CHAPTER 11

Camera Models and Calibration

Vision begins with the detection of light from the world. That light begins as rays emanating from some source (e.g., a light bulb or the sun), which then travels through space until striking some object. When that light strikes the object, much of the light is absorbed, and what is not absorbed we perceive as the color of the light. Reflected light that makes its way to our eye (or our camera) is collected on our retina (or our imager). The geometry of this arrangement—particularly of the ray's travel from the object, through the lens in our eye or camera, and to the retina or imager—is of particular importance to practical computer vision.

A simple but useful model of how this happens is the pinhole camera model.* A *pinhole* is an imaginary wall with a tiny hole in the center that blocks all rays except those passing through the tiny aperture in the center. In this chapter, we will start with a pinhole camera model to get a handle on the basic geometry of projecting rays. Unfortunately, a real pinhole is not a very good way to make images because it does not gather enough light for rapid exposure. This is why our eyes and cameras use lenses to gather more light than what would be available at a single point. The downside, however, is that gathering more light with a lens not only forces us to move beyond the simple geometry of the pinhole model but also introduces distortions from the lens itself.

In this chapter we will learn how, using *camera calibration*, to correct (mathematically) for the main deviations from the simple pinhole model that the use of lenses imposes on us. Camera calibration is important also for relating camera measurements with measurements in the real, three-dimensional world. This is important because scenes are not only three-dimensional; they are also physical spaces with physical units. Hence, the relation between the camera's natural units (pixels) and the units of the

* Knowledge of lenses goes back at least to Roman times. The pinhole camera model goes back at least 987 years to al-Hytham [1021] and is the classic way of introducing the geometric aspects of vision. Mathematical and physical advances followed in the 1600s and 1700s with Descartes, Kepler, Galileo, Newton, Hooke, Euler, Fermat, and Snell (see O'Connor [O'Connor02]). Some key modern texts for geometric vision include those by Trucco [Trucco98], Jaehne (also sometimes spelled Jähne) [Jaehne95; Jaehne97], Hartley and Zisserman [Hartley06], Forsyth and Ponce [Forsyth03], Shapiro and Stockman [Shapiro02], and Xu and Zhang [Xu96].

physical world (e.g., meters) is a critical component in any attempt to reconstruct a three-dimensional scene.

The process of camera calibration gives us both a model of the camera's geometry and a *distortion* model of the lens. These two informational models define the *intrinsic parameters* of the camera. In this chapter we use these models to correct for lens distortions; in Chapter 12, we will use them to interpret a physical scene.

We shall begin by looking at camera models and the causes of lens distortion. From there we will explore the *homography transform*, the mathematical instrument that allows us to capture the effects of the camera's basic behavior and of its various distortions and corrections. We will take some time to discuss exactly how the transformation that characterizes a particular camera can be calculated mathematically. Once we have all this in hand, we'll move on to the OpenCV function that does most of this work for us.

Just about all of this chapter is devoted to building enough theory that you will truly understand what is going into (and what is coming out of) the OpenCV function `cvCalibrateCamera2()` as well as what that function is doing "under the hood". This is important stuff if you want to use the function responsibly. Having said that, if you are already an expert and simply want to know how to use OpenCV to do what you already understand, jump right ahead to the "Calibration Function" section and get to it.

Camera Model

We begin by looking at the simplest model of a camera, the pinhole camera model. In this simple model, light is envisioned as entering from the scene or a distant object, but only a single ray enters from any particular point. In a physical pinhole camera, this point is then "projected" onto an imaging surface. As a result, the image on this *image plane* (also called the *projective plane*) is always in focus, and the size of the image relative to the distant object is given by a single parameter of the camera: its *focal length*. For our idealized pinhole camera, the distance from the pinhole aperture to the screen is precisely the focal length. This is shown in Figure 11-1, where f is the focal length of the camera, Z is the distance from the camera to the object, X is the length of the object, and x is the object's image on the imaging plane. In the figure, we can see by similar triangles that $-x/f = X/Z$, or

$$-x = f\frac{X}{Z}$$

We shall now rearrange our pinhole camera model to a form that is equivalent but in which the math comes out easier. In Figure 11-2, we swap the pinhole and the image plane.* The main difference is that the object now appears rightside up. The point in the pinhole is reinterpreted as the *center of projection*. In this way of looking at things, every

* Typical of such mathematical abstractions, this new arrangement is not one that can be built physically; the image plane is simply a way of thinking of a "slice" through all of those rays that happen to strike the center of projection. This arrangement is, however, much easier to draw and do math with.

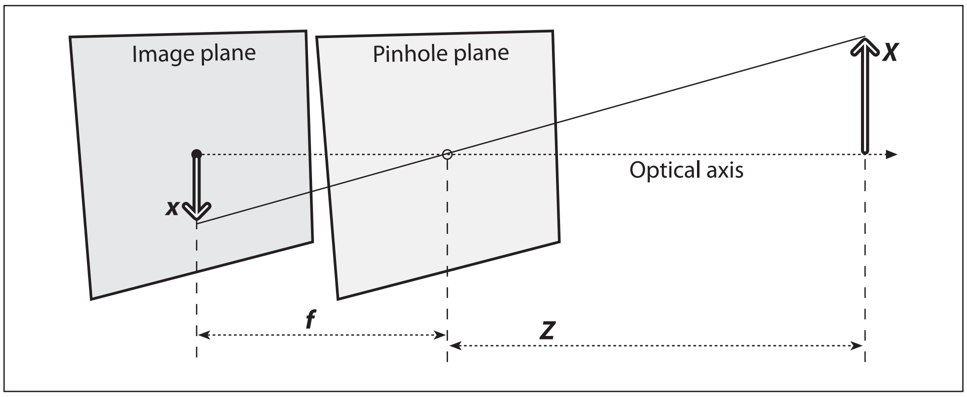


Figure 11-1. Pinhole camera model: a pinhole (the pinhole aperture) lets through only those light rays that intersect a particular point in space; these rays then form an image by "projecting" onto an image plane

ray leaves a point on the distant object and heads for the center of projection. The point at the intersection of the image plane and the optical axis is referred to as the *principal point*. On this new frontal image plane (see Figure 11-2), which is the equivalent of the old projective or image plane, the image of the distant object is exactly the same size as it was on the image plane in Figure 11-1. The image is generated by intersecting these rays with the image plane, which happens to be exactly a distance f from the center of projection. This makes the similar triangles relationship $x/f = X/Z$ more directly evident than before. The negative sign is gone because the object image is no longer upside down.

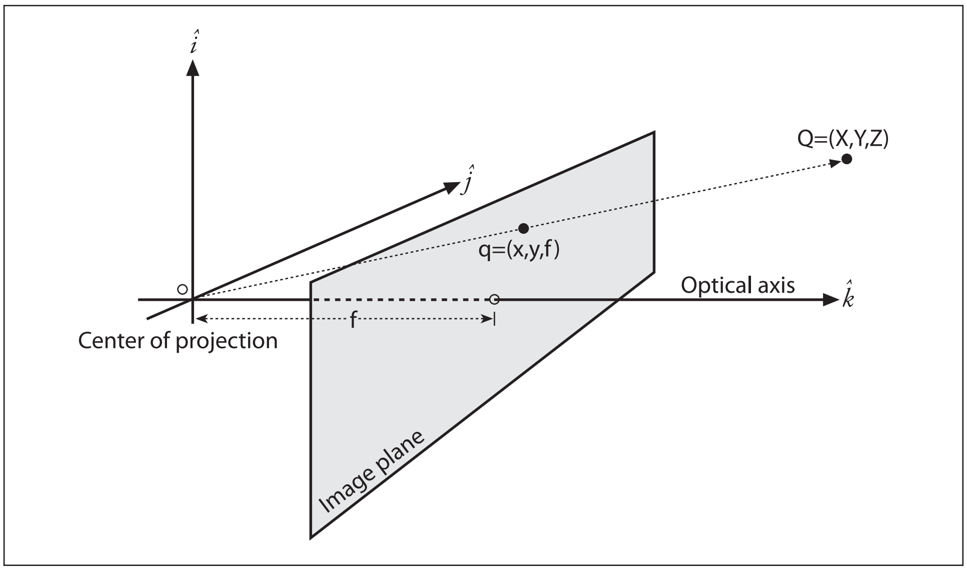


Figure 11-2. A point $Q = (X, Y, Z)$ is projected onto the image plane by the ray passing through the center of projection, and the resulting point on the image is $q = (z, y, f)$; the image plane is really just the projection screen "pushed" in front of the pinhole (the math is equivalent but simpler this way)

You might think that the principle point is equivalent to the center of the imager; yet this would imply that some guy with tweezers and a tube of glue was able to attach the imager in your camera to micron accuracy. In fact, the center of the chip is usually not on the optical axis. We thus introduce two new parameters, c_x and c_y, to model a possible displacement (away from the optic axis) of the center of coordinates on the projection screen. The result is that a relatively simple model in which a point Q in the physical world, whose coordinates are (X, Y, Z), is projected onto the screen at some pixel location given by $(x_{\text{screen}}, y_{\text{screen}})$ in accordance with the following equations:*

$$x_{\text{screen}} = f_x\left(\frac{X}{Z}\right) + c_x, \qquad y_{\text{screen}} = f_y\left(\frac{Y}{Z}\right) + c_y$$

Note that we have introduced two different focal lengths; the reason for this is that the individual pixels on a typical low-cost imager are rectangular rather than square. The focal length f_x (for example) is actually the product of the physical focal length of the lens and the size s_x of the individual imager elements (this should make sense because s_x has units of pixels per millimeter† while F has units of millimeters, which means that f_x is in the required units of pixels). Of course, similar statements hold for f_y and s_y. It is important to keep in mind, though, that s_x and s_y cannot be measured directly via any camera calibration process, and neither is the physical focal length F directly measurable. Only the combinations $f_x = Fs_x$ and $f_y = Fs_y$ can be derived without actually dismantling the camera and measuring its components directly.

Basic Projective Geometry

The relation that maps the points Q_i in the physical world with coordinates (X_i, Y_i, Z_i) to the points on the projection screen with coordinates (x_i, y_i) is called a *projective transform*. When working with such transforms, it is convenient to use what are known as *homogeneous coordinates*. The homogeneous coordinates associated with a point in a projective space of dimension n are typically expressed as an $(n + 1)$-dimensional vector (e.g., x, y, z becomes x, y, z, w), with the additional restriction that any two points whose values are proportional are equivalent. In our case, the image plane is the projective space and it has two dimensions, so we will represent points on that plane as three-dimensional vectors $q = (q_1, q_2, q_3)$. Recalling that all points having proportional values in the projective space are equivalent, we can recover the actual pixel coordinates by dividing through by q_3. This allows us to arrange the parameters that define our camera (i.e., f_x, f_y, c_x, and c_y) into a single 3-by-3 matrix, which we will call the *camera intrinsics matrix* (the approach OpenCV takes to camera intrinsics is derived from Heikkila and

* Here the subscript "screen" is intended to remind you that the coordinates being computed are in the coordinate system of the screen (i.e., the imager). The difference between $(x_{\text{screen}}, y_{\text{screen}})$ in the equation and (x, y) in Figure 11-2 is precisely the point of c_x and c_y. Having said that, we will subsequently drop the "screen" subscript and simply use lowercase letters to describe coordinates on the imager.

† Of course, "millimeter" is just a stand-in for any physical unit you like. It could just as easily be "meter," "micron," or "furlong." The point is that s_x converts physical units to pixel units.

Silven [Heikkila97]). The projection of the points in the physical world into the camera is now summarized by the following simple form:

$$q = MQ, \quad \text{where} \quad q = \begin{bmatrix} x \\ y \\ w \end{bmatrix}, \quad M = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}, \quad Q = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

Multiplying this out, you will find that $w = Z$ and so, since the point q is in homogeneous coordinates, we should divide through by w (or Z) in order to recover our earlier definitions. (The minus sign is gone because we are now looking at the noninverted image on the projective plane in front of the pinhole rather than the inverted image on the projection screen behind the pinhole.)

While we are on the topic of homogeneous coordinates, there is a function in the OpenCV library which would be appropriate to introduce here: `cvConvertPointsHomogenious()`* is handy for converting to and from homogeneous coordinates; it also does a bunch of other useful things.

```
void cvConvertPointsHomogenious(
  const CvMat* src,
  CvMat*       dst
);
```

Don't let the simple arguments fool you; this routine does a whole lot of useful stuff. The input array `src` can be M_{scr}-by-N or N-by-M_{scr} (for $M_{scr} = 2, 3,$ or 4); it can also be 1-by-N or N-by-1, with the array having $M_{scr} = 2, 3,$ or 4 channels (N can be any number; it is essentially the number of points that you have stuffed into the matrix `src` for conversion). The output array `dst` can be any of these types as well, with the additional restriction that the dimensionality M_{dst} must be equal to M_{scr}, $M_{scr} - 1$, or $M_{scr} + 1$.

When the input dimension M_{scr} is equal to the output dimension M_{dst}, the data is simply copied (and, if necessary, transposed). If $M_{scr} > M_{dst}$, then the elements in `dst` are computed by dividing all but the last elements of the corresponding vector from `src` by the last element of that same vector (i.e., `src` is assumed to contain homogeneous coordinates). If $M_{scr} < M_{dst}$, then the points are copied but with a 1 being inserted into the final coordinate of every vector in the `dst` array (i.e., the vectors in `src` are extended to homogeneous coordinates). In these cases, just as in the trivial case of $M_{scr} = M_{dst}$, any necessary transpositions are also done.

One word of warning about this function is that there can be cases (when $N < 5$) where the input and output dimensionality are ambiguous. In this event, the function will throw an error. If you find yourself in this situation, you can just pad out the matrices with some bogus values. Alternatively, the user may pass multichannel N-by-1 or 1-by-N matrices, where the number of channels is M_{scr} (M_{dst}). The function `cvReshape()` can be used to convert single-channel matrices to multichannel ones without copying any data.

* Yes, "Homogenious" in the function name is misspelled.

With the ideal pinhole, we have a useful model for some of the three-dimensional geometry of vision. Remember, however, that very little light goes through a pinhole; thus, in practice such an arrangement would make for very slow imaging while we wait for enough light to accumulate on whatever imager we are using. For a camera to form images at a faster rate, we must gather a lot of light over a wider area and bend (i.e., focus) that light to converge at the point of projection. To accomplish this, we use a lens. A lens can focus a large amount of light on a point to give us fast imaging, but it comes at the cost of introducing distortions.

Lens Distortions

In theory, it is possible to define a lens that will introduce no distortions. In practice, however, no lens is perfect. This is mainly for reasons of manufacturing; it is much easier to make a "spherical" lens than to make a more mathematically ideal "parabolic" lens. It is also difficult to mechanically align the lens and imager exactly. Here we describe the two main lens distortions and how to model them.* *Radial distortions* arise as a result of the shape of lens, whereas *tangential distortions* arise from the assembly process of the camera as a whole.

We start with radial distortion. The lenses of real cameras often noticeably distort the location of pixels near the edges of the imager. This bulging phenomenon is the source of the "barrel" or "fish-eye" effect (see the room-divider lines at the top of Figure 11-12 for a good example). Figure 11-3 gives some intuition as to why radial distortion occurs. With some lenses, rays farther from the center of the lens are bent more than those closer in. A typical inexpensive lens is, in effect, stronger than it ought to be as you get farther from the center. Barrel distortion is particularly noticeable in cheap web cameras but less apparent in high-end cameras, where a lot of effort is put into fancy lens systems that minimize radial distortion.

For radial distortions, the distortion is 0 at the (optical) center of the imager and increases as we move toward the periphery. In practice, this distortion is small and can be characterized by the first few terms of a Taylor series expansion around $r = 0$.† For cheap web cameras, we generally use the first two such terms; the first of which is conventionally called k_1 and the second k_2. For highly distorted cameras such as fish-eye lenses we can use a third radial distortion term k_3. In general, the radial location of a point on the imager will be rescaled according to the following equations:

* The approach to modeling lens distortion taken here derives mostly from Brown [Brown71] and earlier Fryer and Brown [Fryer86].

† If you don't know what a Taylor series is, don't worry too much. The Taylor series is a mathematical technique for expressing a (potentially) complicated function in the form of a polynomial of similar value to the approximated function in at least a small neighborhood of some particular point (the more terms we include in the polynomial series, the more accurate the approximation). In our case we want to expand the distortion function as a polynomial in the neighborhood of $r = 0$. This polynomial takes the general form $f(r) = a_0 + a_1 r + a_2 r^2 + \cdots$, but in our case the fact that $f(r) = 0$ at $r = 0$ implies $a_0 = 0$. Similarly, because the function must be symmetric in r, only the coefficients of even powers of r will be nonzero. For these reasons, the only parameters that are necessary for characterizing these radial distortions are the coefficients of r^2, r^4, and (sometimes) r^6.

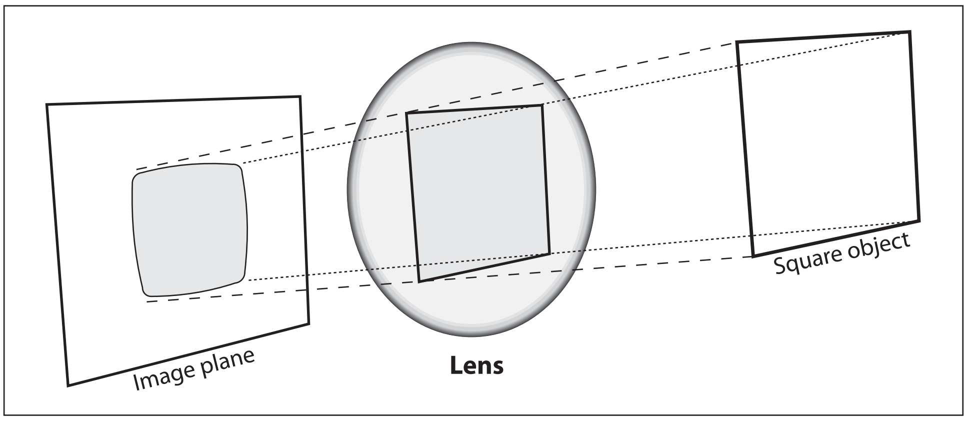


Figure 11-3. Radial distortion: rays farther from the center of a simple lens are bent too much compared to rays that pass closer to the center; thus, the sides of a square appear to bow out on the image plane (this is also known as barrel distortion)

$$x_{\text{corrected}} = x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6)$$
$$y_{\text{corrected}} = y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6)$$

Here, (x, y) is the original location (on the imager) of the distorted point and $(x_{\text{corrected}}, y_{\text{corrected}})$ is the new location as a result of the correction. Figure 11-4 shows displacements of a rectangular grid that are due to radial distortion. External points on a front-facing rectangular grid are increasingly displaced inward as the radial distance from the optical center increases.

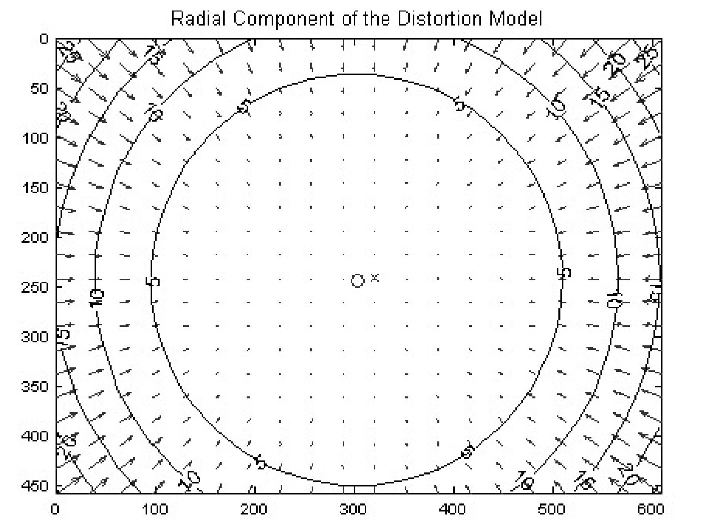
The second-largest common distortion is *tangential distortion*. This distortion is due to manufacturing defects resulting from the lens not being exactly parallel to the imaging plane; see Figure 11-5.

Tangential distortion is minimally characterized by two additional parameters, p_1 and p_2, such that:*

$$x_{\text{corrected}} = x + [2p_1 y + p_2(r^2 + 2x^2)]$$
$$y_{\text{corrected}} = y + [p_1(r^2 + 2y^2) + 2p_2 x]$$

Thus in total there are five distortion coefficients that we require. Because all five are necessary in most of the OpenCV routines that use them, they are typically bundled into one *distortion vector*; this is just a 5-by-1 matrix containing k_1, k_2, p_1, p_2, and k_3 (in that order). Figure 11-6 shows the effects of tangential distortion on a front-facing external rectangular grid of points. The points are displaced elliptically as a function of location and radius.

* The derivation of these equations is beyond the scope of this book, but the interested reader is referred to the "plumb bob" model; see D. C. Brown, "Decentering Distortion of Lenses", *Photometric Engineering* 32(3) (1966), 444–462.



```
Pixel error              = [0.1174, 0.1159]
Focal Length             = (657.303, 657.744)              +/- [0.2849, 0.2894]
Principal Point          = (302.717, 242.334)              +/- [0.5912, 0.5571]
Skew                     = 0.0004198                          +/- 0.0001905
Radial coefficients      = (-0.2535, 0.1187, 0)          +/- [0.00231, 0.009418, 0]
Tangential coefficients  = (-0.0002789, 5.174e-005)    +/- [0.0001217, 0.0001208]
```

Figure 11-4. Radial distortion plot for a particular camera lens: the arrows show where points on an external rectangular grid are displaced in a radially distorted image (courtesy of Jean-Yves Bouguet)

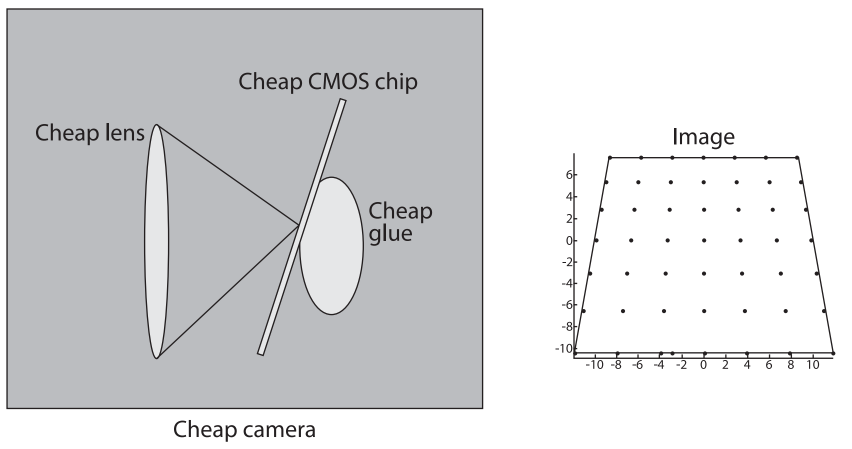
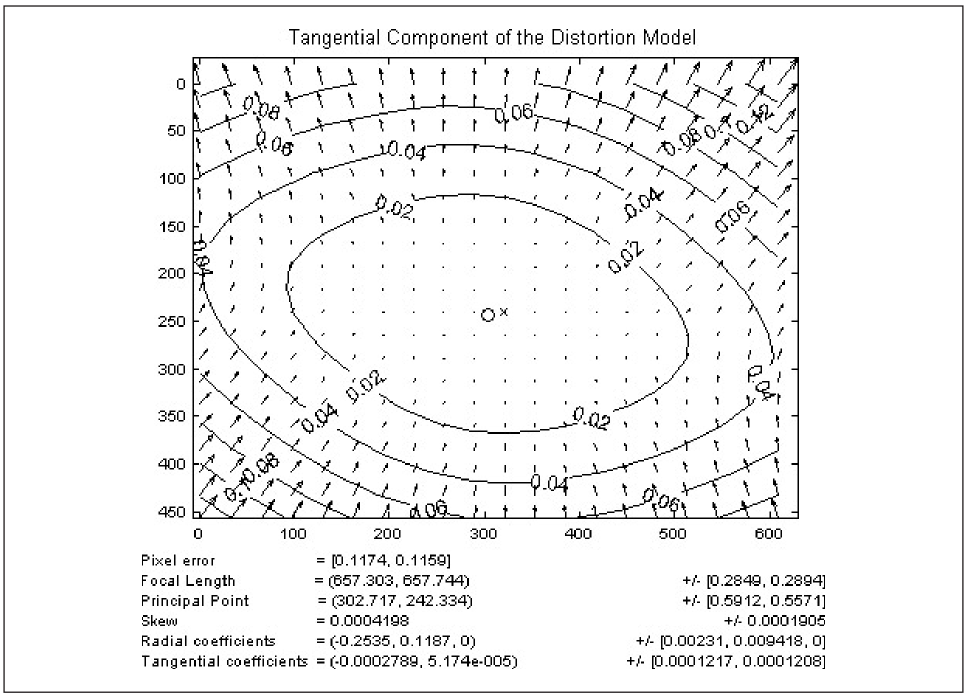


Figure 11-5. Tangential distortion results when the lens is not fully parallel to the image plane; in cheap cameras, this can happen when the imager is glued to the back of the camera (image courtesy of Sebastian Thrun)

There are many other kinds of distortions that occur in imaging systems, but they typically have lesser effects than radial and tangential distortions. Hence neither we nor OpenCV will deal with them further.



```
Pixel error              = [0.1174, 0.1159]
Focal Length             = (657.303, 657.744)                +/- [0.2849, 0.2894]
Principal Point          = (302.717, 242.334)                +/- [0.5912, 0.5571]
Skew                     = 0.0004198                            +/- 0.0001905
Radial coefficients      = (-0.2535, 0.1187, 0)          +/- [0.00231, 0.009418, 0]
Tangential coefficients  = (-0.0002789, 5.174e-005)     +/- [0.0001217, 0.0001208]
```

Figure 11-6. Tangential distortion plot for a particular camera lens: the arrows show where points on an external rectangular grid are displaced in a tangentially distorted image (courtesy of Jean-Yves Bouguet)

Calibration

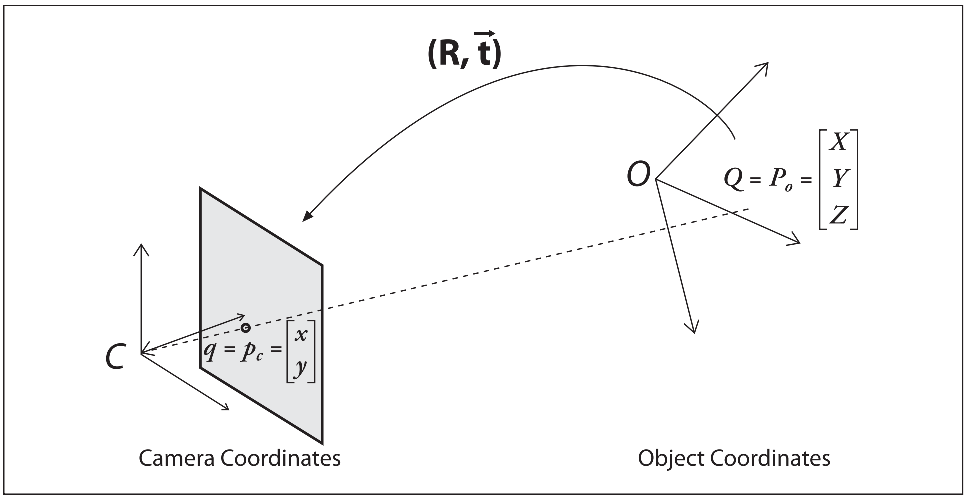
Now that we have some idea of how we'd describe the intrinsic and distortion properties of a camera mathematically, the next question that naturally arises is how we can use OpenCV to compute the intrinsics matrix and the distortion vector.*

OpenCV provides several algorithms to help us compute these intrinsic parameters. The actual calibration is done via `cvCalibrateCamera2()`. In this routine, the method of calibration is to target the camera on a known structure that has many individual and identifiable points. By viewing this structure from a variety of angles, it is possible to then compute the (relative) location and orientation of the camera at the time of each image as well as the intrinsic parameters of the camera (see Figure 11-9 in the "Chessboards" section). In order to provide multiple views, we rotate and translate the object, so let's pause to learn a little more about rotation and translation.

* For a great online tutorial of camera calibration, see Jean-Yves Bouguet's calibration website (*http://www.vision.caltech.edu/bouguetj/calib_doc*).

Rotation Matrix and Translation Vector

For each image the camera takes of a particular object, we can describe the *pose* of the object relative to the camera coordinate system in terms of a rotation and a translation; see Figure 11-7.



(R, $\vec{t}$)
O
$Q = P_o = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$
C
$q = p_c = \begin{bmatrix} x \\ y \end{bmatrix}$
Camera Coordinates
Object Coordinates

Figure 11-7. Converting from object to camera coordinate systems: the point P on the object is seen as point p on the image plane; the point p is related to point P by applying a rotation matrix R and a translation vector **t** *to P*

In general, a rotation in any number of dimensions can be described in terms of multiplication of a coordinate vector by a square matrix of the appropriate size. Ultimately, a rotation is equivalent to introducing a new description of a point's location in a different coordinate system. Rotating the coordinate system by an angle θ is equivalent to counterrotating our target point around the origin of that coordinate system by the same angle θ. The representation of a two-dimensional rotation as matrix multiplication is shown in Figure 11-8. Rotation in three dimensions can be decomposed into a two-dimensional rotation around each axis in which the pivot axis measurements remain constant. If we rotate around the *x*-, *y*-, and *z*-axes in sequence[*] with respective rotation angles ψ, φ, and θ, the result is a total rotation matrix *R* that is given by the product of the three matrices $R_x(\psi)$, $R_y(\varphi)$, and $R_z(\theta)$, where:

$$R_x(\psi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\psi & \sin\psi \\ 0 & -\sin\psi & \cos\psi \end{bmatrix}$$

[*] Just to be clear: the rotation we are describing here is first around the *z*-axis, then around the *new* position of the *y*-axis, and finally around the *new* position of the *x*-axis.

$$R_y(\varphi) = \begin{bmatrix} \cos\varphi & 0 & -\sin\varphi \\ 0 & 1 & 0 \\ \sin\varphi & 0 & \cos\varphi \end{bmatrix}$$

$$R_z(\theta) = \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

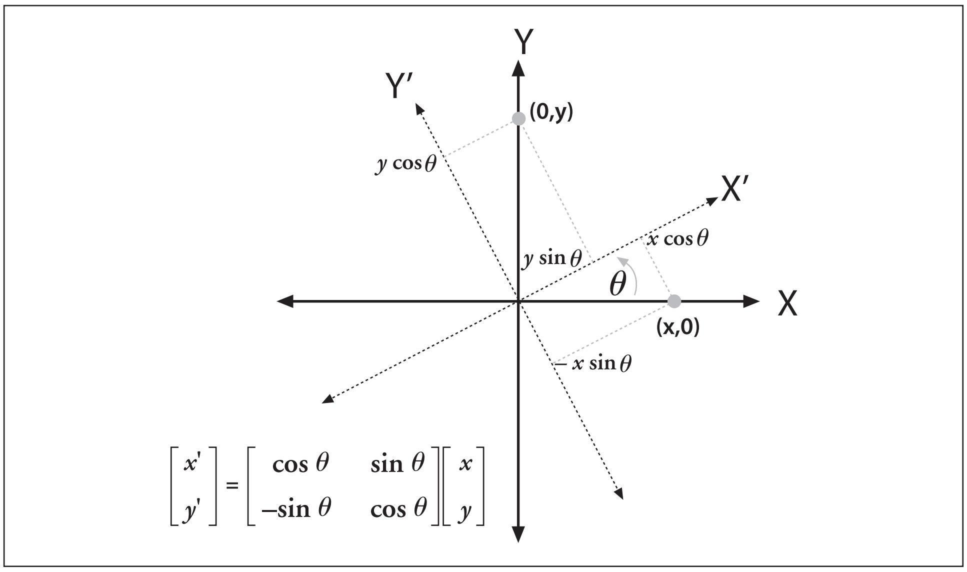

Y
Y′
(0,y)
y cos θ
X′
x cos θ
y sin θ
θ
X
(x,0)
− x sin θ
$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$

Figure 11-8. Rotating points by θ (in this case, around the Z-axis) is the same as counterrotating the coordinate axis by θ; by simple trigonometry, we can see how rotation changes the coordinates of a point

Thus, $R = R_z(\theta), R_y(\varphi), R_x(\psi)$. The rotation matrix R has the property that its inverse is its transpose (we just rotate back); hence we have $R^T R = RR^T = I$, where I is the identity matrix consisting of 1s along the diagonal and 0s everywhere else.

The *translation vector* is how we represent a shift from one coordinate system to another system whose origin is displaced to another location; in other words, the translation vector is just the offset from the origin of the first coordinate system to the origin of the second coordinate system. Thus, to shift from a coordinate system centered on an object to one centered at the camera, the appropriate translation vector is simply $T = \text{origin}_{\text{object}} - \text{origin}_{\text{camera}}$. We then have (with reference to Figure 11-7) that a point in the object (or world) coordinate frame P_o has coordinates P_c in the camera coordinate frame:

$$P_c = R(P_o - T)$$

Combining this equation for P_c above with the camera intrinsic corrections will form the basic system of equations that we will be asking OpenCV to solve. The solution to these equations will be the camera calibration parameters we seek.

We have just seen that a three-dimensional rotation can be specified with three angles and that a three-dimensional translation can be specified with the three parameters (x, y, z); thus we have six parameters so far. The OpenCV intrinsics matrix for a camera has four parameters (f_x, f_y, c_x, and c_y), yielding a grand total of ten parameters that must be solved for each view (but note that the camera intrinsic parameters stay the same between views). Using a planar object, we'll soon see that each view fixes eight parameters. Because the six parameters of rotation and translation change between views, for each view we have constraints on two additional parameters that we use to resolve the camera intrinsic matrix. We'll then need at least two views to solve for all the geometric parameters.

We'll provide more details on the parameters and their constraints later in the chapter, but first we discuss the *calibration object*. The calibration object used in OpenCV is a flat grid of alternating black and white squares that is usually called a "chessboard" (even though it needn't have eight squares, or even an equal number of squares, in each direction).

Chessboards

In principle, any appropriately characterized object could be used as a calibration object, yet the practical choice is a regular pattern such as a chessboard.* Some calibration methods in the literature rely on three-dimensional objects (e.g., a box covered with markers), but flat chessboard patterns are much easier to deal with; it is difficult to make (and to store and distribute) precise 3D calibration objects. OpenCV thus opts for using multiple views of a planar object (a chessboard) rather than one view of a specially constructed 3D object. We use a pattern of alternating black and white squares (see Figure 11-9), which ensures that there is no bias toward one side or the other in measurement. Also, the resulting grid corners lend themselves naturally to the subpixel localization function discussed in Chapter 10.

Given an image of a chessboard (or a person holding a chessboard, or any other scene with a chessboard and a reasonably uncluttered background), you can use the OpenCV function `cvFindChessboardCorners()` to locate the corners of the chessboard.

```
int cvFindChessboardCorners(
  const void*   image,
  CvSize        pattern_size,
  CvPoint2D32f* corners,
  int*          corner_count = NULL,
  int           flags        = CV_CALIB_CB_ADAPTIVE_THRESH
);
```

* The specific use of this calibration object—and much of the calibration approach itself—comes from Zhang [Zhang99; Zhang00] and Sturm [Sturm99].

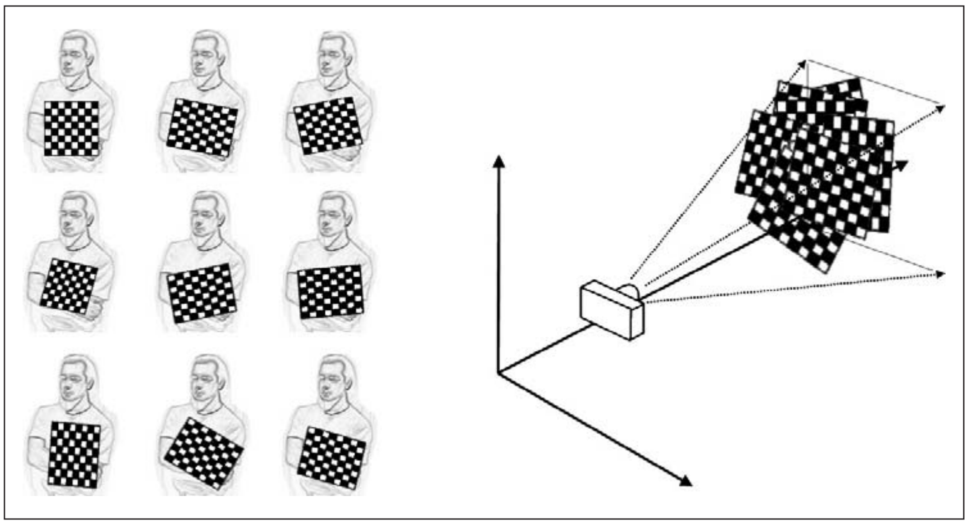


Figure 11-9. Images of a chessboard being held at various orientations (left) provide enough information to completely solve for the locations of those images in global coordinates (relative to the camera) and the camera intrinsics

This function takes as arguments a single image containing a chessboard. This image must be an 8-bit grayscale (single-channel) image. The second argument, `pattern_size`, indicates how many corners are in each row and column of the board. This count is the number of *interior* corners; thus, for a standard chess game board the correct value would be `cvSize(7,7)`.* The next argument, `corners`, is a pointer to an array where the corner locations can be recorded. This array must be preallocated and, of course, must be large enough for all of the corners on the board (49 on a standard chess game board). The individual values are the locations of the corners in pixel coordinates. The `corner_count` argument is optional; if non-`NULL`, it is a pointer to an integer where the number of corners found can be recorded. If the function is successful at finding all of the corners,† then the return value will be a nonzero number. If the function fails, 0 will be returned. The final `flags` argument can be used to implement one or more additional filtration steps to help find the corners on the chessboard. Any or all of the arguments may be combined using a Boolean OR.

`CV_CALIB_CB_ADAPTIVE_THRESH`
: The default behavior of `cvFindChessboardCorners()` is first to threshold the image based on average brightness, but if this flag is set then an adaptive threshold will be used instead.

* In practice, it is often more convenient to use a chessboard grid that is asymmetric and of even and odd dimensions—for example, (5, 6). Using such even-odd asymmetry yields a chessboard that has only one symmetry axis, so the board orientation can always be defined uniquely.

† Actually, the requirement is slightly stricter: not only must all the corners be found, they must also be ordered into rows and columns as expected. Only if the corners can be found and ordered correctly will the return value of the function be nonzero.

`CV_CALIB_CB_NORMALIZE_IMAGE`

If set, this flag causes the image to be normalized via `cvEqualizeHist()` before the thresholding is applied.

`CV_CALIB_CB_FILTER_QUADS`

Once the image is thresholded, the algorithm attempts to locate the quadrangles resulting from the perspective view of the black squares on the chessboard. This is an approximation because the lines of each edge of a quadrangle are assumed to be straight, which isn't quite true when there is radial distortion in the image. If this flag is set, then a variety of additional constraints are applied to those quadrangles in order to reject false quadrangles.

Subpixel corners

The corners returned by `cvFindChessboardCorners()` are only approximate. What this means in practice is that the locations are accurate only to within the limits of our imaging device, which means accurate to within one pixel. A separate function must be used to compute the exact locations of the corners (given the approximate locations and the image as input) to subpixel accuracy. This function is the same `cvFindCornerSubPix()` function that we used for tracking in Chapter 10. It should not be surprising that this function can be used in this context, since the chessboard interior corners are simply a special case of the more general Harris corners; the chessboard corners just happen to be particularly easy to find and track. Neglecting to call subpixel refinement after you first locate the corners can cause substantial errors in calibration.

Drawing chessboard corners

Particularly when debugging, it is often desirable to draw the found chessboard corners onto an image (usually the image that we used to compute the corners in the first place); this way, we can see whether the projected corners match up with the observed corners. Toward this end, OpenCV provides a convenient routine to handle this common task. The function `cvDrawChessboardCorners()` draws the corners found by `cvFindChessboardCorners()` onto an image that you provide. If not all of the corners were found, the available corners will be represented as small red circles. If the entire pattern was found, then the corners will be painted into different colors (each row will have its own color) and connected by lines representing the identified corner order.

```
void cvDrawChessboardCorners(
    CvArr*        image,
    CvSize        pattern_size,
    CvPoint2D32f* corners,
    int           count,
    int           pattern_was_found
);
```

The first argument to `cvDrawChessboardCorners()` is the image to which the drawing will be done. Because the corners will be represented as colored circles, this must be an 8-bit color image; in most cases, this will be a copy of the image you gave to `cvFindChessboardCorners()` (but you must convert it to a three-channel image yourself).

The next two arguments, pattern_size and corners, are the same as the corresponding arguments for cvFindChessboardCorners(). The argument count is an integer equal to the number of corners. Finally the argument pattern_was_found indicates whether the entire chessboard pattern was successfully found; this can be set to the return value from cvFindChessboardCorners(). Figure 11-10 shows the result of applying cvDrawChessboardCorners() to a chessboard image.

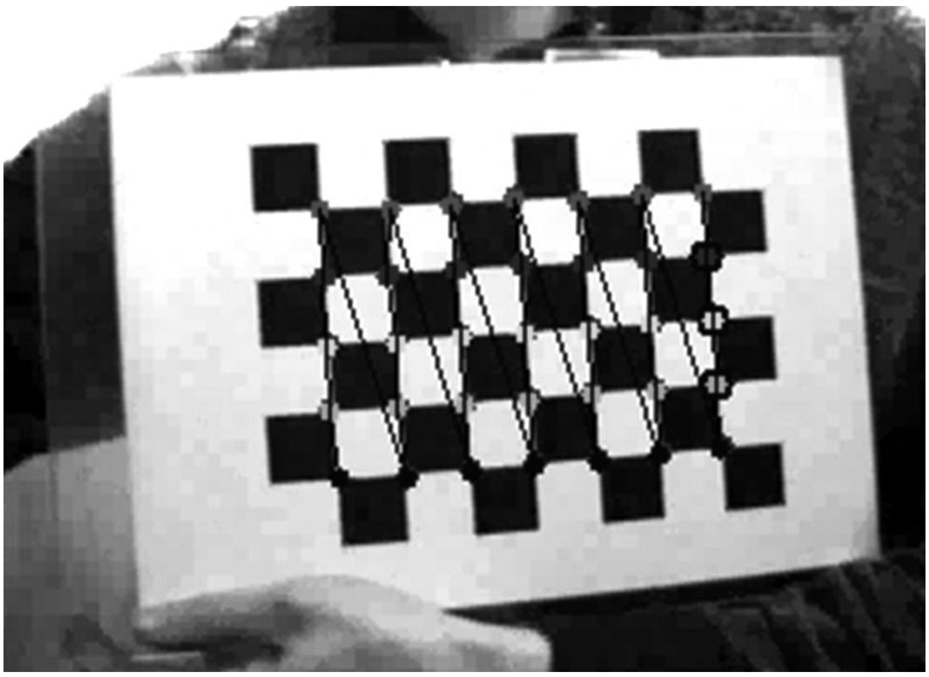


Figure 11-10. Result of cvDrawChessboardCorners(); once you find the corners using cvFindChessboardCorners(), you can project where these corners were found (small circles on corners) and in what order they belong (as indicated by the lines between circles)

We now turn to what a planar object can do for us. Points on a plane undergo perspective transform when viewed through a pinhole or lens. The parameters for this transform are contained in a 3-by-3 *homography* matrix, which we describe next.

Homography

In computer vision, we define *planar homography* as a projective mapping from one plane to another.* Thus, the mapping of points on a two-dimensional planar surface to

* The term "homography" has different meanings in different sciences; for example, it has a somewhat more general meaning in mathematics. The homographies of greatest interest in computer vision are a subset of the other, more general, meanings of the term.

the imager of our camera is an example of planar homography. It is possible to express this mapping in terms of matrix multiplication if we use homogeneous coordinates to express both the viewed point *Q* and the point *q* on the imager to which *Q* is mapped. If we define:

$$\tilde{Q} = \begin{bmatrix} X & Y & Z & 1 \end{bmatrix}^T$$
$$\tilde{q} = \begin{bmatrix} x & y & 1 \end{bmatrix}^T$$

then we can express the action of the homography simply as:

$$\tilde{q} = sH\tilde{Q}$$

Here we have introduced the parameter *s*, which is an arbitrary scale factor (intended to make explicit that the homography is defined only up to that factor). It is conventionally factored out of *H*, and we'll stick with that convention here.

With a little geometry and some matrix algebra, we can solve for this transformation matrix. The most important observation is that *H* has two parts: the physical transformation, which essentially locates the object plane we are viewing; and the projection, which introduces the camera intrinsics matrix. See Figure 11-11.

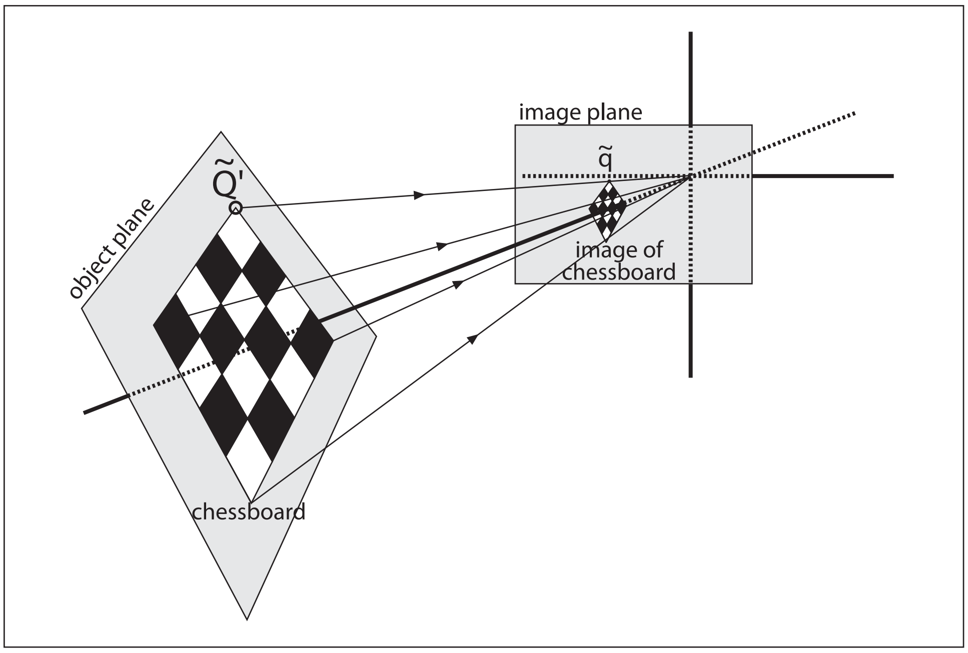


Figure 11-11. View of a planar object as described by homography: a mapping—from the object plane to the image plane—that simultaneously comprehends the relative locations of those two planes as well as the camera projection matrix

The physical transformation part is the sum of the effects of some rotation R and some translation $\mathbf{t}$ that relate the plane we are viewing to the image plane. Because we are working in homogeneous coordinates, we can combine these within a single matrix as follows:*

$$W = \begin{bmatrix} R & \mathbf{t} \end{bmatrix}$$

Then, the action of the camera matrix M (which we already know how to express in projective coordinates) is multiplied by $W\tilde{Q}$; this yields:

$$\tilde{q} = sMW\tilde{Q}, \quad \text{where} \quad M = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

It would seem that we are done. However, it turns out that in practice our interest is not the coordinate $\tilde{Q}$, which is defined for all of space, but rather a coordinate $\tilde{Q}'$, which is defined only on the plane we are looking at. This allows for a slight simplification.

Without loss of generality, we can choose to define the object plane so that $Z = 0$. We do this because, if we also break up the rotation matrix into three 3-by-1 columns (i.e., $R = [r_1\ r_2\ r_3]$), then one of those columns is not needed. In particular:

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = sM\begin{bmatrix} r_1 & r_2 & r_3 & \mathbf{t} \end{bmatrix}\begin{bmatrix} X \\ Y \\ 0 \\ 1 \end{bmatrix} = sM\begin{bmatrix} r_1 & r_2 & \mathbf{t} \end{bmatrix}\begin{bmatrix} X \\ Y \\ 1 \end{bmatrix}$$

The homography matrix H that maps a planar object's points onto the imager is then described completely by $H = sM[r_1\ r_2\ \mathbf{t}]$, where:

$$\tilde{q} = sH\tilde{Q}'$$

Observe that H is now a 3-by-3 matrix.

OpenCV uses the preceding equations to compute the homography matrix. It uses multiple images of the same object to compute both the individual translations and rotations for each view as well as the intrinsics (which are the same for all views). As we have discussed, rotation is described by three angles and translation is defined by three offsets; hence there are six unknowns for each view. This is OK, because a known planar object (such as our chessboard) gives us eight equations—that is, the mapping of a square into a quadrilateral can be described by four (x, y) points. Each new frame gives us eight equations at the cost of six new extrinsic unknowns, so given enough images we should be able to compute any number of intrinsic unknowns (more on this shortly).

* Here $W = [R\ \mathbf{t}]$ is a 3-by-4 matrix whose first three columns comprise the nine entries of R and whose last column consists of the three-component vector $\mathbf{t}$.

The homography matrix H relates the positions of the points on a source image plane to the points on the destination image plane (usually the imager plane) by the following simple equations:

$$p_{\text{dst}} = Hp_{\text{src}}, \quad p_{\text{src}} = H^{-1}p_{\text{dst}}$$

$$p_{\text{dst}} = \begin{bmatrix} x_{\text{dst}} \\ y_{\text{dst}} \\ 1 \end{bmatrix}, \quad p_{\text{src}} = \begin{bmatrix} x_{\text{src}} \\ y_{\text{src}} \\ 1 \end{bmatrix}$$

Notice that we can compute H without knowing anything about the camera intrinsics. In fact, computing multiple homographies from multiple views is the method OpenCV uses to solve for the camera intrinsics, as we'll see.

OpenCV provides us with a handy function, `cvFindHomography()`, which takes a list of correspondences and returns the homography matrix that best describes those correspondences. We need a minimum of four points to solve for H, but we can supply many more if we have them* (as we will with any chessboard bigger than 3-by-3). Using more points is beneficial, because invariably there will be noise and other inconsistencies whose effect we would like to minimize.

```
void cvFindHomography(
    const CvMat* src_points,
    const CvMat* dst_points,
    CvMat*       homography
);
```

The input arrays `src_points` and `dst_points` can be either N-by-2 matrices or N-by-3 matrices. In the former case the points are pixel coordinates, and in the latter they are expected to be homogeneous coordinates. The final argument, `homography`, is just a 3-by-3 matrix to be filled by the function in such a way that the back-projection error is minimized. Because there are only eight free parameters in the homography matrix, we chose a normalization where $H_{33} = 1$. Scaling the homography could be applied to the ninth homography parameter, but usually scaling is instead done by multiplying the entire homography matrix by a scale factor.

Camera Calibration

We finally arrive at camera calibration for camera intrinsics and distortion parameters. In this section we'll learn how to compute these values using `cvCalibrateCamera2()` and also how to use these models to correct distortions in the images that the calibrated camera would have otherwise produced. First we say a little more about how many views of a chessboard are necessary in order to solve for the intrinsics and distortion. Then we'll offer a high-level overview of how OpenCV actually solves this system before moving on to the code that makes it all easy to do.

* Of course, an exact solution is guaranteed only when there are four correspondences. If more are provided, then what's computed is a solution that is optimal in the sense of least-squares error.

How many chess corners for how many parameters?

It will prove instructive to review our unknowns. That is, how many parameters are we attempting to solve for through calibration? In the OpenCV case, we have four *intrinsic* parameters (f_x, f_y, c_x, c_y,) and five *distortion* parameters: three radial (k_1, k_2, k_3) and two tangential (p_1, p_2). Intrinsic parameters are directly tied to the 3D geometry (and hence the extrinsic parameters) of where the chessboard is in space; distortion parameters are tied to the 2D geometry of how the pattern of points gets distorted, so we deal with the constraints on these two classes of parameters separately. Three corner points in a known pattern yielding six pieces of information are (in principle) all that is needed to solve for our five distortion parameters (of course, we use much more for robustness). Thus, one view of a chessboard is all that we need to compute our distortion parameters. The same chessboard view could also be used in our intrinsics computation, which we consider next, starting with the extrinsic parameters. For the *extrinsic* parameters we'll need to know where the chessboard is. This will require three rotation parameters (ψ, ϕ, θ) and three translation parameters (T_x, T_y, T_z) for a total of six per view of the chessboard, because in each image the chessboard will move. Together, the four intrinsic and six extrinsic parameters make for ten altogether that we must solve for each view.

Let's say we have N corners and K images of the chessboard (in different positions). How many views and corners must we see so that there will be enough constraints to solve for all these parameters?

- K images of the chessboard provide $2NK$ constraints (we use the multiplier 2 because each point on the image has both an x and a y coordinate).
- Ignoring the distortion parameters for the moment, we have 4 intrinsic parameters and $6K$ extrinsic parameters (since we need to find the 6 parameters of the chessboard location in each of the K views).
- Solving then requires that $2NK \geq 6K + 4$ hold (or, equivalently, $(N - 3)\, K \geq 2$).

It seems that if $N = 5$ then we need only $K = 1$ image, but watch out! For us, K (the number of images) must be more than 1. The reason for requiring $K > 1$ is that we're using chessboards for calibration to fit a homography matrix for each of the K views. As discussed previously, a homography can yield at most eight parameters from four (x, y) pairs. This is because only four points are needed to express everything that a planar perspective view can do: it can stretch a square in four different directions at once, turning it into any quadrilateral (see the perspective images in Chapter 6). So, no matter how many corners we detect on a plane, we only get four corners' worth of information. Per chessboard view, then, the equation can give us only four corners of information or $(4 - 3)\, K > 1$, which means $K > 1$. This implies that two views of a 3-by-3 chessboard (counting only internal corners) are the minimum that could solve our calibration problem. Consideration for noise and numerical stability is typically what requires the collection of more images of a larger chessboard. In practice, for high-quality results, you'll need at least ten images of a 7-by-8 or larger chessboard (and that's only if you move the chessboard enough between images to obtain a "rich" set of views).

What's under the hood?

This subsection is for those who want to go deeper; it can be safely skipped if you just want to call the calibration functions. If you are still with us, the question remains: how is all this mathematics used for calibration? Although there are many ways to solve for the camera parameters, OpenCV chose one that works well on planar objects. The algorithm OpenCV uses to solve for the focal lengths and offsets is based on Zhang's method [Zhang00], but OpenCV uses a different method based on Brown [Brown71] to solve for the distortion parameters.

To get started, we pretend that there is no distortion in the camera while solving for the other calibration parameters. For each view of the chessboard, we collect a homography H as described previously. We'll write H out as column vectors, $H = [h_1\ h_2\ h_3]$, where each h is a 3-by-1 vector. Then, in view of the preceding homography discussion, we can set H equal to the camera intrinsics matrix M multiplied by a combination of the first two rotation matrix columns, r_1 and r_2, and the translation vector **t**; after including the scale factor s, this yields:

$$H = \begin{bmatrix} h_1 & h_2 & h_3 \end{bmatrix} = sM \begin{bmatrix} r_1 & r_2 & \mathbf{t} \end{bmatrix}$$

Reading off these equations, we have:

$$h_1 = sMr_1 \quad \text{or} \quad r_1 = \lambda M^{-1} h_1$$

$$h_2 = sMr_2 \quad \text{or} \quad r_2 = \lambda M^{-1} h_2$$

$$h_3 = sMt \quad \text{or} \quad t = \lambda M^{-1} h_3$$

Here, $\lambda = 1/s$.

The rotation vectors are orthogonal to each other by construction, and since the scale is extracted it follows that r_1 and r_2 are orthonormal. Orthonormal implies two things: the rotation vector's dot product is 0, and the vectors' magnitudes are equal. Starting with the dot product, we have:

$$r_1^{\mathrm{T}} r_2 = 0$$

For any vectors a and b we have $(ab)^{\mathrm{T}} = b^{\mathrm{T}} a^{\mathrm{T}}$, so we can substitute for r_1 and r_2 to derive our first constraint:

$$h_1^{\mathrm{T}} M^{-\mathrm{T}} M^{-1} h_2 = 0$$

where $A^{-\mathrm{T}}$ is shorthand for $(A^{-1})^{\mathrm{T}}$. We also know that the magnitudes of the rotation vectors are equal:

$$\|r_1\| = \|r_2\| \quad \text{or} \quad r_1^{\mathrm{T}} r_1 = r_2^{\mathrm{T}} r_2$$

Substituting for r_1 and r_2 yields our second constraint:

$$h_1^{\mathrm{T}} M^{-\mathrm{T}} M^{-1} h_1 = h_2^{\mathrm{T}} M^{-\mathrm{T}} M^{-1} h_2$$

To make things easier, we set $B = M^{-T}M^{-1}$. Writing this out, we have:

$$B = M^{-T}M^{-1} = \begin{bmatrix} B_{11} & B_{12} & B_{13} \\ B_{12} & B_{22} & B_{23} \\ B_{13} & B_{23} & B_{33} \end{bmatrix}$$

It so happens that this matrix B has a general closed-form solution:

$$B = \begin{bmatrix} \frac{1}{f_x^2} & 0 & \frac{-c_x}{f_x^2} \\ 0 & \frac{1}{f_y^2} & \frac{-c_y}{f_y^2} \\ \frac{-c_x}{f_x^2} & \frac{-c_y}{f_y^2} & \frac{c_x^2}{f_x^2} + \frac{c_y^2}{f_y^2} + 1 \end{bmatrix}$$

Using the B-matrix, both constraints have the general form $h_i^T B h_j$ in them. Let's multiply this out to see what the components are. Because B is symmetric, it can be written as one six-dimensional vector dot product. Arranging the necessary elements of B into the new vector b, we have:

$$h_i^T B h_j = v_{ij}^T b = \begin{bmatrix} h_{i1}h_{j1} \\ h_{i1}h_{j2} + h_{i2}h_{j1} \\ h_{i2}h_{j2} \\ h_{i3}h_{j1} + h_{i1}h_{j3} \\ h_{i3}h_{j2} + h_{i2}h_{j3} \\ h_{i3}h_{i3} \end{bmatrix}^T \begin{bmatrix} B_{11} \\ B_{12} \\ B_{22} \\ B_{13} \\ B_{23} \\ B_{33} \end{bmatrix}^T$$

Using this definition for v_{ij}^T, our two constraints may now be written as:

$$\begin{bmatrix} v_{12}^T \\ (v_{11} - v_{22})^T \end{bmatrix} b = 0$$

If we collect K images of chessboards together, then we can stack K of these equations together:

$$Vb = 0$$

where V is a $2K$-by-6 matrix. As before, if $K \geq 2$ then this equation can be solved for our $b = [B_{11}, B_{12}, B_{22}, B_{13}, B_{23}, B_{33}]^T$. The camera intrinsics are then pulled directly out of our closed-form solution for the B-matrix:

$$f_x = \sqrt{\lambda / B_{11}}$$
$$f_y = \sqrt{\lambda B_{11} / (B_{11}B_{22} - B_{12}^2)}$$
$$c_x = -B_{13} f_x^2 / \lambda$$
$$c_y = (B_{12}B_{13} - B_{11}B_{23}) / (B_{11}B_{22} - B_{12}^2)$$

where:

$$\lambda = B_{33} - (B_{13}^2 + c_y(B_{12}B_{13} - B_{11}B_{23})) / B_{11}$$

The extrinsics (rotation and translation) are then computed from the equations we read off of the homography condition:

$$r_1 = \lambda M^{-1} h_1$$
$$r_2 = \lambda M^{-1} h_2$$
$$r_3 = r_1 \times r_2$$
$$t = \lambda M^{-1} h_3$$

Here the scaling parameter is determined from the orthonormality condition $\lambda = 1/\|M^{-1}h_1\|$.

Some care is required because, when we solve using real data and put the *r*-vectors together ($R = [r_1\ r_2\ r_3]$), we will not end up with an exact rotation matrix for which $R^TR = RR^T = I$ holds.

To get around this problem, the usual trick is to take the singular value decomposition (SVD) of *R*. As discussed in Chapter 3, SVD is a method of factoring a matrix into two orthonormal matrices, *U* and *V*, and a middle matrix *D* of scale values on its diagonal. This allows us to turn *R* into $R = UDV^T$. Because *R* is itself orthonormal, the matrix *D* must be the identity matrix *I* such that $R = UIV^T$. We can thus "coerce" our computed *R* into being a rotation matrix by taking *R*'s singular value decomposition, setting its *D* matrix to the identity matrix, and multiplying by the SVD again to yield our new, conforming rotation matrix *R′*.

Despite all this work, we have not yet dealt with lens distortions. We use the camera intrinsics found previously—together with the distortion parameters set to 0—for our initial guess to start solving a larger system of equations.

The points we "perceive" on the image are really in the wrong place owing to distortion. Let (x_p, y_p) be the point's location if the pinhole camera were perfect and let (x_d, y_d) be its distorted location; then:

$$\begin{bmatrix} x_p \\ y_p \end{bmatrix} = \begin{bmatrix} f_x X^W / Z^W + c_x \\ f_y X^W / Z^W + c_y \end{bmatrix}$$

We use the results of the calibration without distortion via the following substitution:

$$\begin{bmatrix} x_p \\ y_p \end{bmatrix} = (1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \begin{bmatrix} x_d \\ y_d \end{bmatrix} + \begin{bmatrix} 2p_1 x_d y_d + p_2 (r^2 + 2x_d^2) \\ p_1 (r^2 + 2y_d^2) + 2p_2 x_d y_d \end{bmatrix}$$

A large list of these equations are collected and solved to find the distortion parameters, after which the intrinsics and extrinsics are reestimated. That's the heavy lifting that the single function `cvCalibrateCamera2()`[*] does for you!

Calibration function

Once we have the corners for several images, we can call `cvCalibrateCamera2()`. This routine will do the number crunching and give us the information we want. In particular, the results we receive are the *camera intrinsics matrix*, the *distortion coefficients*, the *rotation vectors*, and the *translation vectors*. The first two of these constitute the intrinsic parameters of the camera, and the latter two are the extrinsic measurements that tell us where the objects (i.e., the chessboards) were found and what their orientations were. The distortion coefficients (k_1, k_2, p_1, p_2, and k_3)[†] are the coefficients from the radial and tangential distortion equations we encountered earlier; they help us when we want to correct that distortion away. The camera intrinsic matrix is perhaps the most interesting final result, because it is what allows us to transform from 3D coordinates to the image's 2D coordinates. We can also use the camera matrix to do the reverse operation, but in this case we can only compute a line in the three-dimensional world to which a given image point must correspond. We will return to this shortly.

Let's now examine the camera calibration routine itself.

```
void cvCalibrateCamera2(
  CvMat*    object_points,
  CvMat*    image_points,
  int*      point_counts,
  CvSize    image_size,
  CvMat*    intrinsic_matrix,
  CvMat*    distortion_coeffs,
  CvMat*    rotation_vectors    = NULL,
  CvMat*    translation_vectors = NULL,
  int       flags               = 0
);
```

When calling `cvCalibrateCamera2()`, there are many arguments to keep straight. Yet we've covered (almost) all of them already, so hopefully they'll make sense.

* The `cvCalibrateCamera2()` function is used internally in the stereo calibration functions we will see in Chapter 12. For stereo calibration, we'll be calibrating two cameras at the same time and will be looking to relate them together through a rotation matrix and a translation vector.

† The third radial distortion component k_3 comes last because it was a late addition to OpenCV to allow better correction to highly distorted fish eye type lenses and should only be used in such cases. We will see momentarily that k_3 can be set to 0 by first initializing it to 0 and then setting the flag to `CV_CALIB_FIX_K3`.

The first argument is the object_points, which is an N-by-3 matrix containing the physical coordinates of each of the K points on each of the M images of the object (i.e., $N = K \times M$). These points are located in the coordinate frame attached to the object.* This argument is a little more subtle than it appears in that your manner of describing the points on the object will implicitly define your physical units and the structure of your coordinate system hereafter. In the case of a chessboard, for example, you might define the coordinates such that all of the points on the chessboard had a z-value of 0 while the x- and y-coordinates are measured in centimeters. Had you chosen inches, all computed parameters would then (implicitly) be in inches. Similarly if you had chosen all the x-coordinates (rather than the z-coordinates) to be 0, then the implied location of the chessboards relative to the camera would be largely in the x-direction rather than the z-direction. The squares define one unit, so that if, for example, your squares are 90 mm on each side, your camera world, object and camera coordinate units would be in mm/90. In principle you can use an object other than a chessboard, so it is not really necessary that all of the object points lie on a plane, but this is usually the easiest way to calibrate a camera.† In the simplest case, we simply define each square of the chessboard to be of dimension one "unit" so that the coordinates of the corners on the chessboard are just integer corner rows and columns. Defining S_{width} as the number of squares across the width of the chessboard and S_{height} as the number of squares over the height:

$$(0,0),(0,1),(0,2),\ldots,(1,0),(2,0),\ldots,(1,1),\ldots,(S_{\text{width}}-1,S_{\text{height}}-1)$$

The second argument is the image_points, which is an N-by-2 matrix containing the pixel coordinates of all the points supplied in object_points. If you are performing a calibration using a chessboard, then this argument consists simply of the return values for the M calls to cvFindChessboardCorners() but now rearranged into a slightly different format.

The argument point_counts indicates the number of points in each image; this is supplied as an M-by-1 matrix. The image_size is just the size, in pixels, of the images from which the image points were extracted (e.g., those images of yourself waving a chessboard around).

The next two arguments, intrinsic_matrix and distortion_coeffs, constitute the intrinsic parameters of the camera. These arguments can be both outputs (filling them in is the main reason for calibration) and inputs. When used as inputs, the values in these matrices when the function is called will affect the computed result. Which of these matrices will be used as input will depend on the flags parameter; see the following discussion. As we discussed earlier, the intrinsic matrix completely specifies the behavior

* Of course, it's normally the same object in every image, so the N points described are actually M repeated listings of the locations of the K points on a single object.

† At the time of this writing, automatic initialization of the intrinsic matrix before the optimization algorithm runs has been implemented only for planar calibration objects. This means that if you have a nonplanar object then you must provide a starting guess for the principal point and focal lengths (see CV_CALIB_USE_INTRINSIC_GUESS to follow).

of the camera in our ideal camera model, while the distortion coefficients characterize much of the camera's nonideal behavior. The camera matrix is always 3-by-3 and the distortion coefficients always number five, so the `distortion_coeffs` argument should be a pointer to a 5-by-1 matrix (they will be recorded in the order k_1, k_2, p_1, p_2, k_3).

Whereas the previous two arguments summarized the camera's intrinsic information, the next two summarize the extrinsic information. That is, they tell us where the calibration objects (e.g., the chessboards) were located relative to the camera in each picture. The locations of the objects are specified by a rotation and a translation.* The rotations, `rotation_vectors`, are defined by M three-component vectors arranged into an M-by-3 matrix (where M is the number of images). Be careful, these are not in the form of the 3-by-3 rotation matrix we discussed previously; rather, each vector represents an axis in three-dimensional space in the camera coordinate system around which the chessboard was rotated and where the length or magnitude of the vector encodes the counterclockwise angle of the rotation. Each of these rotation vectors can be converted to a 3-by-3 rotation matrix by calling `cvRodrigues2()`, which is described in its own section to follow. The translations, `translation_vectors`, are similarly arranged into a second M-by-3 matrix, again in the camera coordinate system. As stated before, the units of the camera coordinate system are exactly those assumed for the chessboard. That is, if a chessboard square is 1 inch by 1 inch, the units are inches.

Finding parameters through optimization can be somewhat of an art. Sometimes trying to solve for all parameters at once can produce inaccurate or divergent results if your initial starting position in parameter space is far from the actual solution. Thus, it is often better to "sneak up" on the solution by getting close to a good parameter starting position in stages. For this reason, we often hold some parameters fixed, solve for other parameters, then hold the other parameters fixed and solve for the original and so on. Finally, when we think all of our parameters are close to the actual solution, we use our close parameter setting as the starting point and solve for everything at once. OpenCV allows you this control through the `flags` setting. The `flags` argument allows for some finer control of exactly how the calibration will be performed. The following values may be combined together with a Boolean OR operation as needed.

`CV_CALIB_USE_INTRINSIC_GUESS`

Normally the intrinsic matrix is computed by `cvCalibrateCamera2()` with no additional information. In particular, the initial values of the parameters c_x and c_y (the image center) are taken directly from the `image_size` argument. If this argument is set, then `intrinsic_matrix` is assumed to contain valid values that will be used as an initial guess to be further optimized by `cvCalibrateCamera2()`.

* You can envision the chessboard's location as being expressed by (1) "creating" a chessboard at the origin of your camera coordinates, (2) rotating that chessboard by some amount around some axis, and (3) moving that oriented chessboard to a particular place. For those who have experience with systems like OpenGL, this should be a familiar construction.

`CV_CALIB_FIX_PRINCIPAL_POINT`
: This flag can be used with or without `CV_CALIB_USE_INTRINSIC_GUESS`. If used without, then the principle point is fixed at the center of the image; if used with, then the principle point is fixed at the supplied initial value in the `intrinsic_matrix`.

`CV_CALIB_FIX_ASPECT_RATIO`
: If this flag is set, then the optimization procedure will only vary f_x and f_y together and will keep their ratio fixed to whatever value is set in the `intrinsic_matrix` when the calibration routine is called. (If the `CV_CALIB_USE_INTRINSIC_GUESS` flag is not also set, then the values of f_x and f_y in `intrinsic_matrix` can be any arbitrary values and only their ratio will be considered relevant.)

`CV_CALIB_FIX_FOCAL_LENGTH`
: This flag causes the optimization routine to just use the f_x and f_y that were passed in in the `intrinsic_matrix`.

`CV_CALIB_FIX_K1`, `CV_CALIB_FIX_K2` and `CV_CALIB_FIX_K3`
: Fix the radial distortion parameters k_1, k_2, and k_3. The radial parameters may be set in any combination by adding these flags together. In general, the last parameter should be fixed to 0 unless you are using a fish-eye lens.

`CV_CALIB_ZERO_TANGENT_DIST`:
: This flag is important for calibrating high-end cameras which, as a result of precision manufacturing, have very little tangential distortion. Trying to fit parameters that are near 0 can lead to noisy spurious values and to problems of numerical stability. Setting this flag turns off fitting the tangential distortion parameters p_1 and p_2, which are thereby both set to 0.

Computing extrinsics only

In some cases you will already have the intrinsic parameters of the camera and therefore need only to compute the location of the object(s) being viewed. This scenario clearly differs from the usual camera calibration, but it is nonetheless a useful task to be able to perform.

```
void cvFindExtrinsicCameraParams2(
    const CvMat* object_points,
    const CvMat* image_points,
    const CvMat* intrinsic_matrix,
    const CvMat* distortion_coeffs,
    CvMat*       rotation_vector,
    CvMat*       translation_vector
);
```

The arguments to `cvFindExtrinsicCameraParams2()` are identical to the corresponding arguments for `cvCalibrateCamera2()` with the exception that the intrinsic matrix and the distortion coefficients are being supplied rather than computed. The rotation output is in the form of a 1-by-3 or 3-by-1 `rotation_vector` that represents the 3D axis around which the chessboard or points were rotated, and the vector magnitude or length represents the counterclockwise angle of rotation. This rotation vector can be converted into the 3-by-3 rotation matrix we've discussed before via the cvRodrigues2() function. The translation vector is the offset in camera coordinates to where the chessboard origin is located.

Undistortion

As we have alluded to already, there are two things that one often wants to do with a calibrated camera. The first is to correct for distortion effects, and the second is to construct three-dimensional representations of the images it receives. Let's take a moment to look at the first of these before diving into the more complicated second task in Chapter 12.

OpenCV provides us with a ready-to-use undistortion algorithm that takes a raw image and the distortion coefficients from cvCalibrateCamera2() and produces a corrected image (see Figure 11-12). We can access this algorithm either through the function cvUndistort2(), which does everything we need in one shot, or through the pair of routines cvInitUndistortMap() and cvRemap(), which allow us to handle things a little more efficiently for video or other situations where we have many images from the same camera.*

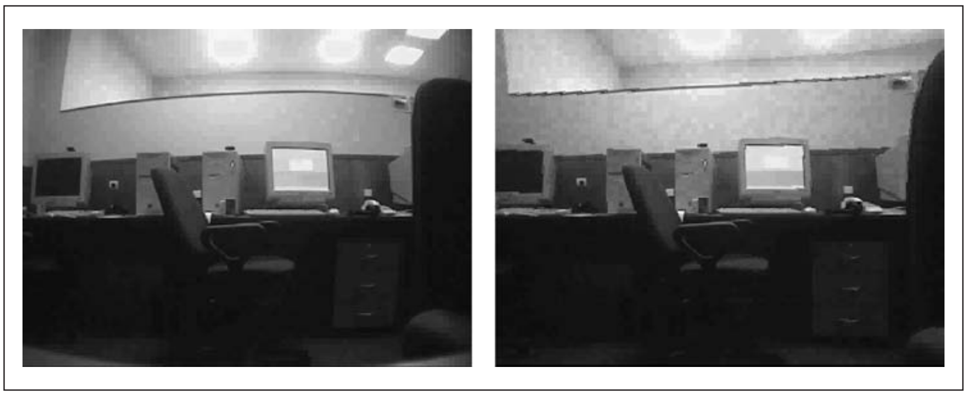


Figure 11-12. Camera image before undistortion (left) and after undistortion (right)

The basic method is to compute a *distortion map*, which is then used to correct the image. The function cvInitUndistortMap() computes the distortion map, and cvRemap() can be used to apply this map to an arbitrary image.† The function cvUndistort2() does one after the other in a single call. However, computing the distortion map is a time-consuming operation, so it's not very smart to keep calling cvUndistort2() if the distortion map is not changing. Finally, if we just have a list of 2D points, we can call the function cvUndistortPoints() to transform them from their original coordinates to their undistorted coordinates.

* We should take a moment to clearly make a distinction here between *undistortion*, which mathematically removes lens distortion, and *rectification*, which mathematically aligns the images with respect to each other.

† We first encountered cvRemap() in the context of image transformations (Chapter 6).

```
// Undistort images
void cvInitUndistortMap(
  const CvMat*   intrinsic_matrix,
  const CvMat*   distortion_coeffs,
  cvArr*         mapx,
  cvArr*         mapy
);
void cvUndistort2(
  const CvArr*   src,
  CvArr*         dst,
  const cvMat*   intrinsic_matrix,
  const cvMat*   distortion_coeffs
);
// Undistort a list of 2D points only
void cvUndistortPoints(
  const CvMat* _src,
  CvMat*       dst,
  const CvMat*  intrinsic_matrix,
  const CvMat*  distortion_coeffs,
  const CvMat*  R  = 0,
  const CvMat*  Mr = 0;
);
```

The function `cvInitUndistortMap()` computes the distortion map, which relates each point in the image to the location where that point is mapped. The first two arguments are the camera intrinsic matrix and the distortion coefficients, both in the expected form as received from `cvCalibrateCamera2()`. The resulting distortion map is represented by two separate 32-bit, single-channel arrays: the first gives the *x*-value to which a given point is to be mapped and the second gives the *y*-value. You might be wondering why we don't just use a single two-channel array instead. The reason is so that the results from `cvUnitUndistortMap()` can be passed directly to `cvRemap()`.

The function `cvUndistort2()` does all this in a single pass. It takes your initial (distorted image) as well as the camera's intrinsic matrix and distortion coefficients, and then outputs an undistorted image of the same size. As mentioned previously, `cvUndistortPoints()` is used if you just have a list of 2D point coordinates from the original image and you want to compute their associated undistorted point coordinates. It has two extra parameters that relate to its use in stereo rectification, discussed in Chapter 12. These parameters are `R`, the rotation matrix between the two cameras, and `Mr`, the camera intrinsic matrix of the rectified camera (only really used when you have two cameras as per Chapter 12). The rectified camera matrix `Mr` can have dimensions of 3-by-3 or 3-by-4 deriving from the first three or four columns of `cvStereoRectify()`'s return value for camera matrices `P1` or `P2` (for the left or right camera; see Chapter 12). These parameters are by default NULL, which the function interprets as identity matrices.

Putting Calibration All Together

OK, now it's time to put all of this together in an example. We'll present a program that performs the following tasks: it looks for chessboards of the dimensions that the user specified, grabs as many full images (i.e., those in which it can find all the chessboard corners) as the user requested, and computes the camera intrinsics and distortion parameters. Finally, the program enters a display mode whereby an undistorted version of the camera image can be viewed; see Example 11-1. When using this algorithm, you'll want to substantially change the chessboard views between successful captures. Otherwise, the matrices of points used to solve for calibration parameters may form an ill-conditioned (rank deficient) matrix and you will end up with either a bad solution or no solution at all.

Example 11-1. Reading a chessboard's width and height, reading and collecting the requested number of views, and calibrating the camera

```
// calib.cpp
// Calling convention:
// calib board_w board_h number_of_views
//
// Hit 'p' to pause/unpause, ESC to quit
//
#include <cv.h>
#include <highgui.h>
#include <stdio.h>
#include <stdlib.h>

int n_boards = 0; //Will be set by input list
const int board_dt = 20; //Wait 20 frames per chessboard view
int board_w;
int board_h;

int main(int argc, char* argv[]) {

  if(argc != 4){
    printf("ERROR: Wrong number of input parameters\n");
    return -1;
  }
  board_w  = atoi(argv[1]);
  board_h  = atoi(argv[2]);
  n_boards = atoi(argv[3]);
  int board_n  = board_w * board_h;
  CvSize board_sz = cvSize( board_w, board_h );
  CvCapture* capture = cvCreateCameraCapture( 0 );
  assert( capture );

  cvNamedWindow( "Calibration" );
  //ALLOCATE STORAGE
  CvMat* image_points      = cvCreateMat(n_boards*board_n,2,CV_32FC1);
  CvMat* object_points     = cvCreateMat(n_boards*board_n,3,CV_32FC1);
  CvMat* point_counts      = cvCreateMat(n_boards,1,CV_32SC1);
  CvMat* intrinsic_matrix  = cvCreateMat(3,3,CV_32FC1);
  CvMat* distortion_coeffs = cvCreateMat(5,1,CV_32FC1);

  CvPoint2D32f* corners = new CvPoint2D32f[ board_n ];
  int corner_count;
  int successes = 0;
  int step, frame = 0;
```

Example 11-1. Reading a chessboard's width and height, reading and collecting the requested number of views, and calibrating the camera (continued)

```
IplImage *image = cvQueryFrame( capture );
IplImage *gray_image = cvCreateImage(cvGetSize(image),8,1);//subpixel

// CAPTURE CORNER VIEWS LOOP UNTIL WE'VE GOT n_boards
// SUCCESSFUL CAPTURES (ALL CORNERS ON THE BOARD ARE FOUND)
//
while(successes < n_boards) {
  //Skip every board_dt frames to allow user to move chessboard
  if(frame++ % board_dt == 0) {
     //Find chessboard corners:
     int found = cvFindChessboardCorners(
              image, board_sz, corners, &corner_count,
              CV_CALIB_CB_ADAPTIVE_THRESH | CV_CALIB_CB_FILTER_QUADS
     );

     //Get Subpixel accuracy on those corners
     cvCvtColor(image, gray_image, CV_BGR2GRAY);
     cvFindCornerSubPix(gray_image, corners, corner_count,
                cvSize(11,11),cvSize(-1,-1), cvTermCriteria(
                CV_TERMCRIT_EPS+CV_TERMCRIT_ITER, 30, 0.1 ));

     //Draw it
     cvDrawChessboardCorners(image, board_sz, corners,
                corner_count, found);
     cvShowImage( "Calibration", image );

     // If we got a good board, add it to our data
     if( corner_count == board_n ) {
        step = successes*board_n;
        for( int i=step, j=0; j<board_n; ++i,++j ) {
           CV_MAT_ELEM(*image_points, float,i,0) = corners[j].x;
           CV_MAT_ELEM(*image_points, float,i,1) = corners[j].y;
           CV_MAT_ELEM(*object_points,float,i,0) = j/board_w;
           CV_MAT_ELEM(*object_points,float,i,1) = j%board_w;
           CV_MAT_ELEM(*object_points,float,i,2) = 0.0f;
        }
        CV_MAT_ELEM(*point_counts, int,successes,0) = board_n;
        successes++;
     }
  } //end skip board_dt between chessboard capture

  //Handle pause/unpause and ESC
  int c = cvWaitKey(15);
  if(c == 'p'){
     c = 0;
     while(c != 'p' && c != 27){
          c = cvWaitKey(250);
     }
   }
   if(c == 27)
      return 0;
```

Example 11-1. Reading a chessboard's width and height, reading and collecting the requested number of views, and calibrating the camera (continued)

```
    image = cvQueryFrame( capture ); //Get next image
  } //END COLLECTION WHILE LOOP.

  //ALLOCATE MATRICES ACCORDING TO HOW MANY CHESSBOARDS FOUND
  CvMat* object_points2  = cvCreateMat(successes*board_n,3,CV_32FC1);
  CvMat* image_points2   = cvCreateMat(successes*board_n,2,CV_32FC1);
  CvMat* point_counts2   = cvCreateMat(successes,1,CV_32SC1);
  //TRANSFER THE POINTS INTO THE CORRECT SIZE MATRICES
  //Below, we write out the details in the next two loops. We could
  //instead have written:
  //image_points->rows = object_points->rows  = \
  //successes*board_n; point_counts->rows = successes;
  //
  for(int i = 0; i<successes*board_n; ++i) {
      CV_MAT_ELEM( *image_points2, float, i, 0) =
             CV_MAT_ELEM( *image_points, float, i, 0);
      CV_MAT_ELEM( *image_points2, float,i,1) =
             CV_MAT_ELEM( *image_points, float, i, 1);
      CV_MAT_ELEM(*object_points2, float, i, 0) =
             CV_MAT_ELEM( *object_points, float, i, 0) ;
      CV_MAT_ELEM( *object_points2, float, i, 1) =
             CV_MAT_ELEM( *object_points, float, i, 1) ;
      CV_MAT_ELEM( *object_points2, float, i, 2) =
             CV_MAT_ELEM( *object_points, float, i, 2) ;
  }
  for(int i=0; i<successes; ++i){ //These are all the same number
    CV_MAT_ELEM( *point_counts2, int, i, 0) =
             CV_MAT_ELEM( *point_counts, int, i, 0);
  }
  cvReleaseMat(&object_points);
  cvReleaseMat(&image_points);
  cvReleaseMat(&point_counts);

  // At this point we have all of the chessboard corners we need.
  // Initialize the intrinsic matrix such that the two focal
  // lengths have a ratio of 1.0
  //
  CV_MAT_ELEM( *intrinsic_matrix, float, 0, 0 ) = 1.0f;
  CV_MAT_ELEM( *intrinsic_matrix, float, 1, 1 ) = 1.0f;

  //CALIBRATE THE CAMERA!
  cvCalibrateCamera2(
      object_points2, image_points2,
      point_counts2,  cvGetSize( image ),
      intrinsic_matrix, distortion_coeffs,
      NULL, NULL,0  //CV_CALIB_FIX_ASPECT_RATIO
  );

  // SAVE THE INTRINSICS AND DISTORTIONS
  cvSave("Intrinsics.xml",intrinsic_matrix);
  cvSave("Distortion.xml",distortion_coeffs);
```

Example 11-1. Reading a chessboard's width and height, reading and collecting the requested number of views, and calibrating the camera (continued)

```
  // EXAMPLE OF LOADING THESE MATRICES BACK IN:
  CvMat *intrinsic = (CvMat*)cvLoad(“Intrinsics.xml”);
  CvMat *distortion = (CvMat*)cvLoad(“Distortion.xml”);

  // Build the undistort map that we will use for all
  // subsequent frames.
  //
  IplImage* mapx = cvCreateImage( cvGetSize(image), IPL_DEPTH_32F, 1 );
  IplImage* mapy = cvCreateImage( cvGetSize(image), IPL_DEPTH_32F, 1 );
  cvInitUndistortMap(
    intrinsic,
    distortion,
    mapx,
    mapy
  );
  // Just run the camera to the screen, now showing the raw and
  // the undistorted image.
  //
  cvNamedWindow( “Undistort” );
  while(image) {
    IplImage *t = cvCloneImage(image);
    cvShowImage( “Calibration”, image ); // Show raw image
    cvRemap( t, image, mapx, mapy );     // Undistort image
    cvReleaseImage(&t);
    cvShowImage(“Undistort”, image);     // Show corrected image

    //Handle pause/unpause and ESC
    int c = cvWaitKey(15);
    if(c == ‘p’) {
       c = 0;
       while(c != ‘p’ && c != 27) {
            c = cvWaitKey(250);
       }
    }
    if(c == 27)
        break;
    image = cvQueryFrame( capture );
  }

  return 0;
}
```

Rodrigues Transform

When dealing with three-dimensional spaces, one most often represents rotations in that space by 3-by-3 matrices. This representation is usually the most convenient because multiplication of a vector by this matrix is equivalent to rotating the vector in some way. The downside is that it can be difficult to intuit just what 3-by-3 matrix goes with what rotation. An alternate and somewhat easier-to-visualize* representation for a rotation is in the form of a vector about which the rotation operates together with a single angle. In this case it is standard practice to use only a single vector whose direction encodes the direction of the axis to be rotated around and to use the size of the vector to encode the amount of rotation in a counterclockwise direction. This is easily done because the direction can be equally well represented by a vector of any magnitude; hence we can choose the magnitude of our vector to be equal to the magnitude of the rotation. The relationship between these two representations, the matrix and the vector, is captured by the Rodrigues transform.† Let r be the three-dimensional vector $r = [r_x\ r_y\ r_z]$; this vector implicitly defines θ, the magnitude of the rotation by the length (or magnitude) of r. We can then convert from this axis-magnitude representation to a rotation matrix R as follows:

$$R = \cos(\theta)\cdot I + (1-\cos(\theta))\cdot rr^{\mathrm{T}} + \sin(\theta)\cdot \begin{bmatrix} 0 & -r_z & r_y \\ r_z & 0 & -r_x \\ r_y & r_x & 0 \end{bmatrix}$$

We can also go from a rotation matrix back to the axis-magnitude representation by using:

$$\sin(\theta)\cdot \begin{bmatrix} 0 & -r_z & r_y \\ r_z & 0 & -r_x \\ r_y & r_x & 0 \end{bmatrix} = \frac{(R-R^{\mathrm{T}})}{2}$$

Thus we find ourselves in the situation of having one representation (the matrix representation) that is most convenient for computation and another representation (the Rodrigues representation) that is a little easier on the brain. OpenCV provides us with a function for converting from either representation to the other.

```
void  cvRodrigues2(
  const CvMat*  src,
  CvMat*        dst,
  CvMat*        jacobian = NULL
);
```

Suppose we have the vector r and need the corresponding rotation matrix representation R; we set `src` to be the 3-by-1 vector r and `dst` to be the 3-by-3 rotation matrix R. Conversely, we can set `src` to be a 3-by-3 rotation matrix R and `dst` to be a 3-by-1 vector r. In either case, `cvRodrigues2()` will do the right thing. The final argument is optional. If `jacobian` is not `NULL`, then it should be a pointer to a 3-by-9 or a 9-by-3 matrix that will

* This "easier" representation is not just for humans. Rotation in 3D space has only three components. For numerical optimization procedures, it is more efficient to deal with the three components of the Rodrigues representation than with the nine components of a 3-by-3 rotation matrix.

† Rodrigues was a 19th-century French mathematician.

be filled with the partial derivatives of the output array components with respect to the input array components. The `jacobian` outputs are mainly used for the internal optimization algorithms of `cvFindExtrinsicCameraParameters2()` and `cvCalibrateCamera2()`; your use of the `jacobian` function will mostly be limited to converting the outputs of `cvFindExtrinsicCameraParameters2()` and `cvCalibrateCamera2()` from the Rodrigues format of 1-by-3 or 3-by-1 axis-angle vectors to rotation matrices. For this, you can leave `jacobian` set to `NULL`.

Exercises

1. Use Figure 11-2 to derive the equations $x = f_x \cdot (X/Z) + c_x$ and $y - f_y \cdot (Y/Z) + c_y$ using similar triangles with a center-position offset.
2. Will errors in estimating the true center location (c_x, c_y) affect the estimation of other parameters such as focus?

   Hint: See the $q = MQ$ equation.
3. Draw an image of a square:
   a. Under radial distortion.
   b. Under tangential distortion.
   c. Under both distortions.
4. Refer to Figure 11-13. For perspective views, explain the following.
   a. Where does the "line at infinity" come from?
   b. Why do parallel lines on the object plane converge to a point on the image plane?
   c. Assume that the object and image planes are perpendicular to one another. On the object plane, starting at a point p_1, move 10 units directly away from the image plane to p_2. What is the corresponding movement distance on the image plane?
5. Figure 11-3 shows the outward-bulging "barrel distortion" effect of radial distortion, which is especially evident in the left panel of Figure 11-12. Could some lenses generate an inward-bending effect? How would this be possible?
6. Using a cheap web camera or cell phone, create examples of radial and tangential distortion in images of concentric squares or chessboards.
7. Calibrate the camera in exercise 6. Display the pictures before and after undistortion.
8. Experiment with numerical stability and noise by collecting many images of chessboards and doing a "good" calibration on all of them. Then see how the calibration parameters change as you reduce the number of chessboard images. Graph your results: camera parameters as a function of number of chessboard images.

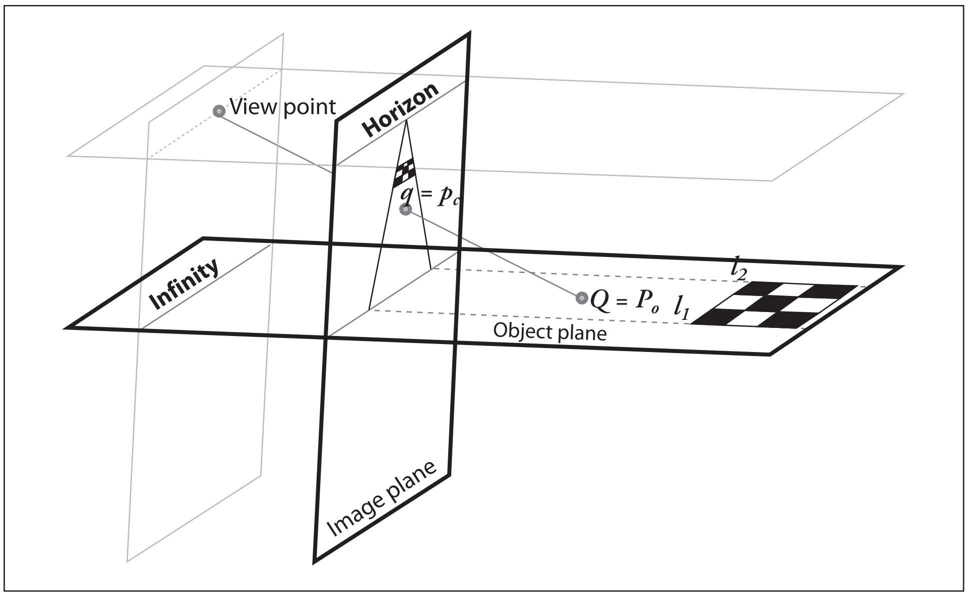


Figure 11-13. Homography diagram showing intersection of the object plane with the image plane and a viewpoint representing the center of projection

9. With reference to exercise 8, how do calibration parameters change when you use (say) 10 images of a 3-by-5, a 4-by-6, and a 5-by-7 chessboard? Graph the results.

10. High-end cameras typically have systems of lens that correct physically for distortions in the image. What might happen if you nevertheless use a multiterm distortion model for such a camera?

    Hint: This condition is known as *overfitting*.

11. *Three-dimensional joystick trick.* Calibrate a camera. Using video, wave a chessboard around and use `cvFindExtrinsicCameraParams2()` as a 3D joystick. Remember that `cvFindExtrinsicCameraParams2()` outputs rotation as a 3-by-1 or 1-by-3 vector axis of rotation, where the magnitude of the vector represents the counterclockwise angle of rotation along with a 3D translation vector.

    a. Output the chessboard's axis and angle of the rotation along with where it is (i.e., the translation) in real time as you move the chessboard around. Handle cases where the chessboard is not in view.

    b. Use `cvRodrigues2()` to translate the output of `cvFindExtrinsicCameraParams2()` into a 3-by-3 rotation matrix and a translation vector. Use this to animate a simple 3D stick figure of an airplane rendered back into the image in real time as you move the chessboard in view of the video camera.