LAB EXPERIMENT 3
BASCS OF IMAGE PROCESSING WITH JETSON ORIN NANO

By: Gowtham Kumar Kamuni
Instructor: Dr. Jafar Saniie
ECE 501, Artificial Intelligence and Edge Computing
Lab Date: 05-30-2025
Due Date: 06-06-2025

Acknowledgment: I acknowledge all of the work (including figures and codes) belongs to me and/or persons who are referenced.

Signature: Gowtham Kumar Kamuni

I. Introduction

A. Purpose

The purpose of this lab was to explore the fundamentals of image processing using Python and OpenCV, specifically running on the Jetson Orin Nano platform. The lab was divided into several phases that guided us through thresholding, edge detection, morphological operations, shape classification, and video processing. Each step was designed to help us understand how images can be segmented, analyzed, and interpreted in real world applications. The end goal was not just to run code, but to actually see how each processing step affects the output, both visually and structurally.

B. Background

Image processing is a core part of modern computer vision systems, and OpenCV is one of the most powerful tools for working with images and video streams. In this lab, we worked with grayscale images, applied thresholding methods like Otsu's algorithm, and used edge detection techniques such as Canny, Sobel, and Laplacian. These helped us isolate the shapes and boundaries of objects in the image.

We also explored morphological operations like erosion, dilation, opening, and closing to clean up noisy binary images and refine object shapes. Then, using shape detection logic from Appendix A, we classified shapes based on their contours and number of edges. This was applied not only to static images but also across frames of a live video stream, bringing everything together in the final phase.

One of the more exciting parts was using the Jetson Orin Nano, which brings GPU acceleration through CUDA. This allowed us to run OpenCV operations more efficiently, showing why hardware matters when you're processing frames in real-time. This lab tied together everything we've been learning, coding, debugging, experimenting, and visualizing results, in a way that felt both challenging and rewarding.

II. Lab Procedure and Equipment List

A. Equipment

- Jetson Orin Nano Developer Kit (Ubuntu-based platform)
- Python 3 (programming language)
- OpenCV Library (for image processing)
- NumPy and Matplotlib (for matrix operations and visualizations)
- Test Images:
  - test.bmp – used for segmentation, morphology, and shape detection
  - shapes.png – used for clean geometric shape recognition
- Test Video:
  - video.mp4 – used in the final phase for frame based shape annotation
- Peripheral Devices:
  - USB drive (for backups and video transfer)
  - HDMI monitor, keyboard, and mouse (connected to Jetson)

B. Procedure

This lab was executed in four phases, each building on the previous one:

Phase A – Image Segmentation and Edge Detection

- Loaded a grayscale image and applied various thresholding methods.
- Applied edge detection using Canny, Sobel, and Laplacian filters.
- Observed and compared how each method captured object outlines.

Phase B – Morphological Operations and Histogram Equalization

- Performed dilation, erosion, opening, and closing on a noisy image.
- Applied histogram equalization to improve contrast and detail visibility.
- Saved and compared each operation's effect on the image structure.

Phase C – Shape Detection

- Implemented the shape classification logic from Appendix A.
- Processed both clean and noisy shape images to detect and label basic shapes.
- Used contour detection and polygon approximation to classify circles, rectangles, triangles, and more.

Phase D – Video Processing with Shape Annotation

- Extracted frames from a video using OpenCV.
- Applied shape detection to each frame and overlaid labels.
- Regenerated a new annotated video from the processed frames.
- Compared the result against expected performance and visual accuracy.

III. Results and Analysis

Phase A – Image Segmentation and Edge Detection

In this phase, I tested different segmentation and edge detection methods on a grayscale image. I used Otsu's thresholding to binarize the image, then applied Canny, Sobel, and Laplacian edge detectors. Each one reacted differently depending on the noise and structure in the image. Canny gave me clean outlines with strong edges, while Laplacian and Sobel picked up more intensity shifts and curves.

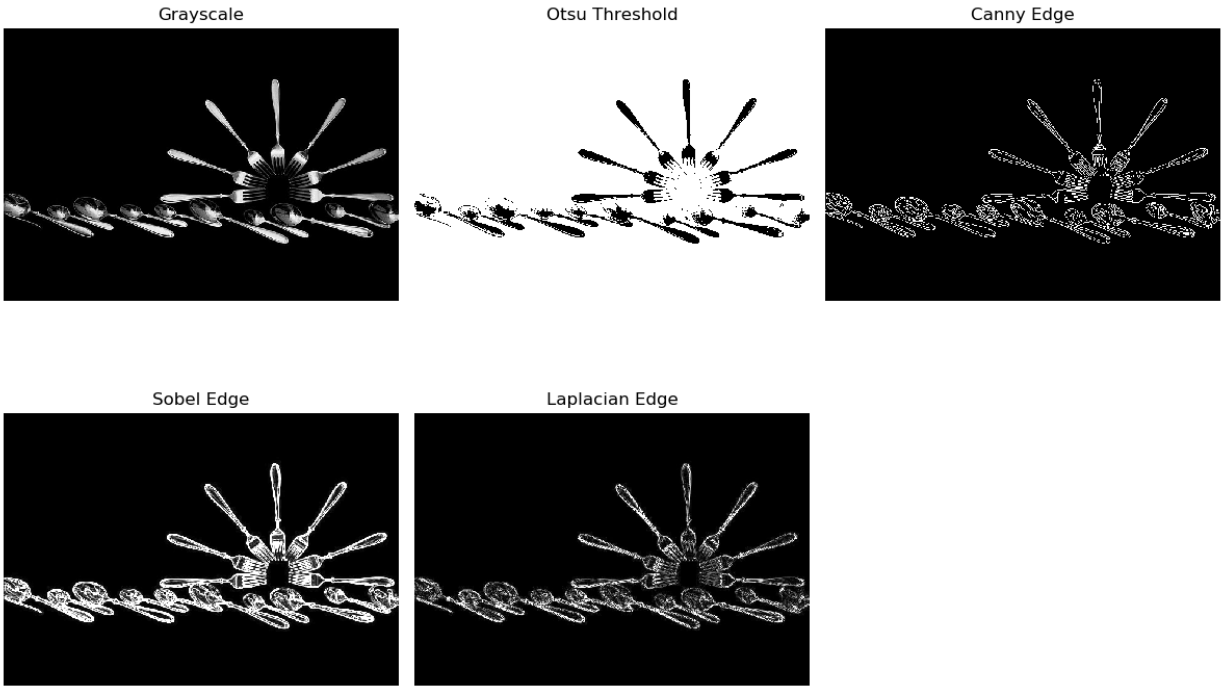


Figure 1: Edge Detection and Segmentation Outputs

This image shows the results of applying different image segmentation and edge detection methods on a grayscale test image. Otsu thresholding produced a clean binary map, while Canny highlighted sharp contours. Sobel and Laplacian detectors picked up gradient-based edges and inner shape curves. This helped build intuition on how each method sees the structure in an image.

Phase B – Morphological Operations and Histogram Equalization

Here, I used morphological filters to clean up a noisy binary image. Dilation and erosion allowed me to either grow or shrink the white regions, while opening and closing helped remove noise and fill small holes. I also equalized the histogram to increase contrast, which made low-detail areas stand out more clearly. These preprocessing steps are super useful when you're trying to detect shapes or isolate foreground objects in cluttered images.

Figure 2: Morphological Operations and Histogram Equalization

This image illustrates how different morphological transformations affect a noisy binary input. Dilation expanded the object boundaries, erosion removed minor artifacts, and the combination (opening/closing) refined the shapes. Histogram equalization enhanced image contrast, making edges and features easier to detect in later stages.

Phase C – Shape Detection and Classification

In this step, I ran the shape detection algorithm based on Appendix A. It detects contours, approximates polygons, and labels shapes like rectangles, circles and ellipses. I tested the script on both clean and noisy images. Even when noise was added, the logic still performed decently, although sometimes it detected circles as ellipses or vice versa, which honestly makes sense depending on contour curves.

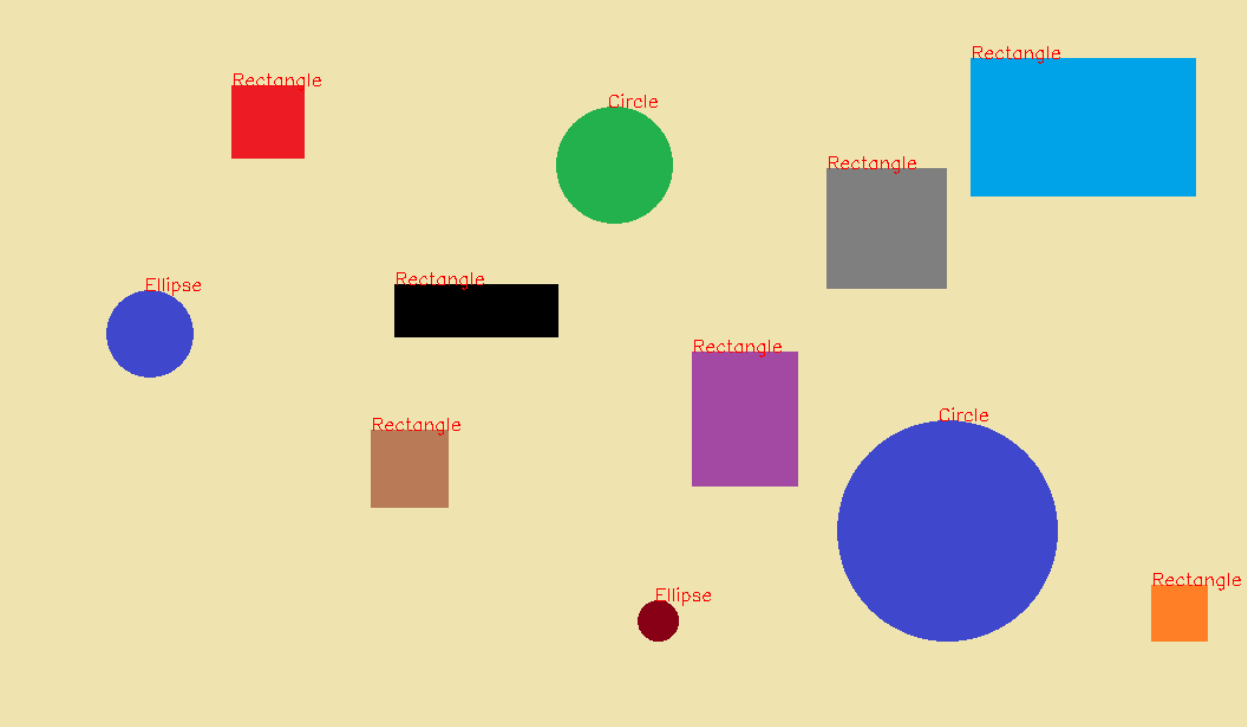


Figure 3: Shape Detection on Clean Input

This figure shows the result of running the shape detection script on a geometric test image. Contours are drawn around each object and labeled using polygonal approximation. Rectangles, circles, and ellipses were identified based on the number of sides and aspect ratios. The output was visually accurate and consistent with expected results.

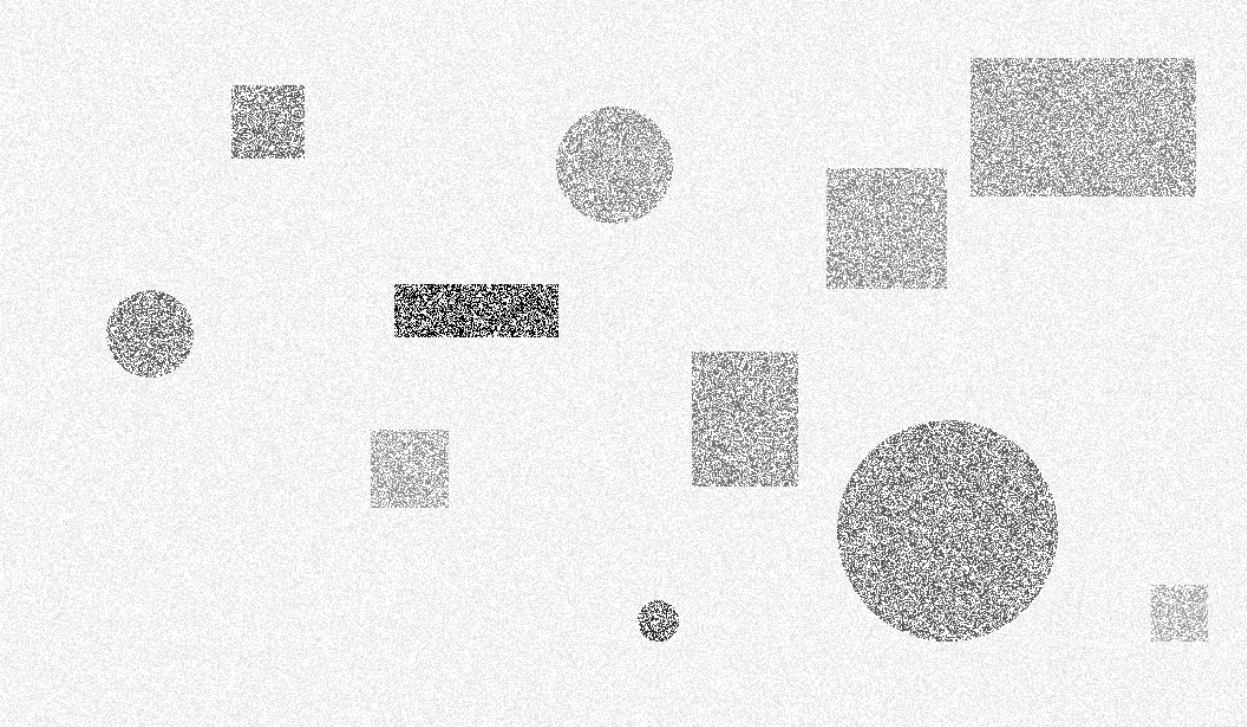


Figure 4: Noisy Input

This figure shows a noisy version of the shape input image, where Gaussian noise has been added to simulate poor image quality. This tests the robustness of the shape detection algorithm when faced with less-than-ideal input conditions.

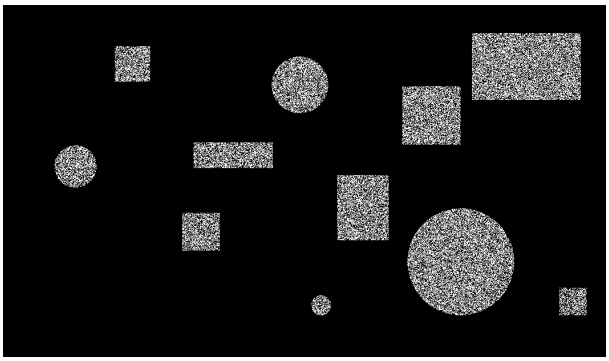


Fig 5: Thresholded Noisy Image

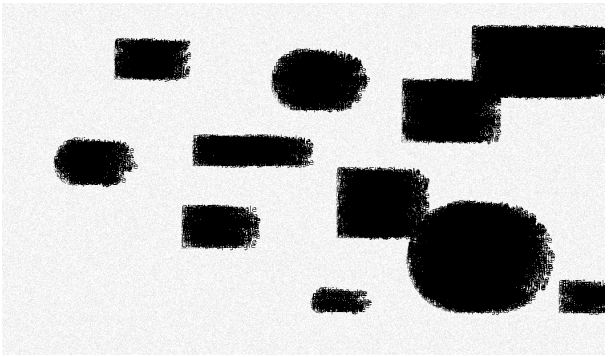


Fig 6:Detected Shapes from Noisy Input

Thresholding applied to the noisy input using the same Otsu method. Despite the added noise, the binary segmentation still preserves most of the shape outlines, although with slight roughness around the boundaries.

This image demonstrates the shape detection performance on noisy input. While the results are slightly less clean compared to the noise-free case, the algorithm still correctly identified most shapes. This confirms that the method holds up reasonably well even under degraded conditions.

These outputs represent the image classification stage in Phase 3, where various test images, including clean and noisy cases, were analyzed for shape detection. The remaining outputs and dynamic results are included in the recorded demo submission.

Phase D – Video Frame Processing and Shape Annotation

This was probably my favorite part of the lab. I extracted frames from a test video, ran shape detection on each one, labeled them, and stitched the frames back into a new video. It was super satisfying to see the shapes appear on live video frames. There were small inconsistencies, like a circle being marked as an ellipse in some frames, but that's just part of the approximation game. The overall output was clean and definitely submission worthy.

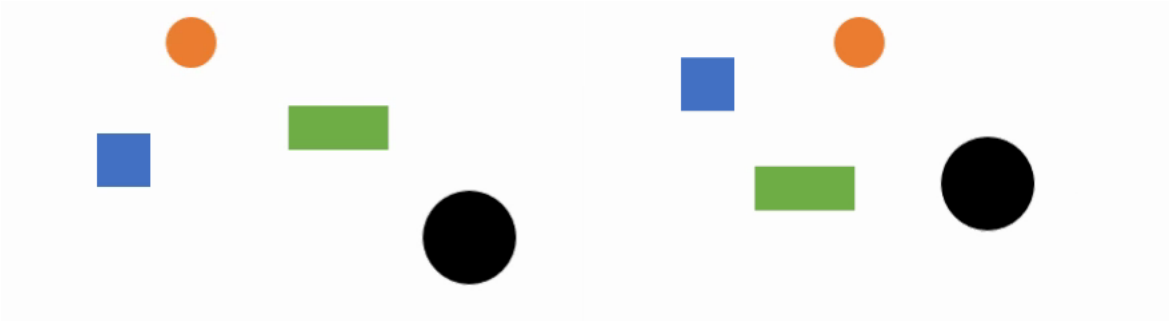


Figure 7: Sample Frames from Original Video

This frame shows one of the original frames extracted from the test video before any processing was applied.

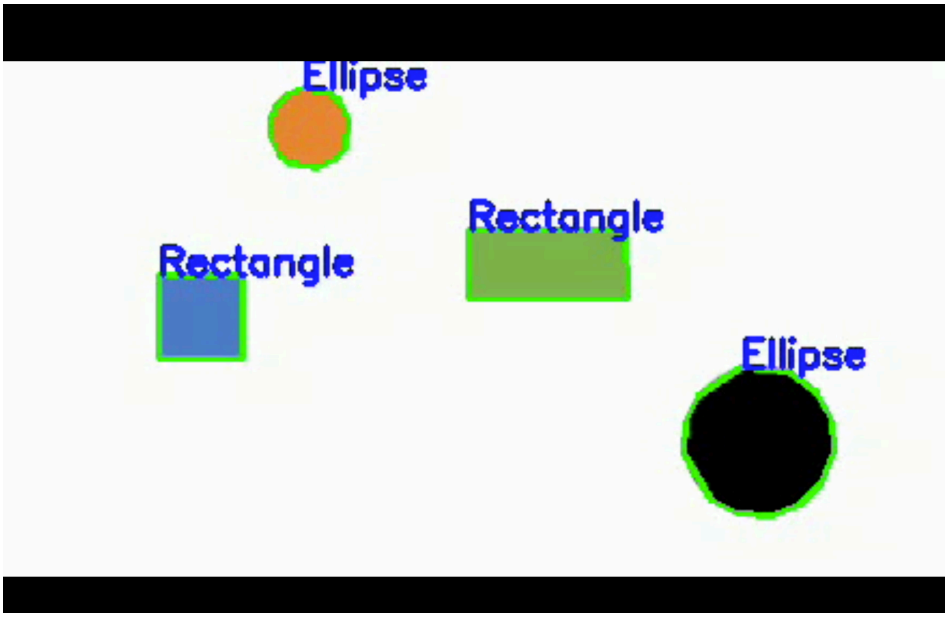


Figure 8: Annotated Frame with Detected Shapes

This frame shows the output after running the shape detection pipeline on the video. Shapes are correctly labeled and annotated in real time. The result was converted back into a video, showing frame by frame classification in action.

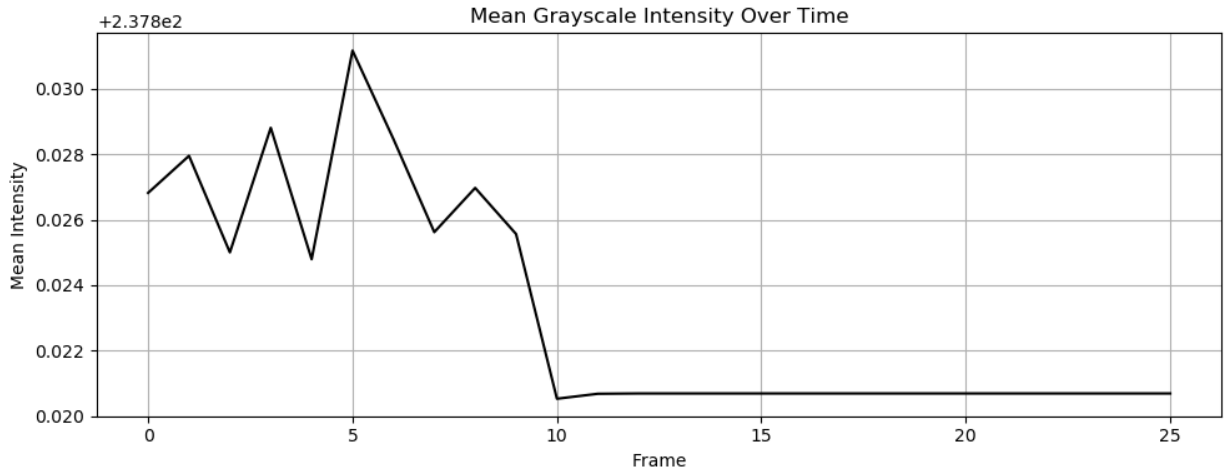


Fig 9: Mean Gray Scale Intensity Over Time

This graph shows the average grayscale intensity of each frame across the 25 frame image classification batch. The y axis represents the mean intensity value, while the x axis represents the frame index. A noticeable drop around frame 10 suggests either a shift in background, lighting, or dominant object tone. This kind of analysis helps track how consistent the scene is across frames, and whether preprocessing adjustments are needed in future phases.

A. Discussion

1. Comment on the advantages and disadvantages of programming with MATLAB and Python, specifically on the image segmentation script/program.

Both MATLAB and Python have their strengths, but when it comes to image segmentation, I found Python, especially with OpenCV, to be more flexible and faster to execute on the Jetson platform. The OpenCV functions are compact, GPU-accelerated, and allow tighter control over thresholds, edge detection, and image manipulation with very few lines of code.

On the other hand, MATLAB was easier to visualize and debug in the beginning. It gives clean plots and GUI features that make image analysis simple, especially for beginners. But when running heavy scripts or doing real time segmentation (like we did in Phase A and Phase D), it started to feel sluggish.

Python's integration with Jetson and support for hardware acceleration made a huge difference in my workflow. It felt more low level and efficient, while MATLAB was more high-level and user-friendly. Overall, Python was the better tool for this lab, especially because of its speed, ecosystem, and compatibility with CUDA.

3. Why are there extra steps involving the range of pixel values in the Python noise generation script/program in Procedure B.2?

The extra steps in the Python noise generation script are there to control the range and intensity of the added noise, so the image doesn't get distorted beyond usable limits. In Procedure B.2, we generate Gaussian noise and add it to the original image, but we also scale and reshape it carefully to match the original image's dimensions and dynamic range.

Since pixel values in OpenCV images are in the range 0–255 for 8-bit images, it's important to scale the Gaussian noise appropriately. If we just added raw noise without control, we'd end up with out of range pixel values, wrapping around or causing visual artifacts.

In our script, we used parameters like mean = 0 and sigma = range*np.sqrt(0.005) to keep the noise realistic, and applied it to every channel if the image had color. Without these steps, the noise would be either too weak to affect anything or so strong that it ruins the image structure. This control is essential when testing how robust shape detection is under noisy conditions, like we did in the noisy_input.png and noisy_detected.png results.

4. Why is GPU acceleration useful/beneficial to image/video processing in general?

GPU acceleration is a game-changer for image and video processing, and I felt that difference firsthand while running these scripts on the Jetson Orin Nano. When you're processing multiple frames, detecting shapes, applying filters, and rendering annotated outputs in real time, the CPU just can't keep up.

The GPU, on the other hand, is built for this kind of parallel processing. Operations like convolution, thresholding, edge detection, and matrix math are all insanely faster when offloaded to CUDA cores. I noticed this especially during the video processing phase, generating 25+ annotated frames and compiling them would've taken forever on a CPU.

GPU acceleration makes image processing scalable. It turns one slow loop into hundreds of operations running side by side. It's not just faster, it enables real time performance, which is exactly what you need for smart cameras, autonomous systems, and live computer vision. In short, if you're serious about image/video processing, the GPU isn't optional, it's essential.

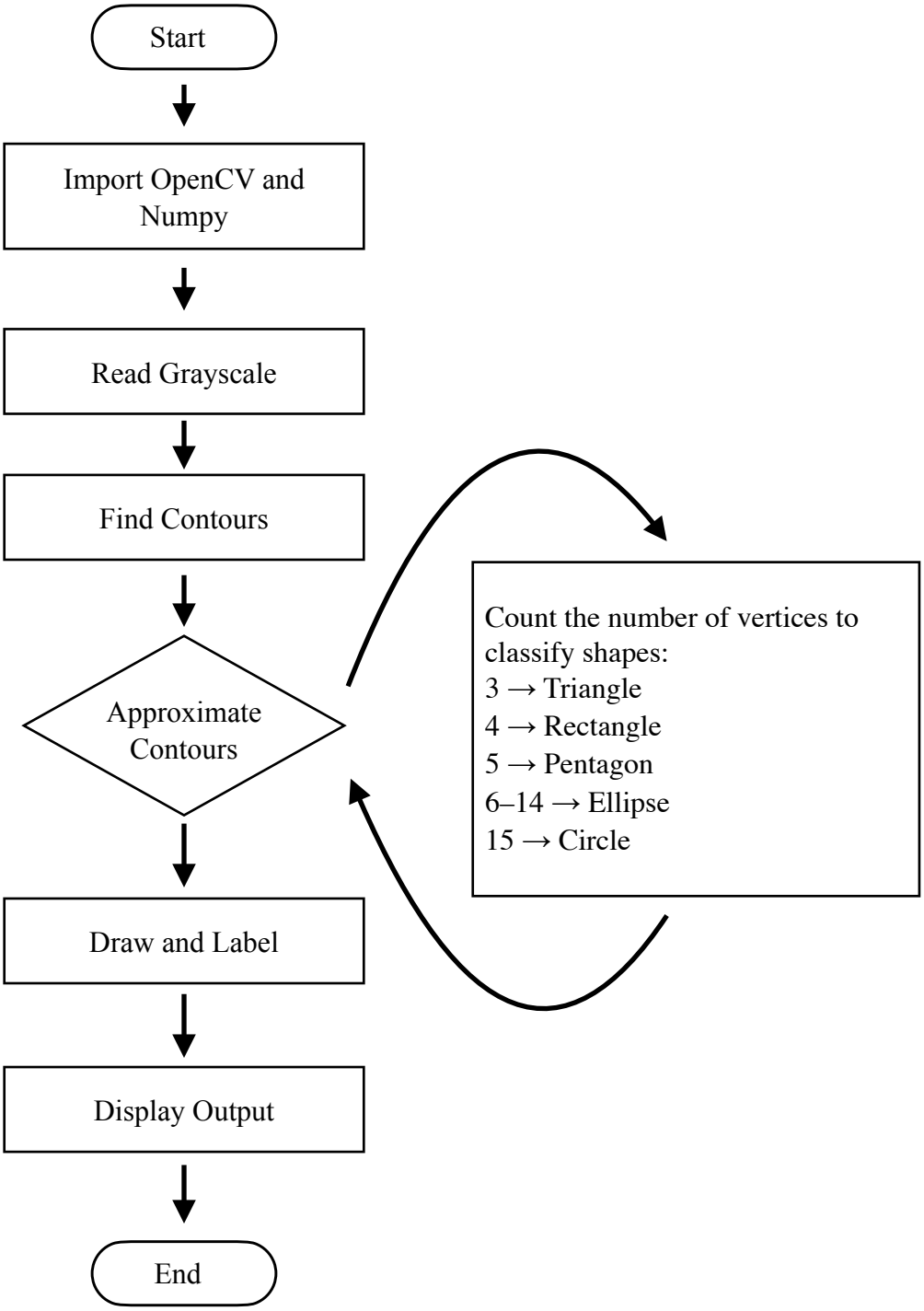
5. In the video processing part, explain why we need to generate separate frames to perform video processing. Assume the video is encoded.

When working with encoded videos, the frames aren't just sitting there like a sequence of images, they're compressed, often using codecs like H.264 or MPEG. That means the pixel data isn't directly accessible until you decode and extract each frame. So to actually apply shape detection or any kind of analysis, we have to convert the video into individual images first.

Once the frames are extracted, it's way easier to process them, we can apply thresholding, contour detection, shape labeling, whatever we need, without worrying about video encoding formats. After all the frames are annotated, we can stitch them back together into a new video.

I saw this firsthand during Phase D. Without separating the frames, we couldn't have cleanly labeled each shape or ensured consistent frame by frame processing. This step isn't just technical overhead, it's necessary to bridge the gap between video compression and per frame computer vision.

2. Draw a flow chart and explain how the shape extraction program in Appendix A works.



The script starts by importing the libraries and reading the shape image in grayscale. We apply thresholding to make the shapes stand out clearly, then detect contours. For each detected shape, we approximate the contour and count how many corner points it has. Based on that, we label it as a triangle, rectangle, ellipse, or circle. The shapes are drawn and labeled directly on the image using OpenCV.

It's simple but effective, everything works just by counting the number of edges and drawing labels. This made it easy to understand and customize during the lab work.

IV. Conclusion

This lab gave me a solid understanding of how different image processing techniques come together to solve real problems. From basic segmentation to shape extraction and finally video frame processing, I got hands on with every step, and saw the output evolve with each phase. Working on Jetson Orin Nano and seeing how GPU acceleration actually speeds things up made this even more real. I learned a lot.

References

1. OpenCV – About
   https://opencv.org/about/

2. NVIDIA CUDA Toolkit
   https://developer.nvidia.com/cuda-toolkit

3. Image Segmentation with Watershed Algorithm
   https://opencv-python-tutorials.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_watershed/py_watershed.html

4. Image Thresholding
   https://opencv-python-tutorials.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_thresholding/py_thresholding.html

5. Canny Edge Detection
   https://opencv-python-tutorials.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_canny/py_canny.html