Raspberry Pi Computer Vision Programming

Design and implement your own computer vision applications with the Raspberry Pi

Ashwin Pajankar
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 4 'Colorspaces, Transformations, and Thresholds'
- A synopsis of the book’s content
- More information on Raspberry Pi Computer Vision Programming

About the Author

Ashwin Pajankar is a Bangalore-based software professional with more than 5 years of experience in software design, development, testing, and automation. He graduated from IIT Hyderabad with an MTech degree in computer science and engineering. He holds multiple professional certifications from Oracle, IBM, Teradata, and ISTQB in development, databases, and testing. Apart from work, he enjoys serving the community. He has won several awards in college through college outreach initiatives and at work for community service through his employers for corporate social responsibility programs. He was introduced to the Raspberry Pi while organizing a hackathon at his workplace, and he's been hooked on to Pi ever since. He writes plenty of code in Bash, Python, and Java on his cluster of Pi. Currently, he's building India's biggest cluster of the recently launched Raspberry Pi 2. He's reviewed two other titles related to Python from Packt and is working on another book on Raspberry Pi.

You can view Ashwin's LinkedIn profile by visiting in.linkedin.com/in/ashwinpajankar.
Raspberry Pi Computer Vision Programming

Raspberry Pi was developed as a low-cost single-board computer with the intention of promoting computer science education in schools. It also represents a welcome return to a simple and fun yet effective way to learn computer science and programming.

You can use Raspberry Pi to learn and implement concepts in computer vision.

With a $35 Raspberry Pi computer and a USB webcam, anyone can afford to become a pro in computer vision in no time and build a real-life computer vision application to impress friends and colleagues.

What This Book Covers

Chapter 1, Introduction to Computer Vision and Raspberry Pi, takes you through the introduction and initial setup of Raspberry Pi and computer vision.

Chapter 2, Working with Images, Webcams, and GUI, teaches you how to work with images, videos, and various cameras.

Chapter 3, Basic Image Processing, explores arithmetic and logical operations on images.

Chapter 4, Colorspaces, Transformations, and Thresholds, introduces you to colorspace conversions, which are then followed by a simple project. This chapter also explores geometric transformations and segmentation by thresholding.

Chapter 5, Let's Make Some Noise, teaches the basics of noise in digital images and low-pass filters. It also discussed their usage in the removal of noise from images.

Chapter 6, Edges, Circles, and Lines' Detection, explores high-pass filters and their applications. It also explores the detection of features like edges, circles, and lines.

Chapter 7, Image Restoration, Quantization, and Depth Map, explores image restoration by inpainting. It also teaches image segmentation, quantization, and depth maps.
Chapter 8, *Histograms, Contours, Morphological Transformations, and Performance Measurement*, introduces the readers to histograms and plotting. It explores the concepts of contours and morphological transformations on an image. It concludes with the basics of performance measurement and improvement.

Chapter 9, *Real-life Computer Vision Applications*, implements various real-life applications of computer vision using Raspberry Pi and a webcam.

Chapter 10, *Introduction to SimpleCV*, teaches the installation and usage of SimpleCV, a powerful yet simple computer vision library, and concludes with a few real-life projects.
In our previous chapter, we saw how to perform basic mathematical and logical operations on images. We also saw how to use these operations to create a film-style smooth image transitioning effect. In this chapter, we will continue to explore a few more intriguing computer vision concepts and their applications in the real world. We will explore the following topics:

- Colorspace and conversions
- Real-time object tracking based on color value
- Geometric transformations on images
- Thresholding an image

**Colors and conversions**

A colorspace is a mathematical model used to represent colors. Usually, colorspace are used to represent the colors in a numerical form and to perform mathematical and logical operations with them. In this book, the colorspace we mostly use are BGR (OpenCV's default colorspace), RGB, HSV, and grayscale. **BGR** stands for **blue**, **green**, and **red**. **HSV** represents colors in **Hue**, **Saturation**, and **Value** format. OpenCV has a function `cv2.cvtColor(img, conv_flag)` that allows us to change the colorspace of an image (`img`), while the source and target colorspace are indicated on the `conv_flag` parameter.
If you remember, in Chapter 2, *Working with Images, Webcams, and GUI*, we discovered that OpenCV loads images in BGR format and matplotlib uses the RGB format for images. So, before displaying an image with matplotlib, we need to convert an image from BGR to RGB colorspace.

Take a look at the following code. The program reads the image in color mode using `cv2.imread()`, which imports the image in the BGR colorspace. Then, it converts it to RGB using `cv2.cvtColor()`, and finally, it uses matplotlib to display the image:

```python
import cv2
import matplotlib.pyplot as plt

img = cv2.imread('/home/pi/book/test_set/4.2.07.tiff',1)
img = cv2.cvtColor ( img , cv2.COLOR_BGR2RGB )
plt.imshow ( img ) , plt.title ('COLOR IMAGE'), plt.xticks([]) , plt.yticks([])
plt.show()
```

Another way to convert an image from BGR to RGB is to first split the image into three separate channels (B, G, and R channels) and merge them in BGR order. However, this takes more time as split and merge operations are inherently computationally costly, making them slower and inefficient. So, for the remainder of this book, we will use the first method. The following code shows this method:

```python
import cv2
import matplotlib.pyplot as plt

img = cv2.imread('/home/pi/book/test_set/4.2.07.tiff',1)
b,g,r = cv2.split ( img )
img=cv2.merge((r,g,b))
plt.imshow ( img ) , plt.title ('COLOR IMAGE'), plt.xticks([]) , plt.yticks([])
plt.show()
```
The output of both the programs is the same as shown in the following image:

![COLOR IMAGE](image)

If you need to know the colorspace conversion flags, then the following snippet of code will assist you in finding the list of available flags for your current OpenCV installation:

```python
import cv2
j=0
for filename in dir(cv2):
    if filename.startswith('COLOR_'):
        print(filename)
        j=j+1

print('There are ' + str(j) + ' Colorspace Conversion flags in OpenCV')
```
The last few lines of the output will be as follows (I am not including the complete output due to space limitation):

```
COLOR_YUV420P2BGRA
COLOR_YUV420P2GRAY
COLOR_YUV420P2RGB
COLOR_YUV420P2RGB
COLOR_YUV420SP2BGR
COLOR_YUV420SP2BGRA
COLOR_YUV420SP2GRAY
COLOR_YUV420SP2RGB
COLOR_YUV420SP2RGBA
```

There are 176 Colorspace Conversion flags in OpenCV.

The following code converts a color from BGR to HSV and prints it:

```python
>>> import cv2
>>> import numpy as np
>>> c = cv2.cvtColor(np.uint8([[255,0,0]]),cv2.COLOR_BGR2HSV)
>>> print c
[[[120 255 255]]]
```

The preceding snippet of code prints the HSV value of the color blue represented in BGR.

Hue, Saturation, Value, or HSV is a color model that describes colors (hue or tint) in terms of their shade (saturation or amount of gray) and their brightness (value or luminance). Hue is expressed as a number representing hues of red, yellow, green, cyan, blue, and magenta. Saturation is the amount of gray in the color. Value works in conjunction with saturation and describes the brightness or intensity of the color.

### Tracking in real time based on color

Let's study a real-life application of this concept. In HSV format, it's much easier to recognize the color range. If we need to track a specific color object, we will have to define a color range in HSV, then convert the captured image in the HSV format, and then check whether the part of that image falls within the HSV color range of our interest. We can use the `cv2.inRange()` function to achieve this. This function takes an image, the upper and lower bounds of the colors, and then checks the range criteria for each pixel. If the pixel value falls in the given color range, the corresponding pixel in the output image is 0; otherwise it is 255, thus creating a binary mask.
We can use `bitwise_and()` to extract the color range we're interested in using this binary mask thereafter. Take a look at the following code to understand this concept:

```python
import numpy as np
import cv2

cam = cv2.VideoCapture(0)

while ( True ):
    ret, frame = cam.read()
    hsv=cv2.cvtColor(frame,cv2.COLOR_BGR2HSV)
    image_mask=cv2.inRange(hsv,np.array([40,50,50]),
                   np.array([80,255,255]))
    output=cv2.bitwise_and(frame,frame,mask=image_mask)
    cv2.imshow('Original',frame)
    cv2.imshow('Output',output)
    if cv2.waitKey(1) == 27:
        break

cv2.destroyAllWindows()
cam.release()
```

We're tracking the green colored objects in this program. The output should be similar to the following one. I used green tea bag tags as the test object.
The mask image is not included in the preceding image. You can see it yourself by adding `cv2.imshow('Image Mask', image_mask)` to the code. It would be a binary (pure black and white) image.

We can also track multiple colors by tweaking this code a bit. We need to modify the preceding code by creating a mask for another color range. Then, we can use `cv2.add()` to get the combined mask for two distinct color ranges, as follows:

```python
blue=cv2.inRange(hsv,np.array([100,50,50]),np.array([140,255,255]))
green=cv2.inRange(hsv,np.array([40,50,50]),np.array([80,255,255]))
image_mask=cv2.add(blue,green)
output=cv2.bitwise_and(frame,frame,mask=image_mask)
```

Try this code and check the output for yourself.

**Image transformations**

In this section, we will see the various transformations on an image, and how to implement them in OpenCV.

**Scaling**

Scaling is the resizing of the image, which can be accomplished by the `cv2.resize()` function. It takes image, scaling factor, and interpolation method as inputs.

The interpolation method parameter can have any one of the following values:

- `INTER_LINEAR`: This deals with bilinear interpolation (default value)
- `INTER_NEAREST`: This deals with the nearest-neighbor interpolation
- `INTER_AREA`: This is associated with resampling using pixel area relation (preferred for shrinking)
- `INTER_CUBIC`: This deals with bicubic interpolation over 4 x 4 pixel neighborhood (preferred for zooming)
- `INTER_LANCZOS4`: This deals with Lanczos interpolation over 8 x 8 pixel neighbourhood

The following example shows the usage for upscaling and downscaling:

```python
import cv2
img = cv2.imread('/home/pi/book/test_set/house.tiff',1)
upscale = cv2.resize(img,None,fx=1.5,fy=1.5,
                     interpolation=cv2.INTER_CUBIC)
```
downscale = cv2.resize(img,None,fx=0.5,fy=0.5,
  interpolation=cv2.INTER_AREA)
cv2.imshow('upscale',UpScale)
cv2.waitKey(0)
cv2.imshow('downscale',DownScale)
cv2.waitKey(0)
cv2.destroyAllWindows()

In the preceding code, we upscale the image in the x and y axes with a factor of 1.5 and downscale in x and y axes with a factor of 0.5. Run the code and see the output for yourself.

**Translation, rotation, and affine transformation**

The cv2.warpAffine() function can be used to perform translation, rotation, and affine transformation. It takes an input image, transformation matrix, and size of the output image as inputs, and returns the transformed image.

You can read more about affine transformations at [http://mathworld.wolfram.com/AffineTransformation.html](http://mathworld.wolfram.com/AffineTransformation.html).

The following examples show different types of transformations which can be implemented with cv.warpAffine().

Translation means shifting the location of the image. The shifting factor in (x,y) can be denoted with the transformation matrix, as follows:

\[
T = \begin{bmatrix}
1 & 0 & x \\
0 & 1 & y
\end{bmatrix}
\]

The following code shifts the location of the image with (-50,50):

```python
import numpy as np
import cv2
import matplotlib.pyplot as plt

img = cv2.imread('/home/pi/book/test_set/house.tiff',1)
input=cv2.cvtColor(img,cv2.COLOR_BGR2RGB)
```
Colors, Transformations, and Thresholds

rows, cols, channel = img.shape

T = np.float32([[1, 0, -50], [0, 1, 50]])
output = cv2.warpAffine(input, T, (cols, rows))

plt.imshow(output), plt.title('Shifted Image')
plt.show()

The output is shown as follows:

Some parts of the image will be cropped as the size of the output is the same as the input.

Similarly, we can use cv2.warpAffine() to apply scaled rotation to an image. For this, we need to define a rotation matrix with the use of cv2.getRotationMatrix2D(), which accepts the center of the rotation, the angle of anti-clockwise rotation (in degrees), and the scale as parameters, and provides a rotation matrix, which can be specified as the parameter to cv2.warpAffine().
The following example rotates the image by 45 degrees with the center of the image as the center of rotation, and scales it down to 50% of the original image:

```python
import cv2
import matplotlib.pyplot as plt

img = cv2.imread('/home/pi/book/test_set/house.tiff',1)
input=cv2.cvtColor(img,cv2.COLOR_BGR2RGB)
rows,cols,channel = img.shape

R = cv2.getRotationMatrix2D((cols/2,rows/2),45,0.5)
output = cv2.warpAffine(input,R,(cols,rows))

plt.imshow ( output ) , plt.title ('Rotated and Downscaled Image')
plt.show()
```

The output will be as follows:
We can create some animation / visual effects by changing the rotation angle at regular intervals and then displaying it in a continuous loop till the Esc key is pressed. Following is the code for this (check the output yourself):

```python
import cv2
from time import sleep

image = cv2.imread('/home/pi/book/test_set/house.tif',1)
rows,cols,channels = image.shape

angle = 0
while(1):
    if angle == 360:
        angle=0
    M = cv2.getRotationMatrix2D((cols/2,rows/2),angle,1)
    rotated = cv2.warpAffine(image,M,(cols,rows))
    cv2.imshow('Rotating Image',rotated)
    angle=angle+1
    sleep(0.2)
    if cv2.waitKey(1) == 27 :
        break

cv2.destroyAllWindows()
```

Try implementing this on the live cam for more fun.

Next, we will see how to implement an affine transformation on any image. An affine transformation is a function between affine spaces. After applying the affine transformation on an image, the parallelism between the lines in an image is preserved. This means that the parallel lines in original images remain parallel even after transformation. The affine transformation needs any three non-collinear points (points which are not on the same line) in the original image and the corresponding points in the transformed image. These points are passed as arguments to `cv2.getAffineTransform()` to get the transformation matrix, and that matrix, in turn, is passed to `cv2.warpAffine()` as an argument. Take a look at the following example:

```python
import cv2
import numpy as np
from matplotlib import pyplot as plt
```
image = cv2.imread('/home/pi/book/test_set/2.1.11.tiff',1)

#changing the colorspace from BGR->RGB
input = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

rows,cols,channels = input.shape

points1 = np.float32([[100,100],[300,100],[100,300]])
points2 = np.float32([[200,150],[400,150],[100,300]])

A = cv2.getAffineTransform(points1,points2)

output = cv2.warpAffine(input,A,(cols,rows))

plt.subplot(121),plt.imshow(input),plt.title('Input')
plt.subplot(122),plt.imshow(output),plt.title('Affine Output')
plt.show()

The output will appear as follows:
Perspective transformation

In perspective transformation, we provide four points from the input image and corresponding four points in the output image. The condition is that any three of these points should not be collinear (again, not in the same line). Like affine transformation, in perspective transformation, a straight line will remain straight. However, the parallelism between the lines will not be preserved. A real-life example of perspective transformation would be the zooming and angled zoom functionality in software. The degree and angle of zoom depends on the transformation matrix, which is defined by a set of four input and four output points. Let’s see an example of the simple zoom functionality with the following code, where we use cv2.getPerspectiveTransform() to generate the transformation matrix and cv2.warpPerspective() to get the transformed output:

```python
import cv2
import numpy as np
from matplotlib import pyplot as plt

image = cv2.imread('/home/pi/book/test_set/ruler.512.tiff',1)

#changing the colorspace from BGR->RGB
input = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

rows,cols,channels = input.shape

points1 = np.float32([[0,0],[400,0],[0,400],[400,400]])
points2 = np.float32([[0,0],[300,0],[0,300],[300,300]])

P = cv2.getPerspectiveTransform(points1,points2)

output = cv2.warpPerspective(input,P,(300,300))

plt.subplot(121),plt.imshow(input),plt.title('Input')
plt.subplot(122),plt.imshow(output),plt.title('Perspective Transform')
plt.show()
```
The output will appear as follows:

![Figure 1](image)

Try passing various combination of the parameters to see how the resultant image changes. In the preceding example, parallelism between the lines is preserved because of the combination of the parameters we used. You might want to try different combinations of the parameters to see that the parallelism between the lines is not preserved.
Thresholding image

Thresholding is the simplest way to segment images. Although thresholding methods and algorithms are available for colored images, it works best on grayscale images. Thresholding usually (but not always) converts grayscale images into binary images (in a binary image, each pixel can only have one of two possible values: white or black). Thresholding the image is usually the first step in many image processing applications.

The way thresholding works is very simple. We define a threshold value. For a pixel in a grayscale image, if the value of grayscale intensity is greater than the threshold, we assign a value to the pixel (for example, white), else we assign a black value to the pixel. This is the simplest form of thresholding and there are many other variations of this method, which we will see now.

In OpenCV, the \texttt{cv2.threshold()} function is used to threshold images. It takes as input, grayscale image, threshold value, \texttt{maxVal}, and threshold method as parameters, and returns the thresholded image as output. The \texttt{maxVal} parameter is the value assigned to the pixel if the pixel intensity is greater (or less in some methods) than the threshold. There are five threshold methods available in OpenCV; in the beginning, the simplest form of thresholding we saw is \texttt{cv2.THRESH_BINARY}.

Let's see the mathematical representation of all the threshold methods.

Say \((x,y)\) is the input pixel; then, operations by threshold methods are as follows:

- **\texttt{cv2.THRESH_BINARY}**
  
  If \(\text{intensity}(x,y) > \text{thresh}\), then set \(\text{intensity}(x,y) = \text{maxVal}\); else set \(\text{intensity}(x,y) = 0\).

- **\texttt{cv2.THRESH_BINARY_INV}**
  
  If \(\text{intensity}(x,y) > \text{thresh}\), then set \(\text{intensity}(x,y) = 0\); else set \(\text{intensity}(x,y) = \text{maxVal}\).

- **\texttt{cv2.THRESH_TRUNC}**
  
  If \(\text{intensity}(x,y) > \text{thresh}\), then set \(\text{intensity}(x,y) = \text{thresh}\); else leave \(\text{intensity}(x,y)\) as it is.
• cv2.THRESH_TOZERO

If intensity(x, y) > thresh; then leave intensity(x, y) as it is; else set intensity(x, y) = 0.

• cv2.THRESH_TOZERO_INV

If intensity(x, y) > thresh, then set intensity(x, y) = 0; else leave intensity(x, y) as it is.

The demonstration of the threshold functionality usually works best on grayscale images with a gradually increasing gradient. In the following example, we chose the value of the threshold as 127, so the image is segmented into two sets of pixels depending on the value of their intensity:

```python
import cv2
import matplotlib.pyplot as plt

img = cv2.imread('/home/pi/book/test_set/gray21.512.tiff',0)
th=127
max_val=255
ret,o1 = cv2.threshold(img,th,max_val,cv2.THRESH_BINARY)
ret,o2 = cv2.threshold(img,th,max_val,cv2.THRESH_BINARY_INV)
ret,o3 = cv2.threshold(img,th,max_val,cv2.THRESH_TOZERO)
ret,o4 = cv2.threshold(img,th,max_val,cv2.THRESH_TOZERO_INV)
ret,o5 = cv2.threshold(img,th,max_val,cv2.THRESH_TRUNC)

titles = ['Input Image','BINARY','BINARY_INV','TOZERO','TOZERO_INV','TRUNC']
output = [img, o1, o2, o3, o4, o5]

for i in xrange(6):
    plt.subplot(2,3,i+1),plt.imshow(output[i],cmap='gray')
    plt.title(titles[i])
    plt.xticks([]),plt.yticks([])
plt.show()
```
Otsu's method

Otsu's method for thresholding automatically determines the value of the threshold for the images, which have two peaks in their histogram (bi-modal histograms). This usually means the image has background and foreground pixels, and Otsu's method is the best way to separate these two sets of pixels automatically without specifying the threshold value.
Otsu’s method is not the best way for images which are not in the background + foreground model, and they may provide improper output if applied. This method is applied in addition to other methods and the threshold is passed as 0. Try implementing the following code:

```python
ret, output = cv2.threshold(image, 0, 255, cv2.THRESH_BINARY + cv2.THRESH_OTSU)
```

The output of this will be as follows. This is an image of a tank in a desert:

![Threshold Image](image)

**Exercise**

Explore `cv2.adaptiveThreshold()`, which is used for adaptive thresholding of images based on uneven lighting conditions (some parts of the image are more illuminated than others).
Summary

In this chapter, we explored colorspaces and its applications in image tracking with one color and multiple colors. Then, we applied transformations on images. Finally, we saw how to threshold an image.

In the next chapter, we will go over the basics of noise and filtering the noise, as well as smoothening/blurring images with Low Pass Filters.
Get more information Raspberry Pi Computer Vision Programming

Where to buy this book
You can buy Raspberry Pi Computer Vision Programming from the
Packt Publishing website.
Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet
book retailers.
Click here for ordering and shipping details.