Detecting Colors in a video

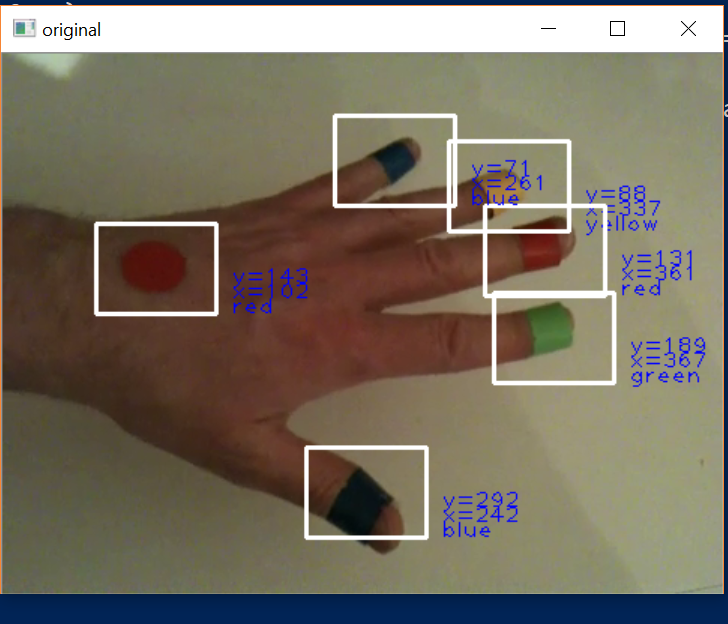
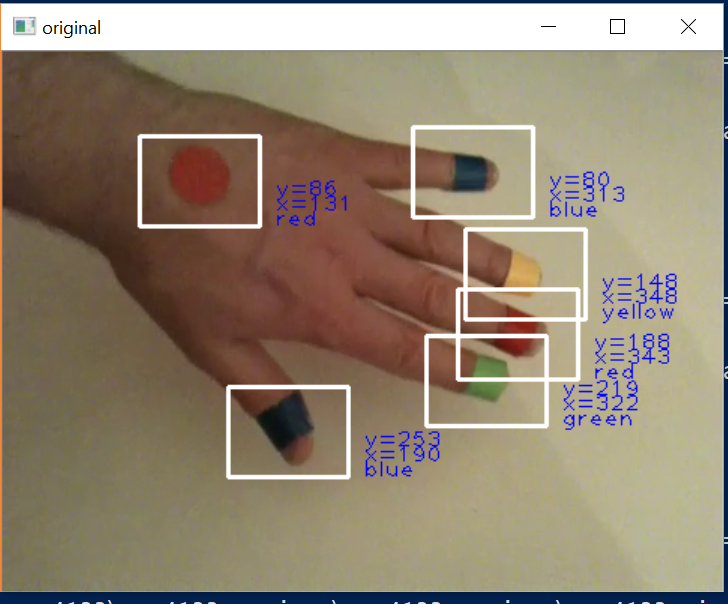
**Intro:**

This document serves as a brief introduction into basic image processing techniques used to detect specific colors in a video. As well, as the steps taken in order to use these techniques. Hopefully, you may become interested in image processing and computer vision and use these practices for your own benefit in the future. Finally, all references to methods and theory can be found at the end of the document. This procedure also assumes the reader has OpenCV already installed as a library to be used in Python. Check out the sample videos to test out your own implementation.

NOTE: This project was completed using OpenCV library in Python so a different approach may be necessary to replicate this in C++.

**Procedure:**

* The first step is to determine which colors you would like to detect.
  + For the sake of this sample video only: green, yellow, red, blue will be considered.
  + There is a multitude of online resources to accurately find a bound for these colors, however in the interest of time some sample upper and lower bounds are provided in the ‘Reference’ section (1).
  + Note: These upper and lower bounds are NOT ABSOLUTE so they may not work perfectly for other sample videos. Finding suitable bounds for different types of videos can prove a fickle task.
* Import the necessary libraries, **OpenCV** as well a **numpy** are necessary for this procedure.
  + ***Import numpy****:* imports a few math functions which will be useful later.
    - See reference (2) for more information on numpy library.
  + ***Import cv2:*** imports the OpenCV library assuming it is already installed.
* For this implementation two differently sized kernels were used: a small kernel of size (1,1) and a large kernel of size (2,2)
  + *Np.ones((2,2),np.uint8):* builds the larger kernel of a matrix of ones.
  + *Np.ones((1,1),np.uint8):* builds the smaller kernel of a matrix of ones.
    - Kernels are useful for applying filters to our images, specifically eroding and dilating. See reference (3) for more information on filter kernels.
* Load the video and save to a global variable. **Loop through each frame of the video.**
  + *Cv2.videoCapture(‘video directory’)* is a basic OpenCV method to return a VideoCapture object from a locally stored video.
  + Combining a *while loop* and *isOpened()* method of the video capture object allows each frame to be looped through for the duration of the video.
* Convert frame to **HSV color space.**
  + *Cv2.cvtColor()* method converts the image from its default color space (BGR) to HSV.
    - Additional documentation about the HSV color space can be found at reference (4).
* Create color masks for each of the colors **using their respective boundary conditions.**
  + *Cv2.inRange()* returns a black and white image where the white pixels fall within the upper and lower bounds, and black otherwise
    - See reference (5) for more information on inRange() method.
* A mask must be created for each color that needs to be detected. So, for this case 4 color are being detected: green, yellow, red, blue.
* **Erode then dilate the masks.** 
  + *Cv2.erode(), and cv2.dilate()* are suitable OpenCV methods to accomplish this respectively. This technique is known as *opening.*
  + This step requires some testing to accurately detect each color using the two different kernels as well as the number of iterations. Suitable parameters of the erosion and dilation are chosen correctly when the masks exclusively display the color they are intended to detect.
    - See reference (6) for more information on eroding and dilating.
* **Find the contours for each color mask.** 
  + Apply the *cv2.findContours()* method to draw all the contours.
  + Contours help to connect pixels located in a close enough neighbourhood to one another.
* For each of the detected contours (for each color mask) draw a rectangle using *cv2.rectangle()*.
  + See reference (7) for more information on drawing rectangles in OpenCV.
  + Only draw the contour if it is larger than a certain radius. This is to avoid drawing circles that are too small.
    - *Cv2.minEnclosingCircle(contour)* can help to identify a sufficiently small circle.
      * see reference (8) for more information on minEnclosingCircle()
* These rectangles are drawn onto the original frame and then displayed.
  + The simple cv*2.imshow()* method for each frame is sufficient.
* Sample images are provided below:

**References**

1. **Boundary colors in HSV color space.** These bounds were obtained through rigorous testing and multiple online resources to pinpoint a sufficient range.
   * Lower bound green [45,100,100]; upper bound green [75,255,255]
   * Lower bound yellow [19,100,100]; upper bound yellow [30,255,255]
   * Lower bound red [0,151,50]; upper bound red [4,255,255]
   * Lower bound blue [95,70,0]; upper bound blue [100,255,255]
2. Numpy official website: <http://www.numpy.org/>
3. Linear filtering and building kernels: <https://docs.opencv.org/2.4/doc/tutorials/imgproc/imgtrans/filter_2d/filter_2d.html>
4. Color spaces including HSV: <https://www.learnopencv.com/color-spaces-in-opencv-cpp-python/>
5. How to use cv2.inRange() with color boundaries: <http://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_colorspaces/py_colorspaces.html>
6. Morphological transformations including erosion and dilation: <https://docs.opencv.org/3.0-beta/doc/py_tutorials/py_imgproc/py_morphological_ops/py_morphological_ops.html>
7. Drawing functions in opencv including rectangles: <https://docs.opencv.org/3.1.0/dc/da5/tutorial_py_drawing_functions.html>
8. Contouring as well as minimum enclosing circle: <https://docs.opencv.org/3.1.0/dd/d49/tutorial_py_contour_features.html>