CHAPTER 5

Object Detection and Recognition

In the previous chapter, you learned about image segmentation and contours. You also learned how to detect lines and circles using Hough lines and circles in OpenCV. In this chapter, you will learn how to detect objects and label them. Object detection is one of the most widely used capabilities of computer vision in multiple domains. In Chapter 1, you saw some real-world use cases. In this chapter, you will start with object detection and then move on to object recognition, landmark identification, and finally handwriting recognition.

The following topics are covered in this chapter:

- Introduction to object detection and its uses
- How objects are stored and the different ways of extracting features such as SIFT, SURF, FAST, BRIEF, and so on
- Handwriting recognition

Basics of Object Detection

Detecting objects in an image is a crucial capability of a computer vision application. Object detection/recognition is used in labeling scenes, robotic navigation, self-driving cars, face and body part recognition, disease and cancer detection, objects in satellite images, handwriting recognition, and many more.

S. Gollapudi, *Learn Computer Vision Using OpenCV*,
https://doi.org/10.1007/978-1-4842-4261-2_5

Figure 5-1 shows an example of real-time object detection and labeling done for a given image.

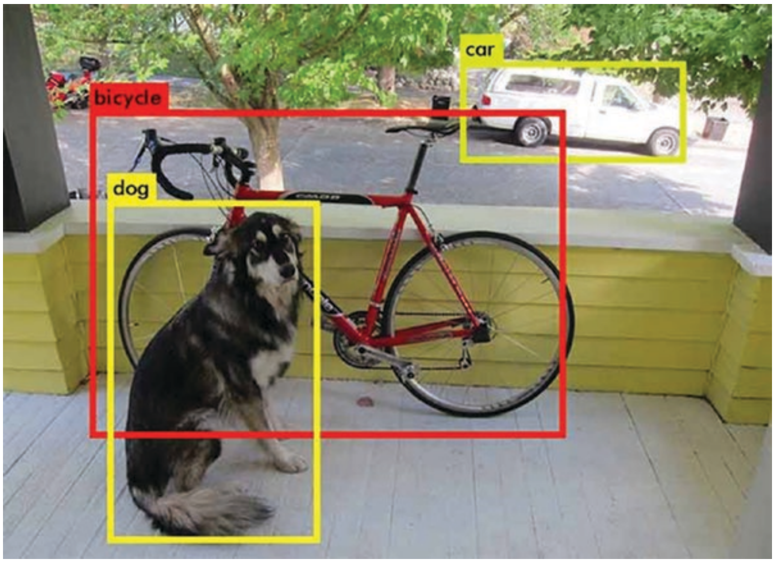

Figure 5-1. *Real-time object detection and labeling*

Object Detection vs. Object Recognition

In Figure 5-1, we only marked or detected if there was a truck or a dog. We did not recognize any specific qualities such as the model or color of the car or the color or breed of the dog because the objective was to just identify what objects are in the image. *Object recognition* is the method of identifying an object within an image. In the case of object recognition, you first detect the car, and on the cropped car you apply recognizers to recognize the features of the car. This is similar with faces as well.

While humans can identify a variety of objects effortlessly, for computers it is a complex problem to solve with accuracy. It has eluded computer vision researchers for decades now and has become the holy grail of computer vision.

Depending on the position and angle of the object, the object detection task is difficult. Defining a bounding box for each object is important.

Template Matching

As part of object detection and recognition, you need to do shape analysis and feature analysis. To do this, there is a robust technique called *template matching*. This technically is a brute-force algorithm or a simple mechanism to extract an object based on a previously acquired template.

OpenCV has a `matchTemplate()` function to perform template matching.

This function takes a “sliding window” of the image being queried and slides it across the image it is searching for to determine its presence. It does this one pixel at a time. Then, for each of these locations, a correlation coefficient is calculated if there is a match at all. Regions with a high correlation are the regions that match.

Figure 5-2 shows a typical object detection using matching. This method uses a template to detect an object after segmentation. If the segmented object is similar to the template, then the object detection process is concluded; otherwise, another template is picked for a similarity check.

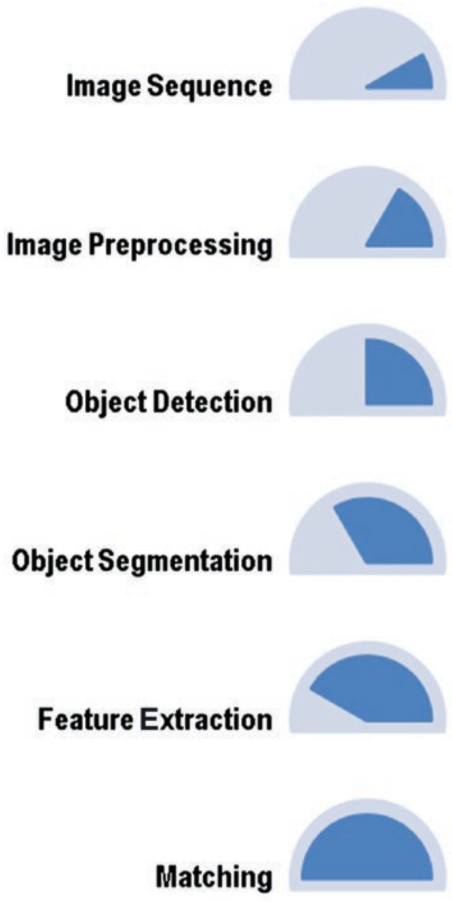

***Figure 5-2.** Object detection using matching*

The following is the example code for template matching in OpenCV using the `matchTemplate()` function:

```
import cv2
import numpy as np

# Load input image and convert to grayscale
image = cv2.imread('./images/inputImage.jpg')
```

```
 6  cv2.imshow('Where is this image?', image)
 7  cv2.waitKey(0)
 8  gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
 9
10  # Load Bigger image
11  bigger_image = cv2.imread('./images/searchImage.jpg',0)
12
13  result = cv2.matchTemplate(gray, template, cv2.TM_CCOEFF)
14  min_val, max_val, min_loc, max_loc = cv2.minMaxLoc(result)
15
16  #Create Bounding Box
17  top_left = max_loc
18  bottom_right = (top_left[0] + 50, top_left[1] + 50)
19  cv2.rectangle(image, top_left, bottom_right, (0,0,255), 5)
20
21  cv2.imshow('Where is input image?', image)
22  cv2.waitKey(0)
23  cv2.destroyAllWindows()
```

The input image is searched in the bigger image. Use the `matchTemplate()` function by passing the grayscale image.

Lines 1 and 2: Import the OpenCV and NumPy libraries.

Lines 4 through 8: Load the image that needs to be searched for and convert it to grayscale.

Line 10: Loads the bigger image in which the input image needs to be searched for.

Lines 13 and 14: `cv2.matchTemplate()` returns a correlation map, essentially a grayscale image. This image has each pixel that denotes the extent to which its neighborhood matches with the template. The `minMaxLoc` function returns the max and min intensity values as an array that includes the location of these intensities.

MaxVal is the location with the highest intensity in the image. This is returned by matchTemplate() and corresponds to the best matching input image with regard to the defined template.

Lines 16 through 19: Draw a boundary with a padding value of 50 and a thickness value of 5 pixels and in blue around the contours of the matching image.

Challenges with Template Matching

While template matching helps when doing object detection and recognition in an image, there are several challenges with this methodology. If the image is rotated, scaled, modified for colors or brightness, or transformed, it is difficult to match or detect an input object in the image.

Understanding Image "Features"

With the challenges of the template matching approach, you will now learn about image feature–driven object detection and recognition. To start, let's look at what features are in the context of image processing.

Features correspond to the properties or attributes of an image. They play an important role in building accurate computer vision applications. Pixels, as you learned in Chapter 1, are used to compare two images.

The most basic form of feature detection is point features. In applications such as panorama creation on our smartphones, each image is stitched with the corresponding previous image. This stitching requires the correct orientation of an image overlapped with pixel-level accuracy. Computing corresponding pixels between two images requires pixel matching.

Interesting and Uninteresting Points

Within an image there can be interesting and uninteresting points. Interesting points in an image are those that can give the most information about the object in the image, and uninteresting points give either zero or no information about the image or the object in the image. Figure 5-3 shows an image of the Eiffel Tower with the image feature points and what could be an interesting or uninteresting point. The sky could be an uninteresting feature because it hardly gives the context of the monument. A point on the Eiffel Tower does give more information about it and hence becomes an interesting feature.

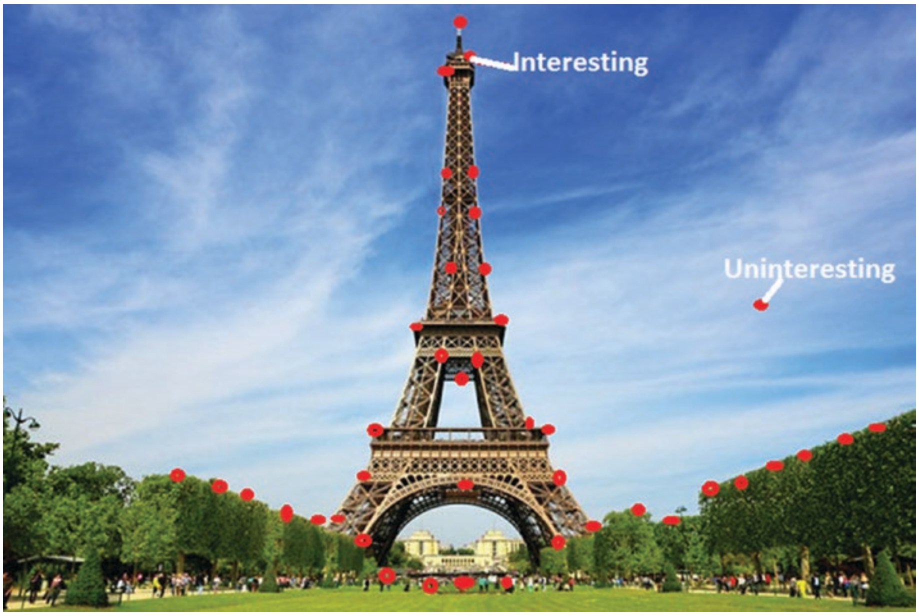

Figure 5-3. *Image feature points*

The following are some characteristics of an interesting, or good, feature:

- **Is repeatable**: The same feature can be found in several other images despite any image transformations.
- **Is salient/distinctive/unique:** The feature is unique and has a distinctive description in the use case context.
- **Is compact in number:** There are a measurable number of pixels that describe the object in context.
- **Is local**: The object in context occupies a relatively smaller area within an image.

Types of Image Features

There are primarily three types of image features: edges, regions, and corner features. These features of objects are used to track objects in an image by observing the change in intensity, as shown in Figure 5-4.

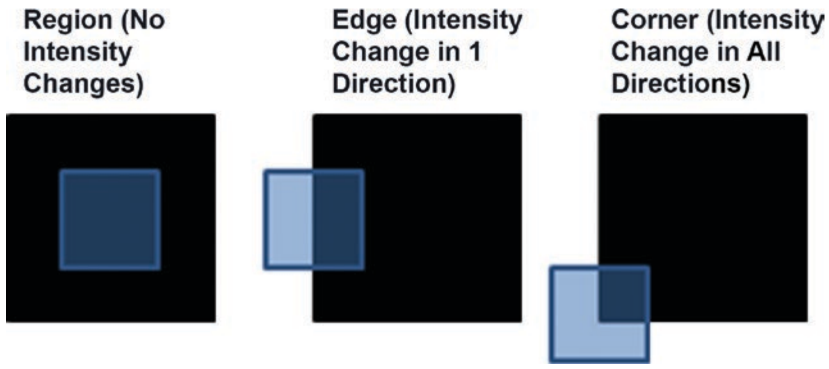
Region (No Intensity Changes)
Edge (Intensity Change in 1 Direction)
Corner (Intensity Change in All Directions)

Figure 5-4. *Changes in intensity*

Feature Matching

Feature matching can be done in one of the following cases:

- There are two images, and you want to quantify whether these images match each other. There will usually be a comparison metric that is applied.
- There is a large database of images, and for every new image, you need to perform matching against the database of images. A smaller search criterion is stored and then compared with the input image instead of recomputing everything for every image in the database. This is called a *feature vector* of the image. For every new input image, a similar vector is extracted and stored.
- As an alternative approach, you have a small portion of the image stored as a template. The goal is to check whether an image has this template. This will require matching key points from the template against the given sample image. If the match value is greater than a threshold, you can say the sample image has a region similar to the given template. There is a possibility of showing where in the sample image your template image is.

Image Corners As Features

In this section, you will learn how to use corners as features for object detection and recognition. While corners do not necessarily provide all the details of the objects, they are helpful in many cases. As indicated in Figure 5-4, when the blue frame is moved around the image and in particular you see that there is an intensity change in all directions (Figure 5-4, section 3), then that is identified as the corner of the image.

Let's look at some OpenCV code that explains how to identify a corner. You will use an algorithm in the OpenCV library called the Harris corner algorithm.

Harris Corner Algorithm

This algorithm helps identify the inside corner of an image by checking the area that has maximum variations in intensity.

In 1988 Chris Harris and Mike Stephens developed this algorithm that can perform both edge detection and corner detection. Hence, this algorithm was named after one of the authors.

In OpenCV, the `cv2.cornerHarris()` function is used to achieve the corner detection.

```
cv2.cornerHarris(image, blockSize, ksize, k)
```

This function takes four arguments.

- `img` is the image to be analyzed; it must be in grayscale and with float32 values.
- `blockSize` is the size of the window considered for the corner detection.
- `ksize` is a parameter for the derivative of Sobel.
- `k` is a free parameter for the Harris equation.

The following OpenCV code takes an image input, identifies the corners, and marks them:

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

img = cv2.imread('blackandwhite.jpg',0)
img = np.float32(img)
```

```
 7  corners = cv2.cornerHarris(img,2,3,0.04)
 8
 9  corners = cv2.cornerHarris(img,2,3,0.04)
10
11  plt.subplot(2,1,1), plt.imshow(corners ,cmap = 'jet')
12  plt.title('Harris Corner Detection'), plt.xticks([]),
    plt.yticks([])
13
14  img2 = cv2.imread('blackandwhite.jpg')
15  corners2 = cv2.dilate(corners, None, iterations=3)
16  img2[corners2>0.01*corners2.max()] = [255,0,0]
17
18  plt.subplot(2,1,2),plt.imshow(img2,cmap = 'gray')
19  plt.title('Canny Edge Detection'), plt.xticks([]),
    plt.yticks([])
20
21  plt.show()
```

In the previous code, line 7 is the place where the Harris corner algorithm is invoked. Once the corners are identified, they are highlighted using the `dilate()` function, and the identified pixels are assigned the color red for showing in a new window. Figure 5-5 shows the input and output images.

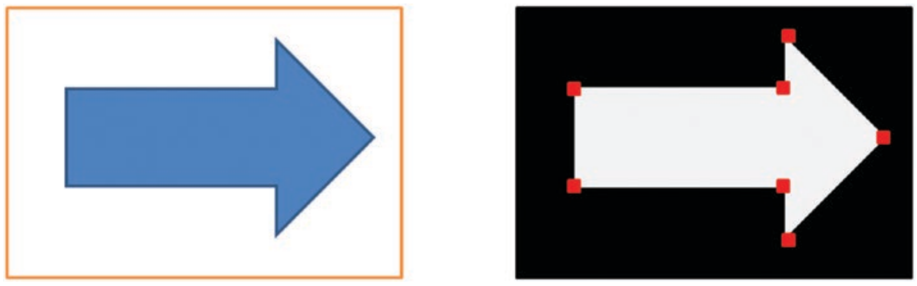

Figure 5-5. *Input and output images, corner detection*

Figure 5-6 shows another example of an input and output of corner detection.

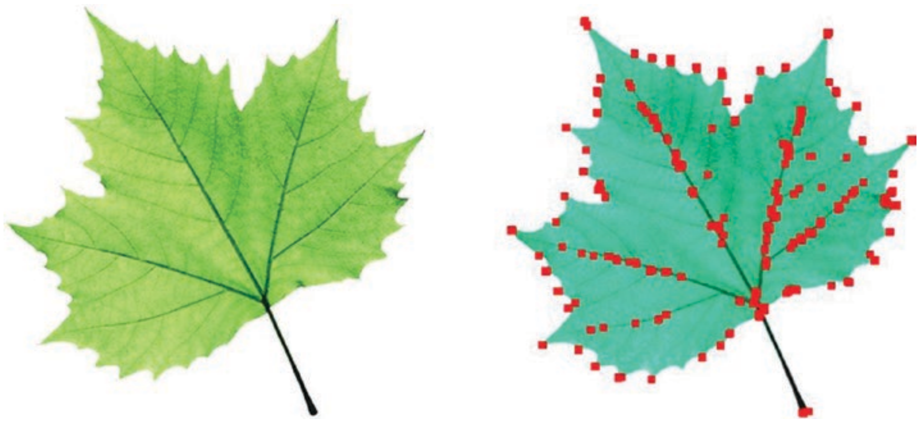

Figure 5-6. Another example of corner detection

However, there are several challenges when corners are used as features for object detection. While corner matching works well with image rotations, translations or any photometric changes such as brightness, intensity changes, and image scaling does not work.

Feature Tracking and Matching Flow

In this section, you will learn the standard flow for feature extraction and matching. Figure 5-7 shows the generic steps involved in feature extraction.

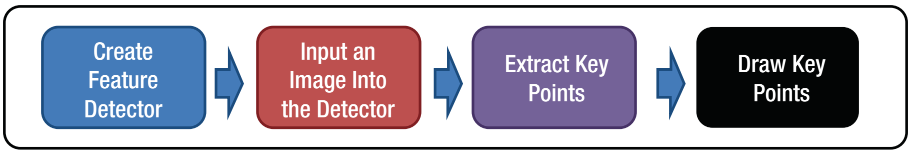

Figure 5-7. Feature extraction workflow

The first step is to create a standard feature extractor and then extract robust features from a given image. This process involves scanning through the whole image for possible features and then thresholding them. There are several techniques for selecting features such as SIFT, SURF, FAST, BRIEF, ORB detectors, and so on. In the next sections, we will cover these methods in depth. The feature extracted, in some cases, needs to be converted into a more descriptive form so that it can be learned by the model or can be stored for re-reading.

In the case of feature matching, say you are given a sample image and want to see whether this matches a reference image. After feature detection and extraction, as shown previously, a distance metric is formed to compute the distance between features of the sample with respect to the features of reference. If this distance is less than the threshold, you can say the two images are similar.

Scale Variant Feature Transform

Scale Variant Feature Transform (SIFT) is currently patented but can be freely used for academic purposes.

You saw in the previous section some of the challenges with using corners for feature extraction and how it doesn't work well when scaling up. In Figure 5-8, you can see how detecting a corner can fail.

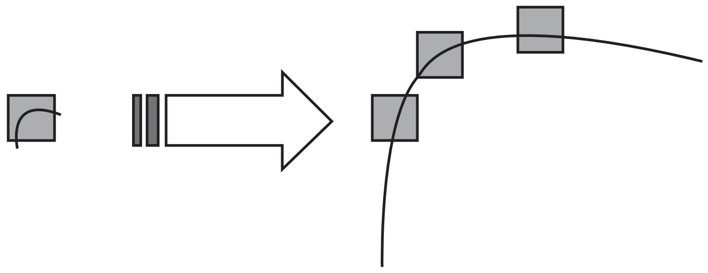

Figure 5-8. *Corner detection failure*

The SIFT approach addresses this challenge. You can find more details about SIFT and how it works at `www.inf.fu-berlin.de/lehre/SS09/CV/uebungen/uebung09/SIFT.pdf`.

OpenCV has built-in functions for SIFT, but they need to be explicitly installed since they are patented.

The following steps and code show how to implement the SIFT functions:

1. Load an image and convert it to grayscale.
2. Construct a SIFT object using the `SIFT()` function.
3. The `sift.detect()` function finds the keypoint in the images. You can pass a mask if you want to search only part of the image. Each keypoint is a special structure that has many attributes such as its (x,y) coordinates, size of the meaningful neighborhood, angle that specifies its orientation, response that specifies the strength of the keypoints, and so on.
4. OpenCV also provides the `cv2.drawKeyPoints()` function, which draws small circles on the locations of the keypoints. If you pass the flag `cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS` to it, it will draw a circle with the size of the keypoint, and it will even show its orientation.

```
import cv2
import numpy as np

image = cv2.imread('images/input.jpg')
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

```

```
#Create SIFT Feature Detector object
sift = cv2.SIFT()

#Detect key points
keypoints = sift.detect(gray, None)
print("Number of keypoints Detected: ", len(keypoints))

# Draw rich key points on input image
image = cv2.drawKeypoints(image, keypoints,
flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)

cv2.imshow('Feature Method - SIFT', image)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

The program results are shown in Figure 5-9; the input image is shown on top, and the output image is shown at the bottom.

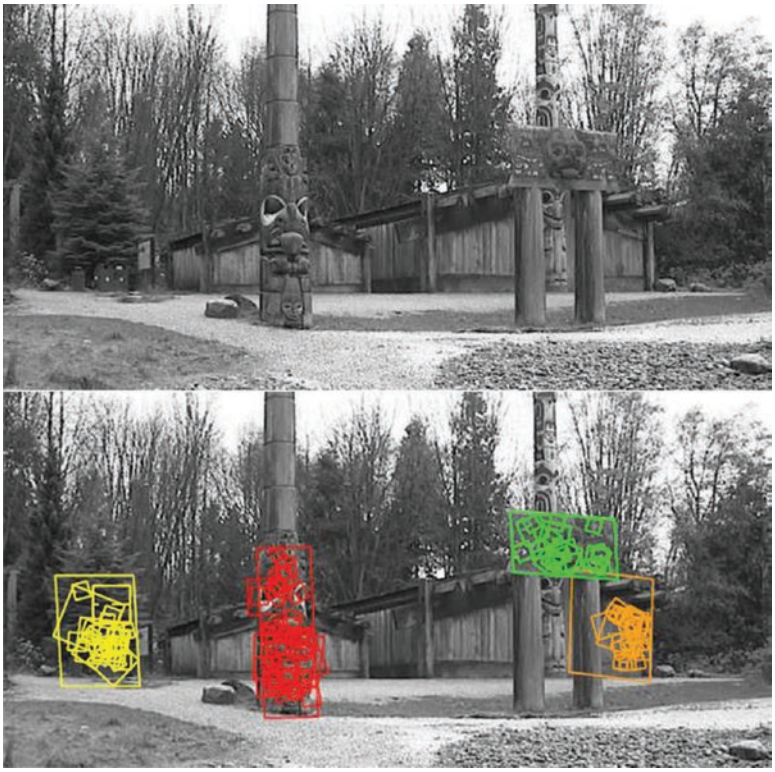

Figure 5-9. *SIFT example (source: AIShack)*

Speeded-Up Robust Features

Like SIFT, Speeded-Up Robust Features (SURF) is patented but can be openly used for academic purposes. It needs to be explicitly imported since it is patented. You can find more details on what SURF is and how it works at www.vision.ee.ethz.ch/~surf/eccv06.pdf.

OpenCV provides functions for SURF like SIFT. Similar to SIFT, SURF has functions such as `detect()` and `compute()`. The following code sample shows the implementation steps:

```
import cv2
import numpy as np
```

```

image = cv2.imread('images/input.jpg')
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

#Create SURF Feature Detector object
surf = cv2.SURF()

# Only features, whose hessian is larger than hessianThreshold are retained by the detector
surf.hessianThreshold = 500
keypoints, descriptors = surf.detectAndCompute(gray, None)
print "Number of keypoints Detected: ", len(keypoints)

# Draw rich key points on input image
image = cv2.drawKeypoints(image, keypoints,
flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)

cv2.imshow('Feature Method - SURF', image)
cv2.waitKey()
cv2.destroyAllWindows()
```

Features from Accelerated Segment Test

Features from Accelerated Segment Test (FAST) was first introduced in 2006 by Edward Rosten and Tom Drummond. The previous feature detectors are not useful for real-time applications, for example those with video cameras collecting real-time images or robots. These use cases will fail if any delay is caused in feature detection at runtime. The FAST algorithm uses a pixel neighborhood to compute key points in an image.

For the neighborhood, three flags are defined: cv2.FAST_FEATURE_DETECTOR_TYPE_5_8, cv2.FAST_FEATURE_DETECTOR_TYPE_7_12, and cv2.FAST_FEATURE_DETECTOR_TYPE_9_16. The following is some simple code to detect and draw the FAST feature points:

```
import cv2
import numpy as np

image = cv2.imread('images/input.jpg')
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

# Create FAST Detector object
fast = cv2.FastFeatureDetector()

# Obtain Key points, by default non max suppression is On
# to turn off set fast.setBool('nonmaxSuppression', False)
keypoints = fast.detect(gray, None)
print "Number of keypoints Detected: ", len(keypoints)

# Draw rich keypoints on input image
image = cv2.drawKeypoints(image, keypoints,
flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)

cv2.imshow('Feature Method - FAST', image)
cv2.waitKey()
cv2.destroyAllWindows()
```

Binary Robust Independent Elementary Features

Binary Robust Independent Elementary Features (BRIEF) is a relatively faster method feature descriptor calculator and matching algorithm. Additionally, it provides a higher-recognition rate except for the cases where there is plane rotation. You can find more details about what BRIEF is and how it works at http://cvlabwww.epfl.ch/~lepetit/papers/calonder_pami11.pdf.

The following code shows the computation of BRIEF descriptors with the help of a CenSurE detector:

```
import cv2
import numpy as np

image = cv2.imread('images/input.jpg')
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

# Create FAST detector object
fast = cv2.FastFeatureDetector()

# Create BRIEF extractor object
brief = cv2.DescriptorExtractor_create("BRIEF")

# Determine key points
keypoints = fast.detect(gray, None)

# Obtain descriptors and new final keypoints using BRIEF
keypoints, descriptors = brief.compute(gray, keypoints)
print "Number of keypoints Detected: ", len(keypoints)

# Draw rich keypoints on input image
image = cv2.drawKeypoints(image, keypoints, flags=cv2.DRAW_
MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)

cv2.imshow('Feature Method - BRIEF', image)
cv2.waitKey()
cv2.destroyAllWindows()
```

Oriented FAST and Rotated BRIEF

ORB is a combination of a FAST keypoint detector and a BRIEF descriptor with additional performance fixes. This method applies the FAST technique to identify the keypoints followed by the measurement of the top *n* points using the Harris corner method.

OpenCV has an `ORB()` function that can use a feature2d common interface. For more details on what ORB is and how it works, refer to http://www.willowgarage.com/sites/default/files/orb_final.pdf.

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

img = cv2.imread('simple.jpg',0)

# Initiate STAR detector
orb = cv2.ORB()

# find the keypoints with ORB
kp = orb.detect(img,None)

# compute the descriptors with ORB
kp, des = orb.compute(img, kp)

# draw only keypoints location,not size and orientation
img2 = cv2.drawKeypoints(img,kp,color=(0,255,0), flags=0)
plt.imshow(img2),plt.show()
```

Conclusion

In this chapter, you learned about the difference between object detection and recognition. You learned about what image features are and how they are important for object detection and feature tracking. You also learned how to detect corners, especially using OpenCV's built-in functions. Additionally, the chapter covered important detectors such as SIFT, SURF, FAST, BRIEF, and ORB with steps for implementing them using the OpenCV and Python libraries.

In the next chapter, you will learn how to do object tracking in motion using specific OpenCV functions.