobel X (shows vertical edges)

Figure 15: Sobel Y (shows horizontal edges)

Observations:

- Figure 13 shows the original image with the my arm positioned horizontally across the frame, creating strong horizontal edge structure
- Figure 14 (Sobel X) highlights vertical edges, showing strong responses at the vertical boundaries of the face, glasses frames, and body outline while the horizontal arm appears faint
- Figure 15 (Sobel Y) emphasizes horizontal edges, displaying prominent detection of the horizontally positioned arm and shoulders while vertical structures like the face sides are suppressed
- The window blinds in the background (top right corner) show strong horizontal edge response in Sobel Y (Figure 15) due to their horizontal slat structure, while appearing nearly invisible in Sobel X (Figure 14) which detects only vertical transitions

---

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

Summary:

This task implements gradient magnitude calculation by combining Sobel X and Y outputs using Euclidean distance: magnitude = sqrt(sx_val * sx_val + sy_val * sy_val). The function takes two 16-bit signed short (CV_16SC3) Sobel images as input and produces an 8-bit unsigned char (CV_8UC3) output suitable for display. The implementation uses pointer access to iterate through pixels, extracts signed gradient values from both directions, computes the magnitude per channel, and clamps results to 255 to prevent overflow. This combines directional edge information into a single comprehensive edge map.

Demo Images:

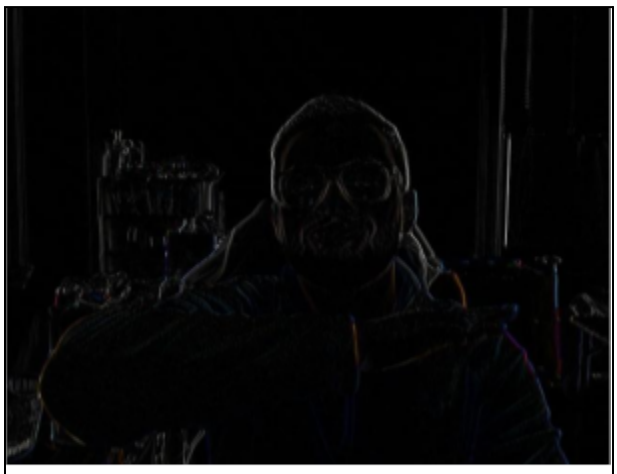


Figure 14: Sobel X (shows vertical edges)

Figure 15: Sobel Y (shows horizontal edges)

Figure 16: Gradient Magnitude image

Observations:

- Figure 14 (Sobel X) shows only vertical edges like face sides and body outline, missing horizontal structures
- Figure 15 (Sobel Y) displays only horizontal edges including the arm and window blinds, missing vertical features
- Figure 16 (Gradient Magnitude) combines both directional gradients, capturing complete edge information including vertical face contours, horizontal arm position, and window blind structures simultaneously

---

9. Implement a function that blurs and quantizes a color image

Summary:

This task implements a blur and quantization filter that reduces the number of distinct colors in an image to create a posterized effect. The function first applies the optimized 5x5 blur (blur5x5_2) to smooth the image, then quantizes each color channel into a fixed number of levels (default 10). Quantization uses bucket-based rounding: bucket size b = 255/levels, then for each pixel value x, compute bucket index xt = x/b and final value xf = xt×b. This maps continuous color values to discrete levels, reducing total possible colors. The implementation uses pointer access to process each pixel and channel independently.

Demo Images:

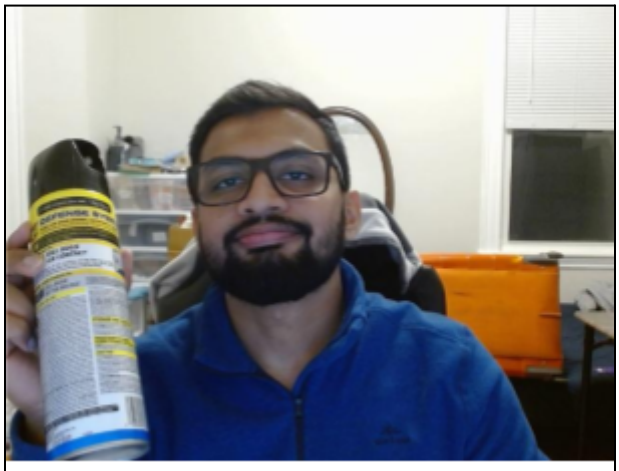


Figure 17: original image

Figure 18: blurred, quantized image

Observations:

- Figure 17 shows the original image with smooth color gradients and clear detail on the spray bottle label
- Figure 18 displays the quantized result with visible color banding and posterization, particularly noticeable in skin tones and background walls where gradual transitions become stepped discrete levels
- The spray bottle label becomes unreadable due to combined blur smoothing and quantization reducing fine detail and color variation

---

10. Detect faces in an image

Summary:

This task integrates Haar cascade-based face detection using pre-built OpenCV functions from the provided faceDetect.cpp file. The detectFaces function converts the input frame to grayscale, applies histogram equalization for better detection accuracy, and uses cv::CascadeClassifier with the haarcascade_frontalface_alt2.xml model to identify face regions. The drawBoxes function renders rectangles around detected faces with customizable color and scale parameters. When activated with the 'f' key, the system processes each frame by converting to grayscale, detecting faces, and drawing bounding boxes in real-time while maintaining the original color video display.

Demo Images:

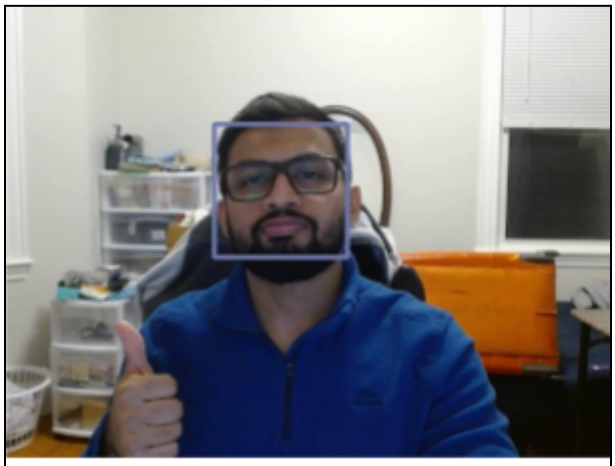


Figure 19: face detection

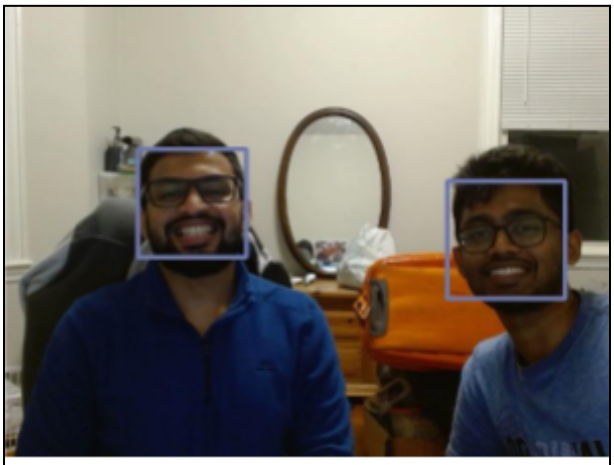


Figure 20: 2 faces correctly detected

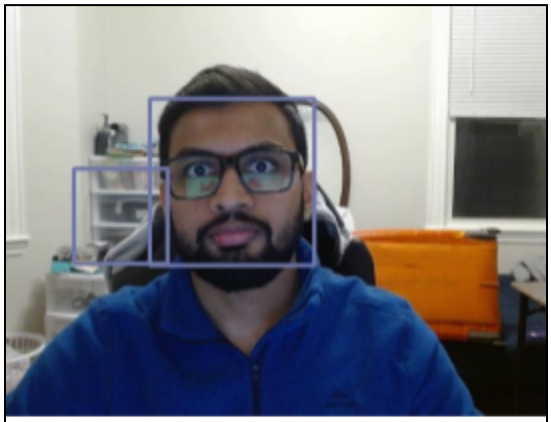


Figure 21: 2 faces WRONGLY detected

Observations:

- Figure 19 demonstrates successful single-face detection with an accurate bounding box around the subject's face despite varying lighting conditions
- Figure 20 shows the system correctly detecting two faces simultaneously when a second person appears in frame, with separate bounding boxes for each detected face
- Figure 21 reveals a false positive detection where the algorithm incorrectly identifies a non-face region as a face, demonstrating the limitations of Haar cascade classifiers with certain background patterns or lighting conditions

---

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

Summary:

This task integrates the Depth Anything V2 neural network using ONNX Runtime to estimate per-pixel depth from monocular images. The DA2Network wrapper class handles network initialization with the model_fp16.onnx file, input preprocessing (resizing, normalization), and output conversion to grayscale depth maps. For performance, frames are resized to 50% before depth inference, then the output is upscaled back to display size. The depth visualization uses the INFERNO colormap where warmer colors (orange/yellow) represent closer objects and cooler colors (purple/blue) indicate farther distances.

The creative depth-based filter combines face detection with depth estimation to display '**Real-Time Face Distance Measurements'**. When a face is detected, the system samples the depth value at the face center coordinates and overlays both a green bounding box and numerical depth value as text on the frame.

Demo Images:

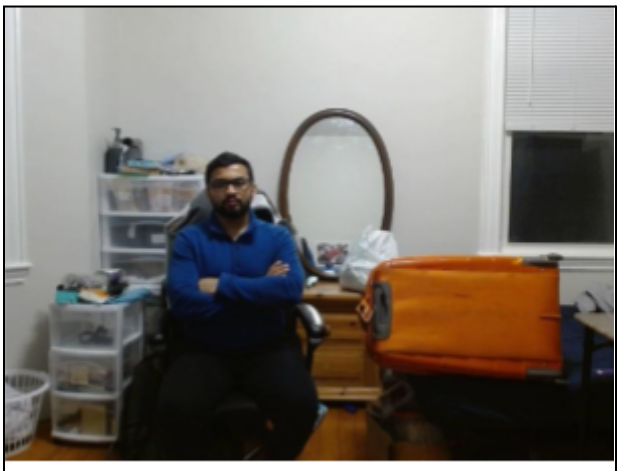


Figure 22: Case 1 (FAR from camera) original image

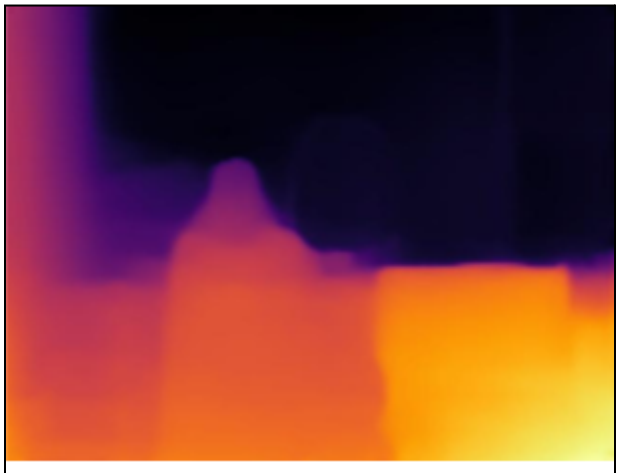


Figure 23: Case 1 (FAR from camera) Depth image

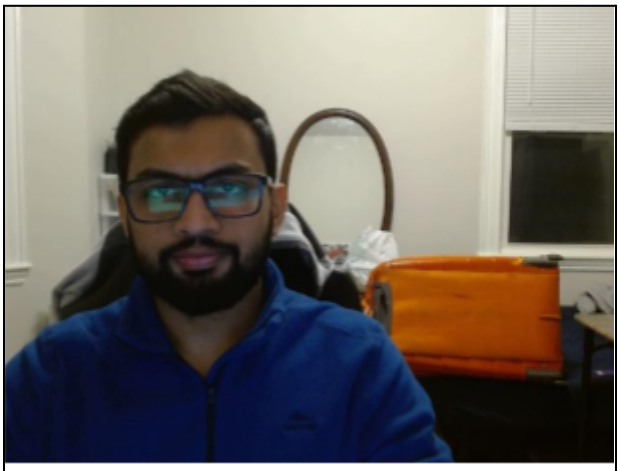


Figure 24: Case 2 (NEAR the camera) original image

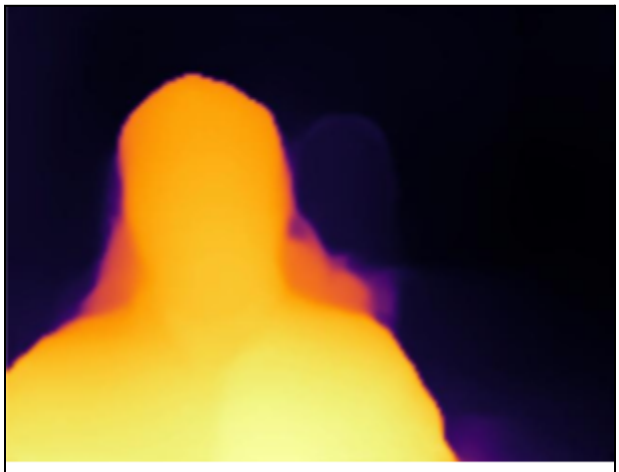


Figure 25: Case 2 (NEAR the camera) depth image

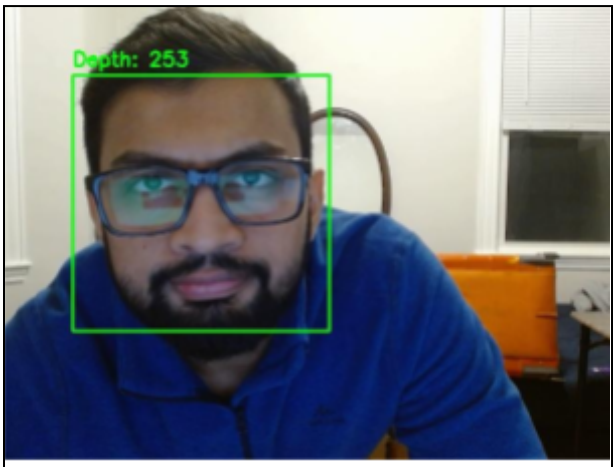


Figure 26: Face Distance (NEAR camera)

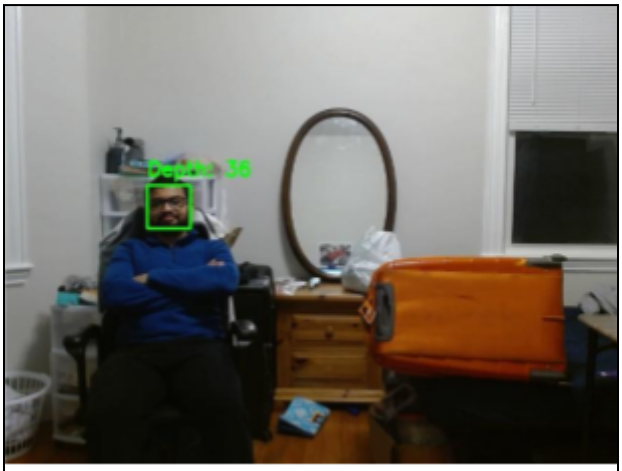


Figure 27: Face Distance (FAR from camera)

Observations:

- Figures 22-23 show depth estimation working correctly when far from camera, with me appearing as cooler purple tones indicating greater distance and background objects showing even darker purple
- Figures 24-25 demonstrate depth response when near the camera, with the my face and torso displaying bright orange/yellow indicating close proximity while background remains purple/dark
- Figure 26 shows the face distance system displaying "Depth: 253" for my close face, with the green box and depth value overlay proving successful integration of face detection and depth estimation
- Figure 27 demonstrates the system maintaining functionality at farther distances, showing lower depth values and correct face detection even when I go farther back

---

12. Implement three more effects on your video

#12.1 - Control based Brightness and Contrast Manipulation

(Type #1 - pixel-wise modification filter)

Summary:

- This effect implements pixel-wise brightness and contrast adjustments that persist across filter modes.
- **Brightness** modification adds a constant offset to all pixel values using *cv::saturate_cast* to prevent overflow/underflow beyond the 0-255 range.
- **Contrast** adjustment scales pixel deviation from middle gray (128) by a multiplicative factor: new_value = (pixel - 128) × factor + 128.
- The controls use keys 1/2 for brightness increase/decrease (±5 per press), keys 3/4 for contrast increase/decrease (±0.1 per press), and key 'r' to reset both to defaults.
- These adjustments are applied after filter processing but before display, meaning they **stack on top** of any active filter (grayscale, sepia, blur, etc.) and persist when switching between filters until explicitly reset.

Demo Images:

(P.T.O.)

A) **Brightness** manipulation:

Figure 28: Original Brightness (no filter)

Figure 29: Original Brightness (+ sepia filter)

Figure 30: Original Brightness (+ grayscale filter)

Figure 31: INCREASED Brightness (no filter)

Figure 32: INCREASED Brightness (+ sepia filter)

Figure 33: INCREASED Brightness (+ grayscale filter)

Figure 34: DECREASED Brightness (no filter)

Figure 35: DECREASED Brightness (+ sepia filter)

Figure 36: DECREASED Brightness (+ grayscale filter)

Observations:

- Figures 28-30 show original images with no filter, sepia filter, and grayscale filter respectively, all displaying normal exposure and contrast as baseline references
- Figures 31-33 demonstrate increased brightness applied to the same three filter states, showing uniformly lighter tones across all pixels while maintaining relative contrast between elements
- Figures 34-36 display decreased brightness with darker overall appearance, demonstrating how the adjustment applies consistently whether viewing the raw feed, color-modified filters, or grayscale conversions

B) **Contrast** manipulation:


Figure 37: Original Contrast (no filter)


Figure 38: Original Contrast (+ sepia filter)


Figure 39: Original Contrast (+ grayscale filter)


Figure 40: INCREASED Contrast (no filter)


Figure 41: INCREASED Contrast (+ sepia filter)


Figure 42: INCREASED Contrast (+ grayscale filter)

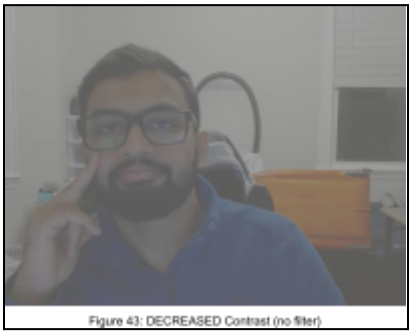


Figure 43: DECREASED Contrast (no filter)

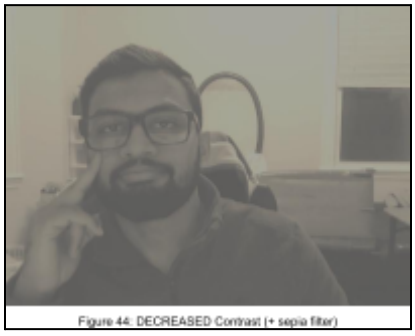


Figure 44: DECREASED Contrast (+ sepia filter)


Figure 45: DECREASED Contrast (+ grayscale filter)

Observations:

- Figures 37-39 show original images with no filter, sepia filter, and grayscale filter respectively, all displaying normal contrast as baseline references
- Figures 40-42 demonstrate increased contrast enhancing the difference between light and dark regions, making colors more vivid in the blue shirt and orange bag while deepening shadows, applied consistently across all three filter modes
- Figures 43-45 display decreased contrast creating a flatter, washed-out appearance with reduced distinction between the subject and background, demonstrating how the adjustment persists across raw, sepia, and grayscale views

---

#12.2 - The ‘Emboss’ effect

(Type #2 - area computation type filter)

Summary:

- This effect creates a 3D relief appearance by computing directional gradients using Sobel X and Y filters, then combining them with a directional lighting assumption.
- The implementation first calculates signed 16-bit Sobel gradients in both horizontal and vertical directions, then computes an emboss value using the dot product with a normalized diagonal direction vector (0.7071, 0.7071) representing light from the **upper-left**, plus a 128 offset to center values in the displayable range.
- This produces highlights on edges where the gradient aligns with the light direction and shadows where it opposes, creating the illusion of raised or carved surfaces.
- The result is converted to unsigned char format using *cv::saturate_cast* for display.

Demo Images:

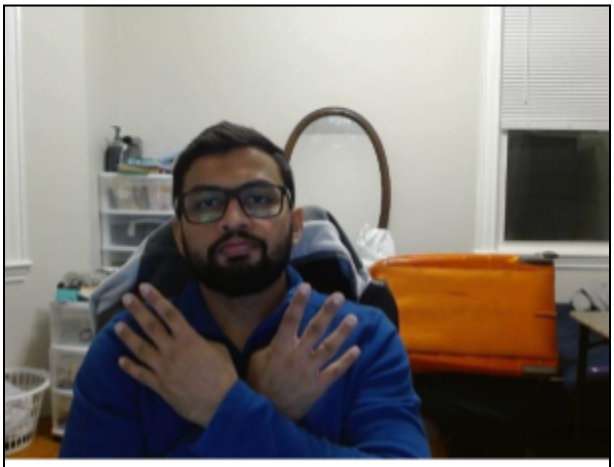


Figure 46: original image

Figure 47: Embossed image

Observations:

- Figure 46 shows the original color image with the subject's hands positioned with palms facing the camera, fingers spread to create distinct edge structures in multiple orientations
- Figure 47 demonstrates the emboss effect highlighting depth perception through directional lighting, where the fingers on the left hand appear brighter with prominent white edges due to alignment with the assumed upper-left light source
- The fingers on the right hand show darker, less pronounced edges because their orientation creates gradients opposing the light direction, demonstrating how the emboss filter responds differently to edge orientation based on the directional lighting model

---

#12.3 - The 'Portrait' effect

(Type #3 - built on ‘face detector’ type filter)

Summary:

- This effect simulates smartphone portrait mode by detecting faces and selectively blurring the background while keeping faces sharp.
- The implementation first creates a copy of the original frame, then applies the optimized blur filter (blur5x5_2) five times iteratively to create a heavily blurred version.
- Using Haar cascade face detection, the system identifies face regions and copies the sharp original content back onto the blurred background only within the detected face bounding boxes.
- This creates a depth-of-field effect where faces remain in focus while the surrounding environment appears out of focus.
- The effect requires face detection to work, making it a multi-step process combining area computation (blur) with face detection.

Demo Images:

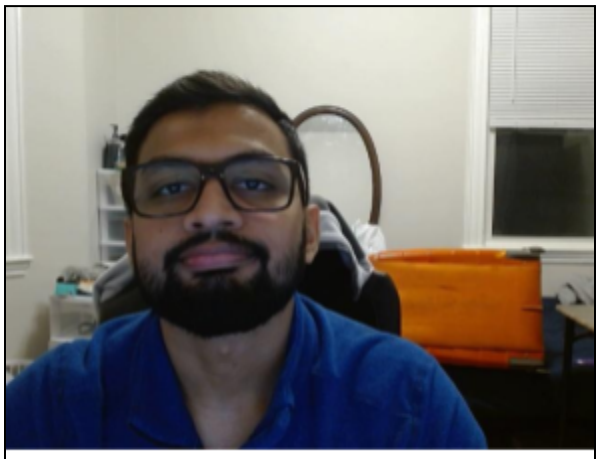


Figure 48: original image

Figure 49: Portrait mode

Observations:

- Figure 48 shows the original image with uniform sharpness across the entire frame, including both, my face and background elements, like the orange bag and window blinds
- Figure 49 demonstrates the portrait mode effect where my face maintains crisp detail and sharpness while the background including the orange bag, mirror, and room environment appears heavily blurred, creating professional depth-of-field separation similar to mobile or DSLR photography

---

Extensions

#1 - The Rockstar Aura (Adaptive Sunglasses + Particle Effects)

#2 - FFT-Based Convolution: A Benchmarking Study Across Kernel Sizes and Filter Approaches

Note: Large Language Models were used to support the development of project extensions. As this is my first exposure to Computer Vision, I wanted to do some fun stuff, learn and do some research. I utilized LLMs to explore implementation possibilities, evaluate the feasibility of ideas, understand new concepts, and refine initial approaches through iterative discussion. All implementations were developed independently, with LLMs serving as a learning resource rather than a solution provider.

#1 - The Rockstar Aura (Adaptive Sunglasses + Particle Effects)

Motivation:

The goal of this extension was to push beyond basic face detection and create something visually engaging and technically sophisticated. The professor's demonstration of halo particle effects was particularly inspiring, and I wanted to build on that concept while adding my own creative interpretation. Rather than simply overlaying static graphics, I aimed to develop an interactive augmented reality-style filter that combines face tracking with adaptive visual elements and animated particle systems, creating a "rockstar aura" effect that responds dynamically to the scene.

What it does:

This extension implements a comprehensive face augmentation system with two primary components:

A. Adaptive Sunglasses System

The sunglasses feature goes beyond simple overlay graphics by implementing three distinct modes, each demonstrating different computer vision and image processing techniques:

Mode 1 - Classic Dope Sunglasses (Key: 7)
Renders stylish oval-shaped sunglasses on detected faces with solid black opaque lenses, complete with realistic highlights, thick black frames, bridge connection, and temple arms extending to the edges of the face. This establishes the foundational overlay system with proper geometric positioning relative to facial features.

Mode 2 - Brightness-Sensitive Adaptive Lenses (Key: 8)
Implements an auto-tinting mechanism inspired by photochromic sunglasses. The lens opacity dynamically adjusts based on the overall scene brightness: in bright environments, lenses become more opaque (darker) to provide better "protection," while in dim lighting, they become semi-transparent, allowing the eyes to remain visible through the lenses. This demonstrates real-time scene analysis and alpha blending techniques.

Mode 3 - Scene-Reflecting Mirror Lenses (Key: 9)
Simulates reflective/mirror sunglasses by displaying a blurred and horizontally flipped version of the scene within the lens regions. This creates the illusion that the sunglasses are reflecting the environment, mimicking the appearance of actual mirrored eyewear. The implementation showcases spatial transformations and selective region-based image compositing.

B. Animated Particle Aura System

Complementing the sunglasses across all modes, golden sparkle particles continuously radiate outward from the face perimeter, creating a dynamic "rockstar aura" effect. Approximately 40 particles are maintained simultaneously, each with independent position, velocity, and visibility states. Particles spawn at random positions along the face boundary and move radially outward at constant speed. They exhibit twinkling behavior: randomly toggling visibility to create a sparkling effect, and automatically respawn at the face edge when they drift too far or their lifetime expires. This creates a continuous, animated halo that emphasizes the detected face with an energetic, celebrity-like visual signature.

Demo Images:

A) Mode #1: Dope Sunglasses + Aura sparkles added.

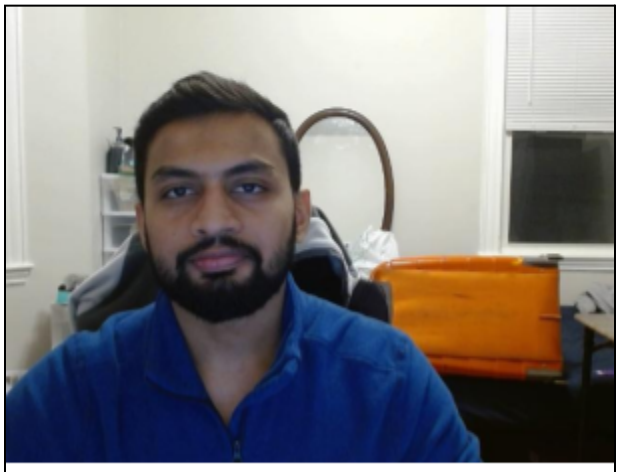


Figure 50: original mode

Figure 51: Mode 1 - Dope Sunglasses + Aura Sparkles

B) Mode #2: Adaptive lenses upgrade

Figure 51: Mode 2 - Brightness Sensitive lenses (more brightness = more opaque, less transparent)

Figure 52: Mode 2 - Brightness Sensitive lenses (less brightness = not opaque, max transparent)

Mode #3: Reflective glasses upgrade.

Figure 52: Mode 3 - Scene reflecting lenses

Working & Code implementations:

A. Sunglasses Implementation

The sunglasses system operates through the addSunglasses() function that adapts its rendering behavior based on the mode parameter. Face detection via Haar cascades provides the face bounding rectangle, which serves as the coordinate reference for overlay positioning.

Geometric Positioning & Base Structure:

All three modes share common geometric calculations. Lens dimensions are computed as fractions of face width and height (lens_width = face.width / 3, lens_height = face.height / 6), ensuring proper scaling. Lenses are positioned at the upper third of the face rectangle (y_pos = face.y + face.height / 3), corresponding to eye level. Centers are calculated equidistant from the face midline. Ellipses are drawn using cv::ellipse() for the lenses, with cv::line() and cv::rectangle() primitives constructing frames, bridges, and temple arms. White highlights positioned at the top-left of each lens simulate light reflection using smaller ellipses with cv::Scalar(200, 200, 200).

Mode 1 - Solid Opaque Lenses:

Straightforward rendering with solid black fill (cv::Scalar(15, 15, 15)). Filled ellipses are drawn directly onto the frame at calculated positions, completely obscuring the eye region. Frame outlines, bridge, and temple arms follow using thicker black lines. This establishes the foundational overlay system with proper geometric positioning.

Mode 2 - Brightness-Adaptive Opacity:

Scene brightness is computed using cv::mean(frame), averaging all color channels to produce a single metric (0-255). This maps to an opacity coefficient via opacity = 0.2 + (avg / 255.0) * 0.7, yielding values between 0.2 (transparent) and 0.9 (opaque). Alpha blending is achieved by cloning the frame, drawing sunglasses on the overlay, then using cv::addWeighted() to blend based on the calculated opacity. Higher scene brightness produces more opaque lenses, mimicking photochromic behavior.

Mode 3 - Scene-Reflecting Lenses:

The frame undergoes Gaussian blur (cv::Size(15, 15)) and horizontal flip (cv::flip() with flip code 1) to create a soft mirror reflection. A binary mask (CV_8UC1) is initialized to black, then white ellipses matching lens positions are drawn to define reflection regions. The copyTo() method with the elliptical mask selectively replaces only the lens areas with the flipped, blurred scene while preserving the rest of the frame. Frames and highlights are added afterward, completing the reflective aesthetic.

B. Particle Aura Implementation

The sparkle system maintains a std::vector<Sparkle> where each struct contains position (x, y), velocity (vx, vy), lifetime counter, and visible boolean for twinkling.

Initialization:

On first execution, 40 particles spawn around the face perimeter. For each particle, a random angle (0-360°) is selected. Position is computed via polar-to-cartesian conversion: x = face_center_x + radius * cos(angle), placing particles on a circle around the face. Velocity vectors point radially outward (vx = cos(angle) * 2.0) at 2 pixels per frame. Random lifetimes (30-90 frames) and initial visibility states are assigned.

Per-Frame Update:

The updateAndDrawSparkles() function processes all particles each frame. Positions update by adding velocity (x += vx), and lifetimes decrement. A random number generator with 10% probability toggles visibility flags, creating twinkling. Euclidean distance from particle to face center is calculated; if exceeding 1.5 * face.width or lifetime expires, the particle respawns at a new random angle on the face perimeter with reset velocity and lifetime. Visible particles render as small circles (cv::circle() with radius 3, color cv::Scalar(0, 215, 255) for gold in BGR).

Integration:

Particles dynamically adapt to detected face position each frame, automatically adjusting spawn points and trajectories as the user moves. The update function executes after sunglasses rendering, ensuring sparkles appear on top. Continuous radial motion, random twinkling, and

automatic respawning create a persistent "rockstar aura" that follows the face naturally throughout the video stream.

Observations:

Figure 50 - Original Mode:
Baseline frame showing unmodified video feed with face detection active. The face is well-lit and centered, providing a clear reference for comparison with the augmented modes.

Figure 51 - Mode 1 (Classic Sunglasses + Aura):
The solid black sunglasses are properly positioned at eye level with accurate geometric scaling relative to the face dimensions. The white highlights on the lenses effectively simulate light reflection, adding depth and realism. Golden sparkle particles are visible radiating from the face, distributed around the head, shoulders, and upper torso region. The particles exhibit varying positions across the frame, demonstrating the outward radial motion and random spawning mechanics. The twinkling effect is evident as some particles are visible while others have toggled off, creating the intended sparkling aesthetic.

Figure 51 & 52 - Mode 2 (Brightness-Sensitive Lenses):
These two images demonstrate the adaptive opacity feature under different lighting conditions. In the brighter environment (Figure 51), the scene has higher overall luminance, resulting in more opaque lenses, where in, the eyes are barely visible through the darkened sunglasses, successfully mimicking auto-darkening behavior. In the darker environment (Figure 52), reduced ambient light causes the lenses to become significantly more transparent, clearly revealing the eyes beneath. The continuous sparkle animation remains active and consistent across both lighting scenarios, maintaining the aura effect regardless of brightness conditions. This demonstrates successful scene-responsive rendering and validates the brightness-to-opacity mapping function.

Figure 52 - Mode 3 (Scene-Reflecting Lenses):
The mirror effect is clearly visible in the lens regions, which display a blurred, flipped version of the surrounding environment. Details such as the shelves, boxes, and background elements are recognizable within the lens areas, though softened by the Gaussian blur. The horizontal flip creates an authentic mirror reflection appearance. The dark background in this particular frame provides strong contrast, making the reflective lens content more prominent and demonstrating that the effect adapts to varying scene compositions. The particle aura continues to function normally, with sparkles distributed around the face boundary.

#2 - FFT-Based Convolution: A Benchmarking Study Across Kernel Sizes and Filter Approaches

Motivation:

This extension was driven by a desire to deeply understand frequency domain processing, a topic that initially proved conceptually challenging despite being fundamental to computer vision. While spatial domain operations like convolution were intuitive after hands-on implementation, the frequency domain remained abstract. Multiple discussions with the professor, including office hours visits, helped clarify the theoretical foundation, but practical questions remained:

- ***How much faster is FFT-based convolution compared to spatial methods?***
- ***At what kernel size does FFT become advantageous?***
- ***How do custom implementations compare against OpenCV's highly optimized built-in functions?***

The professor's assertion that FFT "speeds up" convolution despite transformation overhead was intriguing but required empirical validation. I designed a comprehensive benchmarking study to quantify performance across multiple implementation approaches, kernel sizes, and image types. This experiment serves both as a rigorous performance analysis and a learning exercise to develop intuition about when different algorithmic approaches are optimal.

Consists of 2 parts:

Part #1: Understanding Intermediate FFT Steps.

Part #2: Performance Analysis.

Experiment Setup:

Methods Tested:

Four distinct blur implementations were compared:

1. **Naive Convolution** - Direct 2D kernel application using at<cv::Vec3b>() accessor for pixel access.
2. **Separable Filter** - Optimized approach applying 1D horizontal and vertical passes sequentially, using row pointer access for improved cache performance.
3. **FFT-Based Convolution** - Frequency domain processing: image and kernel transformed via cv::dft(), multiplied using cv::mulSpectrums(), and converted back via cv::idft(), with optimal padding to power-of-2 dimensions.
4. **OpenCV GaussianBlur** - Industry-standard baseline implementation representing best-case optimized performance.

Kernel Sizes:

Six kernel sizes were tested: 3×3, 5×5, 9×9, 15×15, 21×21, and 31×31.
Small kernels test scenarios where separable filtering should dominate.
Medium kernels represent the transition zone.
Large kernels evaluate whether FFT achieves superiority for computationally intensive convolutions.

Test Images:

Two 640×480 images were selected:

- **High Frequency**: City aerial photograph with dense edges and textures
- **Low Frequency**: Sky photograph with smooth gradients

Timing Methodology:

Each test was preceded by 3 warmup iterations, followed by 10 timed iterations using std::chrono::high_resolution_clock. Average time per iteration was recorded in milliseconds, with results exported to CSV for analysis.

FFT Implementation:

The algorithm processes each color channel independently:

1. **Padding**: Image expanded to optimal DFT size using cv::copyMakeBorder() with zero-padding.
2. **Forward DFT**: Padded image converted to frequency domain via cv::dft().

3. **Kernel Preparation**: Gaussian kernel generated in spatial domain, then padded to match image dimensions with circular wrapping.
4. **Kernel DFT**: Padded kernel transformed to frequency domain.
5. **Spectral Multiplication**: Element-wise complex multiplication using cv::mulSpectrums(), implementing the convolution theorem where spatial convolution becomes frequency domain multiplication.
6. **Inverse DFT**: Product spectrum transformed back using cv::idft() with normalization.
7. **Extraction**: Padding removed and result converted to 8-bit format.

Part #1: Understanding Intermediate FFT Steps-

A) High Frequency Image (City Grid):

Figure 53: Original Image - City Grid (High Frequency)

Figure 54: Stage 0: Original grayscale channel

Figure 55: Stage 1: Padded to optimal DFT size

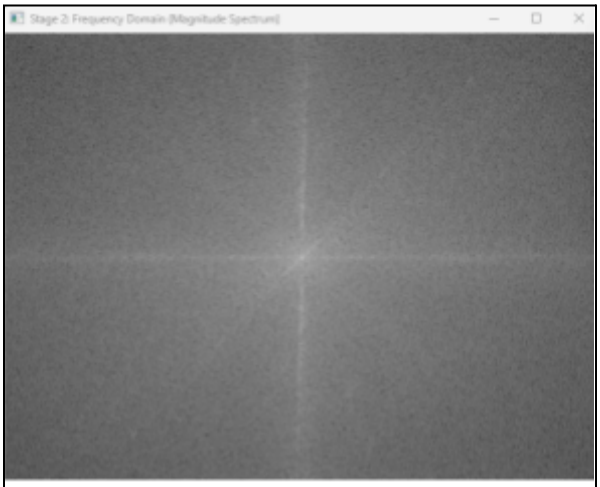


Figure 56: Stage 2: After Forward DFT - converted to frequency domain

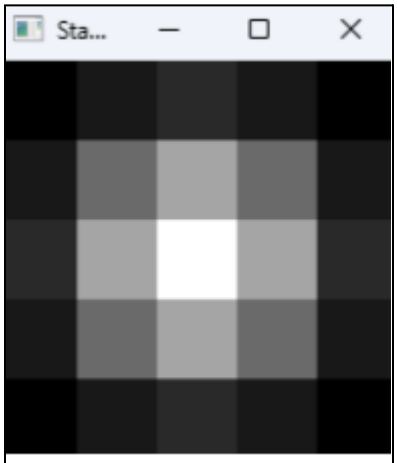


Figure 57: Stage 3: Gaussian Kernel

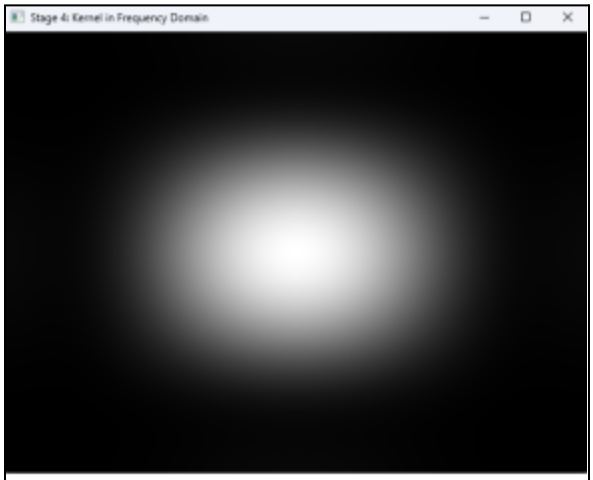


Figure 58: Stage 4: Kernel after DFT - in frequency domain

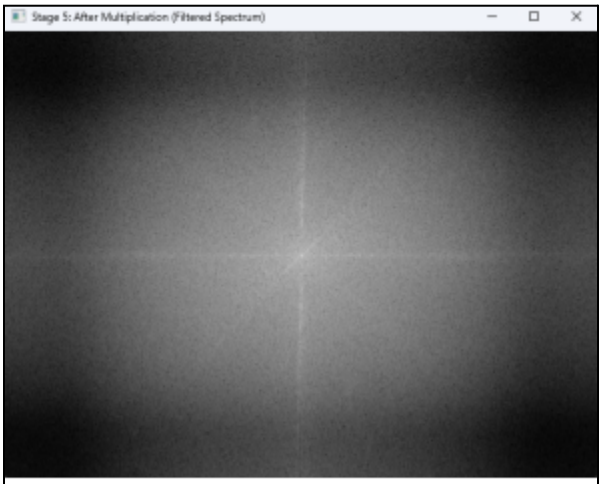


Figure 59: Stage 5: Image spectrum * Kernel spectrum (High frequencies attenuated by Gaussian)

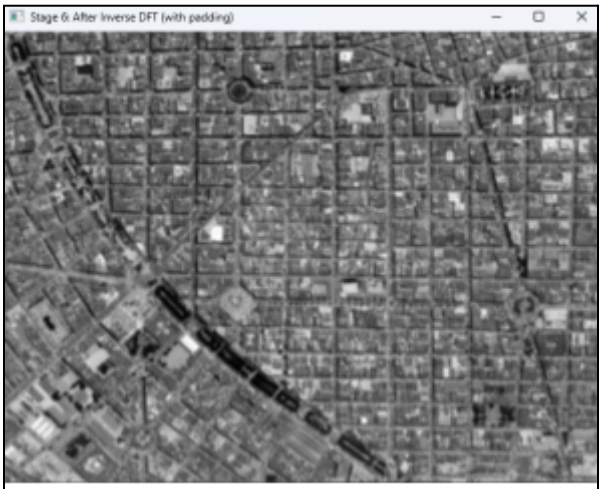


Figure 60: Stage 6: After Inverse DFT - back to spatial domain

Figure 61: Stage 7: Final Result (padding removed)

FFT Steps-

- **Figure 53-54**: Original city aerial shows dense urban structures with abundant sharp edges and fine-grained texture. The grayscale channel preserves all spatial details with high contrast between adjacent pixels.
- **Figure 55-56**: After padding to optimal DFT dimensions (1024×512), the frequency domain reveals a bright, concentrated center (low frequencies) with scattered bright pixels throughout (high frequencies from edges and textures). The widespread distribution confirms broad frequency range.
- **Figure 57-58**: The 5×5 Gaussian kernel shows classic bell-shaped distribution in spatial domain. In frequency domain, it appears as a bright Gaussian blob at center, indicating it predominantly passes low frequencies while attenuating high frequencies.
- **Figure 59**: After spectral multiplication, high-frequency components are significantly dimmed compared to the original spectrum, while the central low-frequency region remains prominent. This directly visualizes the Gaussian filter attenuating high-frequency energy.
- **Figure 60-61**: Inverse transformation back to spatial domain shows heavy blur with sharp edges softened, building details smoothed, and textures homogenized. The aggressive filtering successfully removed high-frequency content.

B) Low Frequency Image (Sky):

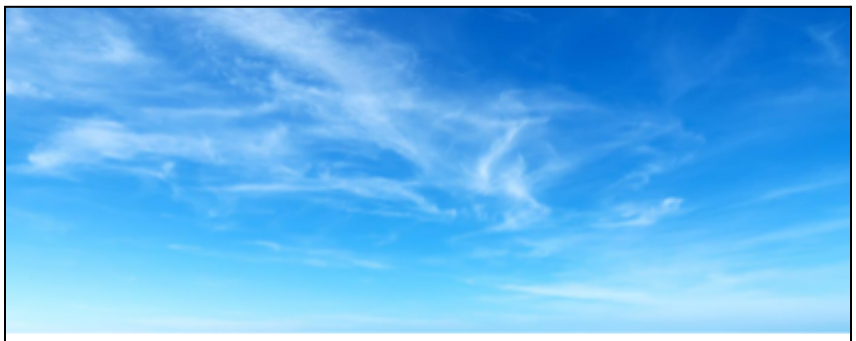


Figure 62: Original Image - Calm Sky (Low Frequency)

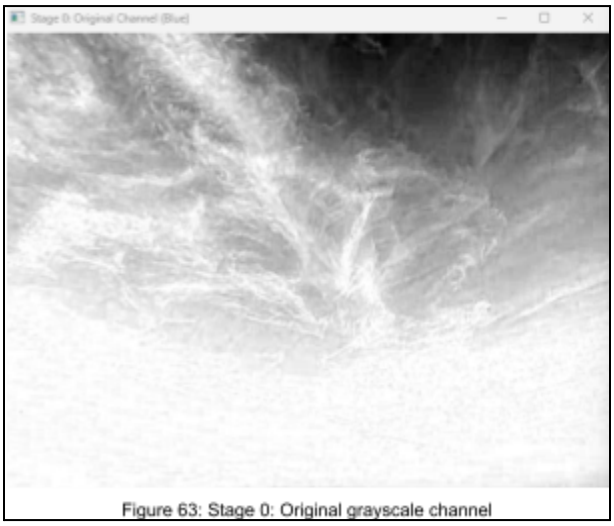


Figure 63: Stage 0: Original grayscale channel

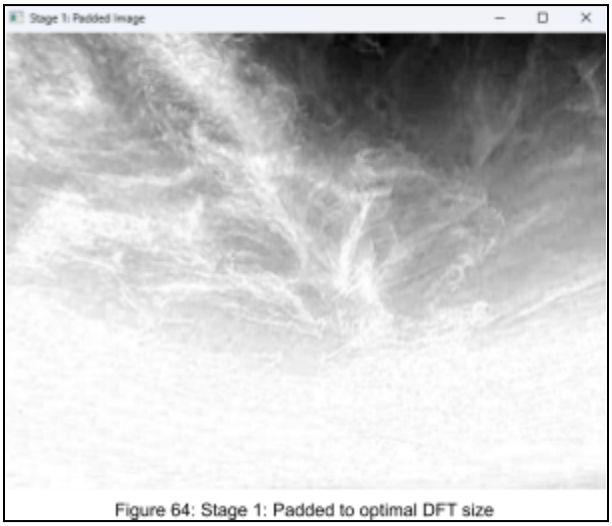


Figure 64: Stage 1: Padded to optimal DFT size

Figure 65: Stage 2: After Forward DFT - converted to frequency domain

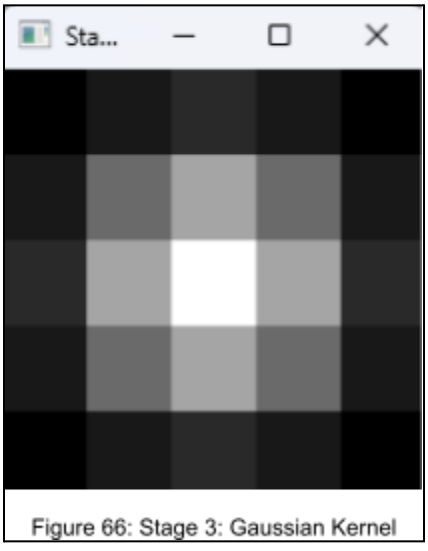


Figure 66: Stage 3: Gaussian Kernel

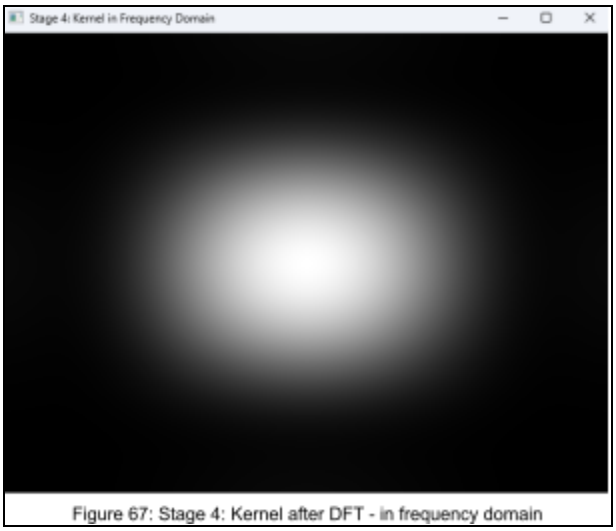


Figure 67: Stage 4: Kernel after DFT - in frequency domain

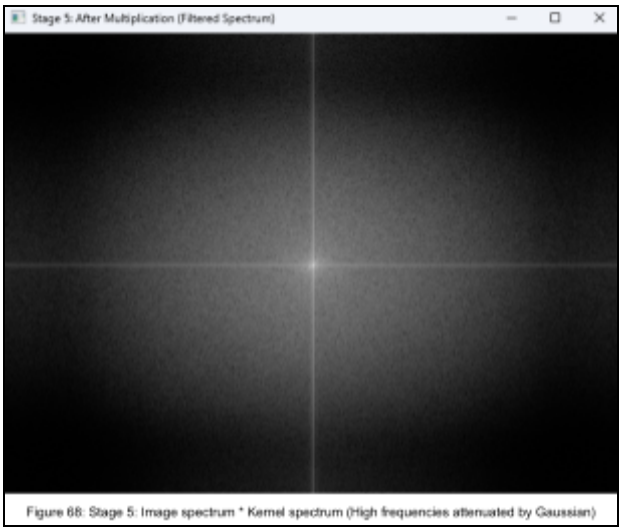


Figure 68: Stage 5: Image spectrum * Kernel spectrum (High frequencies attenuated by Gaussian)

Figure 69: Stage 6: After Inverse DFT - back to spatial domain

Figure 70: Stage 7: Final Result (padding removed)

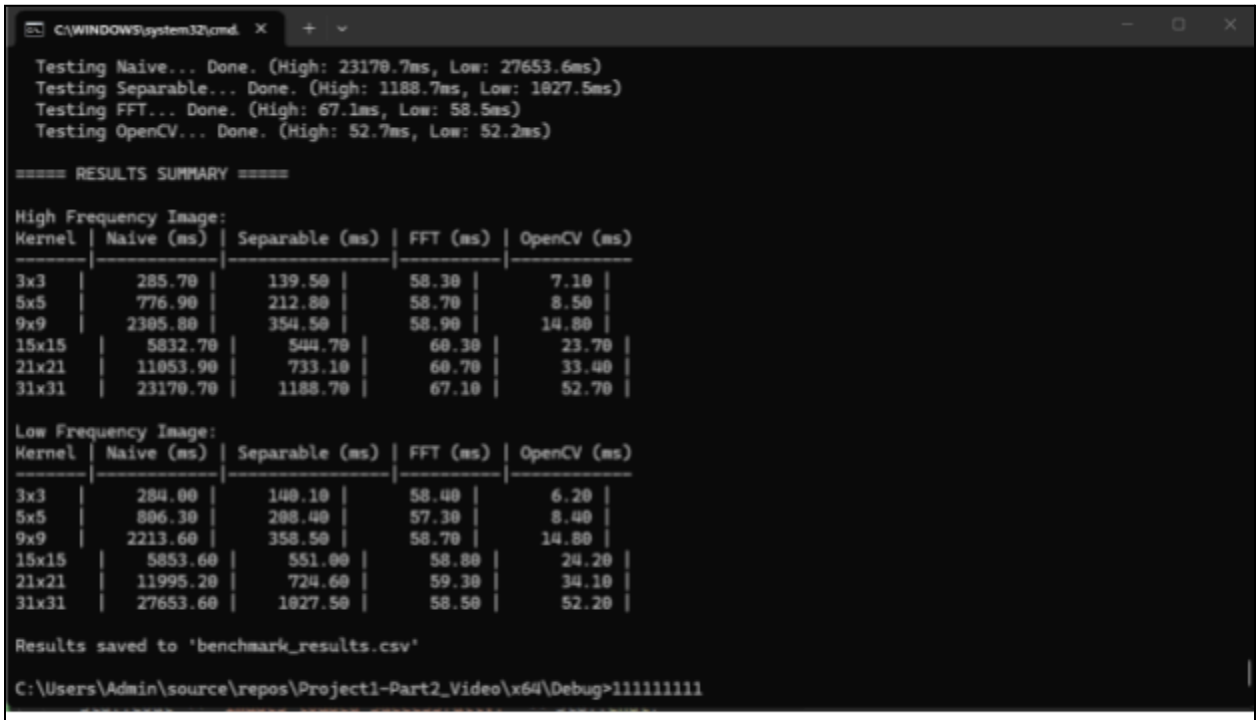
FFT Steps-

- **Figure 62-64**: The original sky image shows smooth gradients with minimal sharp edges. The grayscale channel reveals gentle tonal transitions characteristic of low-frequency content.
- **Figure 65**: The frequency spectrum shows a highly concentrated bright region at center with very little energy at periphery. This compact, centralized distribution indicates the image is dominated by low frequencies with virtually no high-frequency components.
- **Figure 68**: After spectral multiplication, the spectrum appears nearly identical to the original. Because the image had almost no high-frequency content initially, the Gaussian attenuation has minimal visible effect.
- **Figure 69-70**: The spatial domain result shows only subtle softening. The inherent smoothness means there were few sharp features to blur, so the final result is visually very similar to the original.

Comparative Analysis:

- The FFT visualization illustrates the fundamental difference between image types:
- **Frequency Domain**: High frequency (city) shows spread-out spectrum; low frequency (sky) shows compact, centralized spectrum.
- **Filtering Effect**: High frequency experiences dramatic attenuation and heavy visual blur; low frequency shows minimal spectral change and subtle visual effect.
- **Performance**: Despite vastly different frequency content, both images took nearly identical processing time (~58ms for FFT). This demonstrates that FFT-based convolution performance is content-independent, depending solely on image dimensions and kernel size, not on frequency distribution or edge density.

Part #2: Performance Analysis-



```
C:\WINDOWS\system32\cmd.

  Testing Naive... Done. (High: 23170.7ms, Low: 27653.6ms)
  Testing Separable... Done. (High: 1188.7ms, Low: 1027.5ms)
  Testing FFT... Done. (High: 67.1ms, Low: 58.5ms)
  Testing OpenCV... Done. (High: 52.7ms, Low: 52.2ms)

===== RESULTS SUMMARY =====

High Frequency Image:
Kernel | Naive (ms) | Separable (ms) | FFT (ms) | OpenCV (ms)
-------|------------|----------------|----------|------------
3x3    |     285.70 |         139.50 |    58.30 |       7.10 |
5x5    |     776.90 |         212.80 |    58.70 |       8.50 |
9x9    |    2305.80 |         354.50 |    58.90 |      14.80 |
15x15  |    5832.70 |         544.70 |    60.30 |      23.70 |
21x21  |   11053.90 |         733.10 |    60.70 |      33.40 |
31x31  |   23170.70 |        1188.70 |    67.10 |      52.70 |

Low Frequency Image:
Kernel | Naive (ms) | Separable (ms) | FFT (ms) | OpenCV (ms)
-------|------------|----------------|----------|------------
3x3    |     284.00 |         140.10 |    58.40 |       6.20 |
5x5    |     806.30 |         208.40 |    57.30 |       8.40 |
9x9    |    2213.60 |         358.50 |    58.70 |      14.80 |
15x15  |    5853.60 |         551.00 |    58.80 |      24.20 |
21x21  |   11995.20 |         724.60 |    59.30 |      34.10 |
31x31  |   27653.60 |        1027.50 |    58.50 |      52.20 |

Results saved to 'benchmark_results.csv'

C:\Users\Admin\source\repos\Project1-Part2_Video\x64\Debug>111111111
```

Figure 71: Console output (Performance Report - all variants, all kernel sizes, for both frequency images)

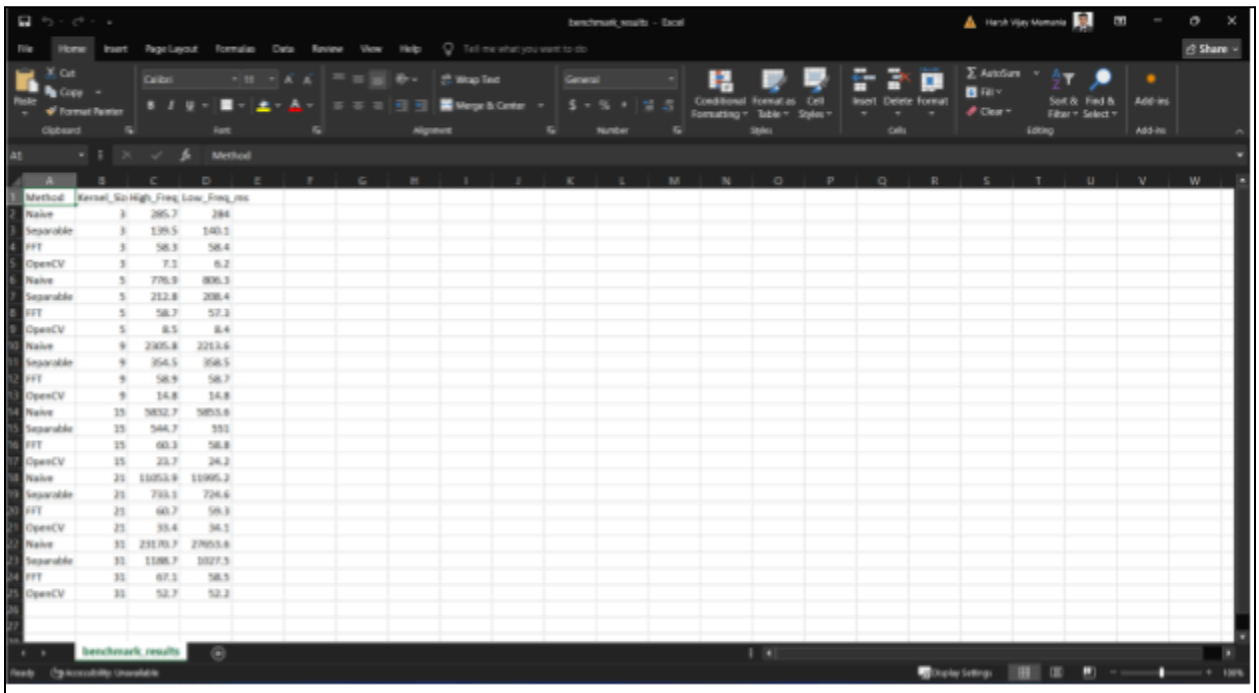


Figure 72: Output csv

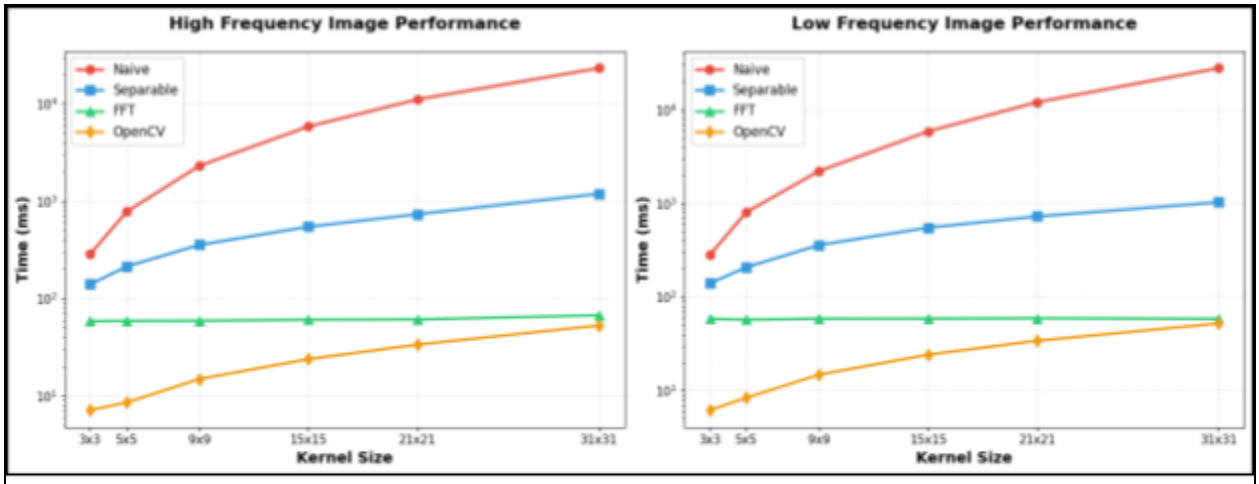


Figure 73: Performance Comparison Graphs

Reading the Results:

- The benchmark results show kernel sizes in rows and methods in columns, with execution time in milliseconds. Two tables (high and low frequency images) allow comparison of content dependency.
- The line graphs use logarithmic y-axis scaling because execution times span three orders of magnitude (7ms to 27,000ms). Four colored lines represent the four methods, with slope indicating scaling behavior and crossover points showing transitions in method superiority.

Interpretation of Results:

- **Naive Method (Red):** Exhibits exponential growth from ~285ms at 3×3 to ~23,000ms at 31×31 (80× slowdown). The dramatic upward curve reflects $O(N^2M^2K^2)$ complexity. For 31×31 kernels, each pixel requires 961 multiplications with individual memory accesses, making this approach computationally prohibitive for large kernels.

- **Separable Method (Blue):** Shows much better scaling with ~140ms at 3×3 to ~1,190ms at 31×31. The improvement stems from separability: a K×K 2D convolution becomes two 1×K 1D convolutions, reducing complexity from $O(K^2)$ to $O(2K)$ per pixel. However, computational cost still scales linearly with kernel size.

- **FFT Method (Green):** The line is nearly flat with execution time remaining stable: 3×3 (58ms), 9×9 (59ms), 21×21 (61ms), 31×31 (67ms). Only 15% variation across 10× kernel size increase demonstrates $O(N \log N)$ complexity where kernel size contributes minimally. The crossover point occurs around 9×9, where FFT becomes competitive with separable filtering. Beyond this threshold, FFT maintains consistent ~60ms performance

while separable methods degrade linearly.

- **OpenCV Method (Orange):** Represents the performance ceiling with ~7ms at 3×3 to ~53ms at 31×31. OpenCV likely uses separable filtering with vectorized operations for small kernels, potentially switching to FFT or other optimized methods for larger kernels. The shallow upward slope indicates near-optimal scaling.

- **Content Independence:** High and low frequency image results are virtually identical for all methods, confirming that blur performance is content-agnostic. The computational workload depends solely on image dimensions and kernel size.

Why We Observe These Results:

- **Theoretical Complexity:** Naive convolution requires $O(N^2M^2K^2)$ operations (every pixel needs K^2 operations), separable filtering achieves $O(N \times M \times 2K)$ (two 1D passes reduce kernel operations from K^2 to $2K$), and FFT operates at $O(N \times M \times \log(N \times M))$ (dominated by transform operations with minimal kernel size overhead).

- **FFT Overhead Justification:** While FFT requires transformation overhead (forward DFT, inverse DFT, padding, kernel preparation), this overhead is amortized for large kernels. The convolution theorem enables spatial domain convolution ($O(K^2)$ per pixel) to become frequency domain multiplication (O(1) per frequency bin). For small kernels (3×3, 5×5), direct convolution with 9-25 multiplications per pixel is faster than transform overhead. For large kernels (21×21, 31×31), avoiding 441-961 multiplications per pixel via FFT becomes highly advantageous.

- **Cache and Memory Effects:** Naive convolution with `at()` accessor incurs function call overhead and poor cache locality. Separable filtering with pointer arithmetic improves cache utilization by processing rows contiguously and columns sequentially. FFT operates on entire matrices, enabling better cache prefetching and vectorization.

- **Practical Implications:** For real-time computer vision applications, use separable filtering for small kernels (≤7×7) due to simplicity and minimal overhead. Use FFT for large kernels (≥15×15) for consistent performance regardless of kernel size. Leverage OpenCV's adaptive implementations for optimal performance across all scenarios.

This benchmarking study empirically validates theoretical complexity predictions and provides actionable guidance for algorithm selection based on kernel size requirements.

Reflections:

- This project served as my comprehensive introduction to computer vision, C++, and Visual Studio, taking me from complete beginner to implementing a full real-time video processing application with multiple sophisticated filters.

- I learned fundamental CV concepts hands-on, starting with basic image representation (BGR channels, cv::Mat objects) and progressing to complex operations like edge detection, depth estimation, and face recognition, understanding how images are just matrices of pixel values that can be mathematically manipulated.

- The experience of implementing filters from scratch using direct pixel manipulation taught me the importance of optimization.

- Working with different color spaces and transformations (BGR to grayscale, sepia tone, depth maps) gave me intuition about how human perception influences algorithm design, such as understanding why green receives the highest weight in perceptual grayscale conversion.

- Integrating modern deep learning (Depth Anything V2) with classical CV techniques (Haar cascades, Sobel filters) demonstrated how contemporary computer vision combines multiple approaches to solve complex problems.

- Experimenting with creative effects like portrait mode, emboss filters, and depth-based face distance measurement was incredibly fun and showed me the artistic potential of computer vision beyond just technical implementation.

- My enthusiastic exploration of bonus features, diverse filter variations, and creative combinations throughout this project reflects not only my genuine passion for computer vision but also my drive to experiment beyond the assignment requirements, discovering the limitless creative possibilities that CV programming offers.

Acknowledgement:

- I would like to express my sincere gratitude to **Professor Bruce Maxwell** for his exceptional teaching and patience in addressing my most fundamental questions, explaining concepts multiple times until they clicked.

- His consideration in granting a 7-day extension for students who joined late allowed me to explore the material thoroughly without rushing, which directly enabled the depth and creativity of the extensions presented in this report.

- I am grateful to the **course teaching assistants** for their consistent support and readiness to help whenever I encountered difficulties. Their guidance on debugging issues and clarifying implementation details was invaluable throughout the project development process.

- The **OpenCV documentation** served as an essential reference for understanding function parameters, image processing techniques, and best practices for efficient implementation.

- **YouTube tutorials** provided supplementary explanations and visual demonstrations that complemented course materials, particularly helpful for grasping complex concepts like frequency domain transformations and optimization techniques.

- Special thanks to **Claude** for being available 24/7 to help me ideate, refine half-formed concepts, explore implementation possibilities, understand new techniques, and provide subject matter expertise that accelerated my learning process throughout this project.

- Finally, a lighthearted but genuine thank you to **my housemate, Rajkesh** for the amusing yet crucial assistance of holding down the 's' key to capture screenshots while I was in proctored poses or positioned away from the keyboard testing various real-time effects. His contribution made documenting this work significantly easier!
