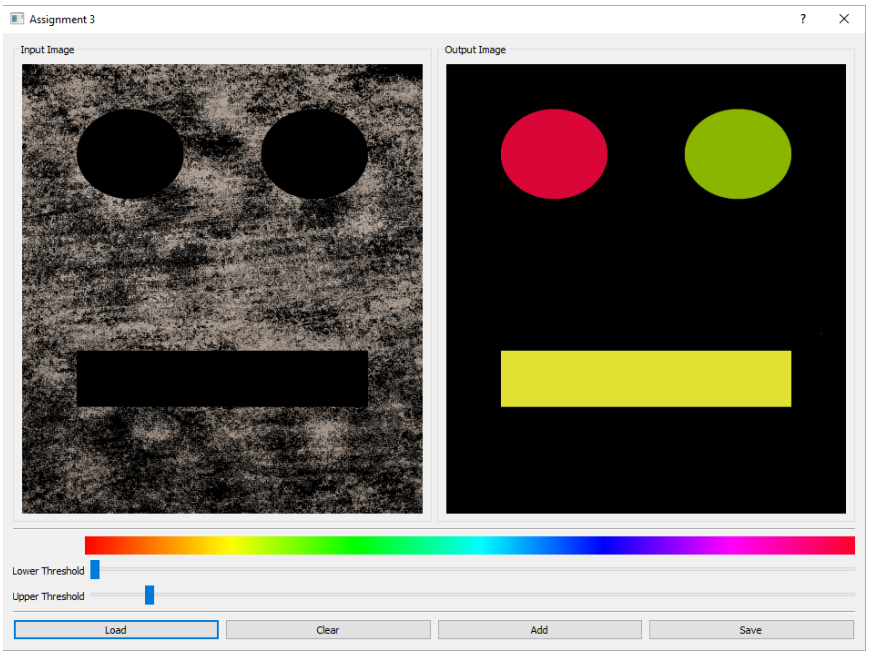
282.762 Robotics and Automation: Assignment 3

Chris Diggle

May 3, 2019

OpenCV image processing and QT interfaces



Contents

1 Introduction

This report will be describing the functionality of my GUI based image thresholding program. The git can be found on https://github.com/Flawededge/OpenCV_Python

The goal of this program was to demonstrate an ability to process images using OpenCV using a threshold function along with an interactive interface made using the QT framework. I have done this by making an app which displays a GUI. The GUI loads an image and then can threshold out different colours by means of thresholding.

To threshold I used HSV instead of the default BGR. This is due to the image we were given to threshold had very clear colors making it possible to threshold with only the hue. If a BGR image was used, the upper and lower threshold of the blue, green and red channel would need to have a slider, as the colours were with HSV, the saturation and the value are irrelevant for the context of this thresholding so only 2 sliders for lower and upper hue are needed.

I decided to use python to make the interface, as I am familiar with PyQt, so I can make a more interactive interface quickly.

2 Interface

The interface for this app is simple and clear. It was made in QT Designer, which is a lightweight designer for QT. This program outputs a .ui file, which can be read directly by the PyQt5 library, but I opted to convert it to a python friendly version using pyuic, which is a module built into PyQt5 to convert ui modules.

```
pyuic5 mainwindow.ui > mainwindow.py
```

This shell command converts mainwindow.ui into mainwindow.py, which allows me to link to signals in the ___init___ function instead of setting up signals and slots. It also allows me to directly interact with the widgets in the interface with autocomplete in the IDE. Overall this makes the converted file far easier to work with and makes the environment more flexible for me.

The two sliders indicate the lower and upper threshold for the hue and the 4 buttons on the bottom are clear in function by reading their name and pressing them.

- Lower Threshold - Adjusts the lower hue threshold
- Upper threshold - Adjusts the upper threshold
- Load [2.2.1] - Loads a new image
- Add [2.2.2] - Adds the currently shown image on the input image to the output
- Clear [2.2.3] - Clears the output image
- Save [2.2.4] - Saves the output image to output.png

2.1 sliders

I have thought about the quality of life with the interface with the sliders, as they are the main point of interaction. When trying to make the upper and lower slider cross, which would make the threshold function not work I made it so the opposing slider starts to move. This is achieved with the code below:

Inside the initialization function

```
def __init__(self, dialog):  # The initialization function for the GUI
    {...}
    self.sLower.sliderMoved.connect(self.upper_slider)  # Attach sliders to update the input image
    self.sUpper.sliderMoved.connect(self.lower_slider)
    {...}
```

This links the upper_slider and lower_slider functions to the action of the slider's value changing. These functions are effectively the same, but are opposite to each other to allow one slider to drag the other one. After they check the position of the sliders, the thresholded input image is recalculated and displayed [3.3]

```
def upper_slider(self):  # Function to keep the slider values consistent
    if self.sLower.value() > self.sUpper.value():  # If the sliders have crossed
```

```
        self.sUpper.setValue(self.sLower.value())  # Update the lower slider to the upper value
    self.update_outline()  # Recalculate threshold

def lower_slider(self):  # Function to keep the slider values consistent
    if self.sLower.value() > self.sUpper.value():  # If the sliders have crossed
        self.sLower.setValue(self.sUpper.value())  # Update the upper slider to the lower value
    self.update_outline()  # Recalculate threshold
```

2.2 Buttons

The four button's function is pretty clear and the implementation is relatively simple.

2.2.1 Load

The load button is named 'bLoad' in the GUI. Inside the init it is linked to the load_file class function. This button is made to load images which aren't the default example.png.

This is accomplished by loading up a file dialog and letting the user browse for a file. The file dialog uses the tkinter, which is one of the native python libraries for dealing with GUI interfaces. By default when running the file dialog a small empty tkinter GUI window shows up in addition to the file interface, which does not disappear after the file dialog disappears. I got around this by manually creating a tkinter interface and then withdrawing it immediately after so the user never sees it. After this I can safely open the file dialog without the extra window showing up.

After the file dialog is completed, the string is stored in the class variable 'currentImage', then the load_image class function is called. A check is added at the start of load_image was added, as if the file dialog is quit before selecting a file and empty string is returned (") or if a non-image or unsupported image file is selected the check will identify that and return before trying to load. This avoids a possible crash. Since the console is hidden in the release version, the "Incorrect file type!" message is never seen and nothing happens instead. A dialogue box indicating the failed file read would make this error more clear.

```
def __init__(self, dialog):  # The initialization function for the GUI
    {...}
    self.bLoad.pressed.connect(self.load_file)  # Connect load file to the load button
    {...}

def load_file(self):
    root = tk.Tk()  # Make tkinter interface
    root.withdraw()  # Hide tkinter interface
    self.currentImage = filedialog.askopenfilename()  # File dialog
    self.load_image()  # Load image

def load_image(self):  # Loads image from a filename
    if not (self.currentImage.endswith('.png') or self.currentImage.endswith('.jpg')):  # Check if valid fil
        print("Incorrect file type!")  # Display error in console
        return  # No point to load an incorrect file, so return
    {...}  # Load the image
```

2.2.2 Add

The add button is named 'bAdd' in the GUI and in the initialization it is linked to the add_mask function. This button is made to update the output image with the shown data on the input image. Once the sliders are adjusted.

To implement this function I ended up modifying the update_outline function [3.3]. This was the easiest way to use the existing threshold. I gave the function an update_global parameter, which defaults to 0. This means that I don't need to change other parts of the code to implement this.

After the initial threshold, the 'if update_global: statement will be true. The next check if update_global is 1 is so the clear button can function.

The mask is a binary image which holds a binary copy of the image. This image is only true for areas which the user has thresholded and is used for the output image. The mask defaults to None, so an if statement deciding whether to assign or add to the variable is there. Adding the images is done with a bitwise_or. The output image is then calculated with a bitwise_and and then displayed. This function is lightweight enough to run this on the fly, so no output is stored.

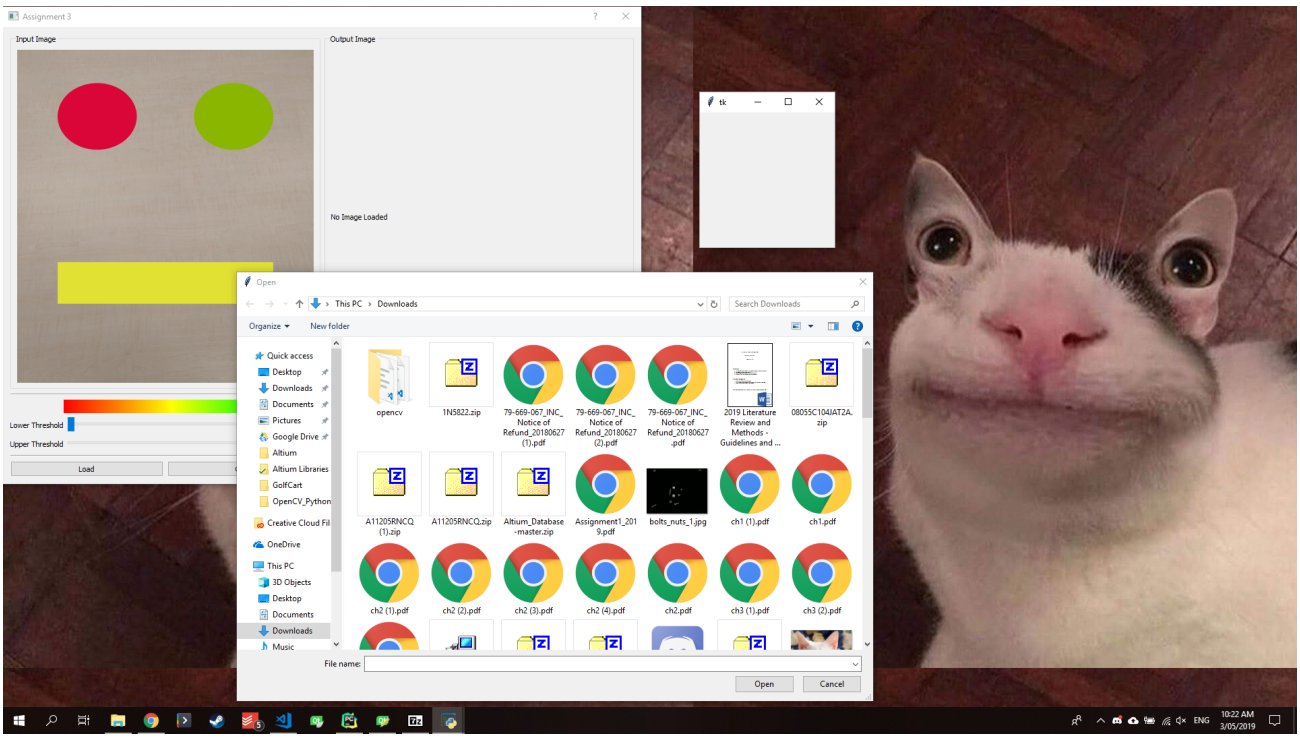


Figure 1: The tkinter interface

```
mask = None

def __init__(self, dialog):
    {...}
    self.bAdd.pressed.connect(self.add_mask)
    {...}

def add_mask(self):
    self.update_outline(1)  # Run the update function with the update_global parameter as 1

def update_outline(self, update_global=0):
    thresh = cv2.inRange(self.curFileImage, (self.sLower.value(), 0, 0), (self.sUpper.value(), 255, 255))  #

    if update_global:  # If add button was pressed
        if update_global == 1:
            if self.mask is None:  # If no global mask currently exists
                self.mask = thresh  # Set global mask
            else:  # If there is a global mask
                self.mask = cv2.bitwise_or(thresh, self.mask)  # Do a bitwise or to add the current mask to

        tmp = cv2.bitwise_and(self.curFileImage, self.curFileImage, mask=self.mask)  # Use the global mask t
        self.disp_image(tmp, 1)  # Display the image on the output panel

    tmp = cv2.bitwise_and(self.curFileImage, self.curFileImage, mask=thresh)  # Use the current mask to gene
    self.disp_image(tmp, 0)  # Display the image on the input panel
```

2.2.3 Clear

The clear button is named 'bClear' in the GUI and the initialization is linked to the clear_mask function. This button is to clear the output back to a blank image when the wrong threshold is displayed.

The first line in the clear_mask button fills the mask with zeroes. This effectively wipes the global mask, resetting the image on the right. update_outline with 2 as the parameter is run to update the shown pictures on the input and output. This works the same as the add button, but instead of the mask being updated with the existing mask, this section is skipped and the output is immediately updated with the newly empty mask. This makes the output image completely black, ready

for new thresholds to be added to it.

```
def __init__(self, dialog):
    {...}
    self.bClear.pressed.connect(self.clear_mask)  # Attach clearing
    {...}

def clear_mask(self):
    self.mask.fill(0)
    self.update_outline(2)

def update_outline(self, update_global=0):
    thresh = cv2.inRange(self.curFileImage, (self.sLower.value(), 0, 0), (self.sUpper.value(), 255, 255))  #

    if update_global:  # If add button was pressed
        if update_global == 1:
            if self.mask is None:  # If no global mask currently exists
                self.mask = thresh  # Set global mask
            else:  # If there is a global mask
                self.mask = cv2.bitwise_or(thresh, self.mask)  # Do a bitwise or to add the current mask to

        tmp = cv2.bitwise_and(self.curFileImage, self.curFileImage, mask=self.mask)  # Use the global mask t
        self.disp_image(tmp, 1)  # Display the image on the output panel

    tmp = cv2.bitwise_and(self.curFileImage, self.curFileImage, mask=thresh)  # Use the current mask to gene
    self.disp_image(tmp, 0)  # Display the image on the input panel
```

2.2.4 Save

The save button is named 'bSave' in the GUI and the initialization is linked to the save_file function. This button is to save the current image to output.png in the running folder. In future I would add a save file dialog to this button, but this function wasn't in the scope of this assignment.

The save_file function starts by calculating the output image in the same way update_outline does by running a bitwise_and with the original image and itself with the mask as the global mask. The image is then converted from HSV to BGR, the default color scheme of OpenCV. After this imwrite is run to save the file to output.png, which saves the file in the running directory.

```
def __init__(self, dialog):
    {...}
    self.bSave.pressed.connect(self.save_file)

def save_file(self):
    tmp = cv2.bitwise_and(self.curFileImage, self.curFileImage, mask=self.mask)
    tmp = cv2.cvtColor(tmp, cv2.COLOR_HSV2BGR)  # Convert to RGB to save
    cv2.imwrite("output.png", tmp)
```

3 Code description

This is a description of all of the primary functions. There are 4 main functions which make up the backbone of the image processing and GUI operations.

3.1 load_image

Load image is used to take a directory and load the relevant image into the curFileImage class variable for the rest of the class to access. This function is only run when needed, as file operations can take a long time depending of the speed of file access. This is crucial in keeping the GUI in feeling fluid to use.

The first check is if the current image string contains a filename with an image in it. This is a simple check to ensure that the GUI doesn't crash, although a crash can be forced by giving a filename with .png at the end without an image actually being at the given location. The print statments are purely for debug, as the console is hidden in the compiled interface and so will not show.

After checking, the file is read using imread and the mask is set to None so it can be regenerated in the size of the new image (mask explained in 3.3). The image is then converted to HSV from BGR. It is done here, as all of the other code in the app needs HSV. This makes it convenient and efficient to convert once when loading and once when saving. The image is then displayed on the left using disp_image 3.2

```
def load_image(self):  # Loads image from a filename
    if not (self.currentImage.endswith('.png') or self.currentImage.endswith('.jpg')):
        print("Incorrect file type!")
        return
    print(f"Loading '{self.currentImage}'")
    self.curFileImage = cv2.imread(self.currentImage)  # Load the image
    self.mask = None
    self.curFileImage = cv2.cvtColor(self.curFileImage, cv2.COLOR_BGR2HSV)  # Convert to HSV
    self.disp_image(self.curFileImage, 0)  # Display the image on the left side
```

3.2 disp_image

disp_image is a function to display a given image on either the left or right panel of the GUI. It takes an OpenCV image in HSV format and a number 0 (left panel) or 1 (right panel) as parameters. This function was intentionally left quite open for input, as I wanted to be able to easily display images on the GUI in one line. This allowed me to keep the code clean and simple instead of dealing with the long and annoying QImage conversion in each function which needs it.

Inside the function, the image is converted to RGB, then converted into a QPixmap. Even though the default format for OpenCV images is BGR, QPixmap uses RGB, so it is cleaner to convert directly to RGB instead of swapping the R and B channels after the image is turned into a QPixmap.

To display the images, there are 2 labels on the GUI. The left one is named inImage and the right one is named outImage. setPixmap is used to apply the pixmap to the

```
def disp_image(self, image, window):  # Shows the given image on 0 (input) or 1 (output)
    image = cv2.cvtColor(image, cv2.COLOR_HSV2RGB)  # Convert to RGB to display
    pix = QtGui.QPixmap(QtGui.QImage(image, image.shape[1], image.shape[0],
                                     image.shape[1] * 3, QtGui.QImage.Format_RGB888))
    if window:
        self.outImage.setPixmap(pix)
    else:
        self.inImage.setPixmap(pix)
```

3.3 update_outline

The update_threshold function is made to do all of the image processing in one function. It has one parameter, which defaults to 0 and also can be 1 or 2 to define what it does.

The first thing this function does is a threshold of the original image. The upper and lower bounds of the hue in the threshold is dictated by the value of the two sliders of the GUI and are collected by getting the .value() of the sliders.

After this there is a check if update_global is 0. If it is anything but 0, there is another check to see if update_global is 1. If this is true the mask is updated. The mask is a binary image which is the same size as the original image. It's

function is to give a global mask to decide what parts of the original image the user wants. If the mask is currently None, this indicated that the mask has never been made for this image, so the binary threshold is equated to the mask, giving the mask a size of the current image. Every time a new image is loaded, the mask is set back to None [3.1] so it can be assigned a new size. If mask has already been set, and is not None, then a bitwise_or is applied with the current threshold and the mask to add them together and update the mask with the new thresholded pixels.

After this the global mask is applied to the original image using a bitwise_and to only show the pixels which the user has added, then the image is displayed on the output (right)

Finally the input (left) panel is updated with only the original threshold. This allows the user to get a live preview of what is added if the add button is pressed. This section is run regardless of the update_global.

```
def update_outline(self, update_global=0):
    thresh = cv2.inRange(self.curFileImage, (self.sLower.value(), 0, 0), (self.sUpper.value(), 255, 255))  #

    if update_global:  # If add button was pressed
        if update_global == 1:
            if self.mask is None:  # If no global mask currently exists
                self.mask = thresh  # Set global mask
            else:  # If there is a global mask
                self.mask = cv2.bitwise_or(thresh, self.mask)  # Do a bitwise or to add the current mask to

        tmp = cv2.bitwise_and(self.curFileImage, self.curFileImage, mask=self.mask)  # Use the global mask t
        self.disp_image(tmp, 1)  # Display the image on the output panel

    tmp = cv2.bitwise_and(self.curFileImage, self.curFileImage, mask=thresh)  # Use the current mask to gene
    self.disp_image(tmp, 0)  # Display the image on the input panel
```

Appendices

A Full main code

```
# # Compile console code
pyinstaller -y -w -i "C:/Users/Ben/PycharmProjects/OpenCV_Python/example.ico" --add-data
"C:/Users/Ben/PycharmProjects/OpenCV_Python/example.png";"." --add-data "C:/Users/Ben/PycharmProjects/OpenCV_
"C:/Users/Ben/PycharmProjects/OpenCV_Python/main.py"

import sys  # Forgot where I actually used this, but it's here
from PyQt5 import QtWidgets, QtGui  # QT stuff
from mainwindow import Ui_OpenCVThresholder  # Import UI
import cv2  # OpenCV
import tkinter as tk            # For file dialog
from tkinter import filedialog  # For file dialog


class MainPlotGui(Ui_OpenCVThresholder):

    # The class variables which are used to pass things around
    currentImage = "example.png"
    curFileImage = None
    mask = None

    def __init__(self, dialog):
        Ui_OpenCVThresholder.__init__(self)  # Set up the GUI
        self.setupUi(dialog)

        self.lColorBar.setPixmap(QtGui.QPixmap("HSV.png"))  # For the pretty bar in the middle of the interf

        self.load_image()  # Load the image and put onto the GUI
        self.update_outline()

        # Connecting GUI elements to functions
        self.sLower.sliderMoved.connect(self.upper_slider)  # Attach sliders to update the input image
        self.sUpper.sliderMoved.connect(self.lower_slider)
        self.bClear.pressed.connect(self.clear_mask)  # Attach clearing
        self.bAdd.pressed.connect(self.add_mask)
        self.bLoad.pressed.connect(self.load_file)
        self.bSave.pressed.connect(self.save_file)

    def load_image(self):  # Loads image from a filename
        if not (self.currentImage.endswith('.png') or self.currentImage.endswith('.jpg')):
            print("Incorrect file type!")
            return
        print(f"Loading '{self.currentImage}'")
        self.curFileImage = cv2.imread(self.currentImage)  # Load the image
        self.mask = None
        self.curFileImage = cv2.cvtColor(self.curFileImage, cv2.COLOR_BGR2HSV)  # Convert to HSV
        self.disp_image(self.curFileImage, 0)  # Display the image on the left side

    def disp_image(self, image, window):  # Shows the given image on 0 (input) or 1 (output)
        image = cv2.cvtColor(image, cv2.COLOR_HSV2RGB)  # Convert to RGB to display
        pix = QtGui.QPixmap(QtGui.QImage(image, image.shape[1], image.shape[0],
                                         image.shape[1] * 3, QtGui.QImage.Format_RGB888))
        if window:
            self.outImage.setPixmap(pix)
        else:
            self.inImage.setPixmap(pix)
```

```
    def upper_slider(self):  # Function to keep the slider values consistent
        if self.sLower.value() > self.sUpper.value():
            self.sUpper.setValue(self.sLower.value())
        self.update_outline()

    def lower_slider(self):  # Function to keep the slider values consistent
        if self.sLower.value() > self.sUpper.value():
            self.sLower.setValue(self.sUpper.value())
        self.update_outline()

    def update_outline(self, update_global=0):
        thresh = cv2.inRange(self.curFileImage, (self.sLower.value(), 0, 0), (self.sUpper.value(), 255, 255))
        if update_global:
            if update_global == 1:
                if self.mask is None:
                    self.mask = thresh
                else:
                    self.mask = cv2.bitwise_or(thresh, self.mask)

            tmp = cv2.bitwise_and(self.curFileImage, self.curFileImage, mask=self.mask)
            self.disp_image(tmp, 1)

        tmp = cv2.bitwise_and(self.curFileImage, self.curFileImage, mask=thresh)
        self.disp_image(tmp, 0)

    def clear_mask(self):
        self.mask.fill(0)
        self.update_outline(2)

    def add_mask(self):
        self.update_outline(1)

    def load_file(self):
        root = tk.Tk()
        root.withdraw()
        self.currentImage = filedialog.askopenfilename()
        self.load_image()

    def save_file(self):
        tmp = cv2.bitwise_and(self.curFileImage, self.curFileImage, mask=self.mask)
        tmp = cv2.cvtColor(tmp, cv2.COLOR_HSV2BGR)  # Convert to RGB to save
        cv2.imwrite("output.png", tmp)


if __name__ == '__main__':
    app = QtWidgets.QApplication(sys.argv)
    dialog = QtWidgets.QDialog()

    prog = MainPlotGui(dialog)

    dialog.show()
    sys.exit(app.exec_())
```