3
Detecting and Tracking Different Body Parts

In this chapter, we are going to learn how to detect and track different body parts in a live video stream. We will start by discussing the face detection pipeline and how it's built from the ground up. We will learn how to use this framework to detect and track other body parts, such as eyes, ears, mouth, and nose.

By the end of this chapter, you will know:

- How to use Haar cascades
- What are integral images
- What is adaptive boosting
- How to detect and track faces in a live video stream
- How to detect and track eyes in a live video stream
- How to automatically overlay sunglasses on top of a person's face
- How to detect ears, nose, and mouth
- How to detect pupils using shape analysis

Using Haar cascades to detect things

When we say Haar cascades, we are actually talking about cascade classifiers based on Haar features. To understand what this means, we need to take a step back and understand why we need this in the first place. Back in 2001, Paul Viola and Michael Jones came up with a very effective object detection method in their seminal paper. It has become one of the major landmarks in the field of machine learning.

In their paper, they have described a machine learning technique where a boosted cascade of simple classifiers is used to get an overall classifier that performs really well. This way, we can circumvent the process of building a single complex classifier that performs with high accuracy. The reason this is so amazing is because building a robust single-step classifier is a computationally intensive process. Besides, we need a lot of training data to build such a classifier. The model ends up becoming complex and the performance might not be up to the mark.

Let's say we want to detect an object like, say, a pineapple. To solve this, we need to build a machine learning system that will learn what a pineapple looks like. It should be able to tell us if an unknown image contains a pineapple or not. To achieve something like this, we need to train our system. In the realm of machine learning, we have a lot of methods available to train a system. It's a lot like training a dog, except that it won't fetch the ball for you! To train our system, we take a lot of pineapple and non-pineapple images, and then feed them into the system. Here, pineapple images are called positive images and the non-pineapple images are called negative images.

As far as the training is concerned, there are a lot of routes available. But all the traditional techniques are computationally intensive and result in complex models. We cannot use these models to build a real time system. Hence, we need to keep the classifier simple. But if we keep the classifier simple, it will not be accurate. The trade off between speed and accuracy is common in machine learning. We overcome this problem by building a set of simple classifiers and then cascading them together to form a unified classifier that's robust. To make sure that the overall classifier works well, we need to get creative in the cascading step. This is one of the main reasons why the **Viola-Jones** method is so effective.

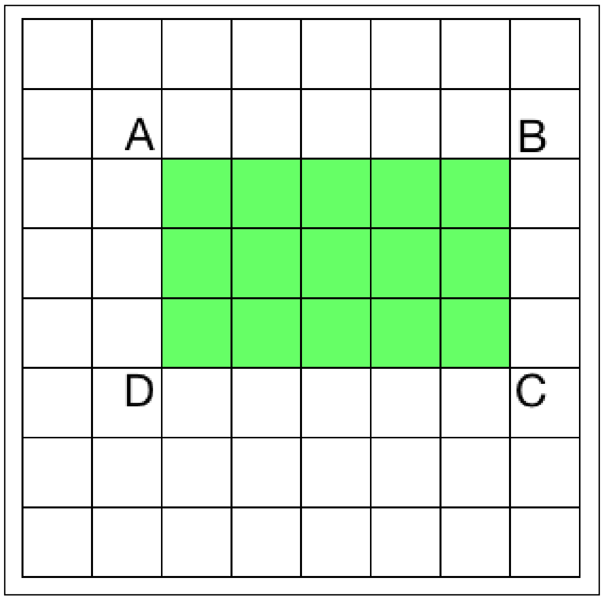
Coming to the topic of face detection, let's see how to train a system to detect faces. If we want to build a machine learning system, we first need to extract features from all the images. In our case, the machine learning algorithms will use these features to learn what a face looks like. We use Haar features to build our feature vectors. Haar features are simple summations and differences of patches across the image. We do this at multiple image sizes to make sure our system is scale invariant.

[If you are curious, you can learn more about the formulation at `http://www.cs.ubc.ca/~lowe/425/slides/13-ViolaJones.pdf`]

Once we extract these features, we pass it through a cascade of classifiers. We just check all the different rectangular sub-regions and keep discarding the ones that don't have faces in them. This way, we arrive at the final answer quickly to see if a given rectangle contains a face or not.

What are integral images?

If we want to compute Haar features, we will have to compute the summations of many different rectangular regions within the image. If we want to effectively build the feature set, we need to compute these summations at multiple scales. This is a very expensive process! If we want to build a real time system, we cannot spend so many cycles in computing these sums. So we use something called integral images.



To compute the sum of any rectangle in the image, we don't need to go through all the elements in that rectangular area. Let's say AP indicates the sum of all the elements in the rectangle formed by the top left point and the point P in the image as the two diagonally opposite corners. So now, if we want to compute the area of the rectangle ABCD, we can use the following formula:

Area of the rectangle ABCD = AC – (AB + AD - AA)

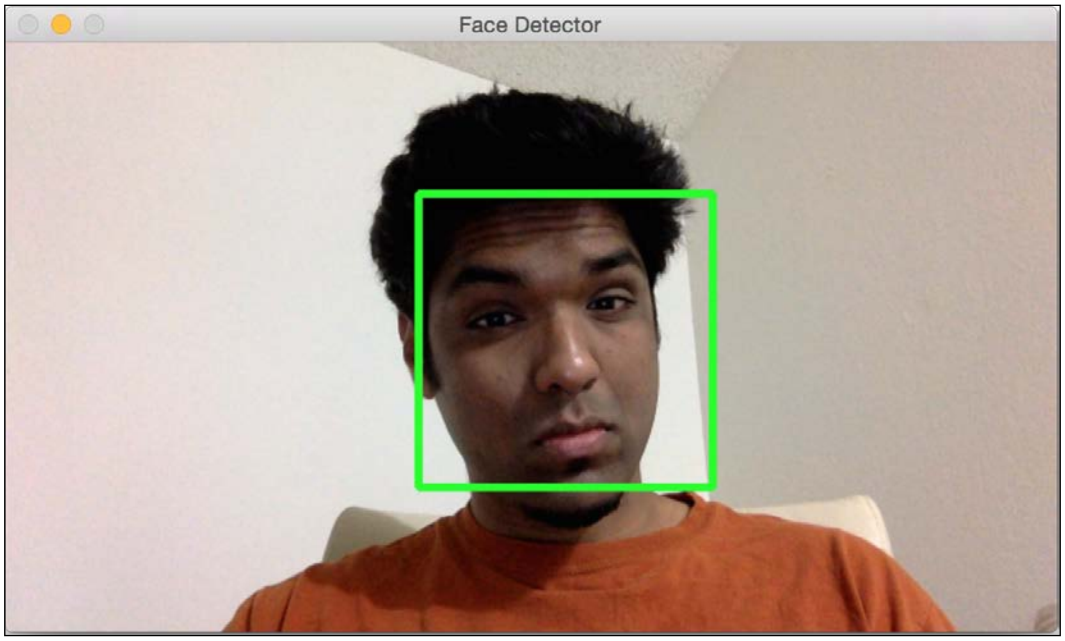
Why do we care about this particular formula? As we discussed earlier, extracting Haar features includes computing the areas of a large number of rectangles in the image at multiple scales. A lot of those computations are repetitive and the overall process is very slow. In fact, it is so slow that we cannot afford to run anything in real time. That's the reason we use this formulation! The good thing about this approach is that we don't have to recalculate anything. All the values for the areas on the right hand side of this equation are already available. So we just use them to compute the area of any given rectangle and extract the features.

Detecting and tracking faces

OpenCV provides a nice face detection framework. We just need to load the cascade file and use it to detect the faces in an image. Let's see how to do it:

```
import cv2
import numpy as np

face_cascade =
cv2.CascadeClassifier('./cascade_files/haarcascade_frontalface_alt.
xml')

cap = cv2.VideoCapture(0)
scaling_factor = 0.5

while True:
    ret, frame = cap.read()
    frame = cv2.resize(frame, None, fx=scaling_factor,
fy=scaling_factor, interpolation=cv2.INTER_AREA)
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

    face_rects = face_cascade.detectMultiScale(gray, 1.3, 5)
    for (x,y,w,h) in face_rects:
        cv2.rectangle(frame, (x,y), (x+w,y+h), (0,255,0), 3)

    cv2.imshow('Face Detector', frame)

    c = cv2.waitKey(1)
    if c == 27:
        break

cap.release()
cv2.destroyAllWindows()
```

If you run the above code, it will look something like the following image:

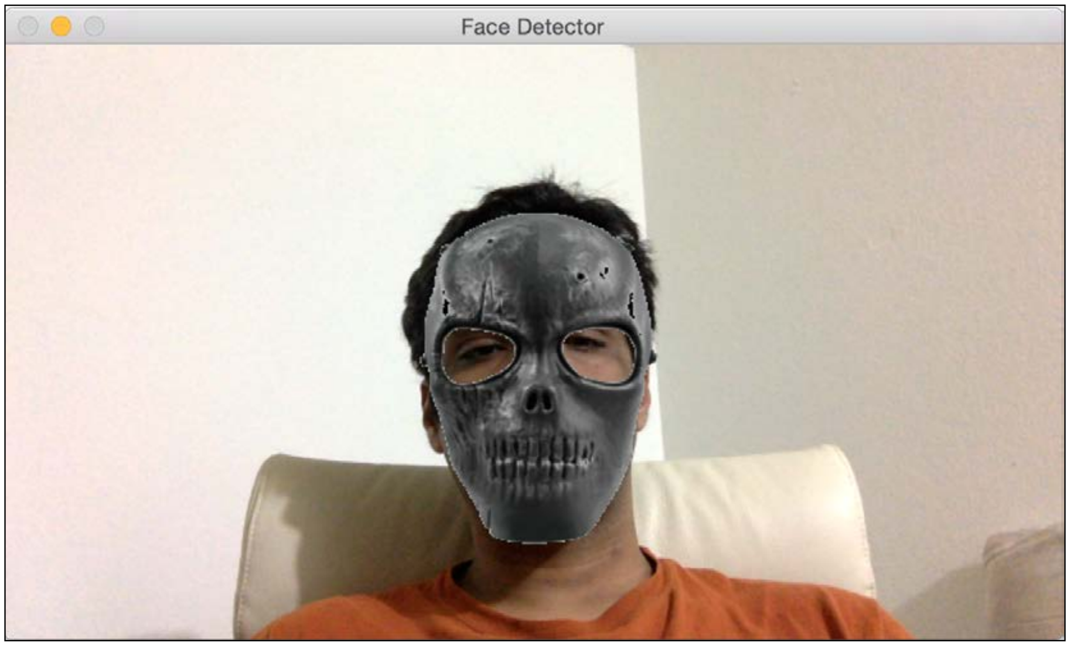


Understanding it better

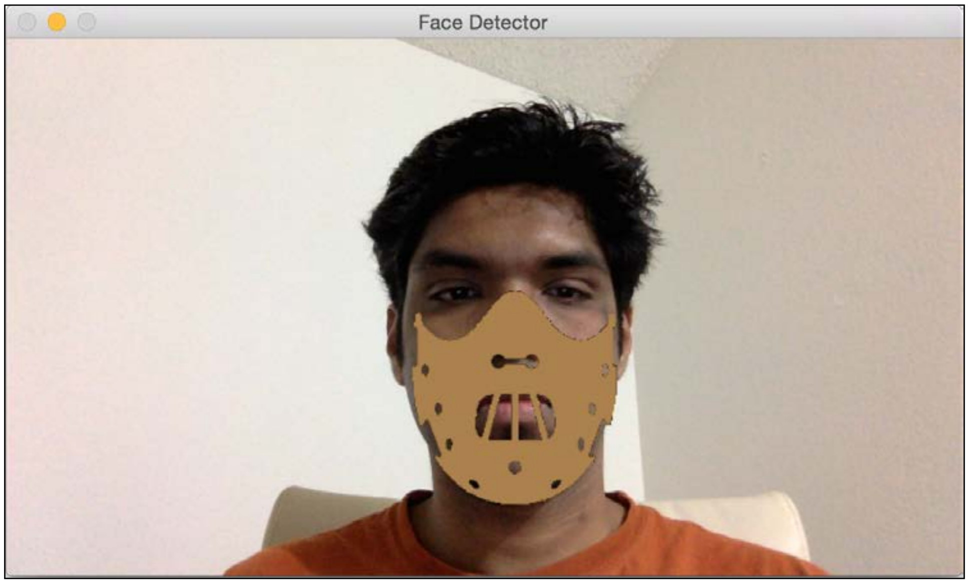
We need a classifier model that can be used to detect the faces in an image. OpenCV provides an xml file that can be used for this purpose. We use the function `CascadeClassifier` to load the xml file. Once we start capturing the input frames from the webcam, we convert it to grayscale and use the function `detectMultiScale` to get the bounding boxes for all the faces in the current image. The second argument in this function specifies the jump in the scaling factor. As in, if we don't find an image in the current scale, the next size to check will be, in our case, 1.3 times bigger than the current size. The last parameter is a threshold that specifies the number of adjacent rectangles needed to keep the current rectangle. It can be used to increase the robustness of the face detector.

Fun with faces

Now that we know how to detect and track faces, let's have some fun with it. When we capture a video stream from the webcam, we can overlay funny masks on top of our faces. It will look something like this next image:



If you are a fan of Hannibal, you can try this next one:



Let's look at the code to see how to overlay the skull mask on top of the face in the input video stream:

```
import cv2
import numpy as np

face_cascade =
cv2.CascadeClassifier('./cascade_files/haarcascade_frontalface_alt.
xml')

face_mask = cv2.imread('mask_hannibal.png')
h_mask, w_mask = face_mask.shape[:2]

if face_cascade.empty():
    raise IOError('Unable to load the face cascade classifier
xml file')

cap = cv2.VideoCapture(0)
scaling_factor = 0.5

while True:
    ret, frame = cap.read()
    frame = cv2.resize(frame, None, fx=scaling_factor,
fy=scaling_factor, interpolation=cv2.INTER_AREA)
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    face_rects = face_cascade.detectMultiScale(gray, 1.3, 5)
    for (x,y,w,h) in face_rects:
        if h > 0 and w > 0:
            # Adjust the height and weight parameters depending
on the sizes and the locations. You need to play around with
these to make sure you get it right.
            h, w = int(1.4*h), int(1.0*w)
            y -= 0.1*h

            # Extract the region of interest from the image
            frame_roi = frame[y:y+h, x:x+w]
            face_mask_small = cv2.resize(face_mask, (w, h),
interpolation=cv2.INTER_AREA)

            # Convert color image to grayscale and threshold it
            gray_mask = cv2.cvtColor(face_mask_small, cv2.COLOR_
BGR2GRAY)
            ret, mask = cv2.threshold(gray_mask, 180, 255,
cv2.THRESH_BINARY_INV)

            # Create an inverse mask
            mask_inv = cv2.bitwise_not(mask)

            # Use the mask to extract the face mask region of
interest
            masked_face = cv2.bitwise_and(face_mask_small, face_mask_
small, mask=mask)

            # Use the inverse mask to get the remaining part of
the image
            masked_frame = cv2.bitwise_and(frame_roi,
frame_roi, mask=mask_inv)

            # add the two images to get the final output
            frame[y:y+h, x:x+w] = cv2.add(masked_face,
masked_frame)

    cv2.imshow('Face Detector', frame)

    c = cv2.waitKey(1)
    if c == 27:
        break

cap.release()
cv2.destroyAllWindows()
```

Under the hood

Just like before, we first load the face cascade classifier xml file. The face detection steps work as usual. We start the infinite loop and keep detecting the face in every frame. Once we know where the face is, we need to modify the coordinates a bit to make sure the mask fits properly. This manipulation process is subjective and depends on the mask in question. Different masks require different levels of adjustments to make it look more natural. We extract the region-of-interest from the input frame in the following line:

```
frame_roi = frame[y:y+h, x:x+w]
```

Now that we have the required region-of-interest, we need to overlay the mask on top of this. So we resize the input mask to make sure it fits in this region-of-interest. The input mask has a white background. So if we just overlay this on top of the region-of-interest, it will look unnatural because of the white background. We need to overlay only the skull-mask pixels and the remaining area should be transparent.

So in the next step, we create a mask by thresholding the skull image. Since the background is white, we threshold the image so that any pixel with an intensity value greater than 180 becomes 0, and everything else becomes 255. As far as the frame region-of-interest is concerned, we need to black out everything in this mask region. We can do that by simply using the inverse of the mask we just created. Once we have the masked versions of the skull image and the input region-of-interest, we just add them up to get the final image.

Detecting eyes

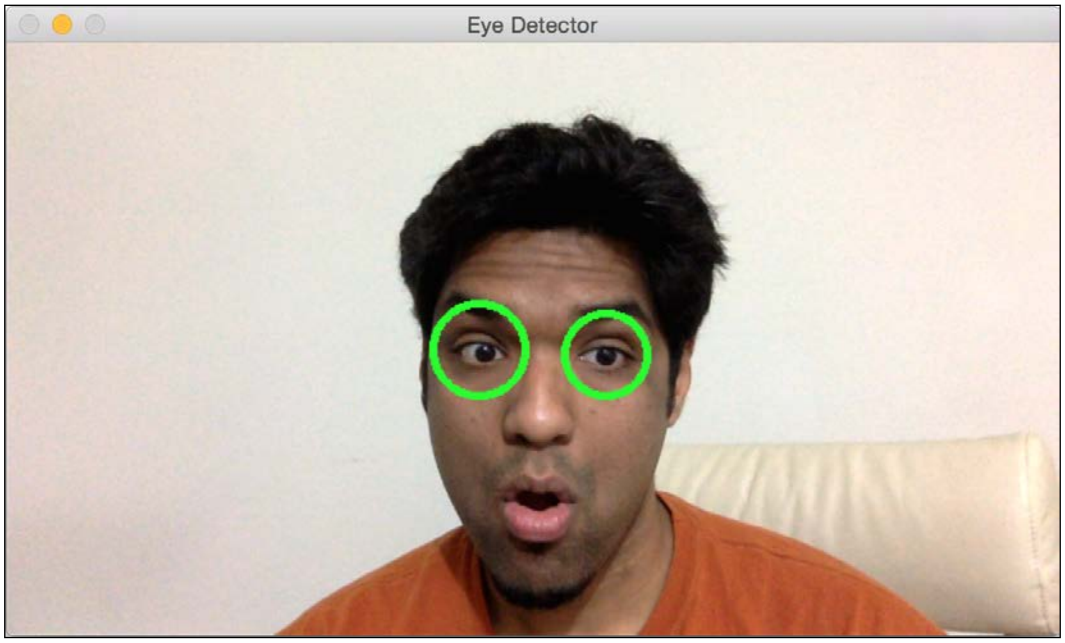
Now that we understand how to detect faces, we can generalize the concept to detect other body parts too. It's important to understand that Viola-Jones framework can be applied to any object. The accuracy and robustness will depend on the uniqueness of the object. For example, a human face has very unique characteristics, so it's easy to train our system to be robust. On the other hand, an object like towel is too generic, and there are no distinguishing characteristics as such; so it's more difficult to build a robust towel detector.

Let's see how to build an eye detector:

```
import cv2
import numpy as np

face_cascade = cv2.CascadeClassifier('./cascade_files/haarcascade_
frontalface_alt.xml')
eye_cascade = cv2.CascadeClassifier('./cascade_files/haarcascade_eye.
xml')
```

```
if face_cascade.empty():
  raise IOError('Unable to load the face cascade classifier xml file')

if eye_cascade.empty():
  raise IOError('Unable to load the eye cascade classifier xml file')

cap = cv2.VideoCapture(0)
ds_factor = 0.5

while True:
    ret, frame = cap.read()
    frame = cv2.resize(frame, None, fx=ds_factor, fy=ds_factor,
interpolation=cv2.INTER_AREA)
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

    faces = face_cascade.detectMultiScale(gray, 1.3, 5)
    for (x,y,w,h) in faces:
        roi_gray = gray[y:y+h, x:x+w]
        roi_color = frame[y:y+h, x:x+w]
        eyes = eye_cascade.detectMultiScale(roi_gray)
        for (x_eye,y_eye,w_eye,h_eye) in eyes:
            center = (int(x_eye + 0.5*w_eye), int(y_eye + 0.5*h_eye))
            radius = int(0.3 * (w_eye + h_eye))
            color = (0, 255, 0)
            thickness = 3
            cv2.circle(roi_color, center, radius, color, thickness)

    cv2.imshow('Eye Detector', frame)

    c = cv2.waitKey(1)
    if c == 27:
        break

cap.release()
cv2.destroyAllWindows()
```

If you run this program, the output will look something like the following image:

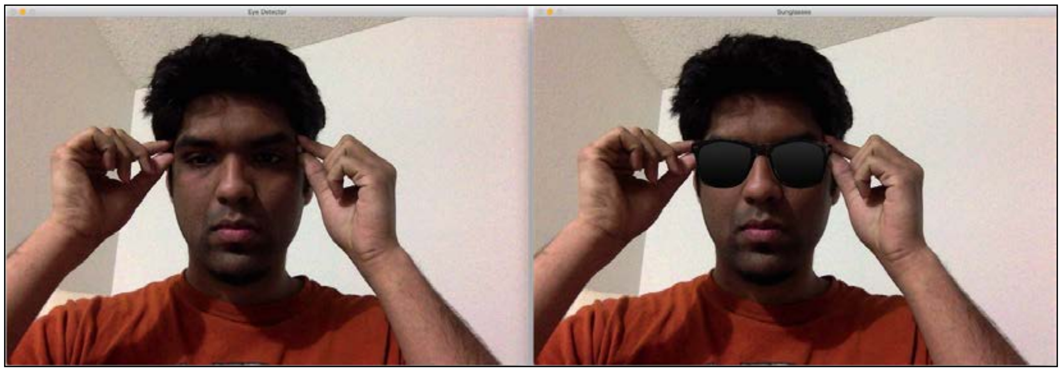


Afterthought

If you notice, the program looks very similar to the face detection program. Along with loading the face detection cascade classifier, we load the eye detection cascade classifier as well. Technically, we don't need to use the face detector. But we know that eyes are always on somebody's face. We use this information and search for eyes only in the relevant region of interest, that is the face. We first detect the face, and then run the eye detector on this sub-image. This way, it's faster and more efficient.

Fun with eyes

Now that we know how to detect eyes in an image, let's see if we can do something fun with it. We can do something like what is shown in the following screenshot:



Let's look at the code to see how to do something like this:

```
import cv2
import numpy as np

face_cascade =
cv2.CascadeClassifier('./cascade_files/haarcascade_frontalface_alt.
xml')
eye_cascade = cv2.CascadeClassifier('./cascade_files/haarcascade_eye.
xml')

if face_cascade.empty():
  raise IOError('Unable to load the face cascade classifier
xml file')

if eye_cascade.empty():
  raise IOError('Unable to load the eye cascade classifier xml
file')

img = cv2.imread('input.jpg')
sunglasses_img = cv2.imread('sunglasses.jpg')

gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

centers = []
faces = face_cascade.detectMultiScale(gray, 1.3, 5)

for (x,y,w,h) in faces:
    roi_gray = gray[y:y+h, x:x+w]
    roi_color = img[y:y+h, x:x+w]
    eyes = eye_cascade.detectMultiScale(roi_gray)
    for (x_eye,y_eye,w_eye,h_eye) in eyes:
        centers.append((x + int(x_eye + 0.5*w_eye), y +
int(y_eye + 0.5*h_eye)))

if len(centers) > 0:
    # Overlay sunglasses; the factor 2.12 is customizable
depending on the size of the face
    sunglasses_width = 2.12 * abs(centers[1][0] -
centers[0][0])
    overlay_img = np.ones(img.shape, np.uint8) * 255
    h, w = sunglasses_img.shape[:2]
    scaling_factor = sunglasses_width / w
    overlay_sunglasses = cv2.resize(sunglasses_img, None,
fx=scaling_factor,
            fy=scaling_factor, interpolation=cv2.INTER_AREA)

    x = centers[0][0] if centers[0][0] < centers[1][0] else
centers[1][0]

    # customizable X and Y locations; depends on the size of
the face
    x -= 0.26*overlay_sunglasses.shape[1]
    y += 0.85*overlay_sunglasses.shape[0]

    h, w = overlay_sunglasses.shape[:2]
    overlay_img[y:y+h, x:x+w] = overlay_sunglasses

    # Create mask
    gray_sunglasses = cv2.cvtColor(overlay_img,
cv2.COLOR_BGR2GRAY)
    ret, mask = cv2.threshold(gray_sunglasses, 110, 255,
cv2.THRESH_BINARY)
    mask_inv = cv2.bitwise_not(mask)
    temp = cv2.bitwise_and(img, img, mask=mask)
    temp2 = cv2.bitwise_and(overlay_img, overlay_img,
mask=mask_inv)
    final_img = cv2.add(temp, temp2)

    cv2.imshow('Eye Detector', img)
    cv2.imshow('Sunglasses', final_img)
    cv2.waitKey()
    cv2.destroyAllWindows()
```

Positioning the sunglasses

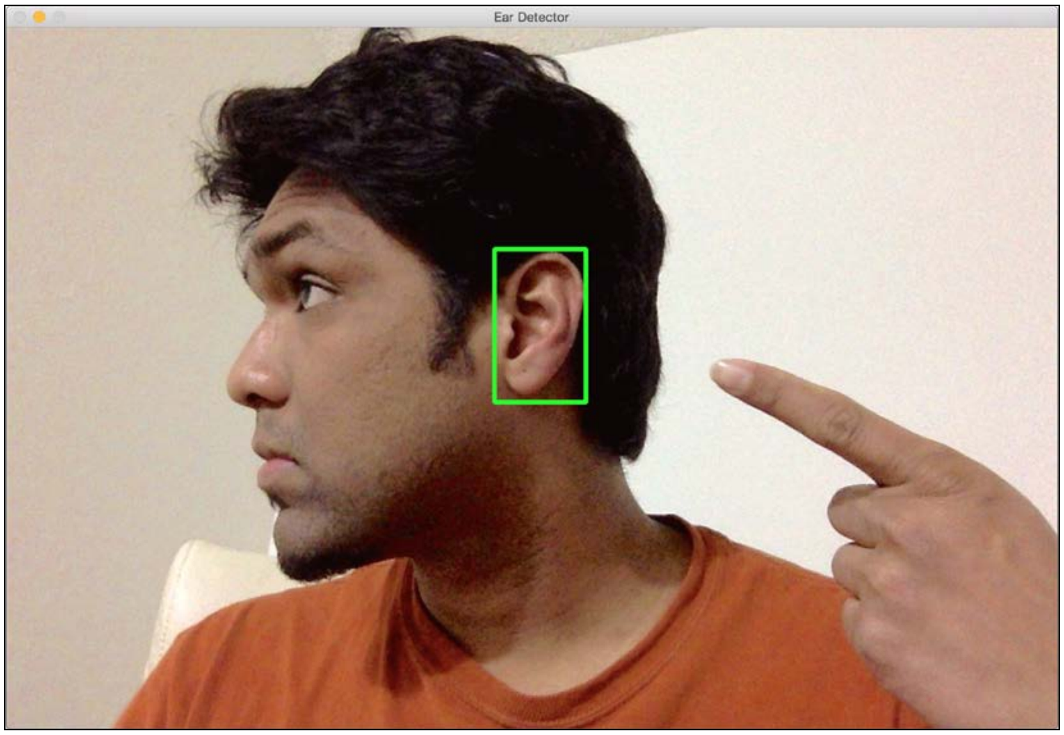
Just like we did earlier, we load the image and detect the eyes. Once we detect the eyes, we resize the sunglasses image to fit the current region of interest. To create the region of interest, we consider the distance between the eyes. We resize the image accordingly and then go ahead to create a mask. This is similar to what we did with the skull mask earlier. The positioning of the sunglasses on the face is subjective. So you will have to tinker with the weights if you want to use a different pair of sunglasses.

Detecting ears

Since we know how the pipeline works, let's just jump into the code:

```
import cv2
import numpy as np

left_ear_cascade =
cv2.CascadeClassifier('./cascade_files/haarcascade_mcs_leftear.xml')
right_ear_cascade = cv2.CascadeClassifier('./cascade_files/
haarcascade_mcs_rightear.xml')

if left_ear_cascade.empty():
  raise IOError('Unable to load the left ear cascade
classifier xml file')

if right_ear_cascade.empty():
  raise IOError('Unable to load the right ear cascade classifier xml
file')

img = cv2.imread('input.jpg')

gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

left_ear = left_ear_cascade.detectMultiScale(gray, 1.3, 5)
right_ear = right_ear_cascade.detectMultiScale(gray, 1.3, 5)

for (x,y,w,h) in left_ear:
    cv2.rectangle(img, (x,y), (x+w,y+h), (0,255,0), 3)

for (x,y,w,h) in right_ear:
    cv2.rectangle(img, (x,y), (x+w,y+h), (255,0,0), 3)

cv2.imshow('Ear Detector', img)
cv2.waitKey()
cv2.destroyAllWindows()
```

If you run the above code on an image, you should see something like the following screenshot:



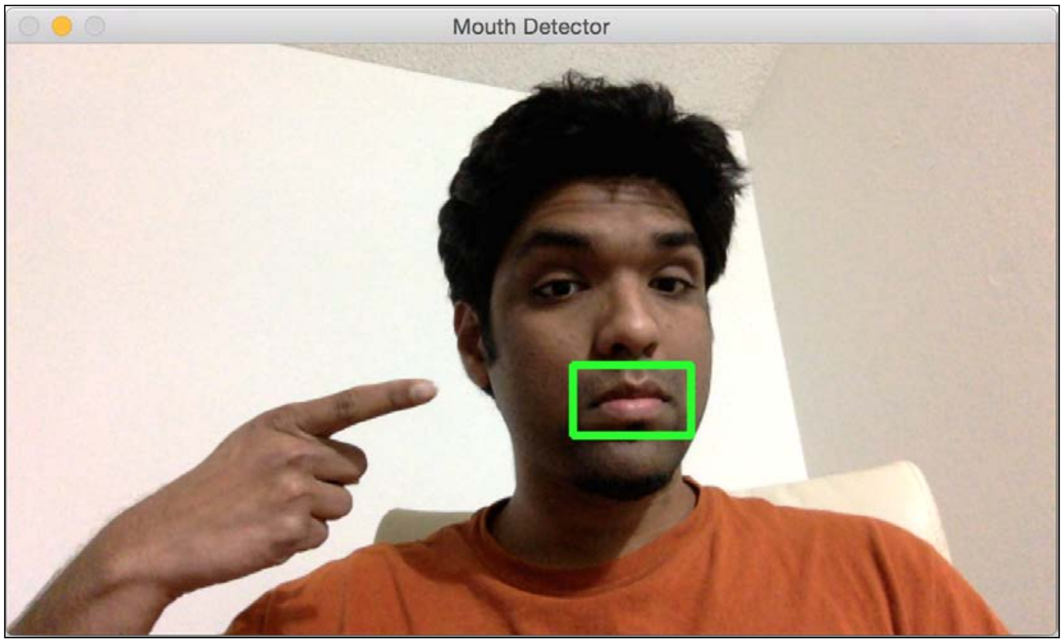
Detecting a mouth

Following is the code:

```
import cv2
import numpy as np

mouth_cascade =
cv2.CascadeClassifier('./cascade_files/haarcascade_mcs_mouth.xml')

if mouth_cascade.empty():
  raise IOError('Unable to load the mouth cascade classifier
xml file')

cap = cv2.VideoCapture(0)
ds_factor = 0.5
```

```
while True:
    ret, frame = cap.read()
    frame = cv2.resize(frame, None, fx=ds_factor, fy=ds_factor,
interpolation=cv2.INTER_AREA)
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

    mouth_rects = mouth_cascade.detectMultiScale(gray, 1.7, 11)
    for (x,y,w,h) in mouth_rects:
        y = int(y - 0.15*h)
        cv2.rectangle(frame, (x,y), (x+w,y+h), (0,255,0), 3)
        break

    cv2.imshow('Mouth Detector', frame)

    c = cv2.waitKey(1)
    if c == 27:
        break

cap.release()
cv2.destroyAllWindows()
```

Following is what the output looks like:



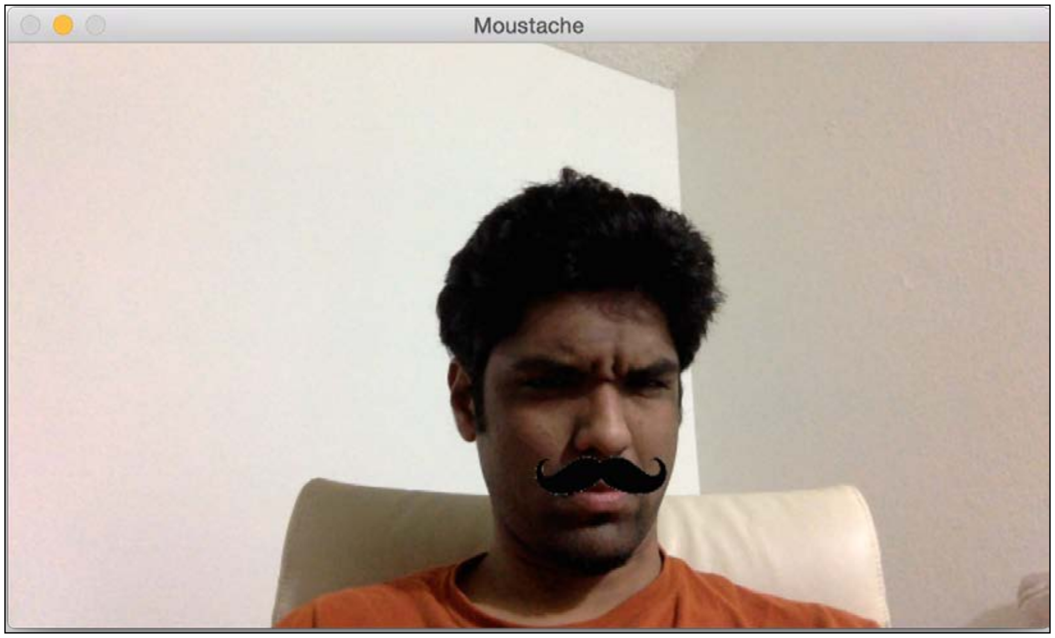
It's time for a moustache

Let's overlay a moustache on top:

```
import cv2
import numpy as np

mouth_cascade =
cv2.CascadeClassifier('./cascade_files/haarcascade_mcs_mouth.xml')

moustache_mask = cv2.imread('../images/moustache.png')
h_mask, w_mask = moustache_mask.shape[:2]

if mouth_cascade.empty():
  raise IOError('Unable to load the mouth cascade classifier
xml file')

cap = cv2.VideoCapture(0)
scaling_factor = 0.5

while True:
    ret, frame = cap.read()
    frame = cv2.resize(frame, None, fx=scaling_factor,
fy=scaling_factor, interpolation=cv2.INTER_AREA)
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

    mouth_rects = mouth_cascade.detectMultiScale(gray, 1.3, 5)
    if len(mouth_rects) > 0:
        (x,y,w,h) = mouth_rects[0]
        h, w = int(0.6*h), int(1.2*w)
        x -= 0.05*w
        y -= 0.55*h
        frame_roi = frame[y:y+h, x:x+w]
        moustache_mask_small = cv2.resize(moustache_mask, (w,
h), interpolation=cv2.INTER_AREA)

        gray_mask = cv2.cvtColor(moustache_mask_small,
cv2.COLOR_BGR2GRAY)
        ret, mask = cv2.threshold(gray_mask, 50, 255,
cv2.THRESH_BINARY_INV)
        mask_inv = cv2.bitwise_not(mask)
        masked_mouth = cv2.bitwise_and(moustache_mask_small,
moustache_mask_small, mask=mask)
        masked_frame = cv2.bitwise_and(frame_roi, frame_roi,
mask=mask_inv)
        frame[y:y+h, x:x+w] = cv2.add(masked_mouth,
masked_frame)
```

```
        cv2.imshow('Moustache', frame)

        c = cv2.waitKey(1)
        if c == 27:
            break

    cap.release()
    cv2.destroyAllWindows()
```

Here's what it looks like:



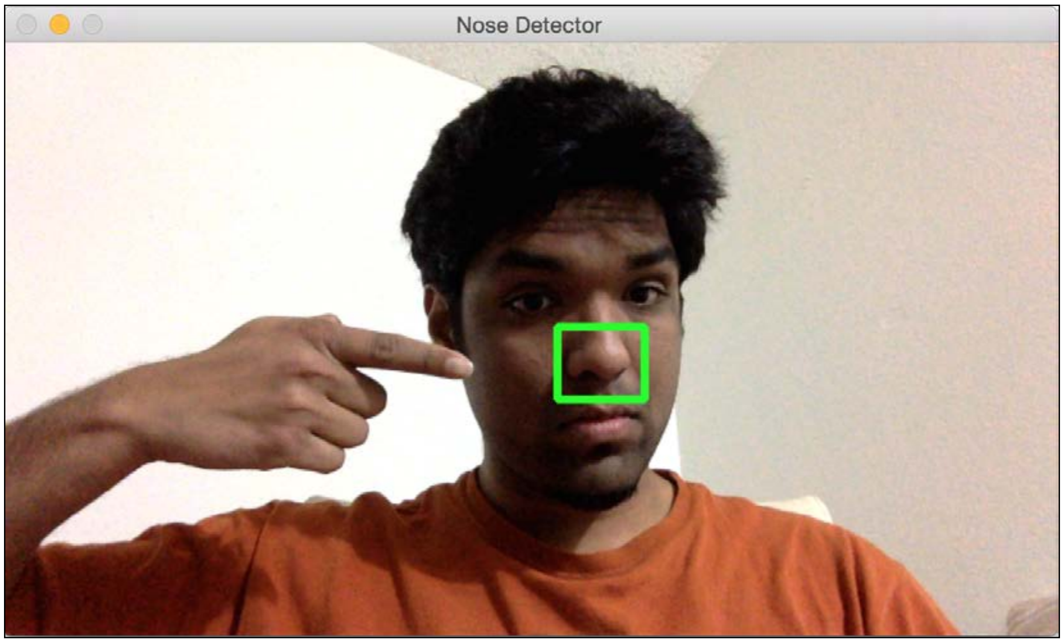
Detecting a nose

The following program shows how you detect a nose:

```
    import cv2
    import numpy as np

    nose_cascade =
    cv2.CascadeClassifier('./cascade_files/haarcascade_mcs_nose.xml')

    if nose_cascade.empty():
      raise IOError('Unable to load the nose cascade classifier
    xml file')
```

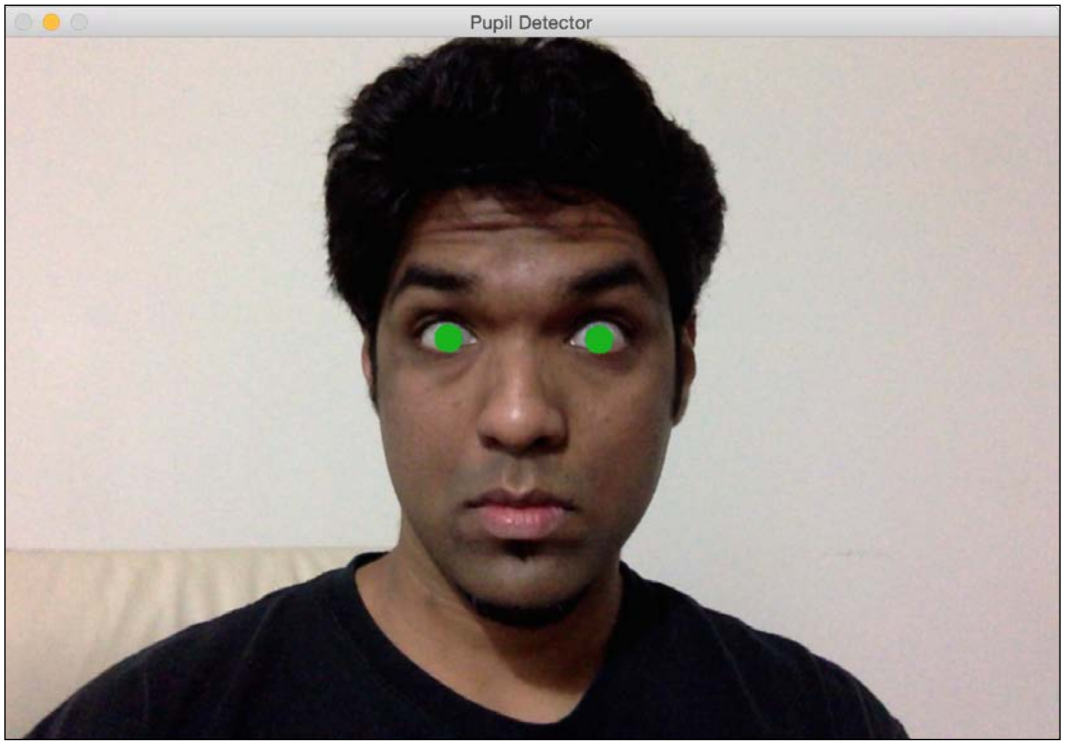
```
cap = cv2.VideoCapture(0)
ds_factor = 0.5

while True:
    ret, frame = cap.read()
    frame = cv2.resize(frame, None, fx=ds_factor, fy=ds_factor,
interpolation=cv2.INTER_AREA)
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

    nose_rects = nose_cascade.detectMultiScale(gray, 1.3, 5)
    for (x,y,w,h) in nose_rects:
        cv2.rectangle(frame, (x,y), (x+w,y+h), (0,255,0), 3)
        break

    cv2.imshow('Nose Detector', frame)

    c = cv2.waitKey(1)
    if c == 27:
        break

cap.release()
cv2.destroyAllWindows()
```

The output looks something like the following image:



Detecting pupils

We are going to take a different approach here. Pupils are too generic to take the Haar cascade approach. We will also get a sense of how to detect things based on their shape. Following is what the output will look like:



Let's see how to build the pupil detector:

```
import math

import cv2
import numpy as np

img = cv2.imread('input.jpg')
scaling_factor = 0.7

img = cv2.resize(img, None, fx=scaling_factor,
fy=scaling_factor, interpolation=cv2.INTER_AREA)
cv2.imshow('Input', img)
gray = cv2.cvtColor(~img, cv2.COLOR_BGR2GRAY)
```

```
ret, thresh_gray = cv2.threshold(gray, 220, 255,
cv2.THRESH_BINARY)
contours, hierarchy = cv2.findContours(thresh_gray,
cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_NONE)

for contour in contours:
    area = cv2.contourArea(contour)
    rect = cv2.boundingRect(contour)
    x, y, width, height = rect
    radius = 0.25 * (width + height)

    area_condition = (100 <= area <= 200)
    symmetry_condition = (abs(1 - float(width)/float(height))
<= 0.2)
    fill_condition = (abs(1 - (area / (math.pi * math.pow(radius,
2.0)))) <= 0.3)

    if area_condition and symmetry_condition and fill_condition:
        cv2.circle(img, (int(x + radius), int(y + radius)),
int(1.3*radius), (0,180,0), -1)

cv2.imshow('Pupil Detector', img)

c = cv2.waitKey()
cv2.destroyAllWindows()
```

If you run this program, you will see the output as shown earlier.

Deconstructing the code

As we discussed earlier, we are not going to use Haar cascade to detect pupils. If we can't use a pre-trained classifier, then how are we going to detect the pupils? Well, we can use shape analysis to detect the pupils. We know that pupils are circular, so we can use this information to detect them in the image. We invert the input image and then convert it into grayscale image as shown in the following line:

```
gray = cv2.cvtColor(~img, cv2.COLOR_BGR2GRAY)
```

As we can see here, we can invert an image using the tilde operator. Inverting the image is helpful in our case because the pupil is black in color, and black corresponds to a low pixel value. We then threshold the image to make sure that there are only black and white pixels. Now, we have to find out the boundaries of all the shapes. OpenCV provides a nice function to achieve this, that is `findContours`. We will discuss more about this in the upcoming chapters. But for now, all we need to know is that this function returns the set of boundaries of all the shapes that are found in the image.

The next step is to identify the shape of the pupil and discard the rest. We will use certain properties of the circle to zero-in on this shape. Let's consider the ratio of width to height of the bounding rectangle. If the shape is a circle, this ratio will be 1. We can use the function `boundingRect` to obtain the coordinates of the bounding rectangle. Let's consider the area of this shape. If we roughly compute the radius of this shape and use the formula for the area of the circle, then it should be close to the area of this contour. We can use the function `contourArea` to compute the area of any contour in the image. So we can use these conditions and filter out the shapes. After we do that, we are left with two pupils in the image. We can refine it further by limiting the search region to the face or the eyes. Since you know how to detect faces and eyes, you can give it a try and see if you can get it working for a live video stream.

Summary

In this chapter, we discussed Haar cascades and integral images. We understood how the face detection pipeline is built. We learnt how to detect and track faces in a live video stream. We discussed how to use the face detection pipeline to detect various body parts like eyes, ears, nose, and mouth. We learnt how to overlay masks on top on the input image using the results of body parts detection. We used the principles of shape analysis to detect the pupils.

In the next chapter, we are going to discuss feature detection and how it can be used to understand the image content.

4
Extracting Features from an Image

In this chapter, we are going to learn how to detect salient points, also known as keypoints, in an image. We will discuss why these keypoints are important and how we can use them to understand the image content. We will talk about different techniques that can be used to detect these keypoints, and understand how we can extract features from a given image.

By the end of this chapter, you will know:

- What are keypoints and why do we care about them
- How to detect keypoints
- How to use keypoints for image content analysis
- The different techniques to detect keypoints
- How to build a feature extractor

Why do we care about keypoints?

Image content analysis refers to the process of understanding the content of an image so that we can take some action based on that. Let's take a step back and talk about how humans do it. Our brain is an extremely powerful machine that can do complicated things very quickly. When we look at something, our brain automatically creates a footprint based on the "interesting" aspects of that image. We will discuss what interesting means as we move along this chapter.