10
Augmented Reality

In this chapter, you are going to learn about augmented reality and how you can use it to build cool applications. We will discuss pose estimation and plane tracking. You will learn how to map the coordinates from 3D to 2D, and how we can overlay graphics on top of a live video.

By the end of this chapter, you will know:

- The premise of augmented reality
- What pose estimation is
- How to track a planar object
- How to map coordinates from 3D to 2D
- How to overlay graphics on top of a video in real time

What is the premise of augmented reality?

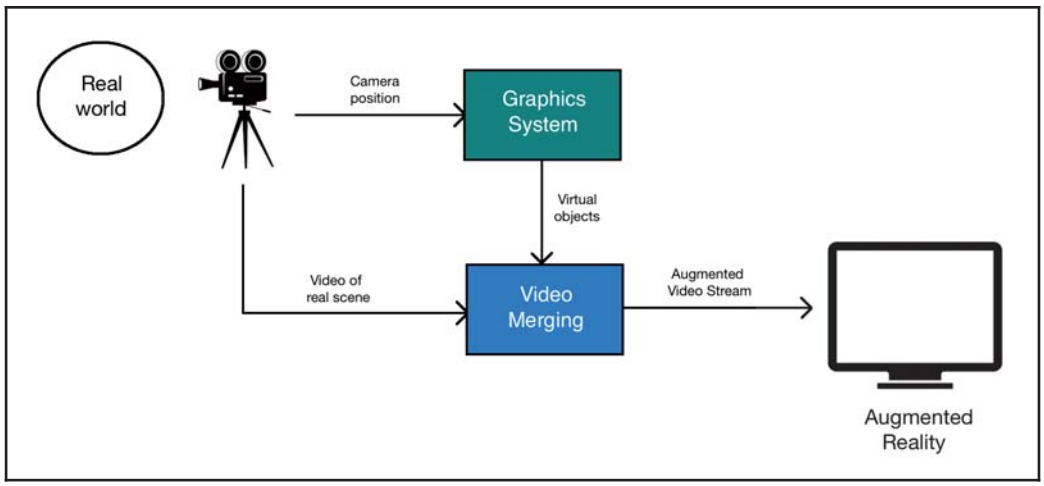
Before we jump into all the fun stuff, let's understand what augmented reality means. You will have probably seen the term augmented reality being used in a variety of contexts. So, we should understand the premise of augmented reality before we start discussing the implementation details. Augmented reality refers to the superimposition of computer-generated input, such as imagery, sounds, graphics, and text, on top of the real world.

Augmented reality tries to blur the line between what's real and what's computer-generated by seamlessly merging information and enhancing what we see and feel. It is actually closely related to a concept called mediated reality, where a computer modifies our view of reality. As a result of this, the technology works by enhancing our current perception of reality. Now, the challenge here is to make it look seamless to the user. It's easy to just overlay something on top of the input video, but we need to make it look as though it is part of the video. The user should feel that the computer-generated input closely reflects the real world. This is what we want to achieve when we build an augmented reality system.

Computer vision research in this context explores how we can apply computer-generated imagery to live video streams so that we can enhance the perception of the real world. Augmented reality technology has a wide variety of applications, including, but not limited to, head-mounted displays, automobiles, data visualization, gaming, construction, and so on. Now that we have powerful smartphones and smarter machines, we can build high-end augmented reality applications with ease.

What does an augmented reality system look like?

Let's consider the following figure:



As we can see here, the camera captures real-world video to get the reference point. The graphics system generates the virtual objects that need to be overlaid on top of the video. Now, the video-merging block is where all the magic happens. This block should be smart enough to understand how to overlay the virtual objects on top of the real world in the best way possible.

Geometric transformations for augmented reality

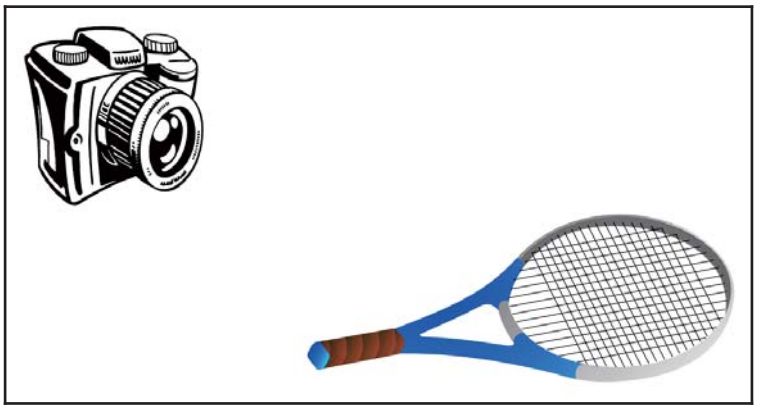
The result of augmented reality is amazing, but there are a lot of mathematical things going on underneath. Augmented reality utilizes a lot of geometric transformations and associated mathematical functions to make sure everything looks smooth. When talking about a live video for augmented reality, we need to precisely register the virtual objects on top of the real world. To understand this better, let's think of it as an alignment of two cameras: the real one through which we see the world, and the virtual one that projects the computer-generated graphical objects.

In order to build an augmented reality system, the following geometric transformations need to be established:

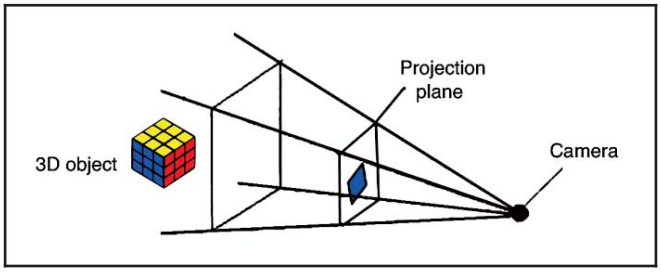
- **Object-to-scene**: This transformation refers to transforming the 3D coordinates of a virtual object and expressing them in the coordinate frame of our real-world scene. This ensures that we are placing the virtual object in the right location.
- **Scene-to-camera**: This transformation refers to the pose of the camera in the real world. By *pose*, we mean the orientation and location of the camera. We need to estimate the point of view of the camera so that we know how to overlay the virtual object.
- **Camera-to-image**: This refers to the calibration parameters of the camera. This defines how we can project a 3D object onto a 2D image plane. This is the image that we will actually see in the end.

Consider the following image:

As we can see here, the car is trying to fit into the scene but it looks very artificial. If we don't convert the coordinates in the right way, the car will look unnatural. This is what we were saying about object-to-scene transformation! Once we transform the 3D coordinates of the virtual object into the coordinate frame of the real world, we need to estimate the pose of the camera:



We need to understand the position and rotation of the camera because that's what the user will see. Once we estimate the camera pose, we are ready to put this 3D scene on a 2D image:

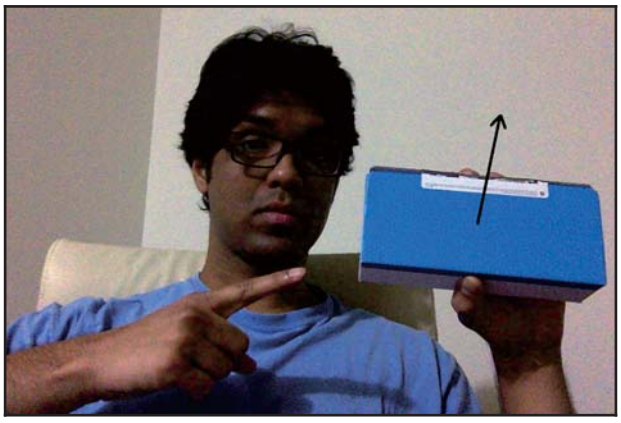


Once we have these transformations, we can build the complete system.

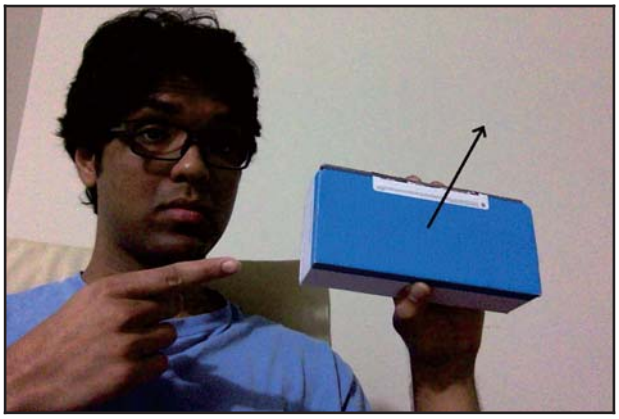
What is pose estimation?

Before we proceed, we need to understand how to estimate the camera pose. This is a very critical step in an augmented reality system and we need to get it right if we want our experience to be seamless. In the world of augmented reality, we overlay graphics on top of an object in real time. In order to do that, we need to know the location and orientation of the camera, and we need to do it quickly. This is where pose estimation becomes very important. If you don't track the pose correctly, the overlaid graphics will look unnatural.

Consider the following image:



The arrow indiactes that the surface is normal. Let's say the object changes its orientation:

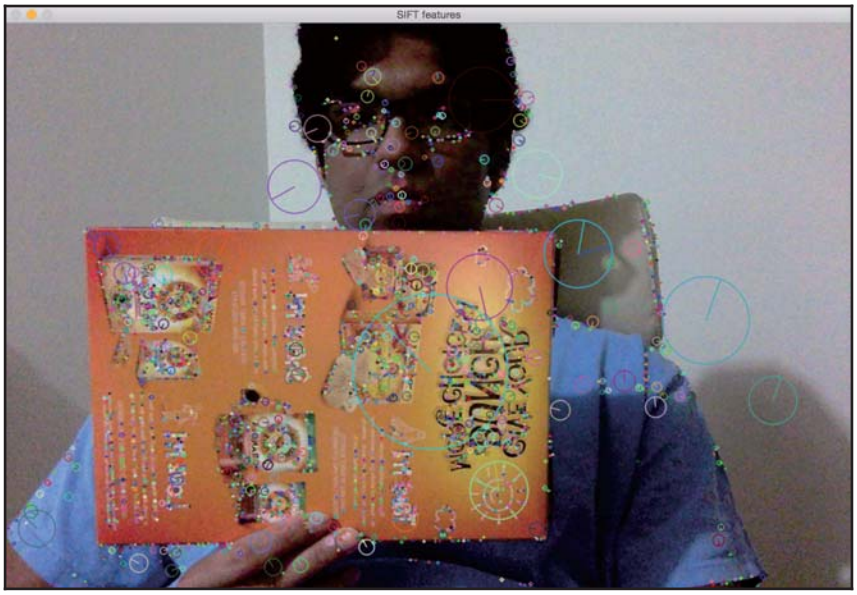


Now, even though the location is the same, the orientation has changed. We need to have this information so that the overlaid graphics look natural. We need to make sure that the graphic is aligned with this orientation and position.

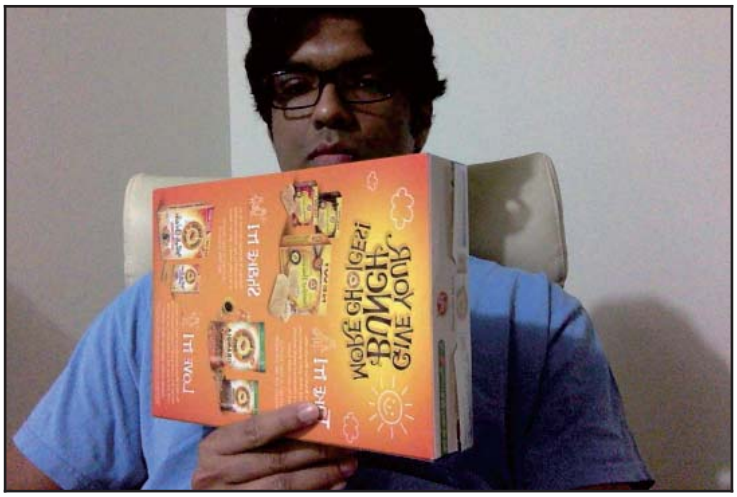
How to track planar objects

Now that you understand what pose estimation is, let's see how you can use it to track planar objects. Let's consider the following planar object:

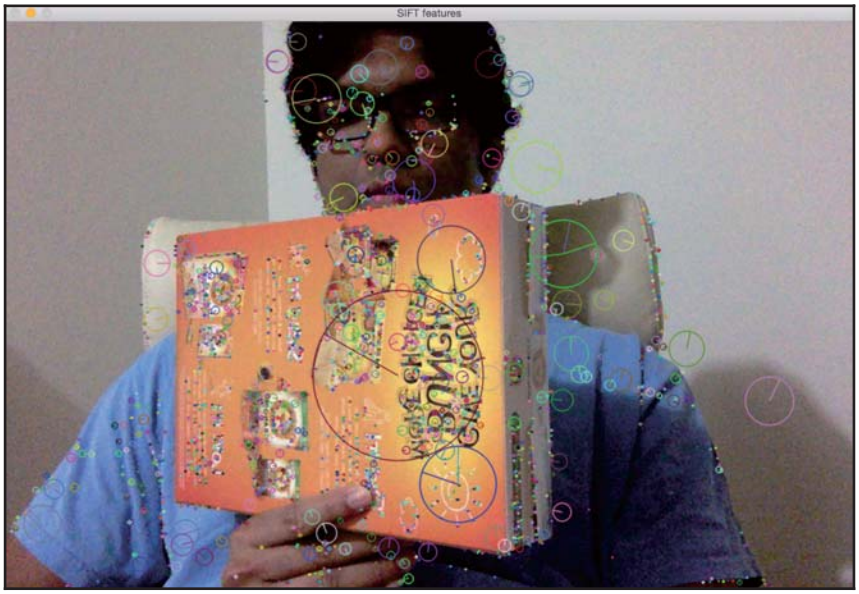
Now, if we extract feature points from this image, we will see something like this:



Let's tilt the cardboard box:



As we can see, the cardboard box is tilted in this image. Now, if we want to make sure our virtual object is overlaid on top of this surface, we need to gather this planar tilt information. One way to do this is by using the relative positions of the feature points. If we extract the feature points from the preceding image, it will look like this:



As you can see, the feature points got closer horizontally on the far end of the plane as compared to the ones on the near end:

So, we can utilize this information to extract the orientation information from the image. If you remember, we discussed perspective transformation in detail when we were discussing geometric transformations, as well as panoramic imaging. All we need to do is use those two sets of points and extract the homography matrix. This homography matrix will tell us how the cardboard box turned.

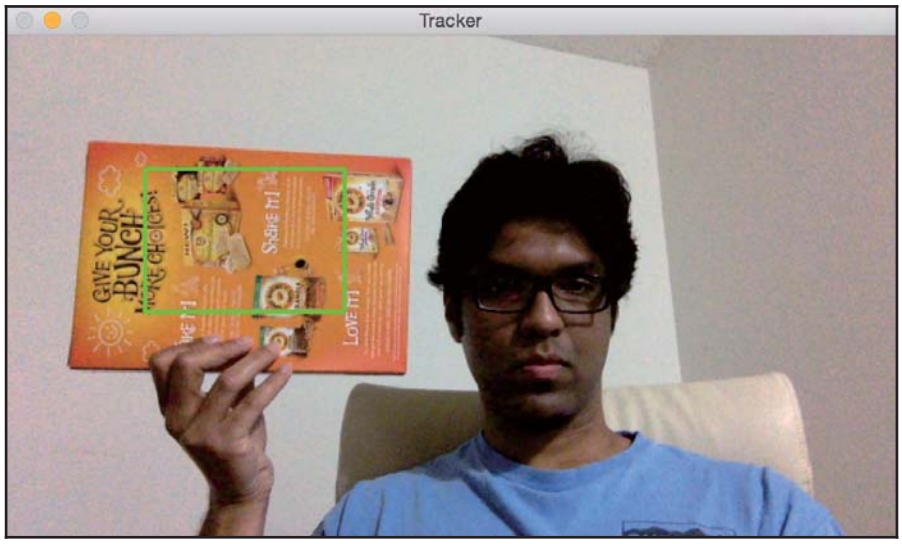
Consider the following image:

First, we will start by selecting the region of interest using the `ROISelector` class, and once we've done that, we will pass those coordinates to `PoseEstimator`:

```
class ROISelector(object):
    def __init__(self, win_name, init_frame, callback_func):
        self.callback_func = callback_func
        self.selected_rect = None
        self.drag_start = None
        self.tracking_state = 0
        event_params = {"frame": init_frame}
        cv2.namedWindow(win_name)
        cv2.setMouseCallback(win_name, self.mouse_event, event_params)

    def mouse_event(self, event, x, y, flags, param):
        x, y = np.int16([x, y])

        # Detecting the mouse button down event
        if event == cv2.EVENT_LBUTTONDOWN:
            self.drag_start = (x, y)
            self.tracking_state = 0
```

```
            if self.drag_start:
                if event == cv2.EVENT_MOUSEMOVE:
                    h, w = param["frame"].shape[:2]
                    xo, yo = self.drag_start
                    x0, y0 = np.maximum(0, np.minimum([xo, yo], [x, y]))
                    x1, y1 = np.minimum([w, h], np.maximum([xo, yo], [x, y]))
                    self.selected_rect = None

                    if x1-x0 > 0 and y1-y0 > 0:
                        self.selected_rect = (x0, y0, x1, y1)

                elif event == cv2.EVENT_LBUTTONUP:
                    self.drag_start = None
                    if self.selected_rect is not None:
                        self.callback_func(self.selected_rect)
                        self.selected_rect = None
                        self.tracking_state = 1

        def draw_rect(self, img, rect):
            if not rect: return False
            x_start, y_start, x_end, y_end = rect
            cv2.rectangle(img, (x_start, y_start), (x_end, y_end), (0, 255, 0),
    2)
            return True
```

In the following image, the region of interest the green rectangle:



We then extract feature points from this region of interest. Since we are tracking planar objects, the algorithm assumes that this region of interest is a plane. That's obvious, but it's better to state it explicitly! So make sure you have a cardboard box in your hand when you select this region of interest. Also, it'll be better if the cardboard box has a bunch of patterns and distinctive points so that it's easy to detect and track the feature points on it.

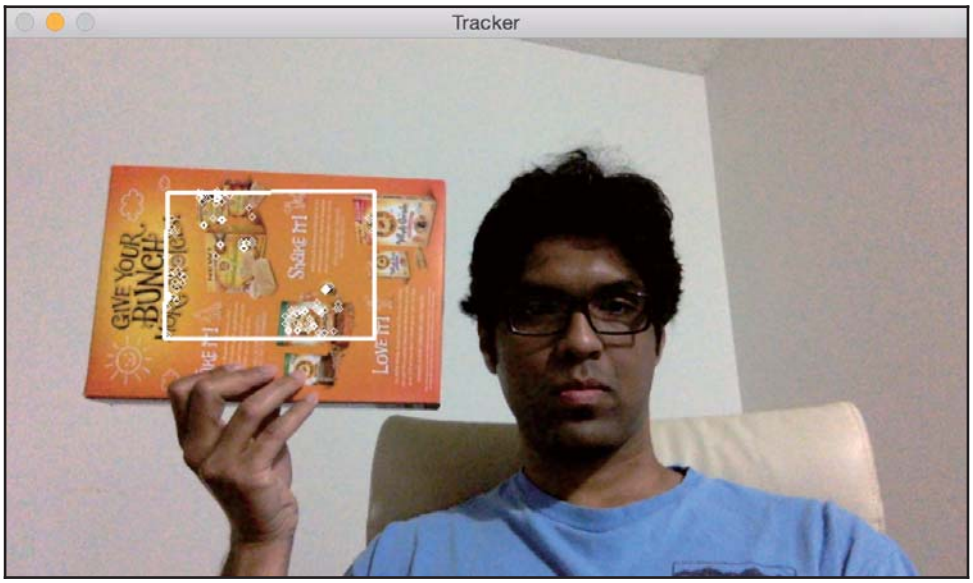
The `PoseEstimator` class will receive areas of interest from its method, `add_target()`, and will extract those feature points from them, which will allow us to track object movements:

```
class PoseEstimator(object):
    def __init__(self):
        # Use locality sensitive hashing algorithm
        flann_params = dict(algorithm = 6, table_number = 6, key_size = 12,
multi_probe_level = 1)

        self.min_matches = 10
        self.cur_target = namedtuple('Current', 'image, rect, keypoints,
descriptors, data')
        self.tracked_target = namedtuple('Tracked', 'target, points_prev,
points_cur, H, quad')

        self.feature_detector = cv2.ORB_create()
        self.feature_detector.setMaxFeatures(1000)
        self.feature_matcher = cv2.FlannBasedMatcher(flann_params, {})
        self.tracking_targets = []

    # Function to add a new target for tracking
    def add_target(self, image, rect, data=None):
        x_start, y_start, x_end, y_end = rect
        keypoints, descriptors = [], []
        for keypoint, descriptor in zip(*self.detect_features(image)):
            x, y = keypoint.pt
            if x_start <= x <= x_end and y_start <= y <= y_end:
                keypoints.append(keypoint)
                descriptors.append(descriptor)

        descriptors = np.array(descriptors, dtype='uint8')
        self.feature_matcher.add([descriptors])
        target = self.cur_target(image=image, rect=rect,
keypoints=keypoints, descriptors=descriptors, data=None)
        self.tracking_targets.append(target)

    # To get a list of detected objects
    def track_target(self, frame):
        self.cur_keypoints, self.cur_descriptors =
```

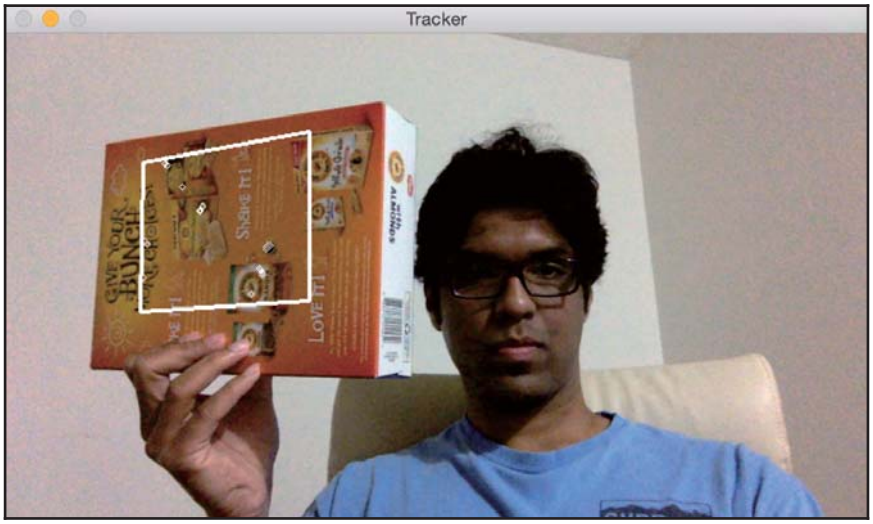
```
self.detect_features(frame)

        if len(self.cur_keypoints) < self.min_matches: return []
        try: matches = self.feature_matcher.knnMatch(self.cur_descriptors,
k=2)
        except Exception as e:
            print('Invalid target, please select another with features to
extract')
            return []
        matches = [match[0] for match in matches if len(match) == 2 and
match[0].distance < match[1].distance * 0.75]
        if len(matches) < self.min_matches: return []

        matches_using_index = [[] for _ in
range(len(self.tracking_targets))]
        for match in matches:
            matches_using_index[match.imgIdx].append(match)

        tracked = []
        for image_index, matches in enumerate(matches_using_index):
            if len(matches) < self.min_matches: continue

            target = self.tracking_targets[image_index]
            points_prev = [target.keypoints[m.trainIdx].pt for m in
matches]
            points_cur = [self.cur_keypoints[m.queryIdx].pt for m in
matches]
            points_prev, points_cur = np.float32((points_prev, points_cur))
            H, status = cv2.findHomography(points_prev, points_cur,
cv2.RANSAC, 3.0)
            status = status.ravel() != 0

            if status.sum() < self.min_matches: continue

            points_prev, points_cur = points_prev[status],
points_cur[status]

            x_start, y_start, x_end, y_end = target.rect
            quad = np.float32([[x_start, y_start], [x_end, y_start],
[x_end, y_end], [x_start, y_end]])
            quad = cv2.perspectiveTransform(quad.reshape(1, -1, 2),
H).reshape(-1, 2)
            track = self.tracked_target(target=target,
points_prev=points_prev, points_cur=points_cur, H=H, quad=quad)
            tracked.append(track)

        tracked.sort(key = lambda x: len(x.points_prev), reverse=True)
        return tracked
```

```
    # Detect features in the selected ROIs and return the keypoints and
descriptors
    def detect_features(self, frame):
        keypoints, descriptors =
self.feature_detector.detectAndCompute(frame, None)
        if descriptors is None: descriptors = []
        return keypoints, descriptors

    # Function to clear all the existing targets
    def clear_targets(self):
        self.feature_matcher.clear()
        self.tracking_targets = []
```

Let the tracking begin! We'll move the cardboard box around to see what happens:



As you can see, the feature points are being tracked inside the region of interest. Let's hold it at an angle and see what happens:



Looks like the feature points are being tracked properly. As we can see, the overlaid rectangle is changing its orientation according to the surface of the cardboard box.

Here is the code to do this:

```
import sys
from collections import namedtuple

import cv2
import numpy as np

class VideoHandler(object):
    def __init__(self, capId, scaling_factor, win_name):
        self.cap = cv2.VideoCapture(capId)
        self.pose_tracker = PoseEstimator()
        self.win_name = win_name
        self.scaling_factor = scaling_factor

        ret, frame = self.cap.read()
        self.rect = None
        self.frame = cv2.resize(frame, None, fx=scaling_factor,
fy=scaling_factor, interpolation=cv2.INTER_AREA)
        self.roi_selector = ROISelector(win_name, self.frame,
self.set_rect)
```

```
    def set_rect(self, rect):
        self.rect = rect
        self.pose_tracker.add_target(self.frame, rect)

    def start(self):
        paused = False
        while True:
            if not paused or self.frame is None:
                ret, frame = self.cap.read()
                scaling_factor = self.scaling_factor
                frame = cv2.resize(frame, None, fx=scaling_factor,
fy=scaling_factor, interpolation=cv2.INTER_AREA)
                if not ret: break
                self.frame = frame.copy()

            img = self.frame.copy()
            if not paused and self.rect is not None:
                tracked = self.pose_tracker.track_target(self.frame)
                for item in tracked:
                    cv2.polylines(img, [np.int32(item.quad)], True, (255,
255, 255), 2)
                    for (x, y) in np.int32(item.points_cur):
                        cv2.circle(img, (x, y), 2, (255, 255, 255))

            self.roi_selector.draw_rect(img, self.rect)
            cv2.imshow(self.win_name, img)
            ch = cv2.waitKey(1)
            if ch == ord(' '): paused = not paused
            if ch == ord('c'): self.pose_tracker.clear_targets()
            if ch == 27: break

if __name__ == '__main__':
    VideoHandler(0, 0.8, 'Tracker').start()
```

What happened inside the code?

To start with, we have a `PoseEstimator` class that does all the heavy lifting here. We need something to detect the features in the image and something to match the features between successive images. So we use the ORB feature detector and the Flann feature matcher for fast nearest neighbor searches within the extracted features. As you can see, we initialize the class with these parameters in the constructor.

Whenever we select a region of interest, we call the `add_target` method to add that to our list of tracking targets. This method just extracts the features from that region of interest and stores them in one of the class variables. Now that we have a target, we are ready to track it!

The `track_target` method handles all the tracking. We take the current frame and extract all the keypoints. However, we are not really interested in all the keypoints in the current frame of the video. We just want the keypoints that belong to our target object. So now our job is to find the closest keypoints in the current frame.

We now have a set of keypoints in the current frame and we have another set of keypoints from our target object in the previous frame. The next step is to extract the homography matrix from these matching points. This homography matrix tells us how to transform the overlaid rectangle so that it's aligned with the surface of the cardboard box. We just need to take this homography matrix and apply it to the overlaid rectangle to obtain the new positions of all the cardboard box's points.

How to augment our reality

Now that we know how to track planar objects, let's see how to overlay 3D objects on top of the real world. The objects are 3D but the video on our screen is 2D. So, the first step here is to understand how to map those 3D objects to 2D surfaces so that they look realistic. We just need to project those 3D points onto planar surfaces.

Mapping coordinates from 3D to 2D

Once we estimate the pose, we project the points from 3D to 2D. Consider the following image:

As we can see here, the TV remote control is a 3D object but we are seeing it on a 2D plane. Now if we move it around, it will look like this:

This 3D object is still on a 2D plane. The object has moved to a different location and the distance from the camera has changed as well. How do we compute these coordinates? We need a mechanism to map this 3D object onto the 2D surface. This is where 3D-to-2D projection becomes really important.

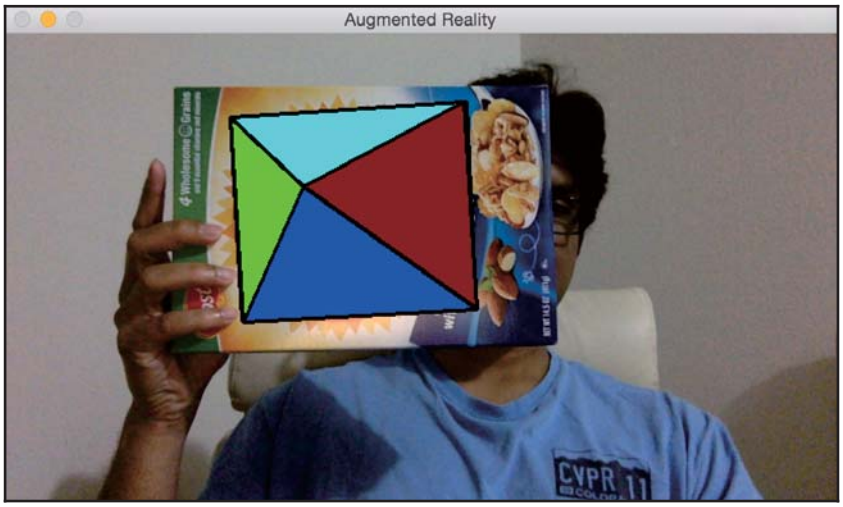
We just need to estimate the initial camera pose to start with. Now, let's assume that the intrinsic parameters of the camera are already known. So, we can just use the `solvePnP` function in OpenCV to estimate the camera's pose. This function is used to estimate the object's pose using a set of points as seen in the following code. You can read more about it at `http://docs.opencv.org/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html#bool`:

```
solvePnP(InputArray objectPoints, InputArray imagePoints, InputArray
cameraMatrix, InputArray distCoeffs, OutputArray rvec, OutputArray tvec,
bool useExtrinsicGuess, int flags)
```

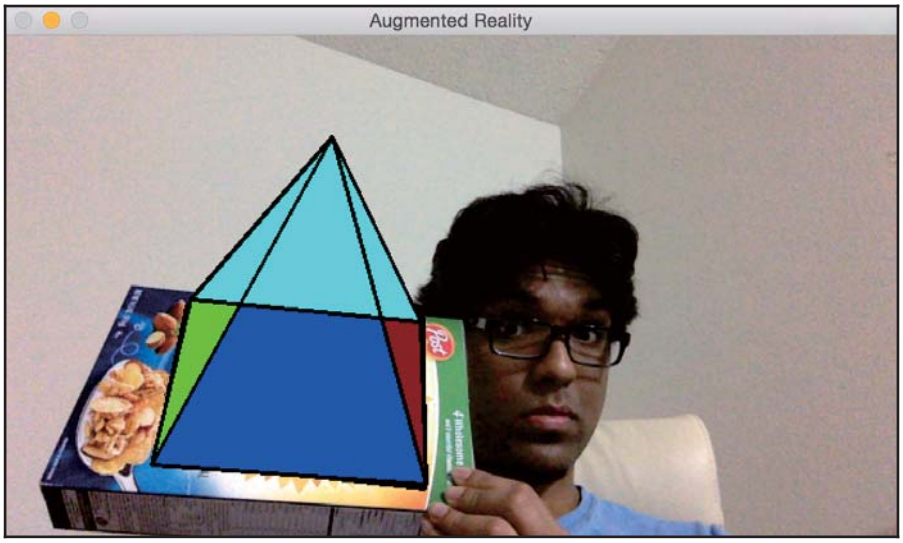
Once we do this, we need to project these points onto a 2D plane. We use the OpenCV `projectPoints` function to do this. This function calculates the projections of those 3D points onto the 2D plane.

How to overlay 3D objects on a video

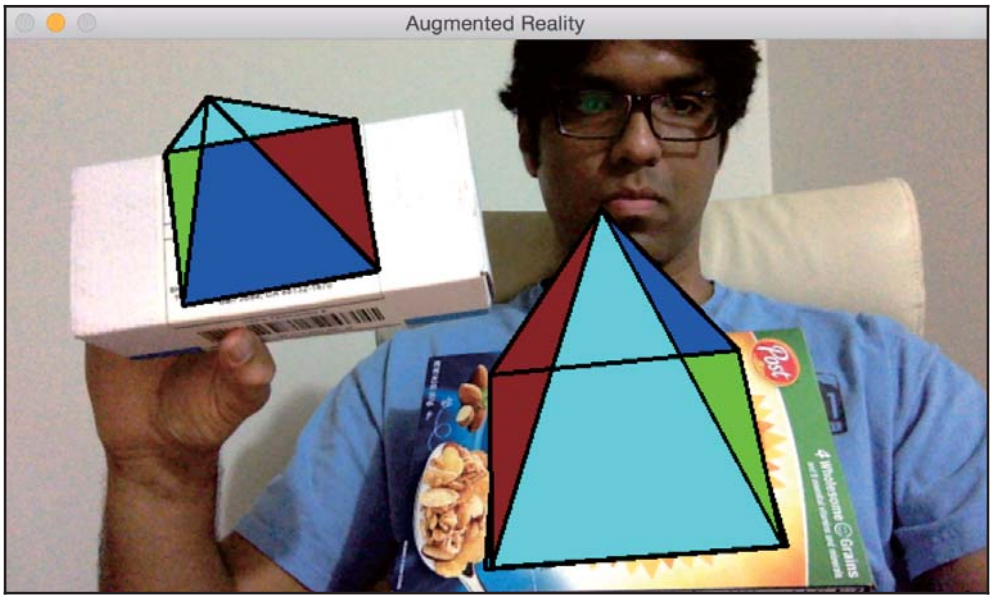
Now that we have all the different blocks, we are ready to build the final system. Let's say we want to overlay a pyramid on top of our cardboard box, as shown here:



Let's tilt the cardboard box to see what happens:



Looks like the pyramid is following the surface. Let's add a second target:



You can keep adding more targets and all those pyramids will be tracked nicely. Let's see how to do this using OpenCV Python. Make sure to save the previous file as `pose_estimation.py` because we will be importing a couple of classes from there:

```
import cv2
import numpy as np

from pose_estimation import PoseEstimator, ROISelector

class Tracker(object):
    def __init__(self, capId, scaling_factor, win_name):
        self.cap = cv2.VideoCapture(capId)
        self.rect = None
        self.win_name = win_name
        self.scaling_factor = scaling_factor
        self.tracker = PoseEstimator()

        ret, frame = self.cap.read()
        self.rect = None
        self.frame = cv2.resize(frame, None, fx=scaling_factor,
fy=scaling_factor, interpolation=cv2.INTER_AREA)

        self.roi_selector = ROISelector(win_name, self.frame,
self.set_rect)
        self.overlay_vertices = np.float32([[0, 0, 0], [0, 1, 0], [1, 1,
0], [1, 0, 0], [0.5, 0.5, 4]])
        self.overlay_edges = [(0, 1), (1, 2), (2, 3), (3, 0), (0,4), (1,4),
(2,4), (3,4)]
        self.color_base = (0, 255, 0)
        self.color_lines = (0, 0, 0)

    def set_rect(self, rect):
        self.rect = rect
        self.tracker.add_target(self.frame, rect)

    def start(self):
        paused = False
        while True:
            if not paused or self.frame is None:
                ret, frame = self.cap.read()
                scaling_factor = self.scaling_factor
                frame = cv2.resize(frame, None, fx=scaling_factor,
fy=scaling_factor,\
                    interpolation=cv2.INTER_AREA)
                if not ret: break

                self.frame = frame.copy()
```

```
            img = self.frame.copy()
            if not paused:
                tracked = self.tracker.track_target(self.frame)
                for item in tracked:
                    cv2.polylines(img, [np.int32(item.quad)],
                     True, self.color_lines, 2)
                    for (x, y) in np.int32(item.points_cur):
                        cv2.circle(img, (x, y), 2,
                         self.color_lines)

                    self.overlay_graphics(img, item)

            self.roi_selector.draw_rect(img, self.rect)
            cv2.imshow(self.win_name, img)
            ch = cv2.waitKey(1)
            if ch == ord(' '): self.paused = not self.paused
            if ch == ord('c'): self.tracker.clear_targets()
            if ch == 27: break

    def overlay_graphics(self, img, tracked):
        x_start, y_start, x_end, y_end = tracked.target.rect
        quad_3d = np.float32([[x_start, y_start, 0], [x_end,
         y_start, 0],
                    [x_end, y_end, 0], [x_start, y_end, 0]])
        h, w = img.shape[:2]
        K = np.float64([[w, 0, 0.5*(w-1)],
                        [0, w, 0.5*(h-1)],
                        [0, 0, 1.0]])
        dist_coef = np.zeros(4)
        ret, rvec, tvec = cv2.solvePnP(objectPoints=quad_3d,
imagePoints=tracked.quad,
                                       cameraMatrix=K,
distCoeffs=dist_coef)
        verts = self.overlay_vertices * \
            [(x_end-x_start), (y_end-y_start), -(x_end-x_start)*0.3] +
(x_start, y_start, 0)
        verts = cv2.projectPoints(verts, rvec, tvec, cameraMatrix=K,
distCoeffs=dist_coef)[0].reshape(-1, 2)

        verts_floor = np.int32(verts).reshape(-1,2)
        cv2.drawContours(img, contours=[verts_floor[:4]], contourIdx=-1,
color=self.color_base, thickness=-3)
        cv2.drawContours(img, contours=[np.vstack((verts_floor[:2],
verts_floor[4:5]))], contourIdx=-1, color=(0,255,0), thickness=-3)
        cv2.drawContours(img, contours=[np.vstack((verts_floor[1:3],
verts_floor[4:5]))], contourIdx=-1, color=(255,0,0), thickness=-3)
        cv2.drawContours(img, contours=[np.vstack((verts_floor[2:4],
verts_floor[4:5]))], contourIdx=-1, color=(0,0,150), thickness=-3)
```

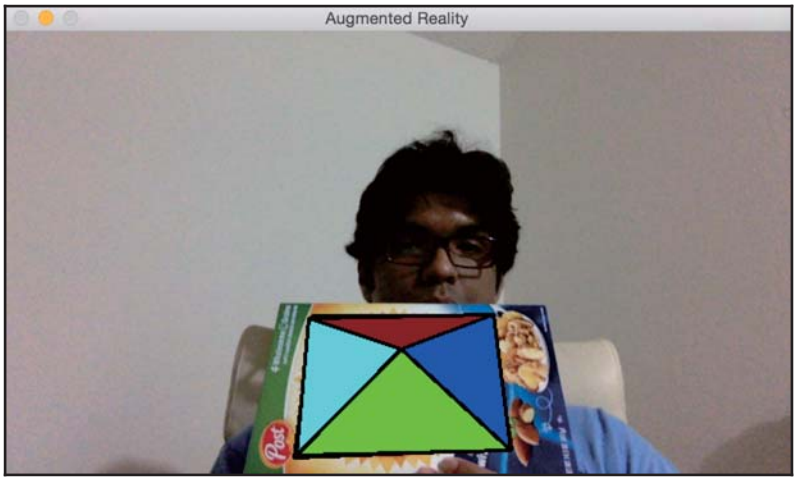
```
        cv2.drawContours(img, contours=[np.vstack((verts_floor[3:4],
verts_floor[0:1], verts_floor[4:5]))], contourIdx=-1, color=(255,255,0),
thickness=-3)

        for i, j in self.overlay_edges:
            (x_start, y_start), (x_end, y_end) = verts[i], verts[j]
            cv2.line(img, (int(x_start), int(y_start)), (int(x_end),
int(y_end)), self.color_lines, 2)

if __name__ == '__main__':
    Tracker(0, 0.8, 'Augmented Reality').start()
```

Let's look at the code

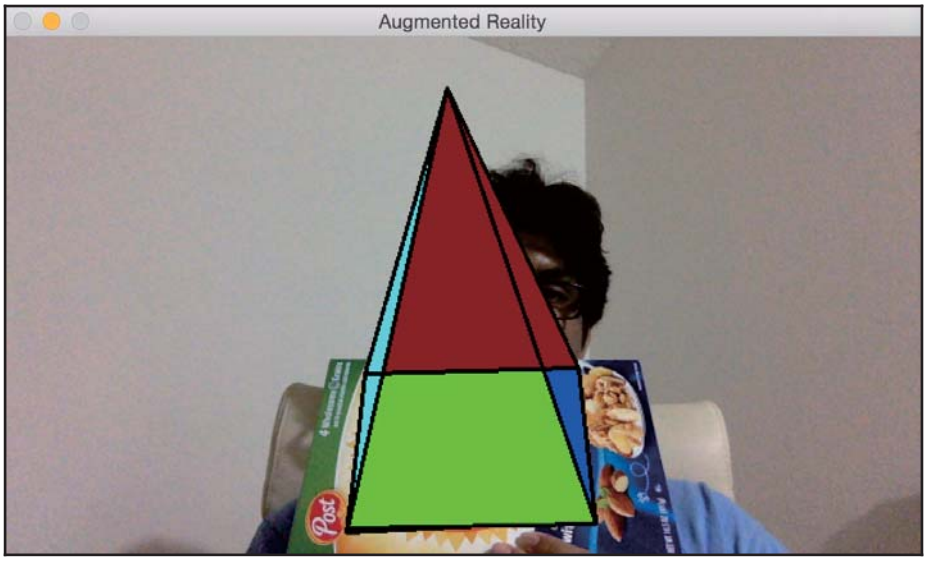
The `Tracker` class is used to perform all the computations here. We initialize the class with the pyramid structure that is defined using edges and vertices. The logic that we use to track the surface is the same as we discussed earlier because we are using the same class. We just need to use `solvePnP` and `projectPoints` to map the 3D pyramid to the 2D surface.

Let's add some movements

Now that we know how to add a virtual pyramid, let's see if we can add some movements. Let's see how we can dynamically change the height of the pyramid. When you start, the pyramid will look like this:



If you wait for some time, the pyramid gets taller and will look like this:



Let's see how to do it in OpenCV Python. Inside the augmented reality code that we just discussed, add the following snippet at the end of the `__init__` method in the `Tracker` class:

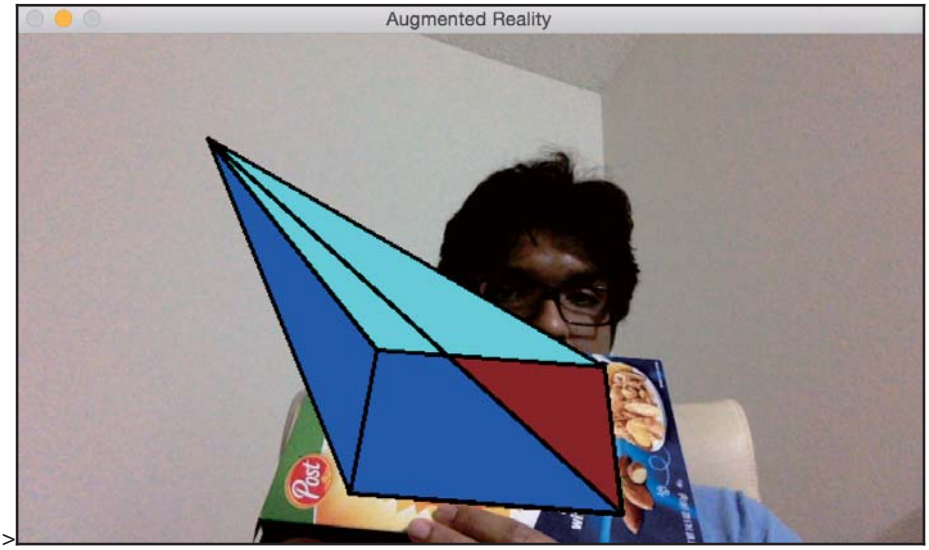
```
self.overlay_vertices = np.float32([[0, 0, 0], [0, 1, 0], [1, 1, 0], [1, 0,
0], [0.5, 0.5, 4]])
self.overlay_edges = [(0, 1), (1, 2), (2, 3), (3, 0),
            (0,4), (1,4), (2,4), (3,4)]
self.color_base = (0, 255, 0)
self.color_lines = (0, 0, 0)

self.graphics_counter = 0
self.time_counter = 0
```

Now that we have the structure, we need to add the code to dynamically change the height. Replace the `overlay_graphics()` method with the following method:

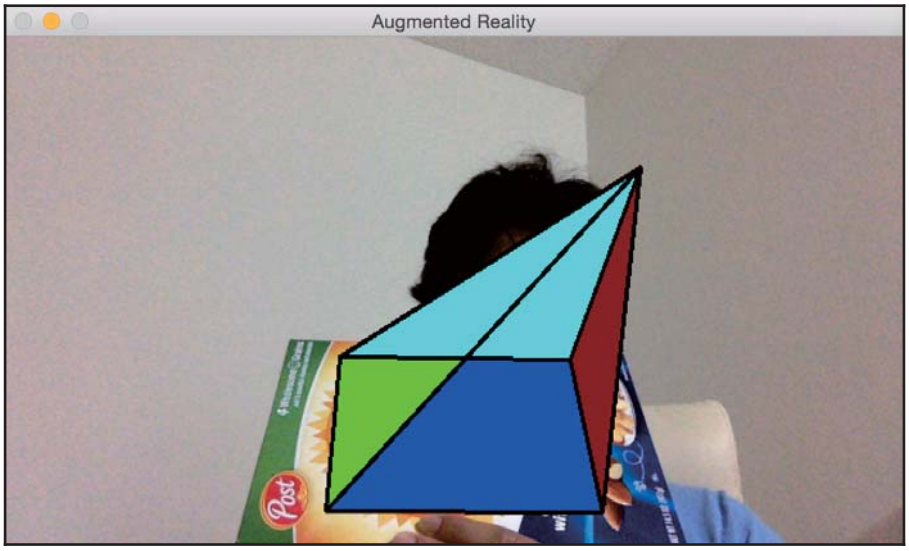
```
    def overlay_graphics(self, img, tracked):
        x_start, y_start, x_end, y_end = tracked.target.rect
        quad_3d = np.float32([[x_start, y_start, 0], [x_end,
         y_start, 0],
                    [x_end, y_end, 0], [x_start, y_end, 0]])
        h, w = img.shape[:2]
        K = np.float64([[w, 0, 0.5*(w-1)],
```

```
                        [0, w, 0.5*(h-1)],
                        [0, 0, 1.0]])
        dist_coef = np.zeros(4)
        ret, rvec, tvec = cv2.solvePnP(objectPoints=quad_3d,
imagePoints=tracked.quad,
                                       cameraMatrix=K,
distCoeffs=dist_coef)
        verts = self.overlay_vertices * \
            [(x_end-x_start), (y_end-y_start), -(x_end-x_start)*0.3] +
(x_start, y_start, 0)
        verts = cv2.projectPoints(verts, rvec, tvec, cameraMatrix=K,
                                  distCoeffs=dist_coef)[0].reshape(-1, 2)

        verts_floor = np.int32(verts).reshape(-1,2)
        cv2.drawContours(img, contours=[verts_floor[:4]],
             contourIdx=-1, color=self.color_base, thickness=-3)
        cv2.drawContours(img, contours=[np.vstack((verts_floor[:2],
            verts_floor[4:5]))], contourIdx=-1, color=(0,255,0),
thickness=-3)
        cv2.drawContours(img, contours=[np.vstack((verts_floor[1:3],
            verts_floor[4:5]))], contourIdx=-1, color=(255,0,0),
thickness=-3)
        cv2.drawContours(img, contours=[np.vstack((verts_floor[2:4],
            verts_floor[4:5]))], contourIdx=-1, color=(0,0,150),
thickness=-3)
        cv2.drawContours(img, contours=[np.vstack((verts_floor[3:4],
            verts_floor[0:1], verts_floor[4:5]))], contourIdx=-1,
color=(255,255,0),thickness=-3)

        for i, j in self.overlay_edges:
            (x_start, y_start), (x_end, y_end) = verts[i], verts[j]
            cv2.line(img, (int(x_start), int(y_start)), (int(x_end),
int(y_end)),
                self.color_lines, 2)
```

Now that we know how to change the height, let's go ahead and make the pyramid dance for us. We can make the tip of the pyramid oscillate periodically. So when you start, it will look like this:

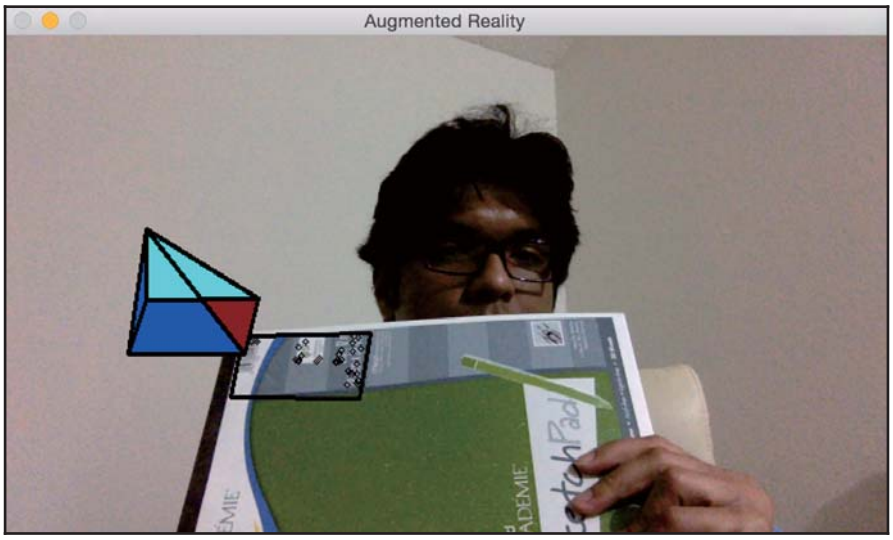


If you wait for some time, it will look like this:

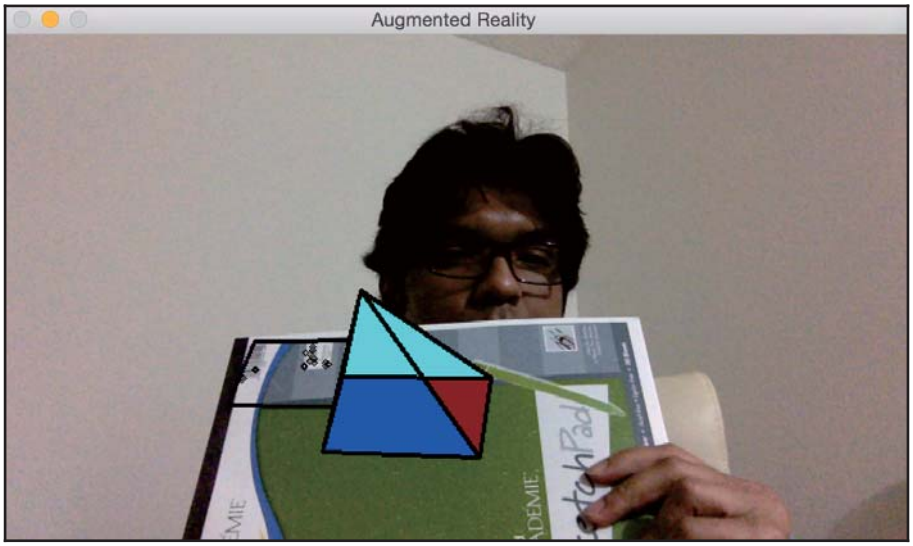


You can look at `augmented_reality_motion.py` for the implementation details.

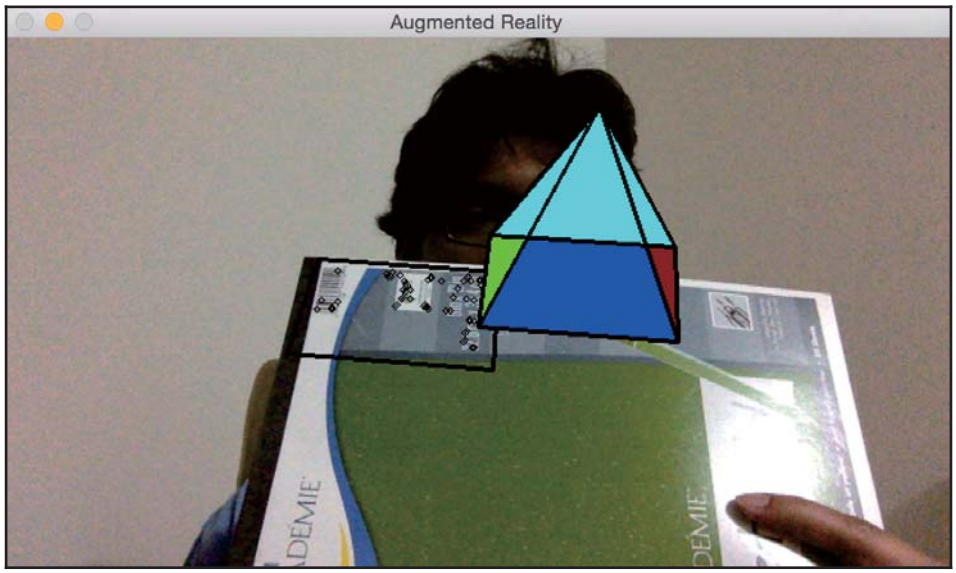
In our next experiment, we will make the whole pyramid move around the region of interest. We can make it move in any way we want. Let's start by adding linear diagonal movement around our selected region of interest. When you start, it will look like this:



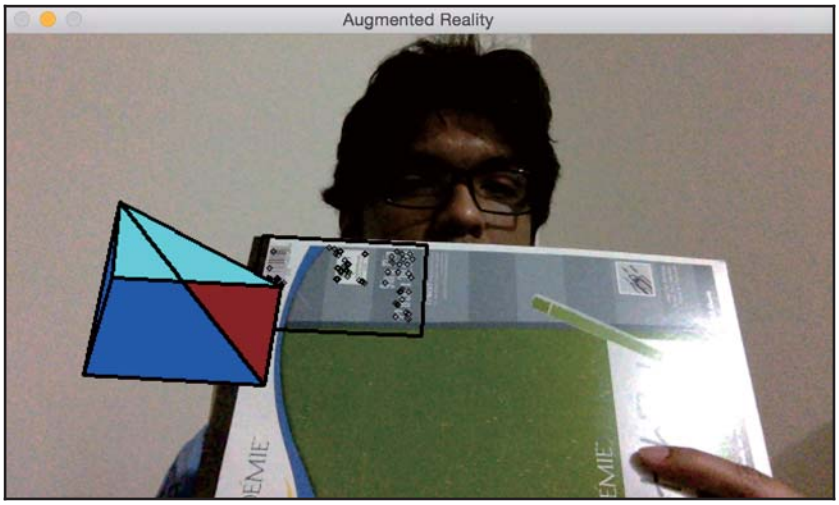
After some time, it will look like this:



Refer to `augmented_reality_dancing.py` to see how to change the `overlay_graphics()` method to make it dance. Let's see if we can make the pyramid go around in circles around our region of interest. When you start, it will look like this:



After some time, it will move to a new position:



You can refer to `augmented_reality_circular_motion.py` to see how to make this happen. You can make it do anything you want. You just need to come up with the right mathematical formula and the pyramid will literally dance to your tune! You can also try out other virtual objects to see what you can do with it. There are a lot of things you can do with a lot of different objects. These examples provide good reference points, on top of which you can build many interesting augmented reality applications.

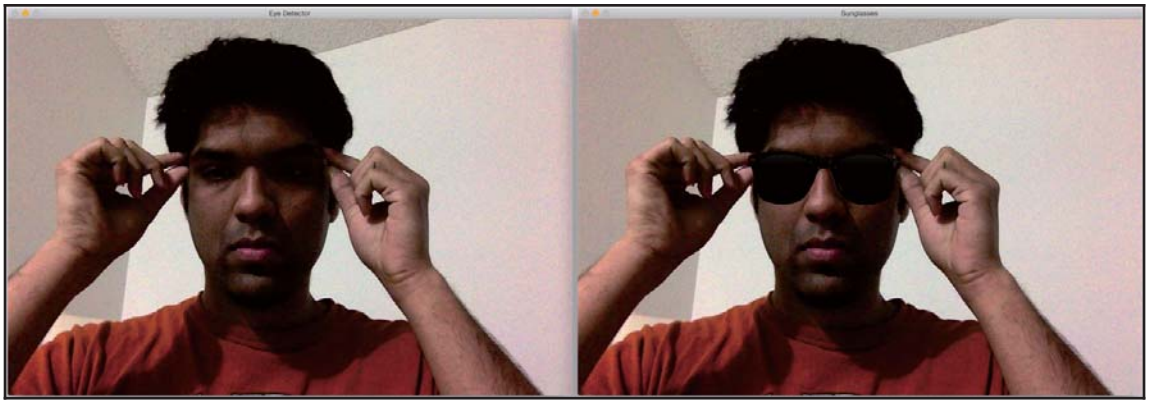
Summary

In this chapter, you learned about the premise of augmented reality and gained an understanding of what an augmented reality system looks like. We discussed the geometric transformations required for augmented reality. You also learned how to use those transformations to estimate the camera pose, and you learned how to track planar objects. We discussed how we can add virtual objects on top of the real world. You learned how to modify virtual objects in different ways to add cool effects.

In the next chapter, we will learn how to apply machine learning techniques, along with artificial neural networks, which will help us to enhance the knowledge already acquired in `Chapter 9`, *Object Recognition*.

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')

cap = cv2.VideoCapture(0)
sunglasses_img = cv2.imread('images/sunglasses.png')

while True:
    ret, frame = cap.read()
    frame = cv2.resize(frame, None, fx=0.5, fy=0.5,
interpolation=cv2.INTER_AREA)
    vh, vw = frame.shape[:2]
    vh, vw = int(vh), int(vw)

    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    faces = face_cascade.detectMultiScale(gray, scaleFactor=1.3,
minNeighbors=1)
```

```
    centers = []

    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:
            centers.append((x + int(x_eye + 0.5*w_eye), y + int(y_eye +
0.5*h_eye)))
    if len(centers) > 1: # if detects both eyes
        h, w = sunglasses_img.shape[:2]
        # Extract the region of interest from the image
        eye_distance = abs(centers[1][0] - centers[0][0])
        # Overlay sunglasses; the factor 2.12 is customizable depending on
the size of the face
        sunglasses_width = 2.12 * eye_distance
        scaling_factor = sunglasses_width / w
        print(scaling_factor, eye_distance)
        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 -= int(0.26*overlay_sunglasses.shape[1])
        y += int(0.26*overlay_sunglasses.shape[0])
        h, w = overlay_sunglasses.shape[:2]
        h, w = int(h), int(w)
        frame_roi = frame[y:y+h, x:x+w]
        # Convert color image to grayscale and threshold it
        gray_overlay_sunglassess = cv2.cvtColor(overlay_sunglasses,
cv2.COLOR_BGR2GRAY)
        ret, mask = cv2.threshold(gray_overlay_sunglassess, 180, 255,
cv2.THRESH_BINARY_INV)

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

        try:
            # Use the mask to extract the face mask region of interest
            masked_face = cv2.bitwise_and(overlay_sunglasses,
overlay_sunglasses, 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)
        except cv2.error as e:
            print('Ignoring arithmentic exceptions: '+ str(e))
            #raise e
```

```
            # add the two images to get the final output
            frame[y:y+h, x:x+w] = cv2.add(masked_face, masked_frame)
        else:
            print('Eyes not detected')

        cv2.imshow('Eye Detector', frame)
        c = cv2.waitKey(1)
        if c == 27:
            break

cap.release()
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 and 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

Once more, through the use of Haar cascade classifier files, the code below will identify each ear, highlighting them once they are detected. As you can notice, two different classifiers are required as the coordinates for each ear will be inverted:

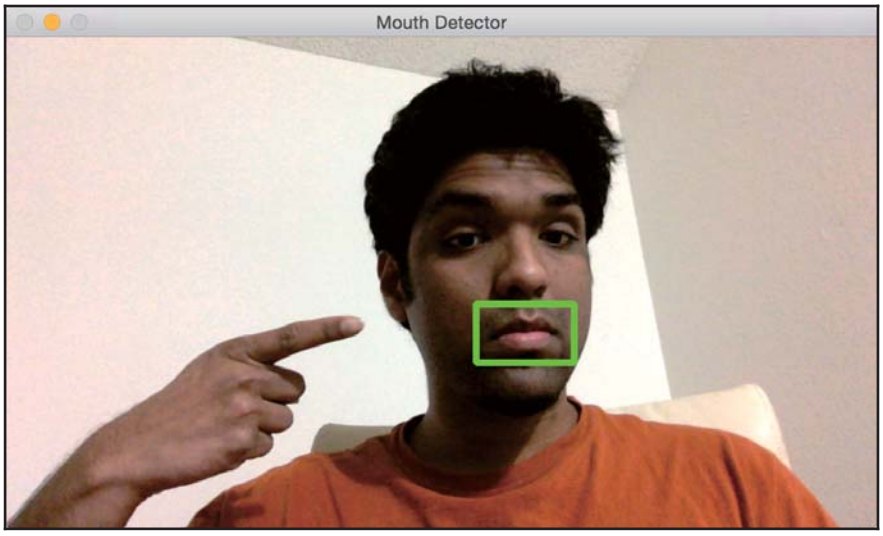
```
import cv2
import numpy as np

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')

cap = cv2.VideoCapture(0)
```

Detecting a mouth

This time, using Haar classifiers, we are going to extract a mouth position from the input video stream, and on the code below this code we are going to use those coordinates to place a mustache on the face:

```
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, scaleFactor=1.7,
minNeighbors=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()
```

The following image shows 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 -= int(0.05*w)
            y -= int(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 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. The following is what the output will look like:

Let's see how to build the pupil detector:

```
import math

import cv2

eye_cascade = cv2.CascadeClassifier('./cascade_files/haarcascade_eye.xml')
if eye_cascade.empty():
  raise IOError('Unable to load the eye cascade classifier xml file')

cap = cv2.VideoCapture(0)
ds_factor = 0.5
ret, frame = cap.read()
contours = []

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)
```

```
  eyes = eye_cascade.detectMultiScale(gray, scaleFactor=1.3,
minNeighbors=1)
  for (x_eye, y_eye, w_eye, h_eye) in eyes:
    pupil_frame = gray[y_eye:y_eye + h_eye, x_eye:x_eye + w_eye]
    ret, thresh = cv2.threshold(pupil_frame, 80, 255, cv2.THRESH_BINARY)
    cv2.imshow("threshold", thresh)
    im2, contours, hierarchy = cv2.findContours(thresh, cv2.RETR_LIST,
cv2.CHAIN_APPROX_SIMPLE)
    print(contours)

    for contour in contours:
      area = cv2.contourArea(contour)
      rect = cv2.boundingRect(contour)
      x, y, w, h = rect
      radius = 0.15 * (w + h)

      area_condition = (100 <= area <= 200)
      symmetry_condition = (abs(1 - float(w)/float(h)) <= 0.2)
      fill_condition = (abs(1 - (area / (math.pi * math.pow(radius, 2.0))))
<= 0.4)
      cv2.circle(frame, (int(x_eye + x + radius), int(y_eye + y + radius)),
int(1.3 * radius), (0, 180, 0), -1)

  cv2.imshow('Pupil Detector', frame)
  c = cv2.waitKey(1)
  if c == 27:
    break
cap.release()
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 a 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 width to height ratio of the bounding rectangle. If the shape is a circle, this ratio will be one. We can use the `boundingRect` function 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 `contourArea` function 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.

If you feel like playing with another kind of body detection, just go to the following link to find the difference classifiers: ;`https://github.com/opencv/opencv/tree/master/data/haarcascades`

Summary

In this chapter, we discussed Haar cascades and integral images. We understood how the face detection pipeline is built. We learned 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, such as eyes, ears, nose, and mouth. We learned 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 image content.