PIC 16, Winter 2018 – Preparation 9F

Assigned 3/5/2018. To be completed by class 3/9/2018.

Intended Learning Outcomes

By the end of this preparatory assignment, students should be able to:

- load, view, and save images and videos using OpenCV-Python,
- manipulate images, or frames from videos, as a NumPy array,
- work in the HSV (Hue, Saturation, Value) color space as an alternative to the familiar RGB (Red, Green, Blue) color space (and its twin, BGR),
- generate contours from binary images/frames and draw them over existing images/frames.

Tasks

- ☐ Skim the Introduction to OpenCV-Python Tutorials.
- ☐ (Optional) Install OpenCV on your personal computer. At the command prompt, try this:
  `conda install -c conda-forge opencv`
  The `-c conda-forge` specifies a location where OpenCV can be found that `conda` doesn't usually check.
  If that doesn't work, instructions for Windows are available here (you should be "Installing OpenCV from prebuilt binaries"). The OpenCV tutorial provides instructions for Fedora here. If you have OS X, attempt install at your own risk. I broke my Anaconda installation before getting it to work, and simple things – like closing windows that show images – still don't work. Some additional notes for Windows users:
  - After downloading OpenCV from sourceforge and extracting it, I moved the resulting folder (opencv) to:
    C:\Users\Matt\Anaconda2\Lib\site-packages\
    Then I copied cv2.pyd from:
    C:\Users\Matt\Anaconda2\Lib\site-packages\opencv\build\python\2.7\x64
    To
    C:\Users\Matt\Anaconda2\Lib\site-packages\
    Of course, the exact paths on your machine will differ but I thought examples path names for an Anaconda installation might be helpful.
  - I was able to import `cv2` in Python after that, but in the tutorials I realized I needed to set up `ffmpeg`, software for encoding and decoding video files. On my 64-bit machine with OpenCV 3.1.0, the file I needed is called `opencv_ffmpeg310_64.dll` and it was located in:
    C:\Users\Matt\Anaconda2\Lib\site-packages\opencv\build\x64\vc12\bin\
    If you have a 32-bit machine, you'll want the file from x86\vc12\bin\ folder.
    I copied it to the root of my Anaconda installation to make it easier for OpenCV to find:
    C:\Users\Matt\Anaconda2\
  - If you run into trouble opening video from files later and there is no error message, try searching for something like "opencv ffmpeg <operating system name> python". The best answer I could find for Windows was here.
- ☐ Follow the Getting Started with Images tutorial, which explains some of the most important OpenCV functions: `imshow`, `waitKey` (which must be used for `imshow` to work), and `destroyAllWindows`. You can use my scaled-down version of the picture in the tutorial, messi.jpg. Note that the image is simply a NumPy array, so everything you learned about NumPy before is applicable!

- Next, follow the Getting Started with Videos tutorial. You can skip the section on capturing video from a camera if you're not interested. The class only deals with video from files.
  You can use ballbounce.mp4, which I downloaded from here. By default OpenCV makes the frame larger than my monitor can handle, so I scaled each frame down to 25% size right after loading it using the instruction at the top of the Geometric Transformations on Images tutorial. You can also make the window size adjustable with what you learned in the last section. Anything that you learn for images works for frames of video!
  When running these examples, you need to press the 'q' key to quit (see code…). Closing the window with the mouse won't work because this doesn't end the `while` loop, which will open up a new window for the next frame. Note that each frame of video is a NumPy array, just like a regular image. How convenient! Try mirroring the video *horizontally* by (flipping each frame individually before `imshow`ing it) using slicing instead of the `flip` function.
  There are a few parts of the code I found confusing. For instance:
  - apparently `cv2.waitKey(1) & 0xFF == ord('q')` returns `True` (and therefore breaks the while loop) when the q key is pressed. This makes sense if you consider:
    - `cv2.waitKey(1)` returns the ASCII character code of the key pressed,
    - `& 0xFF` performs a bit-wise and operation with Hexidecimal value `FF`,
    - Hex value `FF` is binary `11111111`, so the bitwise-and leaves the value returned by `waitKey` unchanged (I have no idea why they do it), and
    - `ord('q')` returns the ASCII character code of "q".

    So you can just write `cv2.waitKey(1) == ord('q')`. It works fine on my machine.
  - In the line `fourcc = cv2.VideoWriter_fourcc(*'XVID')`, the asterisk splits the string `XVID` into four separate arguments. (See second response here and documentation for "Unpacking Argument Lists" here.) I agree that this syntax is strange; it probably has to do with `cv2` being a wrapper for OpenCV, which is actually written in C++.
  - In the line `ret, frame = cap.read()`, `ret` is `True` when the read is successful and `False` if the end of the file has been reached. This is useful for stopping a loop at the end of a video.

  If you have trouble loading/saving video files, make sure you've set up ffmpeg as described above. The PIC computers are already set up. If you're having trouble saving video on a PIC computer, it's much more likely there is a problem with your code. Check that:
  - You have specified the correct resolution of your frames. Note that the *width* (number of columns) is the first element of the tuple, followed by *height* (number of rows). This is opposite what you might expect.
  - If you have converted frames to grayscale, you need to pass `False` as a fifth argument when creating the `VideoWriter`. The default is `True` for color frames. This is not clear in the documentation.

  OpenCV can be annoying in that it often *won't* raise exceptions; it simply won't do what you are trying to do (silently). This can make debugging challenging, so I suggest that you start with the example code provided (changing only the name of the input video file and the frame dimensions) and get that working before making changes.
- Follow the Changing Colorspaces tutorial.
  When we worked with images in NumPy, we usually used the RGB (Red-Green-Blue) color space. You may have noticed that OpenCV loads images in a BGR colorspace; this is essentially the same as the RGB colorspace, just that order in which the colors are specified is reversed. You can convert between the two very easily with some simple array slicing.
  We've seen that there are other ways of representing colors with three numbers. One of the most important is HSV (Hue-Saturation-Value) because in this color space only one number, the hue, affects what we perceive to be as the color itself. The other numbers are for how bold and bright the color is. This makes it somewhat more intuitive to work with.

- Try to isolate the *green* balls in the video (remove all the others). We did basically the same thing to remove the green background from the minion in Assignment 6W using pure NumPy, and you could use those techniques to do this. However, OpenCV has some handy functions to make the task a little simple.
    - First, you'll need to find the right lower and upper bounds for the hue. The tutorial suggests using the `cvtColor` function. That works well for this particular video, but it doesn't work if you don't already know the RGB (BGR) values of the color. In this case, you can take a screenshot, opening the image in Paint (the most basic Windows image editing software), use the color picker tool (that looks like an eyedropper) to select a pixel with the desired color, and record the hue value in the "Edit Colors" dialog. Note that the hue values in Paint go from 0 to 239 whereas in OpenCV they range from 0 to 179; so you'll need to scale the value from Paint into the range used by OpenCV before entering the number in your code.
    - Next, you'll use the `inRange` function to select the pixels with colors that fall within the lower and upper hue bounds. The `inRange` function returns an array with the same dimensions as the original image, but with pure black (0,0,0) in place of all pixels outside the range and pure white (255,255,255) in place of all pixels within the range. (Use `imshow` on the mask to see.) This is sometimes called a binary or threshold image, and is called a mask in this example.
    - `cv2.bitwise_and` is a little complicated. We can talk about exactly what is going on in class if you're interested. You can see how it's used and what it does, however. When you provide the original image as the first *two* inputs and the mask as the third input, it returns an image with black pixels everywhere the mask is black and the original colors everywhere the mask is white.
- The tutorial assigns the exercise of selecting objects of several colors simultaneously. I suggest trying this for the yellow or blue balls. The red balls are tricky because they are actually composed of both red and purple hues, which are at opposite ends of the hue range. (Hint: get separate masks for red and yellow balls, then use `bitwise_or` to combine them, and apply this mask to the original image with `bitwise_and`.)
- Next, read the Contours: Getting Started tutorial. You can read about the `threshold` function in the Image Thresholding tutorial. Its job, like `inRange`, is to convert a color image to a binary image / mask. However it does so purely based on the brightness of each pixel (of a grayscale version of the image) rather than on the basis of color.
- Try drawing red contours around green balls in an image. (Hint: follow the example, but use `inRange` rather than `threshold`. Also, it looks like they made a typo in the first argument to the `drawContours` function. It needs to be the image on which the contours are to be drawn, which would be `im` in the example. Finally, tutorial doesn't mention it, but after you call `drawContours`, you have to `imshow` the image, complete with calls to `waitKey` and `destroyAllWindows` as usual. If you can't get this, review the Preparation 9W notebook during lecture.)
- Finally, use the same technique to draw contours around the green bouncing balls in the (color) video.
- Do something fun with what you've learned here. Suggestions:
    - Take a video of yourself in front of a solid-color background, replace the background with another image (or video!), and save the new video. A little "halo" around you is OK; it doesn't have to be perfect.
    - Take a video of yourself with a "sword" of a solid-color that doesn't appear in the background. Replace the sword with a much brighter color to make it look (somewhat) like a lightsaber, and save the video.

  Show me during class.