Aim: To perform Handling Files, Cameras and GUIs

Objective: To perform Basic I/O Scripts, Reading/Writing an Image File, Converting Between an Image and raw bytes, Accessing image data with numpy.array,Reading /writing a video file, Capturing camera, Displaying images in a window ,Displaying camera frames in a window

Theory:

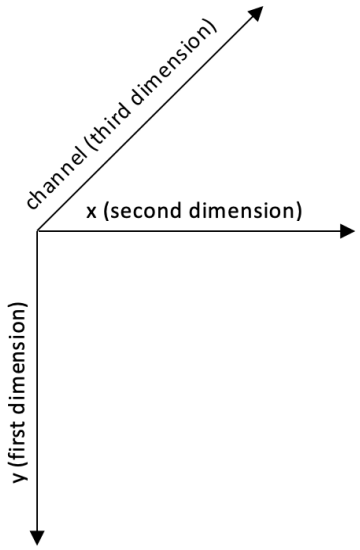
Basic I/O script : Most CV applications need to get images as input. Most also produce images as output. An interactive CV application might require a camera as an input source and a window as an output destination. However, other possible sources and destinations include image files, video files, and raw bytes. For example, raw bytes might be transmitted via a network connection, or they might be generated by an algorithm if we incorporate procedural graphics into our application. Let's look at each of these possibilities.

Reading/Writing an Image File : OpenCV provides the imread function to load an image from a file and the imwrite function to write an image to a file. These functions support various file formats for still images (not videos). The supported formats vary—as formats can be added or removed in a custom build of OpenCV—but normally BMP, PNG, JPEG, and TIFF are among the supported formats.

Converting Between an Image and raw bytes : Conceptually, a byte is an integer ranging from 0 to 255. Throughout real-time graphic applications today, a pixel is typically represented by one byte per channel, though other representations are also possible.

An OpenCV image is a 2D or 3D array of the numpy.array type. An 8-bit grayscale image is a 2D array containing byte values. A 24-bit BGR image is a 3D array, which also contains byte values. We may access these values by using an expression such as image[0, 0] or image[0, 0, 0]. The first index is the pixel's y coordinate or row, 0 being the top. The second index is the pixel's

x coordinate or column, 0 being the leftmost. The third index (if applicable) represents a color channel. The array's three dimensions can be visualized in the following Cartesian coordinate system:



For example, in an 8-bit grayscale image with a white pixel in the upper-left corner, image[0, 0] is 255. For a 24-bit (8-bit-per-channel) BGR image with a blue pixel in the upper-left corner, image[0, 0] is [255, 0, 0].

Accessing image data with numpy. Array : We already know that the easiest (and most common) way to load an image in OpenCV is to use the imread function. We also know that this will return an image, which is really an array (either a 2D or 3D one, depending on the parameters you passed to imread).

The numpy.array class is greatly optimized for array operations, and it allows certain kinds of bulk manipulations that are not available in a plain Python list. These kinds of numpy.array type-specific operations come in handy for image manipulations in OpenCV.

Reading/Writing a video file : OpenCV provides the VideoCapture and VideoWriter classes, which support various video file formats. The supported formats vary depending on the operating system and the build configuration of OpenCV, but normally it is safe to assume that the AVI format is supported. Via its read method, a VideoCapture object may be polled for new frames until it reaches the end of its video file. Each frame is an image in a BGR format.

Conversely, an image may be passed to the write method of the VideoWriter class, which appends the image to a file in VideoWriter

Capturing camera frames: A stream of camera frames is represented by a VideoCapture object too. However, for a camera, we construct a VideoCapture object by passing the camera's device index instead of a video's filename. Let's consider the following example, which captures 10 seconds of video from a camera and writes it to an AVI file.

Displaying images in a window : One of the most basic operations in OpenCV is displaying an image in a window. This can be done with the imshow function. If you come from any other GUI framework background, you might think it sufficient to call imshow to display an image. However, in OpenCV, the window is drawn (or re-drawn) only when you call another function, waitKey. The latter function pumps the window's event queue (allowing various events such as drawing to be handled), and it returns the keycode of any key that the user may have typed within a specified timeout. To some extent, this rudimentary design simplifies the task of developing demos that use video or webcam input; at least the developer has manual control over the capture and display of new frames.

Displaying camera frames in a window : OpenCV allows named windows to be created, redrawn, and destroyed using the namedWindow, imshow, and destroyWindow functions. Also, any window may capture keyboard input via the waitKey function and mouse input via the setMouseCallback function.

Conclusion:

This experiment explored OpenCV's capabilities in handling files, cameras, and GUIs. It covered reading/writing image files, converting between images and raw bytes, and accessing image data with numpy.array. Video files were processed using VideoCapture and VideoWriter classes, and camera frames were displayed in windows. OpenCV's versatility in various data sources and GUI interactions was demonstrated, making it essential for computer vision applications. Understanding file I/O and image processing is crucial for developing interactive computer vision tools. Overall, this experiment provided practical insights into utilizing OpenCV for efficient image and video processing in computer engineering applications.