Student: Aditya Sandhu

University: Colorado State University Global

Course: 25FC - CSC515 - 1 [Module 2 – Image Formation]

Critical Thinking Assignment [OpenCV multi-scale representation of images by pixels matrices]

Instructor: Dr Dong Nguyen

Date – 09/16/2025

GIT LINKS

Document Link –

Python File –

A multi-scale representation of images by pixel matrices involves breaking down an image into multiple levels of resolution or detail, where each level is represented as a matrix of pixel values. This process typically starts with the original image's pixel matrix, which stores intensity or color values (e.g., RGB or grayscale) for each pixel. Through techniques like image pyramids, the image is progressively downsampled or filtered to create smaller matrices representing coarser scales, capturing broader features, while the original matrix retains fine details. This hierarchical approach allows analysis or processing at various scales, such as detecting edges at a detailed level or identifying objects at a lower resolution, enhancing flexibility in computer vision tasks like object recognition or image compression.

Figure 1 – Program of Multi-Level Representation of Puppy Images Using OpenCV

A screenshot of a computer program

AI-generated content may be incorrect.

*import* cv2  
*import* os  
*import* shutil

‘cv2’, is the Python binding of OpenCV (Open Source Computer Vision Library). It loads and saves images with the cv2.imread and cv2.imwrite function and displays with the cv2.imshow. It performs image processing tasks like splitting into channels with the cv2.split and merges images back together with the cv2.merge function. The images are stored in mathematical arrays and OpenCV provides optimized c++ routines for manipulating those arrays at high speed. ‘os’, is pythons system interface module, that handles file system navigation, and for this program, finds where the images are and where to write the output, i.e. Merged images. ‘shutil’ handles file cleanup and copying, i.e. moving clean images to the ‘clean’ folder.

*def* organize\_valid\_photos(base\_path, source\_folder, target\_folder):  
 *"""Prepare a clean folder with only supported photo types."""* full\_source = os.path.join(base\_path, source\_folder)  
 full\_target = os.path.join(base\_path, target\_folder)  
 os.makedirs(full\_target, exist\_ok=*True*)  
  
 allowed\_types = {'.jpg', '.jpeg', '.png'}  
 *for* item *in* os.listdir(full\_source):  
 \_, file\_type = os.path.splitext(item)  
 *if* file\_type.lower() *in* allowed\_types:  
 origin = os.path.join(full\_source, item)  
 destination = os.path.join(full\_target, item)  
 shutil.copy(origin, destination)

The function ‘organize\_valid\_photos’, prepares cleaned up folder with valid images. And the docstring describes the function as such. ‘full\_source’ is the full path to the source folder by combining the base path with the source folder name. ‘full\_targer’ is the full path to the target (clean) folder where valid images will be stored. And ‘ os.makedirs(full\_target, exist\_ok=True)’, creates the target folder if it doesn’t already exist (exist\_ok=True prevents errors if it does). ‘allowed\_types’, defines a set of allowed file extensions, and the for loop, loops through every file and subfolder name inside the source folder. ‘ \_, file\_type = os.path.splitext(item)’, splits the filename into name and extension, and keeps the extension file type specified earlier. ‘ if file\_type.lower() in allowed\_types:’, checks if the files extension matches one of the allowed types, and ‘origin’ object builds the full path to the original file inside the source folder.

‘destination = os.path.join(full\_target, item)’, object builds the full path for the copy inside the target folder. ‘shutil.copy(origin, destination)’, copies the file from the source folder to the targer folder preserving the content.

current\_base = os.path.dirname(os.path.abspath(\_\_file\_\_))

organize\_valid\_photos(current\_base, ‘puppy\_files’, ‘processed\_puppy\_photos’)

current\_base is the directory where this script is located. And ‘organize\_valid\_photos’, cleans up the ‘puppy\_files’ folder by copying only valid images (.jpg, .jpeg, .png) into a new folder called ‘processed\_puppy\_photos’.

processed\_path = os.path.join(current\_base, ‘processed\_puppy\_photos’)

processed\_path builds the full path to the ‘processed\_puppy\_photos’ directory where cleaned up files are.

available\_photos = []

for f in os.listdir(processed\_path):

file\_extension = os.path.splitext(f)[1].lower()

if file\_extension in {'.jpg', '.jpeg', '.png'}:

available\_photos.append(f)

available\_photos is an empty list object, and the for loop iterates through the files in the processed path folder. Then the code extracts the files with the extensions that are listed in the if statement, and if there is a match, appends the file to the available\_photos list via the append function. The if statement has a ValueError that is printed out to the console if no suitable photos are found in the target folder.

Then the below code selects the first valid photo in the list, and build the full path on that photo. And, the ‘print’, statements prints the f string parameter of ‘selected\_photo’ that will pe selected.

selected\_photo = os.path.join(processed\_path, available\_photos[0])

print(f"Processing this photo: {selected\_photo}")

This chunk of code prepares a clean folder of valid photos, checks for valid image files, ensures at least one exists, and selects the first available one for processing.

Figure 2 - Visualization of Color Channel Extraction and Recombination in Puppy Image Processing with OpenCV

A screenshot of a computer program

AI-generated content may be incorrect.

*# Read the photo*photo\_data = cv2.imread(selected\_photo)  
*if* photo\_data *is None*:  
 *raise* ValueError(f"Could not read the photo at {selected\_photo}")

Photo data object loads the selected image file into memory using OpenCV’s imread function, which represents the photo as a three-dimensional NumPy array of pixel intensities in Blue, Green, and Red (BGR) order. . Each pixel is stored as an 8-bit integer ranging from 0 to 255, corresponding to color intensity. The if statement checks whether the image was successfully read. if imread fails (for example, due to a missing, corrupted, or unsupported file), it returns None. In that case, the program raises a ValueError with a descriptive message, ensuring the workflow halts before attempting to process invalid data.

*# Extract individual color layers*blue\_layer, green\_layer, red\_layer = cv2.split(photo\_data)

This line uses OpenCV’s cv2.split function to separate the loaded image (photo\_data) into its three individual color channels. Each channel is returned as a two-dimensional NumPy array of the same height and width as the original image, but containing only intensity values for a single color: the first array corresponds to the blue layer, the second to green, and the third to red. Together, these grayscale matrices represent how much of each primary color contributes to every pixel in the full-color image.

# Show each layer in gray

cv2.imshow('Blue Layer View', blue\_layer)

cv2.imshow('Green Layer View', green\_layer)

cv2.imshow('Red Layer View', red\_layer)

These three display windows, one for each color channel extracted from the image. Since each channel is a 2D matrix of intensity values, OpenCV renders them as grayscale images where brighter areas indicate stronger presence of that color. The blue channel window shows how much blue contributes to each pixel, the green channel shows the green intensities, and the red channel shows the red intensities, making it easy to visualize the distribution of each primary color separately.

*# Recombine to form the initial photo*recombined\_photo = cv2.merge([blue\_layer, green\_layer, red\_layer])  
cv2.imshow('Recombined Original Photo', recombined\_photo)

This merges the three color layers back into one full-color image and displays it in a window, reconstructing the original photo from its channels.

*# Create variant by exchanging green and red layers*variant\_photo = cv2.merge([blue\_layer, red\_layer, green\_layer])  
cv2.imshow('Variant Photo with Layers Exchanged', variant\_photo)

This swaps the red and green channels, producing a color-shifted version of the photo and displaying it in a new window.

*# Hold displays until a key is hit*cv2.waitKey(0)  
cv2.destroyAllWindows()

This keeps all image windows open until a key is pressed, then closes them to end the display session.

Figure 3 - Grayscale Visualization of Individual Color Channels in Puppy Image Processing Using OpenCV

A screenshot of a computer

AI-generated content may be incorrect.

The Figure represents the grayscale outputs of the blue, green, and red color channels extracted from a puppy photograph, captured in a wet outdoor setting, as processed by an OpenCV program. Labeled as "Blue Layer View," "Green Layer View," and "Red Layer View," each window displays the intensity distribution of its respective channel, revealing how the puppy's fur and background contribute to the overall RGB composition.

Figure 4 - Comparative Display of Recombined and Variant Color Manipulations in Puppy Image Processing with OpenCV

A screenshot of a computer screen

AI-generated content may be incorrect.

This Image showcases the visual outcomes the program, using OpenCV to manipulate a puppy photograph taken in a wet outdoor environment, with two side-by-side windows labeled "Recombined Original Photo" and "Variant Photo with Layers Exchanged." The recombined image restores the puppy's natural golden-brown fur and surrounding hues by merging blue, green, and red channels in their original order, while the variant image swaps the green and red channels, resulting in a striking, green-tinted puppy against a purple-hued background.

The script successfully demonstrates multi-scale image representation by processing a puppy photo at both pixel-level detail and broader color scales using OpenCV. It effectively organizes files, extracts and manipulates RGB channels, and visualizes results, enhancing understanding of image composition. The creation of a channel-swapped variant highlights practical color manipulation techniques. This approach supports educational goals in computer vision, showcasing efficient array handling and error management. Overall, it provides a solid foundation for exploring advanced image processing tasks.

**References**

Bradski, G. (2000). The OpenCV library. *Dr. Dobb’s Journal of Software Tools*. Retrieved from <https://opencv.org/>

GeeksforGeeks. (n.d.). *How to install OpenCV for Python in Windows*. Retrieved September 16, 2025, from <https://www.geeksforgeeks.org/python/how-to-install-opencv-for-python-in-windows/>

Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., … Oliphant, T. E. (2020). Array programming with NumPy. *Nature, 585*(7825), 357–362. <https://doi.org/10.1038/s41586-020-2649-2>

OpenCV. (n.d.). *Structure from motion (SfM) — scene reconstruction*. In *OpenCV documentation (v4.x)*. Retrieved September 16, 2025, from <https://docs.opencv.org/4.x/d4/d18/tutorial_sfm_scene_reconstruction.html>

OpenCVHelp. (n.d.). *Advanced reconstruction tutorial*. Retrieved September 16, 2025, from <https://www.opencvhelp.org/tutorials/advanced/reconstruction-opencv/>

Python Software Foundation. (2025). *Python 3.12 documentation*. Retrieved from <https://docs.python.org/3/>

Stack Overflow. (2018, August 23). *How to 3D reconstruct robustly from multiple images with known poses in OpenCV?* Retrieved September 16, 2025, from <https://stackoverflow.com/questions/51991271/how-to-3d-reconstruct-robustly-from-multiple-images-with-known-poses-in-opencv>