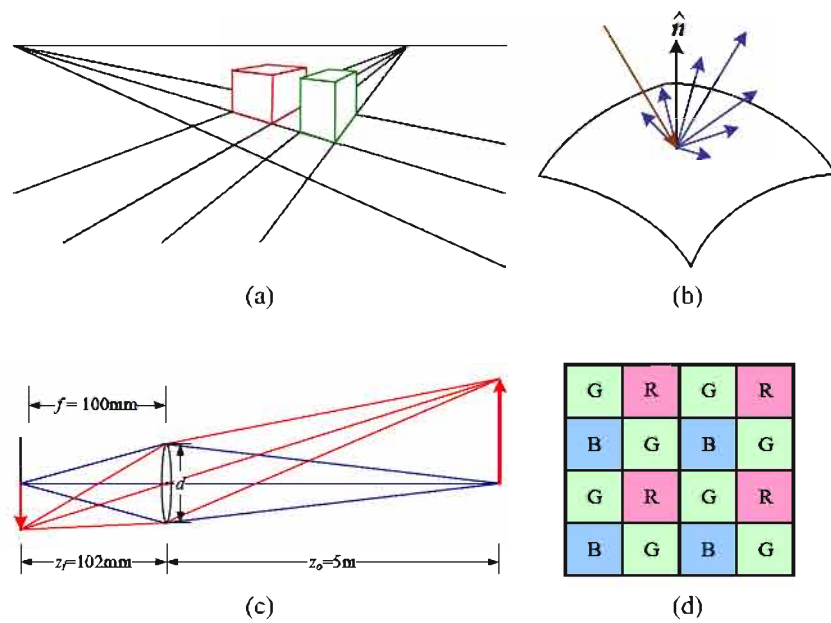


## Chapter 2

# Image formation

2.1	Geometric primitives and transformations . . . . .	31
2.1.1	Geometric primitives . . . . .	32
2.1.2	2D transformations . . . . .	35
2.1.3	3D transformations . . . . .	39
2.1.4	3D rotations . . . . .	41
2.1.5	3D to 2D projections . . . . .	46
2.1.6	Lens distortions . . . . .	58
2.2	Photometric image formation . . . . .	60
2.2.1	Lighting . . . . .	60
2.2.2	Reflectance and shading . . . . .	62
2.2.3	Optics . . . . .	68
2.3	The digital camera . . . . .	73
2.3.1	Sampling and aliasing . . . . .	77
2.3.2	Color . . . . .	80
2.3.3	Compression . . . . .	90
2.4	Additional reading . . . . .	93
2.5	Exercises . . . . .	93



**Figure 2.1** A few components of the image formation process: (a) perspective projection; (b) light scattering when hitting a surface; (c) lens optics; (d) Bayer color filter array.

Before we can intelligently analyze and manipulate images, we need to establish a vocabulary for describing the geometry of a scene. We also need to understand the image formation process that produced a particular image given a set of lighting conditions, scene geometry, surface properties, and camera optics. In this chapter, we present a simplified model of such an image formation process.

Section 2.1 introduces the basic geometric primitives used throughout the book (points, lines, and planes) and the *geometric* transformations that project these 3D quantities into 2D image features (Figure 2.1a). Section 2.2 describes how lighting, surface properties (Figure 2.1b), and camera *optics* (Figure 2.1c) interact in order to produce the color values that fall onto the image sensor. Section 2.3 describes how continuous color images are turned into discrete digital *samples* inside the image sensor (Figure 2.1d) and how to avoid (or at least characterize) sampling deficiencies, such as aliasing.

The material covered in this chapter is but a brief summary of a very rich and deep set of topics, traditionally covered in a number of separate fields. A more thorough introduction to the geometry of points, lines, planes, and projections can be found in textbooks on multi-view geometry (Hartley and Zisserman 2004; Faugeras and Luong 2001) and computer graphics (Foley, van Dam, Feiner *et al.* 1995). The image formation (synthesis) process is traditionally taught as part of a computer graphics curriculum (Foley, van Dam, Feiner *et al.* 1995; Glassner 1995; Watt 1995; Shirley 2005) but it is also studied in physics-based computer vision (Wolff, Shafer, and Healey 1992a). The behavior of camera lens systems is studied in optics (Möller 1988; Hecht 2001; Ray 2002). Two good books on color theory are (Wyszecki and Stiles 2000; Healey and Shafer 1992), with (Livingstone 2008) providing a more fun and informal introduction to the topic of color perception. Topics relating to sampling and aliasing are covered in textbooks on signal and image processing (Crane 1997; Jähne 1997; Oppenheim and Schaffer 1996; Oppenheim, Schaffer, and Buck 1999; Pratt 2007; Russ 2007; Burger and Burge 2008; Gonzales and Woods 2008).

**A note to students:** If you have already studied computer graphics, you may want to skim the material in Section 2.1, although the sections on projective depth and object-centered projection near the end of Section 2.1.5 may be new to you. Similarly, physics students (as well as computer graphics students) will mostly be familiar with Section 2.2. Finally, students with a good background in image processing will already be familiar with sampling issues (Section 2.3) as well as some of the material in Chapter 3.

## 2.1 Geometric primitives and transformations

In this section, we introduce the basic 2D and 3D primitives used in this textbook, namely points, lines, and planes. We also describe how 3D features are projected into 2D features.

More detailed descriptions of these topics (along with a gentler and more intuitive introduction) can be found in textbooks on multiple-view geometry (Hartley and Zisserman 2004; Faugeras and Luong 2001).

### 2.1.1 Geometric primitives

Geometric primitives form the basic building blocks used to describe three-dimensional shapes. In this section, we introduce points, lines, and planes. Later sections of the book discuss curves (Sections 5.1 and 11.2), surfaces (Section 12.3), and volumes (Section 12.5).

**2D points.** 2D points (pixel coordinates in an image) can be denoted using a pair of values,  $\mathbf{x} = (x, y) \in \mathcal{R}^2$ , or alternatively,

$$\mathbf{x} = \begin{bmatrix} x \\ y \end{bmatrix}. \quad (2.1)$$

(As stated in the introduction, we use the  $(x_1, x_2, \dots)$  notation to denote column vectors.)

2D points can also be represented using *homogeneous coordinates*,  $\tilde{\mathbf{x}} = (\tilde{x}, \tilde{y}, \tilde{w}) \in \mathcal{P}^2$ , where vectors that differ only by scale are considered to be equivalent.  $\mathcal{P}^2 = \mathcal{R}^3 - (0, 0, 0)$  is called the 2D *projective space*.

A homogeneous vector  $\tilde{\mathbf{x}}$  can be converted back into an *inhomogeneous* vector  $\mathbf{x}$  by dividing through by the last element  $\tilde{w}$ , i.e.,

$$\tilde{\mathbf{x}} = (\tilde{x}, \tilde{y}, \tilde{w}) = \tilde{w}(x, y, 1) = \tilde{w}\bar{\mathbf{x}}, \quad (2.2)$$

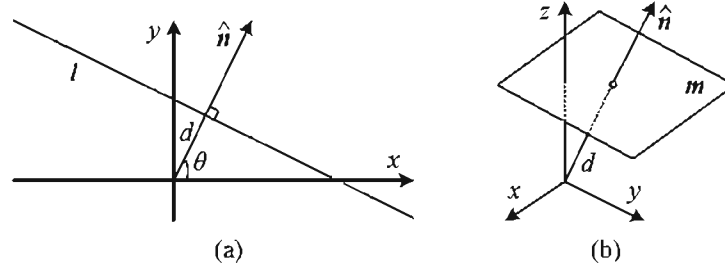
where  $\bar{\mathbf{x}} = (x, y, 1)$  is the *augmented vector*. Homogeneous points whose last element is  $\tilde{w} = 0$  are called *ideal points* or *points at infinity* and do not have an equivalent inhomogeneous representation.

**2D lines.** 2D lines can also be represented using homogeneous coordinates  $\tilde{\mathbf{l}} = (a, b, c)$ . The corresponding *line equation* is

$$\tilde{\mathbf{x}} \cdot \tilde{\mathbf{l}} = ax + by + c = 0. \quad (2.3)$$

We can normalize the line equation vector so that  $\mathbf{l} = (\hat{n}_x, \hat{n}_y, d) = (\hat{\mathbf{n}}, d)$  with  $\|\hat{\mathbf{n}}\| = 1$ . In this case,  $\hat{\mathbf{n}}$  is the *normal vector* perpendicular to the line and  $d$  is its distance to the origin (Figure 2.2). (The one exception to this normalization is the *line at infinity*  $\tilde{\mathbf{l}} = (0, 0, 1)$ , which includes all (ideal) points at infinity.)

We can also express  $\hat{\mathbf{n}}$  as a function of rotation angle  $\theta$ ,  $\hat{\mathbf{n}} = (\hat{n}_x, \hat{n}_y) = (\cos \theta, \sin \theta)$  (Figure 2.2a). This representation is commonly used in the *Hough transform* line-finding



**Figure 2.2** (a) 2D line equation and (b) 3D plane equation, expressed in terms of the normal  $\hat{n}$  and distance to the origin  $d$ .

algorithm, which is discussed in Section 4.3.2. The combination  $(\theta, d)$  is also known as *polar coordinates*.

When using homogeneous coordinates, we can compute the intersection of two lines as

$$\tilde{\mathbf{x}} = \tilde{\mathbf{l}}_1 \times \tilde{\mathbf{l}}_2, \quad (2.4)$$

where  $\times$  is the cross product operator. Similarly, the line joining two points can be written as

$$\tilde{\mathbf{l}} = \tilde{\mathbf{x}}_1 \times \tilde{\mathbf{x}}_2. \quad (2.5)$$

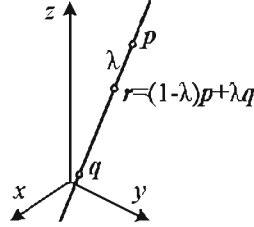
When trying to fit an intersection point to multiple lines or, conversely, a line to multiple points, least squares techniques (Section 6.1.1 and Appendix A.2) can be used, as discussed in Exercise 2.1.

**2D conics.** There are other algebraic curves that can be expressed with simple polynomial homogeneous equations. For example, the *conic sections* (so called because they arise as the intersection of a plane and a 3D cone) can be written using a *quadric* equation

$$\tilde{\mathbf{x}}^T Q \tilde{\mathbf{x}} = 0. \quad (2.6)$$

Quadric equations play useful roles in the study of multi-view geometry and camera calibration (Hartley and Zisserman 2004; Faugeras and Luong 2001) but are not used extensively in this book.

**3D points.** Point coordinates in three dimensions can be written using inhomogeneous coordinates  $\mathbf{x} = (x, y, z) \in \mathcal{R}^3$  or homogeneous coordinates  $\tilde{\mathbf{x}} = (\tilde{x}, \tilde{y}, \tilde{z}, \tilde{w}) \in \mathcal{P}^3$ . As before, it is sometimes useful to denote a 3D point using the augmented vector  $\tilde{\mathbf{x}} = (x, y, z, 1)$  with  $\tilde{x} = \tilde{w}x$ .



**Figure 2.3** 3D line equation,  $\mathbf{r} = (1 - \lambda)\mathbf{p} + \lambda\mathbf{q}$ .

**3D planes.** 3D planes can also be represented as homogeneous coordinates  $\tilde{\mathbf{m}} = (a, b, c, d)$  with a corresponding plane equation

$$\bar{\mathbf{x}} \cdot \tilde{\mathbf{m}} = ax + by + cz + d = 0. \quad (2.7)$$

We can also normalize the plane equation as  $\mathbf{m} = (\hat{n}_x, \hat{n}_y, \hat{n}_z, d) = (\hat{\mathbf{n}}, d)$  with  $\|\hat{\mathbf{n}}\| = 1$ . In this case,  $\hat{\mathbf{n}}$  is the *normal vector* perpendicular to the plane and  $d$  is its distance to the origin (Figure 2.2b). As with the case of 2D lines, the *plane at infinity*  $\tilde{\mathbf{m}} = (0, 0, 0, 1)$ , which contains all the points at infinity, cannot be normalized (i.e., it does not have a unique normal or a finite distance).

We can express  $\hat{\mathbf{n}}$  as a function of two angles  $(\theta, \phi)$ ,

$$\hat{\mathbf{n}} = (\cos \theta \cos \phi, \sin \theta \cos \phi, \sin \phi), \quad (2.8)$$

i.e., using *spherical coordinates*, but these are less commonly used than polar coordinates since they do not uniformly sample the space of possible normal vectors.

**3D lines.** Lines in 3D are less elegant than either lines in 2D or planes in 3D. One possible representation is to use two points on the line,  $(\mathbf{p}, \mathbf{q})$ . Any other point on the line can be expressed as a linear combination of these two points

$$\mathbf{r} = (1 - \lambda)\mathbf{p} + \lambda\mathbf{q}, \quad (2.9)$$

as shown in Figure 2.3. If we restrict  $0 \leq \lambda \leq 1$ , we get the *line segment* joining  $\mathbf{p}$  and  $\mathbf{q}$ .

If we use homogeneous coordinates, we can write the line as

$$\tilde{\mathbf{r}} = \mu\tilde{\mathbf{p}} + \lambda\tilde{\mathbf{q}}. \quad (2.10)$$

A special case of this is when the second point is at infinity, i.e.,  $\tilde{\mathbf{q}} = (\hat{d}_x, \hat{d}_y, \hat{d}_z, 0) = (\hat{\mathbf{d}}, 0)$ . Here, we see that  $\hat{\mathbf{d}}$  is the *direction* of the line. We can then re-write the inhomogeneous 3D line equation as

$$\mathbf{r} = \mathbf{p} + \lambda\hat{\mathbf{d}}. \quad (2.11)$$

A disadvantage of the endpoint representation for 3D lines is that it has too many degrees of freedom, i.e., six (three for each endpoint) instead of the four degrees that a 3D line truly has. However, if we fix the two points on the line to lie in specific planes, we obtain a representation with four degrees of freedom. For example, if we are representing nearly vertical lines, then  $z = 0$  and  $z = 1$  form two suitable planes, i.e., the  $(x, y)$  coordinates in both planes provide the four coordinates describing the line. This kind of two-plane parameterization is used in the *light field* and *Lumigraph* image-based rendering systems described in Chapter 13 to represent the collection of rays seen by a camera as it moves in front of an object. The two-endpoint representation is also useful for representing line segments, even when their exact endpoints cannot be seen (only guessed at).

If we wish to represent all possible lines without bias towards any particular orientation, we can use *Plücker coordinates* (Hartley and Zisserman 2004, Chapter 2; Faugeras and Luong 2001, Chapter 3). These coordinates are the six independent non-zero entries in the  $4 \times 4$  skew symmetric matrix

$$\mathbf{L} = \tilde{\mathbf{p}}\tilde{\mathbf{q}}^T - \tilde{\mathbf{q}}\tilde{\mathbf{p}}^T, \quad (2.12)$$

where  $\tilde{\mathbf{p}}$  and  $\tilde{\mathbf{q}}$  are any two (non-identical) points on the line. This representation has only four degrees of freedom, since  $\mathbf{L}$  is homogeneous and also satisfies  $\det(\mathbf{L}) = 0$ , which results in a quadratic constraint on the Plücker coordinates.

In practice, the minimal representation is not essential for most applications. An adequate model of 3D lines can be obtained by estimating their direction (which may be known ahead of time, e.g., for architecture) and some point within the visible portion of the line (see Section 7.5.1) or by using the two endpoints, since lines are most often visible as finite line segments. However, if you are interested in more details about the topic of minimal line parameterizations, Förstner (2005) discusses various ways to infer and model 3D lines in projective geometry, as well as how to estimate the uncertainty in such fitted models.

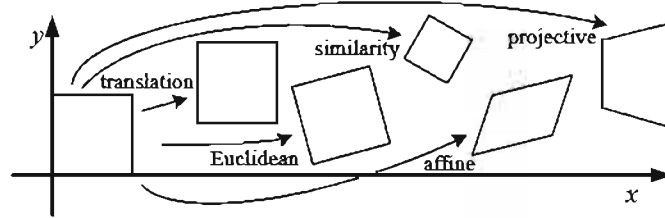
**3D quadrics.** The 3D analog of a conic section is a quadric surface

$$\bar{\mathbf{x}}^T \mathbf{Q} \bar{\mathbf{x}} = 0 \quad (2.13)$$

(Hartley and Zisserman 2004, Chapter 2). Again, while quadric surfaces are useful in the study of multi-view geometry and can also serve as useful modeling primitives (spheres, ellipsoids, cylinders), we do not study them in great detail in this book.

### 2.1.2 2D transformations

Having defined our basic primitives, we can now turn our attention to how they can be transformed. The simplest transformations occur in the 2D plane and are illustrated in Figure 2.4.



**Figure 2.4** Basic set of 2D planar transformations.

**Translation.** 2D translations can be written as  $\mathbf{x}' = \mathbf{x} + \mathbf{t}$  or

$$\mathbf{x}' = \begin{bmatrix} \mathbf{I} & \mathbf{t} \end{bmatrix} \bar{\mathbf{x}} \quad (2.14)$$

where  $\mathbf{I}$  is the  $(2 \times 2)$  identity matrix or

$$\bar{\mathbf{x}}' = \begin{bmatrix} \mathbf{I} & \mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix} \bar{\mathbf{x}} \quad (2.15)$$

where  $\mathbf{0}$  is the zero vector. Using a  $2 \times 3$  matrix results in a more compact notation, whereas using a full-rank  $3 \times 3$  matrix (which can be obtained from the  $2 \times 3$  matrix by appending a  $[\mathbf{0}^T \ 1]$  row) makes it possible to chain transformations using matrix multiplication. Note that in any equation where an augmented vector such as  $\bar{\mathbf{x}}$  appears on both sides, it can always be replaced with a full homogeneous vector  $\tilde{\mathbf{x}}$ .

**Rotation + translation.** This transformation is also known as *2D rigid body motion* or the *2D Euclidean transformation* (since Euclidean distances are preserved). It can be written as  $\mathbf{x}' = \mathbf{R}\mathbf{x} + \mathbf{t}$  or

$$\mathbf{x}' = \begin{bmatrix} \mathbf{R} & \mathbf{t} \end{bmatrix} \bar{\mathbf{x}} \quad (2.16)$$

where

$$\mathbf{R} = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \quad (2.17)$$

is an orthonormal rotation matrix with  $\mathbf{R}\mathbf{R}^T = \mathbf{I}$  and  $|\mathbf{R}| = 1$ .

**Scaled rotation.** Also known as the *similarity transform*, this transformation can be expressed as  $\mathbf{x}' = s\mathbf{R}\mathbf{x} + \mathbf{t}$  where  $s$  is an arbitrary scale factor. It can also be written as

$$\mathbf{x}' = \begin{bmatrix} s\mathbf{R} & \mathbf{t} \end{bmatrix} \bar{\mathbf{x}} = \begin{bmatrix} a & -b & t_x \\ b & a & t_y \end{bmatrix} \bar{\mathbf{x}}, \quad (2.18)$$

where we no longer require that  $a^2 + b^2 = 1$ . The similarity transform preserves angles between lines.



**Affine.** The affine transformation is written as  $\mathbf{x}' = \mathbf{A}\bar{\mathbf{x}}$ , where  $\mathbf{A}$  is an arbitrary  $2 \times 3$  matrix, i.e.,

$$\mathbf{x}' = \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \end{bmatrix} \bar{\mathbf{x}}. \quad (2.19)$$

Parallel lines remain parallel under affine transformations.

**Projective.** This transformation, also known as a *perspective transform* or *homography*, operates on homogeneous coordinates,

$$\tilde{\mathbf{x}}' = \tilde{\mathbf{H}}\tilde{\mathbf{x}}, \quad (2.20)$$

where  $\tilde{\mathbf{H}}$  is an arbitrary  $3 \times 3$  matrix. Note that  $\tilde{\mathbf{H}}$  is homogeneous, i.e., it is only defined up to a scale, and that two  $\tilde{\mathbf{H}}$  matrices that differ only by scale are equivalent. The resulting homogeneous coordinate  $\tilde{\mathbf{x}}'$  must be normalized in order to obtain an inhomogeneous result  $\mathbf{x}$ , i.e.,

$$x' = \frac{h_{00}x + h_{01}y + h_{02}}{h_{20}x + h_{21}y + h_{22}} \quad \text{and} \quad y' = \frac{h_{10}x + h_{11}y + h_{12}}{h_{20}x + h_{21}y + h_{22}}. \quad (2.21)$$

Perspective transformations preserve straight lines (i.e., they remain straight after the transformation).






**Hierarchy of 2D transformations.** The preceding set of transformations are illustrated in Figure 2.4 and summarized in Table 2.1. The easiest way to think of them is as a set of (potentially restricted)  $3 \times 3$  matrices operating on 2D homogeneous coordinate vectors. Hartley and Zisserman (2004) contains a more detailed description of the hierarchy of 2D planar transformations.

The above transformations form a nested set of *groups*, i.e., they are closed under composition and have an inverse that is a member of the same group. (This will be important later when applying these transformations to images in Section 3.6.) Each (simpler) group is a subset of the more complex group below it.

**Co-vectors.** While the above transformations can be used to transform points in a 2D plane, can they also be used directly to transform a line equation? Consider the homogeneous equation  $\tilde{\mathbf{l}} \cdot \tilde{\mathbf{x}} = 0$ . If we transform  $\mathbf{x}' = \tilde{\mathbf{H}}\mathbf{x}$ , we obtain

$$\tilde{\mathbf{l}} \cdot \tilde{\mathbf{x}}' = \tilde{\mathbf{l}}^T \tilde{\mathbf{H}}\tilde{\mathbf{x}} = (\tilde{\mathbf{H}}^T \tilde{\mathbf{l}})^T \tilde{\mathbf{x}} = \tilde{\mathbf{l}} \cdot \tilde{\mathbf{x}} = 0, \quad (2.22)$$

i.e.,  $\tilde{\mathbf{l}} = \tilde{\mathbf{H}}^{-T} \tilde{\mathbf{l}}$ . Thus, the action of a projective transformation on a *co-vector* such as a 2D line or 3D normal can be represented by the transposed inverse of the matrix, which is equivalent to the *adjoint* of  $\tilde{\mathbf{H}}$ , since projective transformation matrices are homogeneous. Jim

Transformation	Matrix	# DoF	Preserves	Icon
translation	$\begin{bmatrix} \mathbf{I} & \mathbf{t} \end{bmatrix}_{2 \times 3}$	2	orientation	
rigid (Euclidean)	$\begin{bmatrix} \mathbf{R} & \mathbf{t} \end{bmatrix}_{2 \times 3}$	3	lengths	
similarity	$\begin{bmatrix} s\mathbf{R} & \mathbf{t} \end{bmatrix}_{2 \times 3}$	4	angles	
affine	$\begin{bmatrix} \mathbf{A} \end{bmatrix}_{2 \times 3}$	6	parallelism	
projective	$\begin{bmatrix} \tilde{\mathbf{H}} \end{bmatrix}_{3 \times 3}$	8	straight lines	

**Table 2.1** Hierarchy of 2D coordinate transformations. Each transformation also preserves the properties listed in the rows below it, i.e.. similarity preserves not only angles but also parallelism and straight lines. The  $2 \times 3$  matrices are extended with a third  $[\mathbf{0}^T \ 1]$  row to form a full  $3 \times 3$  matrix for homogeneous coordinate transformations.

Blinn (1998) describes (in Chapters 9 and 10) the ins and outs of notating and manipulating co-vectors.

While the above transformations are the ones we use most extensively, a number of additional transformations are sometimes used.

**Stretch/squash.** This transformation changes the aspect ratio of an image,






$$\begin{aligned} x' &= s_x x + t_x \\ y' &= s_y y + t_y, \end{aligned}$$

and is a restricted form of an affine transformation. Unfortunately, it does not nest cleanly with the groups listed in Table 2.1.

**Planar surface flow.** This eight-parameter transformation (Horn 1986; Bergen, Anandan, Hanna *et al.* 1992; Girod, Greiner, and Niemann 2000),

$$\begin{aligned} x' &= a_0 + a_1 x + a_2 y + a_6 x^2 + a_7 xy \\ y' &= a_3 + a_4 x + a_5 y + a_7 x^2 + a_6 xy, \end{aligned}$$

arises when a planar surface undergoes a small 3D motion. It can thus be thought of as a small motion approximation to a full homography. Its main attraction is that it is *linear* in the motion parameters,  $a_k$ , which are often the quantities being estimated.

Transformation	Matrix	# DoF	Preserves	Icon
translation	$\begin{bmatrix} \mathbf{I} & \mathbf{t} \end{bmatrix}_{3 \times 4}$	3	orientation	
rigid (Euclidean)	$\begin{bmatrix} \mathbf{R} & \mathbf{t} \end{bmatrix}_{3 \times 4}$	6	lengths	
similarity	$\begin{bmatrix} s\mathbf{R} & \mathbf{t} \end{bmatrix}_{3 \times 4}$	7	angles	
affine	$\begin{bmatrix} \mathbf{A} \end{bmatrix}_{3 \times 4}$	12	parallelism	
projective	$\begin{bmatrix} \tilde{\mathbf{H}} \end{bmatrix}_{4 \times 4}$	15	straight lines	

**Table 2.2** Hierarchy of 3D coordinate transformations. Each transformation also preserves the properties listed in the rows below it, i.e., similarity preserves not only angles but also parallelism and straight lines. The  $3 \times 4$  matrices are extended with a fourth  $[\mathbf{0}^T \ 1]$  row to form a full  $4 \times 4$  matrix for homogeneous coordinate transformations. The mnemonic icons are drawn in 2D but are meant to suggest transformations occurring in a full 3D cube.

**Bilinear interpolant.** This eight-parameter transform (Wolberg 1990),

$$\begin{aligned} x' &= a_0 + a_1x + a_2y + a_6xy \\ y' &= a_3 + a_4x + a_5y + a_7xy, \end{aligned}$$

can be used to interpolate the deformation due to the motion of the four corner points of a square. (In fact, it can interpolate the motion of any four non-collinear points.) While the deformation is linear in the motion parameters, it does not generally preserve straight lines (only lines parallel to the square axes). However, it is often quite useful, e.g., in the interpolation of sparse grids using splines (Section 8.3).

### 2.1.3 3D transformations

The set of three-dimensional coordinate transformations is very similar to that available for 2D transformations and is summarized in Table 2.2. As in 2D, these transformations form a nested set of groups. Hartley and Zisserman (2004, Section 2.4) give a more detailed description of this hierarchy.

**Translation.** 3D translations can be written as  $\mathbf{x}' = \mathbf{x} + \mathbf{t}$  or

$$\mathbf{x}' = \begin{bmatrix} \mathbf{I} & \mathbf{t} \end{bmatrix} \bar{\mathbf{x}} \quad (2.23)$$

where  $I$  is the  $(3 \times 3)$  identity matrix and  $\mathbf{0}$  is the zero vector.

**Rotation + translation.** Also known as 3D *rigid body motion* or the 3D *Euclidean transformation*, it can be written as  $\mathbf{x}' = \mathbf{R}\mathbf{x} + \mathbf{t}$  or

$$\mathbf{x}' = \begin{bmatrix} \mathbf{R} & \mathbf{t} \end{bmatrix} \bar{\mathbf{x}} \quad (2.24)$$

where  $\mathbf{R}$  is a  $3 \times 3$  orthonormal rotation matrix with  $\mathbf{R}\mathbf{R}^T = I$  and  $|\mathbf{R}| = 1$ . Note that sometimes it is more convenient to describe a rigid motion using

$$\mathbf{x}' = \mathbf{R}(\mathbf{x} - \mathbf{c}) = \mathbf{R}\mathbf{x} - \mathbf{R}\mathbf{c}, \quad (2.25)$$

where  $\mathbf{c}$  is the center of rotation (often the camera center).

Compactly parameterizing a 3D rotation is a non-trivial task, which we describe in more detail below.

**Scaled rotation.** The 3D *similarity transform* can be expressed as  $\mathbf{x}' = s\mathbf{R}\mathbf{x} + \mathbf{t}$  where  $s$  is an arbitrary scale factor. It can also be written as

$$\mathbf{x}' = \begin{bmatrix} s\mathbf{R} & \mathbf{t} \end{bmatrix} \bar{\mathbf{x}}. \quad (2.26)$$

This transformation preserves angles between lines and planes.

**Affine.** The affine transform is written as  $\mathbf{x}' = \mathbf{A}\bar{\mathbf{x}}$ , where  $\mathbf{A}$  is an arbitrary  $3 \times 4$  matrix, i.e.,

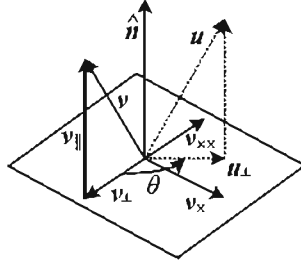
$$\mathbf{x}' = \begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \end{bmatrix} \bar{\mathbf{x}}. \quad (2.27)$$

Parallel lines and planes remain parallel under affine transformations.

**Projective.** This transformation, variously known as a 3D *perspective transform*, *homography*, or *collineation*, operates on homogeneous coordinates,

$$\tilde{\mathbf{x}}' = \tilde{\mathbf{H}}\tilde{\mathbf{x}}, \quad (2.28)$$

where  $\tilde{\mathbf{H}}$  is an arbitrary  $4 \times 4$  homogeneous matrix. As in 2D, the resulting homogeneous coordinate  $\tilde{\mathbf{x}}'$  must be normalized in order to obtain an inhomogeneous result  $\mathbf{x}$ . Perspective transformations preserve straight lines (i.e., they remain straight after the transformation).



**Figure 2.5** Rotation around an axis  $\hat{n}$  by an angle  $\theta$ .

### 2.1.4 3D rotations

The biggest difference between 2D and 3D coordinate transformations is that the parameterization of the 3D rotation matrix  $\mathbf{R}$  is not as straightforward but several possibilities exist.

#### Euler angles

A rotation matrix can be formed as the product of three rotations around three cardinal axes, e.g.,  $x$ ,  $y$ , and  $z$ , or  $x$ ,  $y$ , and  $x$ . This is generally a bad idea, as the result depends on the order in which the transforms are applied. What is worse, it is not always possible to move smoothly in the parameter space, i.e., sometimes one or more of the Euler angles change dramatically in response to a small change in rotation.<sup>1</sup> For these reasons, we do not even give the formula for Euler angles in this book—interested readers can look in other textbooks or technical reports (Faugeras 1993; Diebel 2006). Note that, in some applications, if the rotations are known to be a set of uni-axial transforms, they can always be represented using an explicit set of rigid transformations.

#### Axis/angle (exponential twist)

A rotation can be represented by a rotation axis  $\hat{n}$  and an angle  $\theta$ , or equivalently by a 3D vector  $\omega = \theta\hat{n}$ . Figure 2.5 shows how we can compute the equivalent rotation. First, we project the vector  $v$  onto the axis  $\hat{n}$  to obtain

$$v_{\parallel} = \hat{n}(\hat{n} \cdot v) = (\hat{n}\hat{n}^T)v, \quad (2.29)$$

which is the component of  $v$  that is not affected by the rotation. Next, we compute the perpendicular residual of  $v$  from  $\hat{n}$ ,

$$v_{\perp} = v - v_{\parallel} = (I - \hat{n}\hat{n}^T)v. \quad (2.30)$$

<sup>1</sup> In robotics, this is sometimes referred to as *gimbal lock*.

We can rotate this vector by  $90^\circ$  using the cross product,

$$\mathbf{v}_\times = \hat{\mathbf{n}} \times \mathbf{v} = [\hat{\mathbf{n}}]_\times \mathbf{v}, \quad (2.31)$$

where  $[\hat{\mathbf{n}}]_\times$  is the matrix form of the cross product operator with the vector  $\hat{\mathbf{n}} = (\hat{n}_x, \hat{n}_y, \hat{n}_z)$ ,

$$[\hat{\mathbf{n}}]_\times = \begin{bmatrix} 0 & -\hat{n}_z & \hat{n}_y \\ \hat{n}_z & 0 & -\hat{n}_x \\ -\hat{n}_y & \hat{n}_x & 0 \end{bmatrix}. \quad (2.32)$$

Note that rotating this vector by another  $90^\circ$  is equivalent to taking the cross product again,

$$\mathbf{v}_{\times\times} = \hat{\mathbf{n}} \times \mathbf{v}_\times = [\hat{\mathbf{n}}]_\times^2 \mathbf{v} = -\mathbf{v}_\perp,$$

and hence

$$\mathbf{v}_\parallel = \mathbf{v} - \mathbf{v}_\perp = \mathbf{v} + \mathbf{v}_{\times\times} = (\mathbf{I} + [\hat{\mathbf{n}}]_\times^2) \mathbf{v}.$$

We can now compute the in-plane component of the rotated vector  $\mathbf{u}$  as

$$\mathbf{u}_\perp = \cos \theta \mathbf{v}_\perp + \sin \theta \mathbf{v}_\times = (\sin \theta [\hat{\mathbf{n}}]_\times - \cos \theta [\hat{\mathbf{n}}]_\times^2) \mathbf{v}.$$

Putting all these terms together, we obtain the final rotated vector as

$$\mathbf{u} = \mathbf{u}_\perp + \mathbf{u}_\parallel = (\mathbf{I} + \sin \theta [\hat{\mathbf{n}}]_\times + (1 - \cos \theta) [\hat{\mathbf{n}}]_\times^2) \mathbf{v}. \quad (2.33)$$

We can therefore write the rotation matrix corresponding to a rotation by  $\theta$  around an axis  $\hat{\mathbf{n}}$  as

$$\mathbf{R}(\hat{\mathbf{n}}, \theta) = \mathbf{I} + \sin \theta [\hat{\mathbf{n}}]_\times + (1 - \cos \theta) [\hat{\mathbf{n}}]_\times^2, \quad (2.34)$$

which is known as *Rodriguez's formula* (Ayache 1989).

The product of the axis  $\hat{\mathbf{n}}$  and angle  $\theta$ ,  $\boldsymbol{\omega} = \theta \hat{\mathbf{n}} = (\omega_x, \omega_y, \omega_z)$ , is a minimal representation for a 3D rotation. Rotations through common angles such as multiples of  $90^\circ$  can be represented exactly (and converted to exact matrices) if  $\theta$  is stored in degrees. Unfortunately, this representation is not unique, since we can always add a multiple of  $360^\circ$  ( $2\pi$  radians) to  $\theta$  and get the same rotation matrix. As well,  $(\hat{\mathbf{n}}, \theta)$  and  $(-\hat{\mathbf{n}}, -\theta)$  represent the same rotation.

However, for small rotations (e.g., corrections to rotations), this is an excellent choice. In particular, for small (infinitesimal or instantaneous) rotations and  $\theta$  expressed in radians, Rodriguez's formula simplifies to

$$\mathbf{R}(\boldsymbol{\omega}) \approx \mathbf{I} + \sin \theta [\hat{\mathbf{n}}]_\times \approx \mathbf{I} + [\theta \hat{\mathbf{n}}]_\times = \begin{bmatrix} 1 & -\omega_z & \omega_y \\ \omega_z & 1 & -\omega_x \\ -\omega_y & \omega_x & 1 \end{bmatrix}, \quad (2.35)$$

which gives a nice linearized relationship between the rotation parameters  $\boldsymbol{\omega}$  and  $\mathbf{R}$ . We can also write  $\mathbf{R}(\boldsymbol{\omega})\mathbf{v} \approx \mathbf{v} + \boldsymbol{\omega} \times \mathbf{v}$ , which is handy when we want to compute the derivative of  $\mathbf{R}\mathbf{v}$  with respect to  $\boldsymbol{\omega}$ ,

$$\frac{\partial \mathbf{R}\mathbf{v}}{\partial \boldsymbol{\omega}^T} = -[\mathbf{v}]_{\times} = \begin{bmatrix} 0 & z & -y \\ -z & 0 & x \\ y & -x & 0 \end{bmatrix}. \quad (2.36)$$

Another way to derive a rotation through a finite angle is called the *exponential twist* (Murray, Li, and Sastry 1994). A rotation by an angle  $\theta$  is equivalent to  $k$  rotations through  $\theta/k$ . In the limit as  $k \rightarrow \infty$ , we obtain

$$\mathbf{R}(\hat{\mathbf{n}}, \theta) = \lim_{k \rightarrow \infty} \left( \mathbf{I} + \frac{1}{k} [\theta \hat{\mathbf{n}}]_{\times} \right)^k = \exp[\boldsymbol{\omega}]_{\times}. \quad (2.37)$$

If we expand the matrix exponential as a Taylor series (using the identity  $[\hat{\mathbf{n}}]_{\times}^{k+2} = -[\hat{\mathbf{n}}]_{\times}^k$ ,  $k > 0$ , and again assuming  $\theta$  is in radians),

$$\begin{aligned} \exp[\boldsymbol{\omega}]_{\times} &= \mathbf{I} + \theta [\hat{\mathbf{n}}]_{\times} + \frac{\theta^2}{2} [\hat{\mathbf{n}}]_{\times}^2 + \frac{\theta^3}{3!} [\hat{\mathbf{n}}]_{\times}^3 + \cdots \\ &= \mathbf{I} + \left( \theta - \frac{\theta^3}{3!} + \cdots \right) [\hat{\mathbf{n}}]_{\times} + \left( \frac{\theta^2}{2} - \frac{\theta^4}{4!} + \cdots \right) [\hat{\mathbf{n}}]_{\times}^2 \\ &= \mathbf{I} + \sin \theta [\hat{\mathbf{n}}]_{\times} + (1 - \cos \theta) [\hat{\mathbf{n}}]_{\times}^2, \end{aligned} \quad (2.38)$$

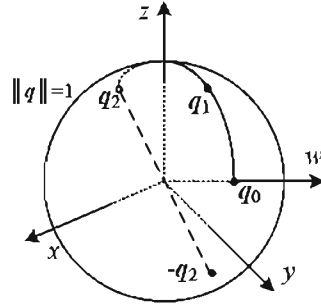
which yields the familiar Rodriguez's formula.

### Unit quaternions

The unit quaternion representation is closely related to the angle/axis representation. A unit quaternion is a unit length 4-vector whose components can be written as  $\mathbf{q} = (q_x, q_y, q_z, q_w)$  or  $\mathbf{q} = (x, y, z, w)$  for short. Unit quaternions live on the unit sphere  $\|\mathbf{q}\| = 1$  and *antipodal* (opposite sign) quaternions,  $\mathbf{q}$  and  $-\mathbf{q}$ , represent the same rotation (Figure 2.6). Other than this ambiguity (dual covering), the unit quaternion representation of a rotation is unique. Furthermore, the representation is *continuous*, i.e., as rotation matrices vary continuously, one can find a continuous quaternion representation, although the path on the quaternion sphere may wrap all the way around before returning to the "origin"  $\mathbf{q}_o = (0, 0, 0, 1)$ . For these and other reasons given below, quaternions are a very popular representation for pose and for pose interpolation in computer graphics (Shoemake 1985).

Quaternions can be derived from the axis/angle representation through the formula

$$\mathbf{q} = (\mathbf{v}, w) = \left( \sin \frac{\theta}{2} \hat{\mathbf{n}}, \cos \frac{\theta}{2} \right), \quad (2.39)$$



**Figure 2.6** Unit quaternions live on the unit sphere  $\|q\| = 1$ . This figure shows a smooth trajectory through the three quaternions  $q_0$ ,  $q_1$ , and  $q_2$ . The *antipodal* point to  $q_2$ , namely  $-q_2$ , represents the same rotation as  $q_2$ .

where  $\hat{n}$  and  $\theta$  are the rotation axis and angle. Using the trigonometric identities  $\sin \theta = 2 \sin \frac{\theta}{2} \cos \frac{\theta}{2}$  and  $(1 - \cos \theta) = 2 \sin^2 \frac{\theta}{2}$ , Rodriguez's formula can be converted to

$$\begin{aligned} \mathbf{R}(\hat{n}, \theta) &= \mathbf{I} + \sin \theta [\hat{n}]_{\times} + (1 - \cos \theta) [\hat{n}]_{\times}^2 \\ &= \mathbf{I} + 2w[\mathbf{v}]_{\times} + 2[\mathbf{v}]_{\times}^2. \end{aligned} \quad (2.40)$$

This suggests a quick way to rotate a vector  $\mathbf{v}$  by a quaternion using a series of cross products, scalings, and additions. To obtain a formula for  $\mathbf{R}(q)$  as a function of  $(x, y, z, w)$ , recall that

$$[\mathbf{v}]_{\times} = \begin{bmatrix} 0 & -z & y \\ z & 0 & -x \\ -y & x & 0 \end{bmatrix} \quad \text{and} \quad [\mathbf{v}]_{\times}^2 = \begin{bmatrix} -y^2 - z^2 & xy & xz \\ xy & -x^2 - z^2 & yz \\ xz & yz & -x^2 - y^2 \end{bmatrix}.$$

We thus obtain

$$\mathbf{R}(q) = \begin{bmatrix} 1 - 2(y^2 + z^2) & 2(xy - zw) & 2(xz + yw) \\ 2(xy + zw) & 1 - 2(x^2 + z^2) & 2(yz - xw) \\ 2(xz - yw) & 2(yz + xw) & 1 - 2(x^2 + y^2) \end{bmatrix}. \quad (2.41)$$

The diagonal terms can be made more symmetrical by replacing  $1 - 2(y^2 + z^2)$  with  $(x^2 + w^2 - y^2 - z^2)$ , etc.

The nicest aspect of unit quaternions is that there is a simple algebra for composing rotations expressed as unit quaternions. Given two quaternions  $q_0 = (v_0, w_0)$  and  $q_1 = (v_1, w_1)$ , the *quaternion multiply* operator is defined as

$$q_2 = q_0 q_1 = (v_0 \times v_1 + w_0 v_1 + w_1 v_0, w_0 w_1 - v_0 \cdot v_1), \quad (2.42)$$



with the property that  $\mathbf{R}(q_2) = \mathbf{R}(q_0)\mathbf{R}(q_1)$ . Note that quaternion multiplication is *not* commutative, just as 3D rotations and matrix multiplications are not.

Taking the inverse of a quaternion is easy: Just flip the sign of  $\mathbf{v}$  or  $\mathbf{w}$  (but not both!). (You can verify this has the desired effect of transposing the  $\mathbf{R}$  matrix in (2.41).) Thus, we can also define *quaternion division* as

$$q_2 = q_0/q_1 = q_0q_1^{-1} = (\mathbf{v}_0 \times \mathbf{v}_1 + w_0\mathbf{v}_1 - w_1\mathbf{v}_0, -w_0w_1 - \mathbf{v}_0 \cdot \mathbf{v}_1). \quad (2.43)$$

This is useful when the *incremental rotation* between two rotations is desired.

In particular, if we want to determine a rotation that is partway between two given rotations, we can compute the incremental rotation, take a fraction of the angle, and compute the new rotation. This procedure is called *spherical linear interpolation* or *slerp* for short (Shoemake 1985) and is given in Algorithm 2.1. Note that Shoemake presents two formulas other than the one given here. The first exponentiates  $q_r$  by alpha before multiplying the original quaternion,

$$q_2 = q_r^\alpha q_0, \quad (2.44)$$

while the second treats the quaternions as 4-vectors on a sphere and uses

$$q_2 = \frac{\sin(1-\alpha)\theta}{\sin\theta} q_0 + \frac{\sin\alpha\theta}{\sin\theta} q_1, \quad (2.45)$$

where  $\theta = \cos^{-1}(q_0 \cdot q_1)$  and the dot product is directly between the quaternion 4-vectors. All of these formulas give comparable results, although care should be taken when  $q_0$  and  $q_1$  are close together, which is why I prefer to use an arctangent to establish the rotation angle.

### Which rotation representation is better?

The choice of representation for 3D rotations depends partly on the application.

The axis/angle representation is minimal, and hence does not require any additional constraints on the parameters (no need to re-normalize after each update). If the angle is expressed in degrees, it is easier to understand the pose (say, 90° twist around  $x$ -axis), and also easier to express exact rotations. When the angle is in radians, the derivatives of  $\mathbf{R}$  with respect to  $\omega$  can easily be computed (2.36).

Quaternions, on the other hand, are better if you want to keep track of a smoothly moving camera, since there are no discontinuities in the representation. It is also easier to interpolate between rotations and to chain rigid transformations (Murray, Li, and Sastry 1994; Bregler and Malik 1998).

My usual preference is to use quaternions, but to update their estimates using an incremental rotation, as described in Section 6.2.2.

**procedure** *slerp*( $q_0, q_1, \alpha$ ):

1.  $q_r = q_1/q_0 = (v_r, w_r)$
2. if  $w_r < 0$  then  $q_r \leftarrow -q_r$
3.  $\theta_r = 2 \tan^{-1}(\|v_r\|/w_r)$
4.  $\tilde{n}_r = \mathcal{N}(v_r) = v_r/\|v_r\|$
5.  $\theta_\alpha = \alpha \theta_r$
6.  $q_\alpha = (\sin \frac{\theta_\alpha}{2} \tilde{n}_r, \cos \frac{\theta_\alpha}{2})$
7. **return**  $q_2 = q_\alpha q_0$

**Algorithm 2.1** Spherical linear interpolation (slerp). The axis and total angle are first computed from the quaternion ratio. (This computation can be lifted outside an inner loop that generates a set of interpolated position for animation.) An incremental quaternion is then computed and multiplied by the starting rotation quaternion.

### 2.1.5 3D to 2D projections

Now that we know how to represent 2D and 3D geometric primitives and how to transform them spatially, we need to specify how 3D primitives are projected onto the image plane. We can do this using a linear 3D to 2D projection matrix. The simplest model is orthography, which requires no division to get the final (inhomogeneous) result. The more commonly used model is perspective, since this more accurately models the behavior of real cameras.

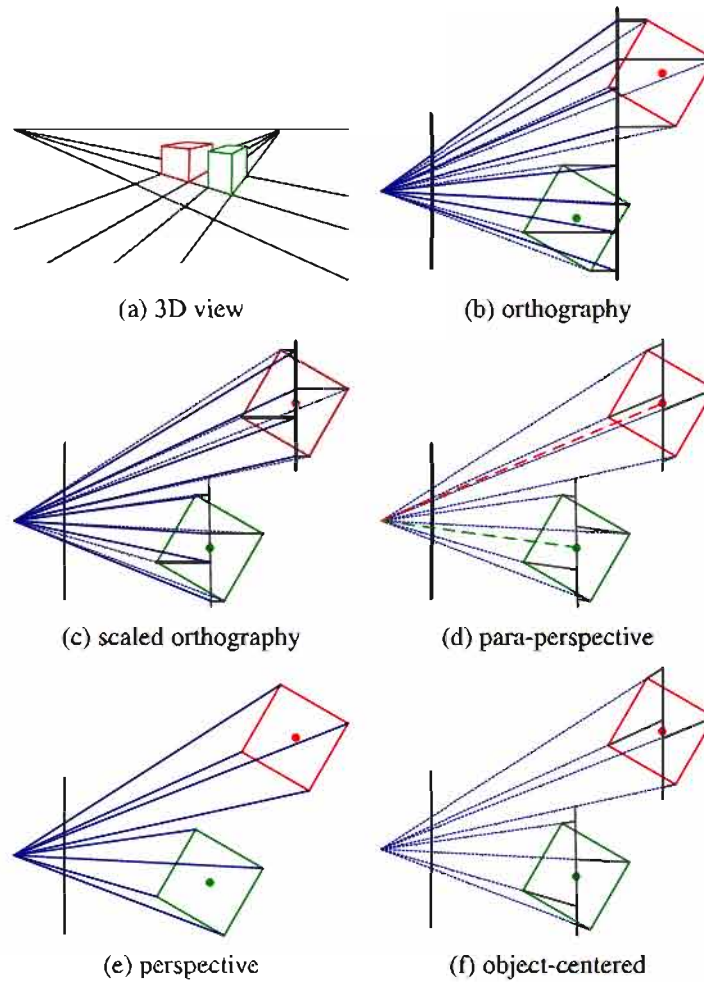
#### Orthography and para-perspective

An orthographic projection simply drops the  $z$  component of the three-dimensional coordinate  $p$  to obtain the 2D point  $x$ . (In this section, we use  $p$  to denote 3D points and  $x$  to denote 2D points.) This can be written as

$$x = [I_{2 \times 2} | 0] p. \quad (2.46)$$

If we are using homogeneous (projective) coordinates, we can write

$$\tilde{x} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tilde{p}, \quad (2.47)$$



**Figure 2.7** Commonly used projection models: (a) 3D view of world, (b) orthography, (c) scaled orthography, (d) para-perspective, (e) perspective, (f) object-centered. Each diagram shows a top-down view of the projection. Note how parallel lines on the ground plane and box sides remain parallel in the non-perspective projections.

i.e., we drop the  $z$  component but keep the  $w$  component. Orthography is an approximate model for long focal length (telephoto) lenses and objects whose depth is *shallow* relative to their distance to the camera (Sawhney and Hanson 1991). It is exact only for *telecentric* lenses (Baker and Nayar 1999, 2001).

In practice, world coordinates (which may measure dimensions in meters) need to be scaled to fit onto an image sensor (physically measured in millimeters, but ultimately measured in pixels). For this reason, *scaled orthography* is actually more commonly used,

$$\mathbf{x} = [s\mathbf{I}_{2 \times 2} | \mathbf{0}] \mathbf{p}. \quad (2.48)$$

This model is equivalent to first projecting the world points onto a local fronto-parallel image plane and then scaling this image using regular perspective projection. The scaling can be the same for all parts of the scene (Figure 2.7b) or it can be different for objects that are being modeled independently (Figure 2.7c). More importantly, the scaling can vary from frame to frame when estimating *structure from motion*, which can better model the scale change that occurs as an object approaches the camera.

Scaled orthography is a popular model for reconstructing the 3D shape of objects far away from the camera, since it greatly simplifies certain computations. For example, *pose* (camera orientation) can be estimated using simple least squares (Section 6.2.1). Under orthography, structure and motion can simultaneously be estimated using *factorization* (singular value decomposition), as discussed in Section 7.3 (Tomasi and Kanade 1992).

A closely related projection model is *para-perspective* (Aloimonos 1990; Poelman and Kanade 1997). In this model, object points are again first projected onto a local reference plane parallel to the image plane. However, rather than being projected orthogonally to this plane, they are projected *parallel* to the line of sight to the object center (Figure 2.7d). This is followed by the usual projection onto the final image plane, which again amounts to a scaling. The combination of these two projections is therefore *affine* and can be written as

$$\tilde{\mathbf{x}} = \begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ 0 & 0 & 0 & 1 \end{bmatrix} \tilde{\mathbf{p}}. \quad (2.49)$$

Note how parallel lines in 3D remain parallel after projection in Figure 2.7b–d. Para-perspective provides a more accurate projection model than scaled orthography, without incurring the added complexity of per-pixel perspective division, which invalidates traditional factorization methods (Poelman and Kanade 1997).

## Perspective

The most commonly used projection in computer graphics and computer vision is true 3D *perspective* (Figure 2.7e). Here, points are projected onto the image plane by dividing them

by their  $z$  component. Using inhomogeneous coordinates, this can be written as

$$\bar{\mathbf{x}} = \mathcal{P}_z(\mathbf{p}) = \begin{bmatrix} x/z \\ y/z \\ 1 \end{bmatrix}. \quad (2.50)$$

In homogeneous coordinates, the projection has a simple linear form,

$$\tilde{\mathbf{x}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tilde{\mathbf{p}}, \quad (2.51)$$

i.e., we drop the  $w$  component of  $\mathbf{p}$ . Thus, after projection, it is not possible to recover the *distance* of the 3D point from the image, which makes sense for a 2D imaging sensor.

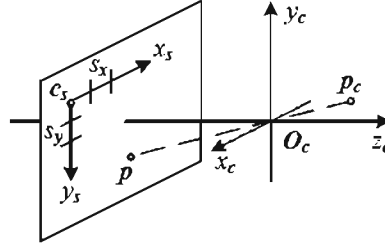
A form often seen in computer graphics systems is a two-step projection that first projects 3D coordinates into *normalized device coordinates* in the range  $(x, y, z) \in [-1, -1] \times [-1, 1] \times [0, 1]$ , and then rescales these coordinates to integer pixel coordinates using a *viewport* transformation (Watt 1995; OpenGL-ARB 1997). The (initial) perspective projection is then represented using a  $4 \times 4$  matrix

$$\tilde{\mathbf{x}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -z_{\text{far}}/z_{\text{range}} & z_{\text{near}}z_{\text{far}}/z_{\text{range}} \\ 0 & 0 & 1 & 0 \end{bmatrix} \tilde{\mathbf{p}}, \quad (2.52)$$

where  $z_{\text{near}}$  and  $z_{\text{far}}$  are the near and far  $z$  *clipping planes* and  $z_{\text{range}} = z_{\text{far}} - z_{\text{near}}$ . Note that the first two rows are actually scaled by the focal length and the aspect ratio so that visible rays are mapped to  $(x, y, z) \in [-1, -1]^2$ . The reason for keeping the third row, rather than dropping it, is that visibility operations, such as *z-buffering*, require a depth for every graphical element that is being rendered.

If we set  $z_{\text{near}} = 1$ ,  $z_{\text{far}} \rightarrow \infty$ , and switch the sign of the third row, the third element of the normalized screen vector becomes the inverse depth, i.e., the *disparity* (Okutomi and Kanade 1993). This can be quite convenient in many cases since, for cameras moving around outdoors, the inverse depth to the camera is often a more well-conditioned parameterization than direct 3D distance.

While a regular 2D image sensor has no way of measuring distance to a surface point, *range sensors* (Section 12.2) and stereo matching algorithms (Chapter 11) can compute such values. It is then convenient to be able to map from a sensor-based depth or disparity value  $d$  directly back to a 3D location using the inverse of a  $4 \times 4$  matrix (Section 2.1.5). We can do this if we represent perspective projection using a full-rank  $4 \times 4$  matrix, as in (2.64).



**Figure 2.8** Projection of a 3D camera-centered point  $p_c$  onto the sensor planes at location  $p$ .  $O_c$  is the camera center (nodal point),  $c_s$  is the 3D origin of the sensor plane coordinate system, and  $s_x$  and  $s_y$  are the pixel spacings.

### Camera intrinsics

Once we have projected a 3D point through an ideal pinhole using a projection matrix, we must still transform the resulting coordinates according to the pixel sensor spacing and the relative position of the sensor plane to the origin. Figure 2.8 shows an illustration of the geometry involved. In this section, we first present a mapping from 2D pixel coordinates to 3D rays using a sensor homography  $M_s$ , since this is easier to explain in terms of physically measurable quantities. We then relate these quantities to the more commonly used camera intrinsic matrix  $K$ , which is used to map 3D camera-centered points  $p_c$  to 2D pixel coordinates  $\bar{x}_s$ .

Image sensors return pixel values indexed by integer *pixel coordinates*  $(x_s, y_s)$ , often with the coordinates starting at the upper-left corner of the image and moving down and to the right. (This convention is not obeyed by all imaging libraries, but the adjustment for other coordinate systems is straightforward.) To map pixel centers to 3D coordinates, we first scale the  $(x_s, y_s)$  values by the pixel spacings  $(s_x, s_y)$  (sometimes expressed in microns for solid-state sensors) and then describe the orientation of the sensor array relative to the camera projection center  $O_c$  with an origin  $c_s$  and a 3D rotation  $R_s$  (Figure 2.8).

The combined 2D to 3D projection can then be written as

$$p = \left[ R_s \mid c_s \right] \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_s \\ y_s \\ 1 \end{bmatrix} = M_s \bar{x}_s. \quad (2.53)$$

The first two columns of the  $3 \times 3$  matrix  $M_s$  are the 3D vectors corresponding to unit steps in the image pixel array along the  $x_s$  and  $y_s$  directions, while the third column is the 3D image array origin  $c_s$ .

The matrix  $M_s$  is parameterized by eight unknowns: the three parameters describing the rotation  $R_s$ , the three parameters describing the translation  $c_s$ , and the two scale factors  $(s_x, s_y)$ . Note that we ignore here the possibility of *skew* between the two axes on the image plane, since solid-state manufacturing techniques render this negligible. In practice, unless we have accurate external knowledge of the sensor spacing or sensor orientation, there are only seven degrees of freedom, since the distance of the sensor from the origin cannot be teased apart from the sensor spacing, based on external image measurement alone.

However, estimating a camera model  $M_s$  with the required seven degrees of freedom (i.e., where the first two columns are orthogonal after an appropriate re-scaling) is impractical, so most practitioners assume a general  $3 \times 3$  homogeneous matrix form.

The relationship between the 3D pixel center  $p$  and the 3D camera-centered point  $p_c$  is given by an unknown scaling  $s$ ,  $p = sp_c$ . We can therefore write the complete projection between  $p_c$  and a homogeneous version of the pixel address  $\tilde{x}_s$  as

$$\tilde{x}_s = \alpha M_s^{-1} p_c = K p_c. \quad (2.54)$$

The  $3 \times 3$  matrix  $K$  is called the *calibration matrix* and describes the camera *intrinsics* (as opposed to the camera's orientation in space, which are called the *extrinsics*).

From the above discussion, we see that  $K$  has seven degrees of freedom in theory and eight degrees of freedom (the full dimensionality of a  $3 \times 3$  homogeneous matrix) in practice. Why, then, do most textbooks on 3D computer vision and multi-view geometry (Faugeras 1993; Hartley and Zisserman 2004; Faugeras and Luong 2001) treat  $K$  as an upper-triangular matrix with five degrees of freedom?

While this is usually not made explicit in these books, it is because we cannot recover the full  $K$  matrix based on external measurement alone. When calibrating a camera (Chapter 6) based on external 3D points or other measurements (Tsai 1987), we end up estimating the intrinsic ( $K$ ) and extrinsic ( $R, t$ ) camera parameters simultaneously using a series of measurements,

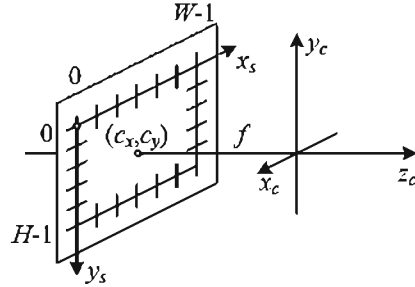
$$\tilde{x}_s = K \begin{bmatrix} R & | & t \end{bmatrix} p_w = P p_w, \quad (2.55)$$

where  $p_w$  are known 3D world coordinates and

$$P = K[R|t] \quad (2.56)$$

is known as the *camera matrix*. Inspecting this equation, we see that we can post-multiply  $K$  by  $R_1$  and pre-multiply  $[R|t]$  by  $R_1^T$ , and still end up with a valid calibration. Thus, it is impossible based on image measurements alone to know the true orientation of the sensor and the true camera intrinsics.

The choice of an upper-triangular form for  $K$  seems to be conventional. Given a full  $3 \times 4$  camera matrix  $P = K[R|t]$ , we can compute an upper-triangular  $K$  matrix using QR



**Figure 2.9** Simplified camera intrinsics showing the focal length  $f$  and the optical center  $(c_x, c_y)$ . The image width and height are  $W$  and  $H$ .

factorization (Golub and Van Loan 1996). (Note the unfortunate clash of terminologies: In matrix algebra textbooks,  $\mathbf{R}$  represents an upper-triangular (right of the diagonal) matrix; in computer vision,  $\mathbf{R}$  is an orthogonal rotation.)

There are several ways to write the upper-triangular form of  $\mathbf{K}$ . One possibility is

$$\mathbf{K} = \begin{bmatrix} f_x & s & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}, \quad (2.57)$$

which uses independent *focal lengths*  $f_x$  and  $f_y$  for the sensor  $x$  and  $y$  dimensions. The entry  $s$  encodes any possible *skew* between the sensor axes due to the sensor not being mounted perpendicular to the optical axis and  $(c_x, c_y)$  denotes the *optical center* expressed in pixel coordinates. Another possibility is

$$\mathbf{K} = \begin{bmatrix} f & s & c_x \\ 0 & af & c_y \\ 0 & 0 & 1 \end{bmatrix}, \quad (2.58)$$

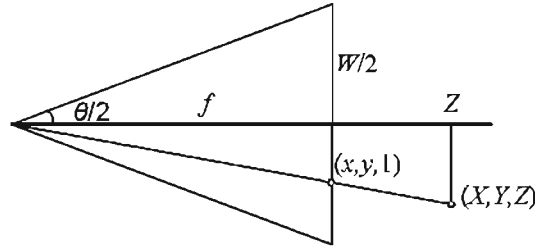
where the *aspect ratio*  $a$  has been made explicit and a common focal length  $f$  is used.

In practice, for many applications an even simpler form can be obtained by setting  $a = 1$  and  $s = 0$ ,

$$\mathbf{K} = \begin{bmatrix} f & 0 & c_x \\ 0 & f & c_y \\ 0 & 0 & 1 \end{bmatrix}. \quad (2.59)$$

Often, setting the origin at roughly the center of the image, e.g.,  $(c_x, c_y) = (W/2, H/2)$ , where  $W$  and  $H$  are the image height and width, can result in a perfectly usable camera model with a single unknown, i.e., the focal length  $f$ .





**Figure 2.10** Central projection, showing the relationship between the 3D and 2D coordinates,  $\mathbf{p}$  and  $\mathbf{x}$ , as well as the relationship between the focal length  $f$ , image width  $W$ , and the field of view  $\theta$ .

Figure 2.9 shows how these quantities can be visualized as part of a simplified imaging model. Note that now we have placed the image plane *in front* of the nodal point (projection center of the lens). The sense of the  $y$  axis has also been flipped to get a coordinate system compatible with the way that most imaging libraries treat the vertical (row) coordinate. Certain graphics libraries, such as Direct3D, use a left-handed coordinate system, which can lead to some confusion.

#### A note on focal lengths

The issue of how to express focal lengths is one that often causes confusion in implementing computer vision algorithms and discussing their results. This is because the focal length depends on the units used to measure pixels.

If we number pixel coordinates using integer values, say  $[0, W) \times [0, H)$ , the focal length  $f$  and camera center  $(c_x, c_y)$  in (2.59) can be expressed as pixel values. How do these quantities relate to the more familiar focal lengths used by photographers?

Figure 2.10 illustrates the relationship between the focal length  $f$ , the sensor width  $W$ , and the field of view  $\theta$ , which obey the formula

$$\tan \frac{\theta}{2} = \frac{W}{2f} \quad \text{or} \quad f = \frac{W}{2} \left[ \tan \frac{\theta}{2} \right]^{-1}. \quad (2.60)$$

For conventional film cameras,  $W = 35\text{mm}$ , and hence  $f$  is also expressed in millimeters. Since we work with digital images, it is more convenient to express  $W$  in pixels so that the focal length  $f$  can be used directly in the calibration matrix  $\mathbf{K}$  as in (2.59).

Another possibility is to scale the pixel coordinates so that they go from  $[-1, 1)$  along the longer image dimension and  $[-a^{-1}, a^{-1})$  along the shorter axis, where  $a \geq 1$  is the *image aspect ratio* (as opposed to the *sensor cell aspect ratio* introduced earlier). This can be

accomplished using *modified normalized device coordinates*,

$$x'_s = (2x_s - W)/S \text{ and } y'_s = (2y_s - H)/S, \text{ where } S = \max(W, H). \quad (2.61)$$

This has the advantage that the focal length  $f$  and optical center  $(c_x, c_y)$  become independent of the image resolution, which can be useful when using multi-resolution, image-processing algorithms, such as image pyramids (Section 3.5).<sup>2</sup> The use of  $S$  instead of  $W$  also makes the focal length the same for landscape (horizontal) and portrait (vertical) pictures, as is the case in 35mm photography. (In some computer graphics textbooks and systems, normalized device coordinates go from  $[-1, 1] \times [-1, 1]$ , which requires the use of two different focal lengths to describe the camera intrinsics (Watt 1995; OpenGL-ARB 1997).) Setting  $S = W = 2$  in (2.60), we obtain the simpler (unitless) relationship

$$f^{-1} = \tan \frac{\theta}{2}. \quad (2.62)$$

The conversion between the various focal length representations is straightforward, e.g., to go from a unitless  $f$  to one expressed in pixels, multiply by  $W/2$ , while to convert from an  $f$  expressed in pixels to the equivalent 35mm focal length, multiply by  $35/W$ .

### Camera matrix

Now that we have shown how to parameterize the calibration matrix  $K$ , we can put the camera intrinsics and extrinsics together to obtain a single  $3 \times 4$  *camera matrix*

$$P = K \left[ R \mid t \right]. \quad (2.63)$$

It is sometimes preferable to use an invertible  $4 \times 4$  matrix, which can be obtained by not dropping the last row in the  $P$  matrix,

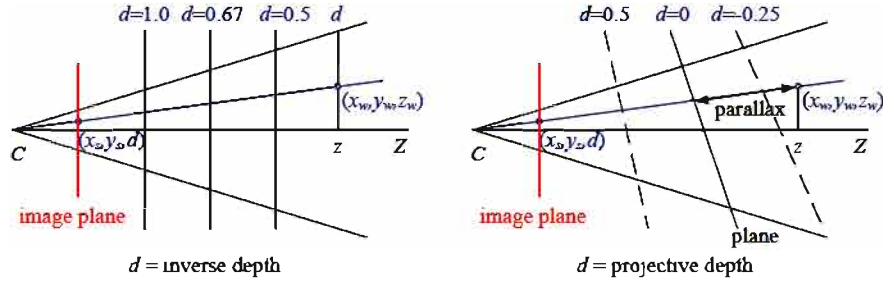
$$\tilde{P} = \begin{bmatrix} K & \mathbf{0} \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} R & t \\ \mathbf{0}^T & 1 \end{bmatrix} = \tilde{K}E, \quad (2.64)$$

where  $E$  is a 3D rigid-body (Euclidean) transformation and  $\tilde{K}$  is the full-rank calibration matrix. The  $4 \times 4$  camera matrix  $\tilde{P}$  can be used to map directly from 3D world coordinates  $\bar{p}_w = (x_w, y_w, z_w, 1)$  to screen coordinates (plus disparity),  $x_s = (x_s, y_s, 1, d)$ ,

$$x_s \sim \tilde{P}\bar{p}_w, \quad (2.65)$$

where  $\sim$  indicates equality up to scale. Note that after multiplication by  $\tilde{P}$ , the vector is divided by the *third* element of the vector to obtain the normalized form  $x_s = (x_s, y_s, 1, d)$ .

<sup>2</sup> To make the conversion truly accurate after a downsampling step in a pyramid, floating point values of  $W$  and  $H$  would have to be maintained since they can become non-integral if they are ever odd at a larger resolution in the pyramid.



**Figure 2.11** Regular disparity (inverse depth) and projective depth (parallax from a reference plane).

### Plane plus parallax (projective depth)

In general, when using the  $4 \times 4$  matrix  $\tilde{P}$ , we have the freedom to remap the last row to whatever suits our purpose (rather than just being the “standard” interpretation of disparity as inverse depth). Let us re-write the last row of  $\tilde{P}$  as  $\mathbf{p}_3 = s_3[\hat{\mathbf{n}}_0|c_0]$ , where  $\|\hat{\mathbf{n}}_0\| = 1$ . We then have the equation

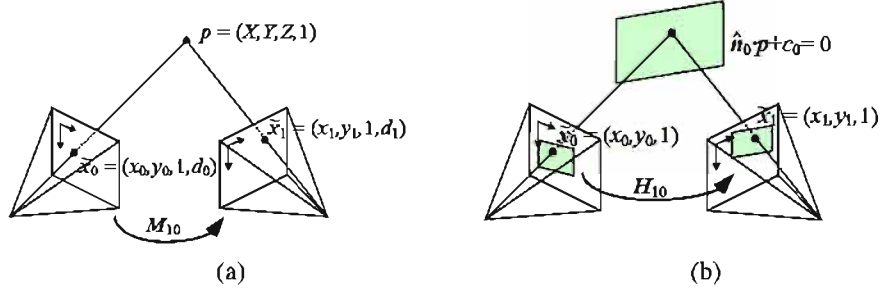
$$d = \frac{s_3}{z}(\hat{\mathbf{n}}_0 \cdot \mathbf{p}_w + c_0), \quad (2.66)$$

where  $z = \mathbf{p}_2 \cdot \bar{\mathbf{p}}_w = \mathbf{r}_z \cdot (\mathbf{p}_w - \mathbf{c})$  is the distance of  $\mathbf{p}_w$  from the camera center  $C$  (2.25) along the optical axis  $Z$  (Figure 2.11). Thus, we can interpret  $d$  as the *projective disparity* or *projective depth* of a 3D scene point  $\mathbf{p}_w$  from the *reference plane*  $\hat{\mathbf{n}}_0 \cdot \mathbf{p}_w + c_0 = 0$  (Szeliski and Coughlan 1997; Szeliski and Golland 1999; Shade, Gortler, He *et al.* 1998; Baker, Szeliski, and Anandan 1998). (The projective depth is also sometimes called *parallax* in reconstruction algorithms that use the term *plane plus parallax* (Kumar, Anandan, and Hanna 1994; Sawhney 1994).) Setting  $\hat{\mathbf{n}}_0 = \mathbf{0}$  and  $c_0 = 1$ , i.e., putting the reference plane at infinity, results in the more standard  $d = 1/z$  version of disparity (Okutomi and Kanade 1993).

Another way to see this is to invert the  $\tilde{P}$  matrix so that we can map pixels plus disparity directly back to 3D points,

$$\bar{\mathbf{p}}_w = \tilde{P}^{-1} \mathbf{x}_s. \quad (2.67)$$

In general, we can choose  $\tilde{P}$  to have whatever form is convenient, i.e., to sample space using an arbitrary projection. This can come in particularly handy when setting up multi-view stereo reconstruction algorithms, since it allows us to sweep a series of planes (Section 11.1.2) through space with a variable (projective) sampling that best matches the sensed image motions (Collins 1996; Szeliski and Golland 1999; Saito and Kanade 1999).



**Figure 2.12** A point is projected into two images: (a) relationship between the 3D point coordinate  $(X, Y, Z, 1)$  and the 2D projected point  $(x, y, 1, d)$ ; (b) planar homography induced by points all lying on a common plane  $\hat{n}_0 \cdot p + c_0 = 0$ .

### Mapping from one camera to another

What happens when we take two images of a 3D scene from different camera positions or orientations (Figure 2.12a)? Using the full rank  $4 \times 4$  camera matrix  $\tilde{P} = \tilde{K}E$  from (2.64), we can write the projection from world to screen coordinates as

$$\tilde{x}_0 \sim \tilde{K}_0 E_0 p = \tilde{P}_0 p. \quad (2.68)$$

Assuming that we know the z-buffer or disparity value  $d_0$  for a pixel in one image, we can compute the 3D point location  $p$  using

$$p \sim E_0^{-1} \tilde{K}_0^{-1} \tilde{x}_0 \quad (2.69)$$

and then project it into another image yielding

$$\tilde{x}_1 \sim \tilde{K}_1 E_1 p = \tilde{K}_1 E_1 E_0^{-1} \tilde{K}_0^{-1} \tilde{x}_0 = \tilde{P}_1 \tilde{P}_0^{-1} \tilde{x}_0 = M_{10} \tilde{x}_0. \quad (2.70)$$

Unfortunately, we do not usually have access to the depth coordinates of pixels in a regular photographic image. However, for a *planar scene*, as discussed above in (2.66), we can replace the last row of  $P_0$  in (2.64) with a general *plane equation*,  $\hat{n}_0 \cdot p + c_0$  that maps points on the plane to  $d_0 = 0$  values (Figure 2.12b). Thus, if we set  $d_0 = 0$ , we can ignore the last column of  $M_{10}$  in (2.70) and also its last row, since we do not care about the final z-buffer depth. The mapping equation (2.70) thus reduces to

$$\tilde{x}_1 \sim \tilde{H}_{10} \tilde{x}_0, \quad (2.71)$$

where  $\tilde{H}_{10}$  is a general  $3 \times 3$  homography matrix and  $\tilde{x}_1$  and  $\tilde{x}_0$  are now 2D homogeneous coordinates (i.e., 3-vectors) (Szeliski 1996). This justifies the use of the 8-parameter homography as a general alignment model for mosaics of planar scenes (Mann and Picard 1994; Szeliski 1996).

The other special case where we do not need to know depth to perform inter-camera mapping is when the camera is undergoing pure rotation (Section 9.1.3), i.e., when  $t_0 = t_1$ . In this case, we can write

$$\tilde{x}_1 \sim \mathbf{K}_1 \mathbf{R}_1 \mathbf{R}_0^{-1} \mathbf{K}_0^{-1} \tilde{x}_0 = \mathbf{K}_1 \mathbf{R}_{10} \mathbf{K}_0^{-1} \tilde{x}_0, \quad (2.72)$$

which again can be represented with a  $3 \times 3$  homography. If we assume that the calibration matrices have known aspect ratios and centers of projection (2.59), this homography can be parameterized by the rotation amount and the two unknown focal lengths. This particular formulation is commonly used in image-stitching applications (Section 9.1.3).

### Object-centered projection

When working with long focal length lenses, it often becomes difficult to reliably estimate the focal length from image measurements alone. This is because the focal length and the distance to the object are highly correlated and it becomes difficult to tease these two effects apart. For example, the change in scale of an object viewed through a zoom telephoto lens can either be due to a zoom change or a motion towards the user. (This effect was put to dramatic use in some of Alfred Hitchcock's film *Vertigo*, where the simultaneous change of zoom and camera motion produces a disquieting effect.)

This ambiguity becomes clearer if we write out the projection equation corresponding to the simple calibration matrix  $\mathbf{K}$  (2.59),

$$x_s = f \frac{\mathbf{r}_x \cdot \mathbf{p} + t_x}{\mathbf{r}_z \cdot \mathbf{p} + t_z} + c_x \quad (2.73)$$

$$y_s = f \frac{\mathbf{r}_y \cdot \mathbf{p} + t_y}{\mathbf{r}_z \cdot \mathbf{p} + t_z} + c_y, \quad (2.74)$$

where  $\mathbf{r}_x$ ,  $\mathbf{r}_y$ , and  $\mathbf{r}_z$  are the three rows of  $\mathbf{R}$ . If the distance to the object center  $t_z \gg \|\mathbf{p}\|$  (the size of the object), the denominator is approximately  $t_z$  and the overall scale of the projected object depends on the ratio of  $f$  to  $t_z$ . It therefore becomes difficult to disentangle these two quantities.

To see this more clearly, let  $\eta_z = t_z^{-1}$  and  $s = \eta_z f$ . We can then re-write the above equations as

$$x_s = s \frac{\mathbf{r}_x \cdot \mathbf{p} + t_x}{1 + \eta_z \mathbf{r}_z \cdot \mathbf{p}} + c_x \quad (2.75)$$

$$y_s = s \frac{\mathbf{r}_y \cdot \mathbf{p} + t_y}{1 + \eta_z \mathbf{r}_z \cdot \mathbf{p}} + c_y \quad (2.76)$$

(Szeliski and Kang 1994; Pighin, Hecker, Lischinski *et al.* 1998). The scale of the projection  $s$  can be reliably estimated if we are looking at a known object (i.e., the 3D coordinates  $\mathbf{p}$

are known). The inverse distance  $\eta_z$  is now mostly decoupled from the estimates of  $s$  and can be estimated from the amount of *foreshortening* as the object rotates. Furthermore, as the lens becomes longer, i.e., the projection model becomes orthographic, there is no need to replace a perspective imaging model with an orthographic one, since the same equation can be used, with  $\eta_z \rightarrow 0$  (as opposed to  $f$  and  $t_z$  both going to infinity). This allows us to form a natural link between orthographic reconstruction techniques such as factorization and their projective/perspective counterparts (Section 7.3).

### 2.1.6 Lens distortions

The above imaging models all assume that cameras obey a *linear* projection model where straight lines in the world result in straight lines in the image. (This follows as a natural consequence of linear matrix operations being applied to homogeneous coordinates.) Unfortunately, many wide-angle lenses have noticeable *radial distortion*, which manifests itself as a visible curvature in the projection of straight lines. (See Section 2.2.3 for a more detailed discussion of lens optics, including chromatic aberration.) Unless this distortion is taken into account, it becomes impossible to create highly accurate photorealistic reconstructions. For example, image mosaics constructed without taking radial distortion into account will often exhibit blurring due to the mis-registration of corresponding features before pixel blending (Chapter 9).

Fortunately, compensating for radial distortion is not that difficult in practice. For most lenses, a simple quartic model of distortion can produce good results. Let  $(x_c, y_c)$  be the pixel coordinates obtained *after* perspective division but *before* scaling by focal length  $f$  and shifting by the optical center  $(c_x, c_y)$ , i.e.,

$$\begin{aligned} x_c &= \frac{r_x \cdot p + t_x}{r_z \cdot p + t_z} \\ y_c &= \frac{r_y \cdot p + t_y}{r_z \cdot p + t_z}. \end{aligned} \quad (2.77)$$

The radial distortion model says that coordinates in the observed images are displaced away (*barrel* distortion) or towards (*pincushion* distortion) the image center by an amount proportional to their radial distance (Figure 2.13a–b).<sup>3</sup> The simplest radial distortion models use low-order polynomials, e.g.,

$$\begin{aligned} \hat{x}_c &= x_c(1 + \kappa_1 r_c^2 + \kappa_2 r_c^4) \\ \hat{y}_c &= y_c(1 + \kappa_1 r_c^2 + \kappa_2 r_c^4), \end{aligned} \quad (2.78)$$

<sup>3</sup> Anamorphic lenses, which are widely used in feature film production, do not follow this radial distortion model. Instead, they can be thought of, to a first approximation, as inducing different vertical and horizontal scalings, i.e., non-square pixels.



**Figure 2.13** Radial lens distortions: (a) barrel, (b) pincushion, and (c) fisheye. The fisheye image spans almost  $180^\circ$  from side-to-side.

where  $r_c^2 = x_c^2 + y_c^2$  and  $\kappa_1$  and  $\kappa_2$  are called the *radial distortion parameters*.<sup>4</sup> After the radial distortion step, the final pixel coordinates can be computed using

$$\begin{aligned}x_s &= fx'_c + c_x \\y_s &= fy'_c + c_y.\end{aligned}\tag{2.79}$$

A variety of techniques can be used to estimate the radial distortion parameters for a given lens, as discussed in Section 6.3.5.

Sometimes the above simplified model does not model the true distortions produced by complex lenses accurately enough (especially at very wide angles). A more complete analytic model also includes *tangential distortions* and *decentering distortions* (Slama 1980), but these distortions are not covered in this book.

Fisheye lenses (Figure 2.13c) require a model that differs from traditional polynomial models of radial distortion. Fisheye lenses behave, to a first approximation, as *equi-distance* projectors of angles away from the optical axis (Xiong and Turkowski 1997), which is the same as the *polar projection* described by Equations (9.22–9.24). Xiong and Turkowski (1997) describe how this model can be extended with the addition of an extra quadratic correction in  $\phi$  and how the unknown parameters (center of projection, scaling factor  $s$ , etc.) can be estimated from a set of overlapping fisheye images using a direct (intensity-based) non-linear minimization algorithm.

For even larger, less regular distortions, a parametric distortion model using splines may be necessary (Goshtasby 1989). If the lens does not have a single center of projection, it

<sup>4</sup> Sometimes the relationship between  $x_c$  and  $\hat{x}_c$  is expressed the other way around, i.e.,  $x_c = \hat{x}_c(1 + \kappa_1 r_c^2 + \kappa_2 r_c^4)$ . This is convenient if we map image pixels into (warped) rays by dividing through by  $f$ . We can then undistort the rays and have true 3D rays in space.



may become necessary to model the 3D *line* (as opposed to *direction*) corresponding to each pixel separately (Gremban, Thorpe, and Kanade 1988; Champleboux, Lavallée, Sautot *et al.* 1992; Grossberg and Nayar 2001; Sturm and Ramalingam 2004; Tardif, Sturm, Trudeau *et al.* 2009). Some of these techniques are described in more detail in Section 6.3.5, which discusses how to calibrate lens distortions.

There is one subtle issue associated with the simple radial distortion model that is often glossed over. We have introduced a non-linearity between the perspective projection and final sensor array projection steps. Therefore, we cannot, in general, post-multiply an arbitrary  $3 \times 3$  matrix  $\mathbf{K}$  with a rotation to put it into upper-triangular form and absorb this into the global rotation. However, this situation is not as bad as it may at first appear. For many applications, keeping the simplified diagonal form of (2.56) is still an adequate model. Furthermore, if we correct radial and other distortions to an accuracy where straight lines are preserved, we have essentially converted the sensor back into a linear imager and the previous decomposition still applies.

## 2.2 Photometric image formation

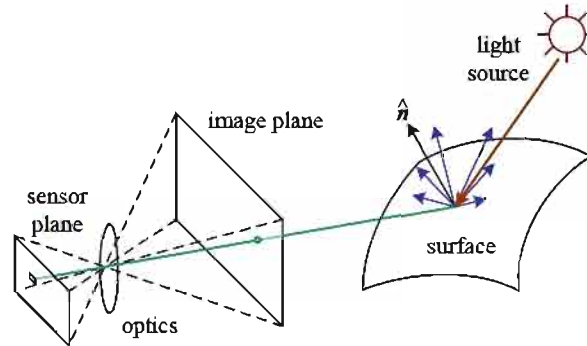
In modeling the image formation process, we have described how 3D geometric features in the world are projected into 2D features in an image. However, images are not composed of 2D features. Instead, they are made up of discrete color or intensity values. Where do these values come from? How do they relate to the lighting in the environment, surface properties and geometry, camera optics, and sensor properties (Figure 2.14)? In this section, we develop a set of models to describe these interactions and formulate a generative process of image formation. A more detailed treatment of these topics can be found in other textbooks on computer graphics and image synthesis (Glassner 1995; Weyrich, Lawrence, Lensch *et al.* 2008; Foley, van Dam, Feiner *et al.* 1995; Watt 1995; Cohen and Wallace 1993; Sillion and Puech 1994).

### 2.2.1 Lighting

Images cannot exist without light. To produce an image, the scene must be illuminated with one or more light sources. (Certain modalities such as fluorescent microscopy and X-ray tomography do not fit this model, but we do not deal with them in this book.) Light sources can generally be divided into point and area light sources.

A point light source originates at a single location in space (e.g., a small light bulb), potentially at infinity (e.g., the sun). (Note that for some applications such as modeling soft shadows (*penumbras*), the sun may have to be treated as an area light source.) In addition to its location, a point light source has an intensity and a color spectrum, i.e., a distribution over





**Figure 2.14** A simplified model of photometric image formation. Light is emitted by one or more light sources and is then reflected from an object’s surface. A portion of this light is directed towards the camera. This simplified model ignores multiple reflections, which often occur in real-world scenes.

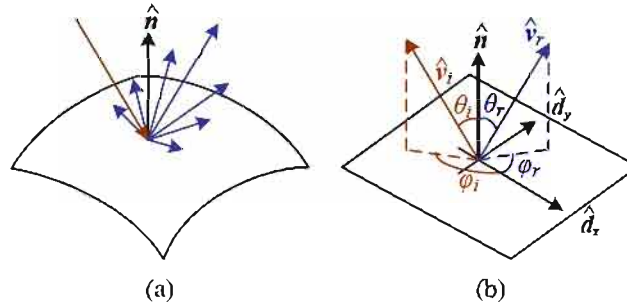
wavelengths  $L(\lambda)$ . The intensity of a light source falls off with the square of the distance between the source and the object being lit, because the same light is being spread over a larger (spherical) area. A light source may also have a directional falloff (dependence), but we ignore this in our simplified model.

Area light sources are more complicated. A simple area light source such as a fluorescent ceiling light fixture with a diffuser can be modeled as a finite rectangular area emitting light equally in all directions (Cohen and Wallace 1993; Sillion and Puech 1994; Glassner 1995). When the distribution is strongly directional, a four-dimensional lightfield can be used instead (Ashdown 1993).

A more complex light distribution that approximates, say, the incident illumination on an object sitting in an outdoor courtyard, can often be represented using an *environment map* (Greene 1986) (originally called a *reflection map* (Blinn and Newell 1976)). This representation maps incident light directions  $\vec{v}$  to color values (or wavelengths,  $\lambda$ ),

$$L(\vec{v}; \lambda), \quad (2.80)$$

and is equivalent to assuming that all light sources are at infinity. Environment maps can be represented as a collection of cubical faces (Greene 1986), as a single longitude–latitude map (Blinn and Newell 1976), or as the image of a reflecting sphere (Watt 1995). A convenient way to get a rough model of a real-world environment map is to take an image of a reflective mirrored sphere and to unwrap this image onto the desired environment map (Debevec 1998). Watt (1995) gives a nice discussion of environment mapping, including the formulas needed to map directions to pixels for the three most commonly used representations.



**Figure 2.15** (a) Light scatters when it hits a surface. (b) The bidirectional reflectance distribution function (BRDF)  $f(\theta_i, \phi_i, \theta_r, \phi_r)$  is parameterized by the angles that the incident,  $\hat{v}_i$ , and reflected,  $\hat{v}_r$ , light ray directions make with the local surface coordinate frame  $(\hat{d}_x, \hat{d}_y, \hat{n})$ .

### 2.2.2 Reflectance and shading

When light hits an object's surface, it is scattered and reflected (Figure 2.15a). Many different models have been developed to describe this interaction. In this section, we first describe the most general form, the bidirectional reflectance distribution function, and then look at some more specialized models, including the diffuse, specular, and Phong shading models. We also discuss how these models can be used to compute the *global illumination* corresponding to a scene.

#### The Bidirectional Reflectance Distribution Function (BRDF)

The most general model of light scattering is the *bidirectional reflectance distribution function* (BRDF).<sup>5</sup> Relative to some local coordinate frame on the surface, the BRDF is a four-dimensional function that describes how much of each wavelength arriving at an *incident* direction  $\hat{v}_i$  is emitted in a *reflected* direction  $\hat{v}_r$  (Figure 2.15b). The function can be written in terms of the angles of the incident and reflected directions relative to the surface frame as

$$f_r(\theta_i, \phi_i, \theta_r, \phi_r; \lambda). \quad (2.81)$$

The BRDF is *reciprocal*, i.e., because of the physics of light transport, you can interchange the roles of  $\hat{v}_i$  and  $\hat{v}_r$  and still get the same answer (this is sometimes called *Helmholtz reciprocity*).

<sup>5</sup> Actually, even more general models of light transport exist, including some that model spatial variation along the surface, sub-surface scattering, and atmospheric effects—see Section 12.7.1—(Dorsey, Rushmeier, and Sillion 2007; Weyrich, Lawrence, Lensch *et al.* 2008).

Most surfaces are *isotropic*, i.e., there are no preferred directions on the surface as far as light transport is concerned. (The exceptions are *anisotropic* surfaces such as brushed (scratched) aluminum, where the reflectance depends on the light orientation relative to the direction of the scratches.) For an isotropic material, we can simplify the BRDF to

$$f_r(\theta_i, \theta_r, |\phi_r - \phi_i|; \lambda) \text{ or } f_r(\hat{\mathbf{v}}_i, \hat{\mathbf{v}}_r, \hat{\mathbf{n}}; \lambda), \quad (2.82)$$

since the quantities  $\theta_i$ ,  $\theta_r$  and  $\phi_r - \phi_i$  can be computed from the directions  $\hat{\mathbf{v}}_i$ ,  $\hat{\mathbf{v}}_r$ , and  $\hat{\mathbf{n}}$ .

To calculate the amount of light exiting a surface point  $\mathbf{p}$  in a direction  $\hat{\mathbf{v}}_r$  under a given lighting condition, we integrate the product of the incoming light  $L_i(\hat{\mathbf{v}}_i; \lambda)$  with the BRDF (some authors call this step a *convolution*). Taking into account the *foreshortening* factor  $\cos^+ \theta_i$ , we obtain

$$L_r(\hat{\mathbf{v}}_r; \lambda) = \int L_i(\hat{\mathbf{v}}_i; \lambda) f_r(\hat{\mathbf{v}}_i, \hat{\mathbf{v}}_r, \hat{\mathbf{n}}; \lambda) \cos^+ \theta_i d\hat{\mathbf{v}}_i, \quad (2.83)$$

where

$$\cos^+ \theta_i = \max(0, \cos \theta_i). \quad (2.84)$$

If the light sources are discrete (a finite number of point light sources), we can replace the integral with a summation,

$$L_r(\hat{\mathbf{v}}_r; \lambda) = \sum_i L_i(\lambda) f_r(\hat{\mathbf{v}}_i, \hat{\mathbf{v}}_r, \hat{\mathbf{n}}; \lambda) \cos^+ \theta_i. \quad (2.85)$$

BRDFs for a given surface can be obtained through physical modeling (Torrance and Sparrow 1967; Cook and Torrance 1982; Glassner 1995), heuristic modeling (Phong 1975), or through empirical observation (Ward 1992; Westin, Arvo, and Torrance 1992; Dana, van Ginneken, Nayar *et al.* 1999; Dorsey, Rushmeier, and Sillion 2007; Weyrich, Lawrence, Lensch *et al.* 2008).<sup>6</sup> Typical BRDFs can often be split into their *diffuse* and *specular* components, as described below.

### Diffuse reflection

The diffuse component (also known as *Lambertian* or *matte* reflection) scatters light uniformly in all directions and is the phenomenon we most normally associate with *shading*, e.g., the smooth (non-shiny) variation of intensity with surface normal that is seen when observing a statue (Figure 2.16). Diffuse reflection also often imparts a strong *body color* to the light since it is caused by selective absorption and re-emission of light inside the object's material (Shafer 1985; Glassner 1995).

<sup>6</sup> See <http://www1.cs.columbia.edu/CAVE/software/curet/> for a database of some empirically sampled BRDFs.



**Figure 2.16** This close-up of a statue shows both diffuse (smooth shading) and specular (shiny highlight) reflection, as well as darkening in the grooves and creases due to reduced light visibility and interreflections. (Photo courtesy of the Caltech Vision Lab, <http://www.vision.caltech.edu/archive.html>.)

While light is scattered uniformly in all directions, i.e., the BRDF is constant,

$$f_d(\hat{\mathbf{v}}_i, \hat{\mathbf{v}}_r, \hat{\mathbf{n}}; \lambda) = f_d(\lambda), \quad (2.86)$$

the amount of light depends on the angle between the incident light direction and the surface normal  $\theta_i$ . This is because the surface area exposed to a given amount of light becomes larger at oblique angles, becoming completely self-shadowed as the outgoing surface normal points away from the light (Figure 2.17a). (Think about how you orient yourself towards the sun or fireplace to get maximum warmth and how a flashlight projected obliquely against a wall is less bright than one pointing directly at it.) The *shading equation* for diffuse reflection can thus be written as

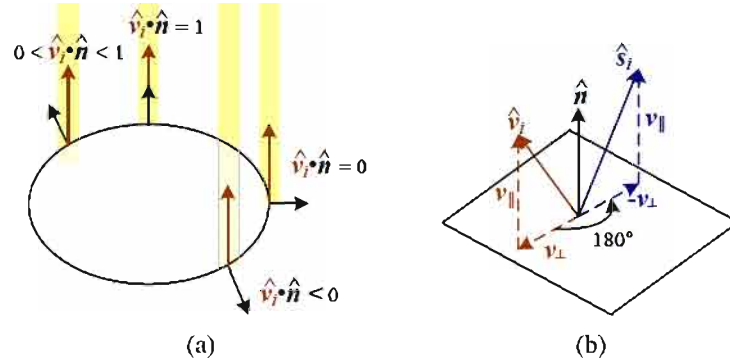
$$L_d(\hat{\mathbf{v}}_r; \lambda) = \sum_i L_i(\lambda) f_d(\lambda) \cos^+ \theta_i = \sum_i L_i(\lambda) f_d(\lambda) [\hat{\mathbf{v}}_i \cdot \hat{\mathbf{n}}]^+, \quad (2.87)$$

where

$$[\hat{\mathbf{v}}_i \cdot \hat{\mathbf{n}}]^+ = \max(0, \hat{\mathbf{v}}_i \cdot \hat{\mathbf{n}}). \quad (2.88)$$

### Specular reflection

The second major component of a typical BRDF is *specular* (gloss or highlight) reflection, which depends strongly on the direction of the outgoing light. Consider light reflecting off a mirrored surface (Figure 2.17b). Incident light rays are reflected in a direction that is rotated by  $180^\circ$  around the surface normal  $\hat{\mathbf{n}}$ . Using the same notation as in Equations (2.29–2.30),



**Figure 2.17** (a) The diminution of returned light caused by *foreshortening* depends on  $\hat{v}_i \cdot \hat{n}$ , the cosine of the angle between the incident light direction  $\hat{v}_i$  and the surface normal  $\hat{n}$ . (b) Mirror (specular) reflection: The incident light ray direction  $\hat{v}_i$  is reflected onto the specular direction  $\hat{s}_i$  around the surface normal  $\hat{n}$ .

we can compute the *specular reflection* direction  $\hat{s}_i$  as

$$\hat{s}_i = \mathbf{v}_{\parallel} - \mathbf{v}_{\perp} = (2\hat{n}\hat{n}^T - \mathbf{I})\mathbf{v}_i. \quad (2.89)$$

The amount of light reflected in a given direction  $\hat{v}_r$  thus depends on the angle  $\theta_s = \cos^{-1}(\hat{v}_r \cdot \hat{s}_i)$  between the view direction  $\hat{v}_r$  and the specular direction  $\hat{s}_i$ . For example, the Phong (1975) model uses a power of the cosine of the angle,

$$f_s(\theta_s; \lambda) = k_s(\lambda) \cos^{k_e} \theta_s, \quad (2.90)$$

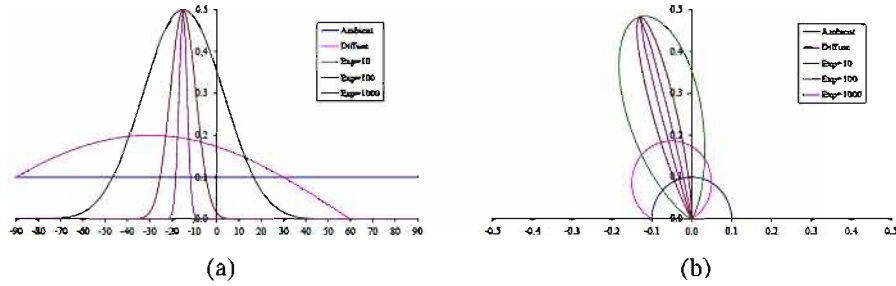
while the Torrance and Sparrow (1967) micro-facet model uses a Gaussian,

$$f_s(\theta_s; \lambda) = k_s(\lambda) \exp(-c_s^2 \theta_s^2). \quad (2.91)$$

Larger exponents  $k_e$  (or inverse Gaussian widths  $c_s$ ) correspond to more specular surfaces with distinct highlights, while smaller exponents better model materials with softer gloss.

### Phong shading

Phong (1975) combined the diffuse and specular components of reflection with another term, which he called the *ambient illumination*. This term accounts for the fact that objects are generally illuminated not only by point light sources but also by a general diffuse illumination corresponding to inter-reflection (e.g., the walls in a room) or distant sources, such as the



**Figure 2.18** Cross-section through a Phong shading model BRDF for a fixed incident illumination direction: (a) component values as a function of angle away from surface normal; (b) polar plot. The value of the Phong exponent  $k_e$  is indicated by the “Exp” labels and the light source is at an angle of  $30^\circ$  away from the normal.

blue sky. In the Phong model, the ambient term does not depend on surface orientation, but depends on the color of both the ambient illumination  $L_a(\lambda)$  and the object  $k_a(\lambda)$ ,

$$f_a(\lambda) = k_a(\lambda)L_a(\lambda). \quad (2.92)$$

Putting all of these terms together, we arrive at the *Phong shading* model,

$$L_r(\vec{v}_r; \lambda) = k_a(\lambda)L_a(\lambda) + k_d(\lambda) \sum_i L_i(\lambda)[\vec{v}_i \cdot \hat{n}]^+ + k_s(\lambda) \sum_i L_i(\lambda)(\vec{v}_r \cdot \hat{s}_i)^{k_e}. \quad (2.93)$$

Figure 2.18 shows a typical set of Phong shading model components as a function of the angle away from the surface normal (in a plane containing both the lighting direction and the viewer).

Typically, the ambient and diffuse reflection color distributions  $k_a(\lambda)$  and  $k_d(\lambda)$  are the same, since they are both due to sub-surface scattering (body reflection) inside the surface material (Shafer 1985). The specular reflection distribution  $k_s(\lambda)$  is often uniform (white), since it is caused by interface reflections that do not change the light color. (The exception to this are *metallic* materials, such as copper, as opposed to the more common *dielectric* materials, such as plastics.)

The ambient illumination  $L_a(\lambda)$  often has a different color cast from the direct light sources  $L_i(\lambda)$ , e.g., it may be blue for a sunny outdoor scene or yellow for an interior lit with candles or incandescent lights. (The presence of ambient sky illumination in shadowed areas is what often causes shadows to appear bluer than the corresponding lit portions of a scene). Note also that the diffuse component of the Phong model (or of any shading model) depends on the angle of the *incoming* light source  $\vec{v}_i$ , while the specular component depends on the relative angle between the viewer  $\vec{v}_r$  and the specular reflection direction  $\hat{s}_i$  (which itself depends on the incoming light direction  $\vec{v}_i$  and the surface normal  $\hat{n}$ ).

The Phong shading model has been superseded in terms of physical accuracy by a number of more recently developed models in computer graphics, including the model developed by Cook and Torrance (1982) based on the original micro-facet model of Torrance and Sparrow (1967). Until recently, most computer graphics hardware implemented the Phong model but the recent advent of programmable pixel shaders makes the use of more complex models feasible.

### Di-chromatic reflection model

The Torrance and Sparrow (1967) model of reflection also forms the basis of Shafer's (1985) *di-chromatic reflection model*, which states that the apparent color of a uniform material lit from a single source depends on the sum of two terms,

$$L_r(\hat{\mathbf{v}}_r; \lambda) = L_i(\hat{\mathbf{v}}_r, \hat{\mathbf{v}}_i, \hat{\mathbf{n}}; \lambda) + L_b(\hat{\mathbf{v}}_r, \hat{\mathbf{v}}_i, \hat{\mathbf{n}}; \lambda) \quad (2.94)$$

$$= c_i(\lambda)m_i(\hat{\mathbf{v}}_r, \hat{\mathbf{v}}_i, \hat{\mathbf{n}}) + c_b(\lambda)m_b(\hat{\mathbf{v}}_r, \hat{\mathbf{v}}_i, \hat{\mathbf{n}}), \quad (2.95)$$

i.e., the radiance of the light reflected at the *interface*,  $L_i$ , and the radiance reflected at the *surface body*,  $L_b$ . Each of these, in turn, is a simple product between a relative power spectrum  $c(\lambda)$ , which depends only on wavelength, and a magnitude  $m(\hat{\mathbf{v}}_r, \hat{\mathbf{v}}_i, \hat{\mathbf{n}})$ , which depends only on geometry. (This model can easily be derived from a generalized version of Phong's model by assuming a single light source and no ambient illumination, and re-arranging terms.) The di-chromatic model has been successfully used in computer vision to segment specular colored objects with large variations in shading (Klinker 1993) and more recently has inspired local two-color models for applications such as Bayer pattern demosaicing (Bennett, Uyttendaele, Zitnick *et al.* 2006).

### Global illumination (ray tracing and radiosity)

The simple shading model presented thus far assumes that light rays leave the light sources, bounce off surfaces visible to the camera, thereby changing in intensity or color, and arrive at the camera. In reality, light sources can be shadowed by occluders and rays can bounce multiple times around a scene while making their trip from a light source to the camera.

Two methods have traditionally been used to model such effects. If the scene is mostly specular (the classic example being scenes made of glass objects and mirrored or highly polished balls), the preferred approach is *ray tracing* or *path tracing* (Glassner 1995; Akenine-Möller and Haines 2002; Shirley 2005), which follows individual rays from the camera across multiple bounces towards the light sources (or vice versa). If the scene is composed mostly of uniform albedo simple geometry illuminators and surfaces, *radiosity* (*global illumination*) techniques are preferred (Cohen and Wallace 1993; Sillion and Puech 1994; Glassner 1995).



Combinations of the two techniques have also been developed (Wallace, Cohen, and Greenberg 1987), as well as more general *light transport* techniques for simulating effects such as the *caustics* cast by rippling water.

The basic ray tracing algorithm associates a light ray with each pixel in the camera image and finds its intersection with the nearest surface. A *primary* contribution can then be computed using the simple shading equations presented previously (e.g., Equation (2.93)) for all light sources that are visible for that surface element. (An alternative technique for computing which surfaces are illuminated by a light source is to compute a *shadow map*, or *shadow buffer*, i.e., a rendering of the scene from the light source's perspective, and then compare the depth of pixels being rendered with the map (Williams 1983; Akenine-Möller and Haines 2002).) Additional *secondary* rays can then be cast along the specular direction towards other objects in the scene, keeping track of any attenuation or color change that the specular reflection induces.

Radiosity works by associating lightness values with rectangular surface areas in the scene (including area light sources). The amount of light interchanged between any two (mutually visible) areas in the scene can be captured as a *form factor*, which depends on their relative orientation and surface reflectance properties, as well as the  $1/r^2$  fall-off as light is distributed over a larger effective sphere the further away it is (Cohen and Wallace 1993; Sillion and Puech 1994; Glassner 1995). A large linear system can then be set up to solve for the final lightness of each area patch, using the light sources as the forcing function (right hand side). Once the system has been solved, the scene can be rendered from any desired point of view. Under certain circumstances, it is possible to recover the global illumination in a scene from photographs using computer vision techniques (Yu, Debevec, Malik *et al.* 1999).

The basic radiosity algorithm does not take into account certain *near field* effects, such as the darkening inside corners and scratches, or the limited ambient illumination caused by partial shadowing from other surfaces. Such effects have been exploited in a number of computer vision algorithms (Nayar, Ikeuchi, and Kanade 1991; Langer and Zucker 1994).

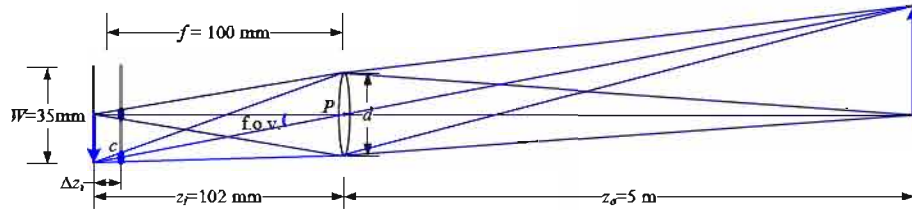
While all of these global illumination effects can have a strong effect on the appearance of a scene, and hence its 3D interpretation, they are not covered in more detail in this book. (But see Section 12.7.1 for a discussion of recovering BRDFs from real scenes and objects.)

### 2.2.3 Optics

Once the light from a scene reaches the camera, it must still pass through the lens before reaching the sensor (analog film or digital silicon). For many applications, it suffices to treat the lens as an ideal pinhole that simply projects all rays through a common center of projection (Figures 2.8 and 2.9).

However, if we want to deal with issues such as focus, exposure, vignetting, and aber-





**Figure 2.19** A thin lens of focal length  $f$  focuses the light from a plane a distance  $z_o$  in front of the lens at a distance  $z_i$  behind the lens, where  $\frac{1}{z_o} + \frac{1}{z_i} = \frac{1}{f}$ . If the focal plane (vertical gray line next to  $c$ ) is moved forward, the images are no longer in focus and the *circle of confusion*  $c$  (small thick line segments) depends on the distance of the image plane motion  $\Delta z_i$  relative to the lens aperture diameter  $d$ . The field of view (f.o.v.) depends on the ratio between the sensor width  $W$  and the focal length  $f$  (or, more precisely, the focusing distance  $z_i$ , which is usually quite close to  $f$ ).

ration, we need to develop a more sophisticated model, which is where the study of *optics* comes in (Möller 1988; Hecht 2001; Ray 2002).

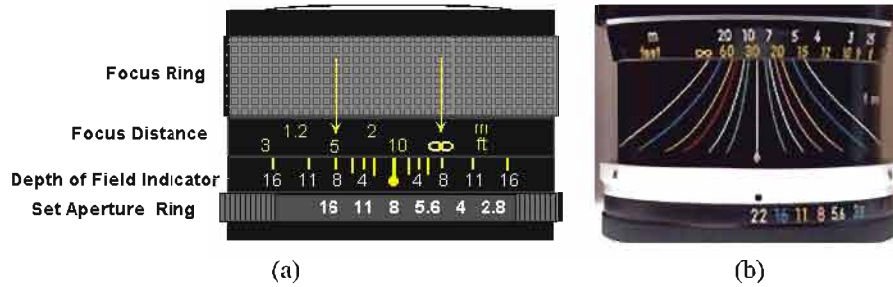
Figure 2.19 shows a diagram of the most basic lens model, i.e., the *thin lens* composed of a single piece of glass with very low, equal curvature on both sides. According to the *lens law* (which can be derived using simple geometric arguments on light ray refraction), the relationship between the distance to an object  $z_o$  and the distance behind the lens at which a focused image is formed  $z_i$  can be expressed as

$$\frac{1}{z_o} + \frac{1}{z_i} = \frac{1}{f}, \quad (2.96)$$

where  $f$  is called the *focal length* of the lens. If we let  $z_o \rightarrow \infty$ , i.e., we adjust the lens (move the image plane) so that objects at infinity are in focus, we get  $z_i = f$ , which is why we can think of a lens of focal length  $f$  as being equivalent (to a first approximation) to a pinhole a distance  $f$  from the focal plane (Figure 2.10), whose field of view is given by (2.60).

If the focal plane is moved away from its proper in-focus setting of  $z_i$  (e.g., by twisting the focus ring on the lens), objects at  $z_o$  are no longer in focus, as shown by the gray plane in Figure 2.19. The amount of mis-focus is measured by the *circle of confusion*  $c$  (shown as short thick blue line segments on the gray plane).<sup>7</sup> The equation for the circle of confusion can be derived using similar triangles; it depends on the distance of travel in the focal plane  $\Delta z_i$  relative to the original focus distance  $z_i$  and the diameter of the aperture  $d$  (see Exercise 2.4).

<sup>7</sup> If the aperture is not completely circular, e.g., if it is caused by a hexagonal diaphragm, it is sometimes possible to see this effect in the actual blur function (Levin, Fergus, Durand *et al.* 2007; Joshi, Szeliski, and Kriegman 2008) or in the “glints” that are seen when shooting into the sun.



**Figure 2.20** Regular and zoom lens depth of field indicators.

The allowable depth variation in the scene that limits the circle of confusion to an acceptable number is commonly called the *depth of field* and is a function of both the focus distance and the aperture, as shown diagrammatically by many lens markings (Figure 2.20). Since this depth of field depends on the aperture diameter  $d$ , we also have to know how this varies with the commonly displayed *f-number*, which is usually denoted as  $f/\#$  or  $N$  and is defined as

$$f/\# = N = \frac{f}{d}, \quad (2.97)$$

where the focal length  $f$  and the aperture diameter  $d$  are measured in the same unit (say, millimeters).

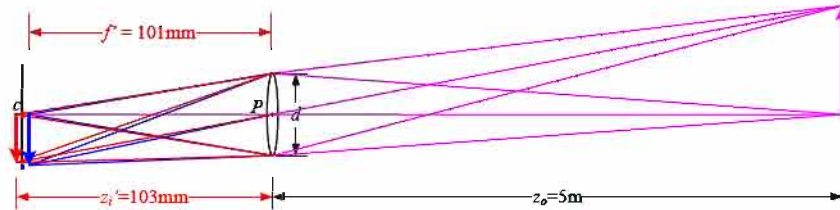
The usual way to write the *f-number* is to replace the  $\#$  in  $f/\#$  with the actual number, i.e.,  $f/1.4$ ,  $f/2$ ,  $f/2.8$ ,  $\dots$ ,  $f/22$ . (Alternatively, we can say  $N = 1.4$ , etc.) An easy way to interpret these numbers is to notice that dividing the focal length by the *f-number* gives us the diameter  $d$ , so these are just formulas for the aperture diameter.<sup>8</sup>

Notice that the usual progression for *f-numbers* is in *full stops*, which are multiples of  $\sqrt{2}$ , since this corresponds to doubling the area of the entrance pupil each time a smaller *f-number* is selected. (This doubling is also called changing the exposure by one *exposure value* or EV. It has the same effect on the amount of light reaching the sensor as doubling the exposure duration, e.g., from  $1/125$  to  $1/250$ , see Exercise 2.5.)

Now that you know how to convert between *f-numbers* and aperture diameters, you can construct your own plots for the depth of field as a function of focal length  $f$ , circle of confusion  $c$ , and focus distance  $z_o$ , as explained in Exercise 2.4 and see how well these match what you observe on actual lenses, such as those shown in Figure 2.20.

Of course, real lenses are not infinitely thin and therefore suffer from *geometric aberrations*, unless compound elements are used to correct for them. The classic five *Seidel aberrations*, which arise when using *third-order optics*, include spherical aberration, coma, astigmatism, curvature of field, and distortion (Möller 1988; Hecht 2001; Ray 2002).

<sup>8</sup> This also explains why, with zoom lenses, the *f-number* varies with the current zoom (focal length) setting.



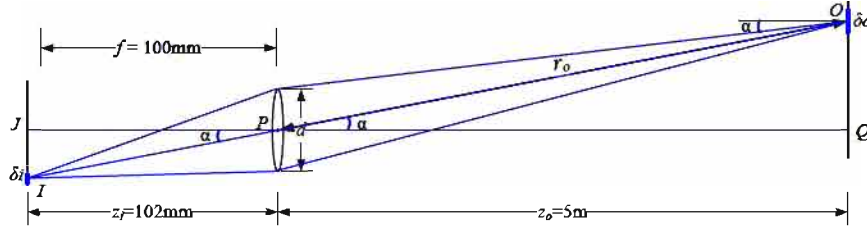
**Figure 2.21** In a lens subject to *chromatic aberration*, light at different wavelengths (e.g., the red and blue arrows) is focused with a different focal length  $f'$  and hence a different depth  $z'_i$ , resulting in both a geometric (in-plane) displacement and a loss of focus.

### Chromatic aberration

Because the index of refraction for glass varies slightly as a function of wavelength, simple lenses suffer from *chromatic aberration*, which is the tendency for light of different colors to focus at slightly different distances (and hence also with slightly different magnification factors), as shown in Figure 2.21. The wavelength-dependent magnification factor, i.e., the *transverse chromatic aberration*, can be modeled as a per-color radial distortion (Section 2.1.6) and, hence, calibrated using the techniques described in Section 6.3.5. The wavelength-dependent blur caused by *longitudinal chromatic aberration* can be calibrated using techniques described in Section 10.1.4. Unfortunately, the blur induced by longitudinal aberration can be harder to undo, as higher frequencies can get strongly attenuated and hence hard to recover.

In order to reduce chromatic and other kinds of aberrations, most photographic lenses today are *compound lenses* made of different glass elements (with different coatings). Such lenses can no longer be modeled as having a single *nodal point*  $P$  through which all of the rays must pass (when approximating the lens with a pinhole model). Instead, these lenses have both a *front nodal point*, through which the rays enter the lens, and a *rear nodal point*, through which they leave on their way to the sensor. In practice, only the location of the front nodal point is of interest when performing careful camera calibration, e.g., when determining the point around which to rotate to capture a parallax-free panorama (see Section 9.1.3).

Not all lenses, however, can be modeled as having a single nodal point. In particular, very wide-angle lenses such as fisheye lenses (Section 2.1.6) and certain *catadioptric* imaging systems consisting of lenses and curved mirrors (Baker and Nayar 1999) do not have a single point through which all of the acquired light rays pass. In such cases, it is preferable to explicitly construct a mapping function (look-up table) between pixel coordinates and 3D rays in space (Gremban, Thorpe, and Kanade 1988; Champlébourg, Lavallée, Sautot *et al.*



**Figure 2.22** The amount of light hitting a pixel of surface area  $\delta_i$  depends on the square of the ratio of the aperture diameter  $d$  to the focal length  $f$ , as well as the fourth power of the off-axis angle  $\alpha$  cosine,  $\cos^4 \alpha$ .

1992; Grossberg and Nayar 2001; Sturm and Ramalingam 2004; Tardif, Sturm, Trudeau *et al.* 2009), as mentioned in Section 2.1.6.

### Vignetting

Another property of real-world lenses is *vignetting*, which is the tendency for the brightness of the image to fall off towards the edge of the image.

Two kinds of phenomena usually contribute to this effect (Ray 2002). The first is called *natural vignetting* and is due to the foreshortening in the object surface, projected pixel, and lens aperture, as shown in Figure 2.22. Consider the light leaving the object surface patch of size  $\delta_o$  located at an *off-axis angle*  $\alpha$ . Because this patch is foreshortened with respect to the camera lens, the amount of light reaching the lens is reduced by a factor  $\cos \alpha$ . The amount of light reaching the lens is also subject to the usual  $1/r^2$  fall-off; in this case, the distance  $r_o = z_o / \cos \alpha$ . The actual area of the aperture through which the light passes is foreshortened by an additional factor  $\cos \alpha$ , i.e., the aperture as seen from point  $O$  is an ellipse of dimensions  $d \times d \cos \alpha$ . Putting all of these factors together, we see that the amount of light leaving  $O$  and passing through the aperture on its way to the image pixel located at  $I$  is proportional to

$$\frac{\delta_o \cos \alpha}{r_o^2} \pi \left( \frac{d}{2} \right)^2 \cos \alpha = \delta_o \frac{\pi d^2}{4 z_o^2} \cos^4 \alpha. \quad (2.98)$$

Since triangles  $\triangle OPQ$  and  $\triangle IPJ$  are similar, the projected areas of the object surface  $\delta_o$  and image pixel  $\delta_i$  are in the same (squared) ratio as  $z_o : z_i$ ,

$$\frac{\delta_o}{\delta_i} = \frac{z_o^2}{z_i^2}. \quad (2.99)$$

Putting these together, we obtain the final relationship between the amount of light reaching

pixel  $i$  and the aperture diameter  $d$ , the focusing distance  $z_i \approx f$ , and the off-axis angle  $\alpha$ ,

$$\delta o \frac{\pi d^2}{4 z_o^2} \cos^4 \alpha = \delta i \frac{\pi d^2}{4 z_i^2} \cos^4 \alpha \approx \delta i \frac{\pi}{4} \left( \frac{d}{f} \right)^2 \cos^4 \alpha, \quad (2.100)$$

which is called the *fundamental radiometric relation* between the scene radiance  $L$  and the light (irradiance)  $E$  reaching the pixel sensor,

$$E = L \frac{\pi}{4} \left( \frac{d}{f} \right)^2 \cos^4 \alpha, \quad (2.101)$$

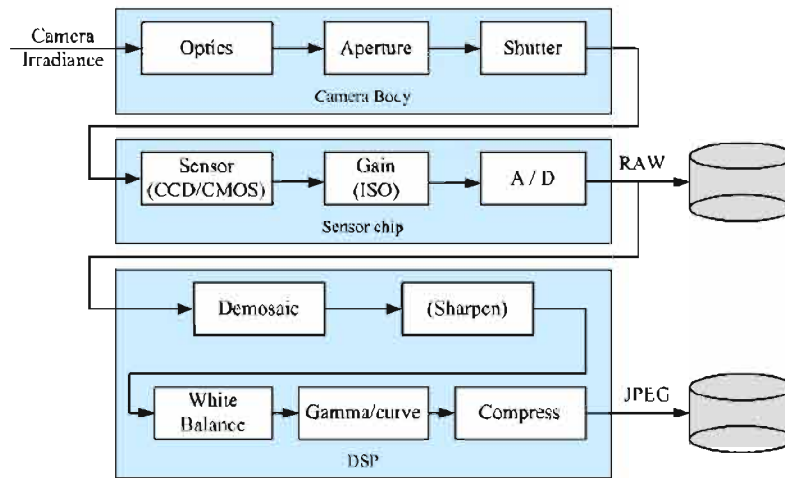
(Horn 1986; Nalwa 1993; Hecht 2001; Ray 2002). Notice in this equation how the amount of light depends on the pixel surface area (which is why the smaller sensors in point-and-shoot cameras are so much noisier than digital single lens reflex (SLR) cameras), the inverse square of the f-stop  $N = f/d$  (2.97), and the fourth power of the  $\cos^4 \alpha$  off-axis fall-off, which is the natural vignetting term.

The other major kind of vignetting, called *mechanical vignetting*, is caused by the internal occlusion of rays near the periphery of lens elements in a compound lens, and cannot easily be described mathematically without performing a full ray-tracing of the actual lens design.<sup>9</sup> However, unlike natural vignetting, mechanical vignetting can be decreased by reducing the camera aperture (increasing the f-number). It can also be calibrated (along with natural vignetting) using special devices such as integrating spheres, uniformly illuminated targets, or camera rotation, as discussed in Section 10.1.3.

## 2.3 The digital camera

After starting from one or more light sources, reflecting off one or more surfaces in the world, and passing through the camera's optics (lenses), light finally reaches the imaging sensor. How are the photons arriving at this sensor converted into the digital (R, G, B) values that we observe when we look at a digital image? In this section, we develop a simple model that accounts for the most important effects such as exposure (gain and shutter speed), non-linear mappings, sampling and aliasing, and noise. Figure 2.23, which is based on camera models developed by Healey and Kondepudy (1994); Tsin, Ramesh, and Kanade (2001); Liu, Szeliski, Kang *et al.* (2008), shows a simple version of the processing stages that occur in modern digital cameras. Chakrabarti, Scharstein, and Zickler (2009) developed a sophisticated 24-parameter model that is an even better match to the processing performed in today's cameras.

<sup>9</sup> There are some empirical models that work well in practice (Kang and Weiss 2000; Zheng, Lin, and Kang 2006).



**Figure 2.23** Image sensing pipeline, showing the various sources of noise as well as typical digital post-processing steps.

Light falling on an imaging sensor is usually picked up by an *active sensing area*, integrated for the duration of the exposure (usually expressed as the shutter speed in a fraction of a second, e.g.,  $\frac{1}{125}$ ,  $\frac{1}{60}$ ,  $\frac{1}{30}$ ), and then passed to a set of *sense amplifiers*. The two main kinds of sensor used in digital still and video cameras today are charge-coupled device (CCD) and complementary metal oxide on silicon (CMOS).

In a CCD, photons are accumulated in each active *well* during the exposure time. Then, in a *transfer* phase, the charges are transferred from well to well in a kind of “bucket brigade” until they are deposited at the sense amplifiers, which amplify the signal and pass it to an analog-to-digital converter (ADC).<sup>10</sup> Older CCD sensors were prone to *blooming*, when charges from one over-exposed pixel spilled into adjacent ones, but most newer CCDs have anti-blooming technology (“troughs” into which the excess charge can spill).

In CMOS, the photons hitting the sensor directly affect the conductivity (or gain) of a photodetector, which can be selectively gated to control exposure duration, and locally amplified before being read out using a multiplexing scheme. Traditionally, CCD sensors outperformed CMOS in quality sensitive applications, such as digital SLRs, while CMOS was better for low-power applications, but today CMOS is used in most digital cameras.

The main factors affecting the performance of a digital image sensor are the shutter speed, sampling pitch, fill factor, chip size, analog gain, sensor noise, and the resolution (and quality)

<sup>10</sup> In digital still cameras, a complete frame is captured and then read out sequentially at once. However, if video is being captured, a *rolling shutter*, which exposes and transfers each line separately, is often used. In older video cameras, the even fields (lines) were scanned first, followed by the odd fields, in a process that is called *interlacing*.



of the analog-to-digital converter. Many of the actual values for these parameters can be read from the EXIF tags embedded with digital images, while others can be obtained from the camera manufacturers' specification sheets or from camera review or calibration Web sites.<sup>11</sup>

**Shutter speed.** The shutter speed (exposure time) directly controls the amount of light reaching the sensor and, hence, determines if images are under- or over-exposed. (For bright scenes, where a large aperture or slow shutter speed are desired to get a shallow depth of field or motion blur, *neutral density filters* are sometimes used by photographers.) For dynamic scenes, the shutter speed also determines the amount of *motion blur* in the resulting picture. Usually, a higher shutter speed (less motion blur) makes subsequent analysis easier (see Section 10.3 for techniques to remove such blur). However, when video is being captured for display, some motion blur may be desirable to avoid stroboscopic effects.

**Sampling pitch.** The sampling pitch is the physical spacing between adjacent sensor cells on the imaging chip. A sensor with a smaller sampling pitch has a higher *sampling density* and hence provides a higher *resolution* (in terms of pixels) for a given active chip area. However, a smaller pitch also means that each sensor has a smaller area and cannot accumulate as many photons; this makes it not as *light sensitive* and more prone to noise.

**Fill factor.** The fill factor is the active sensing area size as a fraction of the theoretically available sensing area (the product of the horizontal and vertical sampling pitches). Higher fill factors are usually preferable, as they result in more light capture and less *aliasing* (see Section 2.3.1). However, this must be balanced with the need to place additional electronics between the active sense areas. The fill factor of a camera can be determined empirically using a photometric camera calibration process (see Section 10.1.4).

**Chip size.** Video and point-and-shoot cameras have traditionally used small chip areas ( $\frac{1}{4}$ -inch to  $\frac{1}{2}$ -inch sensors<sup>12</sup>), while digital SLR cameras try to come closer to the traditional size of a 35mm film frame.<sup>13</sup> When overall device size is not important, having a larger chip size is preferable, since each sensor cell can be more photo-sensitive. (For compact cameras, a smaller chip means that all of the optics can be shrunk down proportionately.) However,

<sup>11</sup> <http://www.clarkvision.com/imagedetail/digital.sensor.performance.summary/>.

<sup>12</sup> These numbers refer to the "tube diameter" of the old vidicon tubes used in video cameras ([http://www.dpreview.com/learn/?/Glossary/Camera\\_System/sensor\\_sizes\\_01.htm](http://www.dpreview.com/learn/?/Glossary/Camera_System/sensor_sizes_01.htm)). The 1/2.5" sensor on the Canon SD800 camera actually measures 5.76mm × 4.29mm, i.e., a sixth of the size (on side) of a 35mm full-frame (36mm × 24mm) DSLR sensor.

<sup>13</sup> When a DSLR chip does not fill the 35mm full frame, it results in a *multiplier effect* on the lens focal length. For example, a chip that is only 0.6 the dimension of a 35mm frame will make a 50mm lens image the same angular extent as a  $50/0.6 = 50 \times 1.6 = 80$ mm lens, as demonstrated in (2.60).

larger chips are more expensive to produce, not only because fewer chips can be packed into each wafer, but also because the probability of a chip defect goes up linearly with the chip area.

**Analog gain.** Before analog-to-digital conversion, the sensed signal is usually boosted by a *sense amplifier*. In video cameras, the gain on these amplifiers was traditionally controlled by *automatic gain control* (AGC) logic, which would adjust these values to obtain a good overall exposure. In newer digital still cameras, the user now has some additional control over this gain through the *ISO setting*, which is typically expressed in ISO standard units such as 100, 200, or 400. Since the automated exposure control in most cameras also adjusts the aperture and shutter speed, setting the ISO manually removes one degree of freedom from the camera's control, just as manually specifying aperture and shutter speed does. In theory, a higher gain allows the camera to perform better under low light conditions (less motion blur due to long exposure times when the aperture is already maxed out). In practice, however, higher ISO settings usually amplify the *sensor noise*.

**Sensor noise.** Throughout the whole sensing process, noise is added from various sources, which may include *fixed pattern noise*, *dark current noise*, *shot noise*, *amplifier noise* and *quantization noise* (Healey and Kondepudy 1994; Ts'in, Ramesh, and Kanade 2001). The final amount of noise present in a sampled image depends on all of these quantities, as well as the incoming light (controlled by the scene radiance and aperture), the exposure time, and the sensor gain. Also, for low light conditions where the noise is due to low photon counts, a Poisson model of noise may be more appropriate than a Gaussian model.

As discussed in more detail in Section 10.1.1, Liu, Szeliski, Kang *et al.* (2008) use this model, along with an empirical database of camera response functions (CRFs) obtained by Grossberg and Nayar (2004), to estimate the *noise level function* (NLF) for a given image, which predicts the overall noise variance at a given pixel as a function of its brightness (a separate NLF is estimated for each color channel). An alternative approach, when you have access to the camera before taking pictures, is to pre-calibrate the NLF by taking repeated shots of a scene containing a variety of colors and luminances, such as the Macbeth Color Chart shown in Figure 10.3b (McCamy, Marcus, and Davidson 1976). (When estimating the variance, be sure to throw away or downweight pixels with large gradients, as small shifts between exposures will affect the sensed values at such pixels.) Unfortunately, the pre-calibration process may have to be repeated for different exposure times and gain settings because of the complex interactions occurring within the sensing system.

In practice, most computer vision algorithms, such as image denoising, edge detection, and stereo matching, all benefit from at least a rudimentary estimate of the noise level. Barring the ability to pre-calibrate the camera or to take repeated shots of the same scene, the simplest



approach is to look for regions of near-constant value and to estimate the noise variance in such regions (Liu, Szeliski, Kang *et al.* 2008).

**ADC resolution.** The final step in the analog processing chain occurring within an imaging sensor is the *analog to digital conversion* (ADC). While a variety of techniques can be used to implement this process, the two quantities of interest are the *resolution* of this process (how many bits it yields) and its noise level (how many of these bits are useful in practice). For most cameras, the number of bits quoted (eight bits for compressed JPEG images and a nominal 16 bits for the RAW formats provided by some DSLRs) exceeds the actual number of usable bits. The best way to tell is to simply calibrate the noise of a given sensor, e.g., by taking repeated shots of the same scene and plotting the estimated noise as a function of brightness (Exercise 2.6).

**Digital post-processing.** Once the irradiance values arriving at the sensor have been converted to digital bits, most cameras perform a variety of *digital signal processing* (DSP) operations to enhance the image before compressing and storing the pixel values. These include color filter array (CFA) demosaicing, white point setting, and mapping of the luminance values through a *gamma function* to increase the perceived dynamic range of the signal. We cover these topics in Section 2.3.2 but, before we do, we return to the topic of aliasing, which was mentioned in connection with sensor array fill factors.

### 2.3.1 Sampling and aliasing

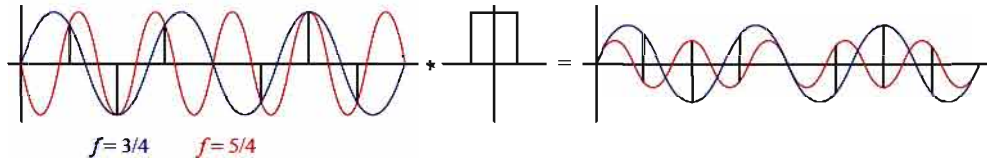
What happens when a field of light impinging on the image sensor falls onto the active sense areas in the imaging chip? The photons arriving at each active cell are integrated and then digitized. However, if the fill factor on the chip is small and the signal is not otherwise *band-limited*, visually displeasing aliasing can occur.

To explore the phenomenon of aliasing, let us first look at a one-dimensional signal (Figure 2.24), in which we have two sine waves, one at a frequency of  $f = 3/4$  and the other at  $f = 5/4$ . If we sample these two signals at a frequency of  $f = 2$ , we see that they produce the same samples (shown in black), and so we say that they are *aliased*.<sup>14</sup> Why is this a bad effect? In essence, we can no longer reconstruct the original signal, since we do not know which of the two original frequencies was present.

In fact, Shannon's Sampling Theorem shows that the minimum sampling (Oppenheim and Schaffer 1996; Oppenheim, Schaffer, and Buck 1999) rate required to reconstruct a signal

---

<sup>14</sup> An alias is an alternate name for someone, so the sampled signal corresponds to two different *aliases*.



**Figure 2.24** Aliasing of a one-dimensional signal: The blue sine wave at  $f = 3/4$  and the red sine wave at  $f = 5/4$  have the same digital samples, when sampled at  $f = 2$ . Even after convolution with a 100% fill factor box filter, the two signals, while no longer of the same magnitude, are still aliased in the sense that the sampled red signal looks like an inverted lower magnitude version of the blue signal. (The image on the right is scaled up for better visibility. The actual sine magnitudes are 30% and  $-18\%$  of their original values.)

from its instantaneous samples must be at least twice the highest frequency,<sup>15</sup>

$$f_s \geq 2f_{\max}. \quad (2.102)$$

The maximum frequency in a signal is known as the *Nyquist frequency* and the inverse of the minimum sampling frequency  $r_s = 1/f_s$  is known as the *Nyquist rate*.

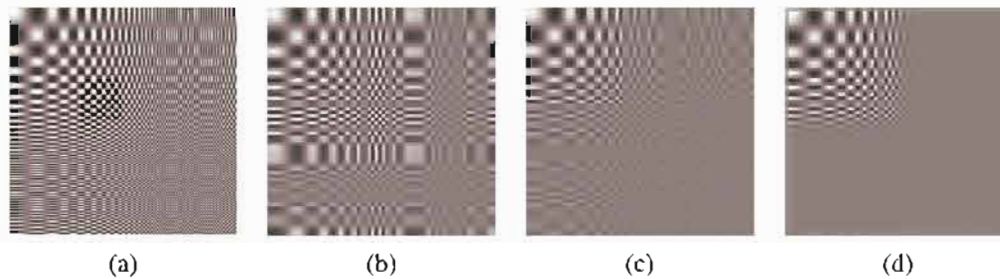
However, you may ask, since an imaging chip actually *averages* the light field over a finite area, are the results on point sampling still applicable? Averaging over the sensor area does tend to attenuate some of the higher frequencies. However, even if the fill factor is 100%, as in the right image of Figure 2.24, frequencies above the Nyquist limit (half the sampling frequency) still produce an aliased signal, although with a smaller magnitude than the corresponding band-limited signals.

A more convincing argument as to why aliasing is bad can be seen by downsampling a signal using a poor quality filter such as a box (square) filter. Figure 2.25 shows a high-frequency *chirp* image (so called because the frequencies increase over time), along with the results of sampling it with a 25% fill-factor area sensor, a 100% fill-factor sensor, and a high-quality 9-tap filter. Additional examples of downsampling (*decimation*) filters can be found in Section 3.5.2 and Figure 3.30.

The best way to predict the amount of aliasing that an imaging system (or even an image processing algorithm) will produce is to estimate the *point spread function* (PSF), which represents the response of a particular pixel sensor to an ideal point light source. The PSF is a combination (convolution) of the blur induced by the optical system (lens) and the finite integration area of a chip sensor.<sup>16</sup>

<sup>15</sup> The actual theorem states that  $f_s$  must be at least twice the signal *bandwidth* but, since we are not dealing with modulated signals such as radio waves during image capture, the maximum frequency suffices.

<sup>16</sup> Imaging chips usually interpose an optical *anti-aliasing filter* just before the imaging chip to reduce or control the amount of aliasing.



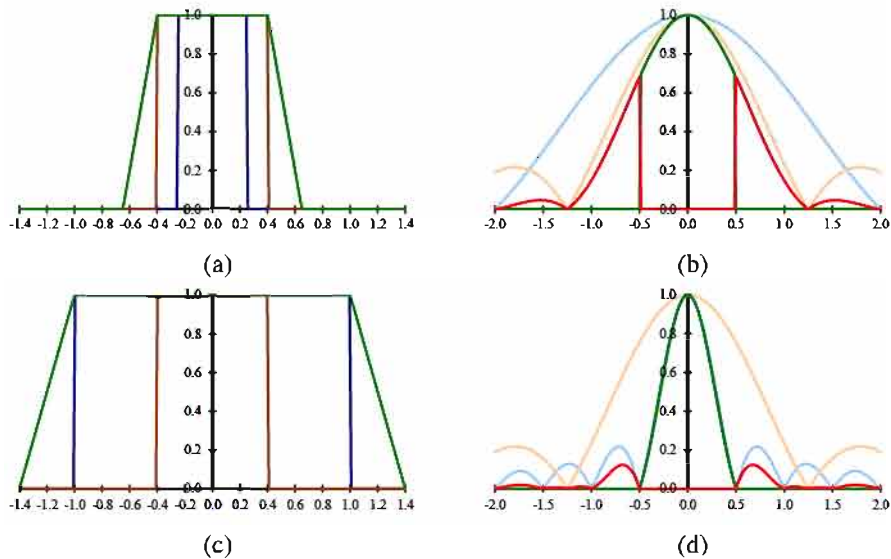
**Figure 2.25** Aliasing of a two-dimensional signal: (a) original full-resolution image; (b) downsampled  $4\times$  with a 25% fill factor box filter; (c) downsampled  $4\times$  with a 100% fill factor box filter; (d) downsampled  $4\times$  with a high-quality 9-tap filter. Notice how the higher frequencies are aliased into visible frequencies with the lower quality filters, while the 9-tap filter completely removes these higher frequencies.

If we know the blur function of the lens and the fill factor (sensor area shape and spacing) for the imaging chip (plus, optionally, the response of the anti-aliasing filter), we can convolve these (as described in Section 3.2) to obtain the PSF. Figure 2.26a shows the one-dimensional cross-section of a PSF for a lens whose blur function is assumed to be a disc of a radius equal to the pixel spacing  $s$  plus a sensing chip whose horizontal fill factor is 80%. Taking the Fourier transform of this PSF (Section 3.4), we obtain the *modulation transfer function* (MTF), from which we can estimate the amount of aliasing as the area of the Fourier magnitude outside the  $f \leq f_s$  Nyquist frequency.<sup>17</sup> If we de-focus the lens so that the blur function has a radius of  $2s$  (Figure 2.26c), we see that the amount of aliasing decreases significantly, but so does the amount of image detail (frequencies closer to  $f = f_s$ ).

Under laboratory conditions, the PSF can be estimated (to pixel precision) by looking at a point light source such as a pin hole in a black piece of cardboard lit from behind. However, this PSF (the actual image of the pin hole) is only accurate to a pixel resolution and, while it can model larger blur (such as blur caused by defocus), it cannot model the sub-pixel shape of the PSF and predict the amount of aliasing. An alternative technique, described in Section 10.1.4, is to look at a calibration pattern (e.g., one consisting of slanted step edges (Reichenbach, Park, and Narayanswamy 1991; Williams and Burns 2001; Joshi, Szeliski, and Kriegman 2008)) whose ideal appearance can be re-synthesized to sub-pixel precision.

In addition to occurring during image acquisition, aliasing can also be introduced in various image processing operations, such as resampling, upsampling, and downsampling. Sections 3.4 and 3.5.2 discuss these issues and show how careful selection of filters can reduce

<sup>17</sup> The complex Fourier transform of the PSF is actually called the *optical transfer function* (OTF) (Williams 1999). Its magnitude is called the *modulation transfer function* (MTF) and its phase is called the *phase transfer function* (PTF).



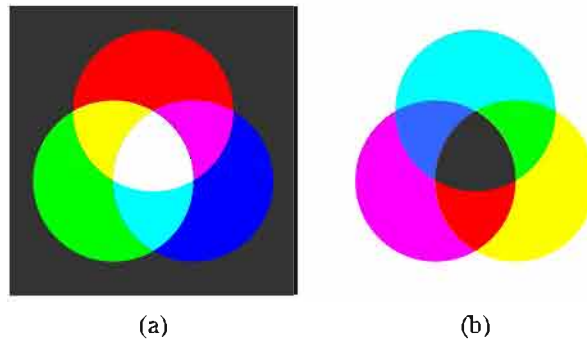
**Figure 2.26** Sample point spread functions (PSF): The diameter of the blur disc (blue) in (a) is equal to half the pixel spacing, while the diameter in (c) is twice the pixel spacing. The horizontal fill factor of the sensing chip is 80% and is shown in brown. The convolution of these two kernels gives the point spread function, shown in green. The Fourier response of the PSF (the MTF) is plotted in (b) and (d). The area above the Nyquist frequency where aliasing occurs is shown in red.

the amount of aliasing that operations inject.

### 2.3.2 Color

In Section 2.2, we saw how lighting and surface reflections are functions of wavelength. When the incoming light hits the imaging sensor, light from different parts of the spectrum is somehow integrated into the discrete red, green, and blue (RGB) color values that we see in a digital image. How does this process work and how can we analyze and manipulate color values?

You probably recall from your childhood days the magical process of mixing paint colors to obtain new ones. You may recall that blue+yellow makes green, red+blue makes purple, and red+green makes brown. If you revisited this topic at a later age, you may have learned that the proper *subtractive* primaries are actually cyan (a light blue-green), magenta (pink), and yellow (Figure 2.27b), although black is also often used in four-color printing (CMYK). (If you ever subsequently took any painting classes, you learned that colors can have even



**Figure 2.27** Primary and secondary colors: (a) additive colors red, green, and blue can be mixed to produce cyan, magenta, yellow, and white; (b) subtractive colors cyan, magenta, and yellow can be mixed to produce red, green, blue, and black.

more fanciful names, such as alizarin crimson, cerulean blue, and chartreuse.) The subtractive colors are called subtractive because pigments in the paint absorb certain wavelengths in the color spectrum.

Later on, you may have learned about the *additive* primary colors (red, green, and blue) and how they can be added (with a slide projector or on a computer monitor) to produce cyan, magenta, yellow, white, and all the other colors we typically see on our TV sets and monitors (Figure 2.27a).

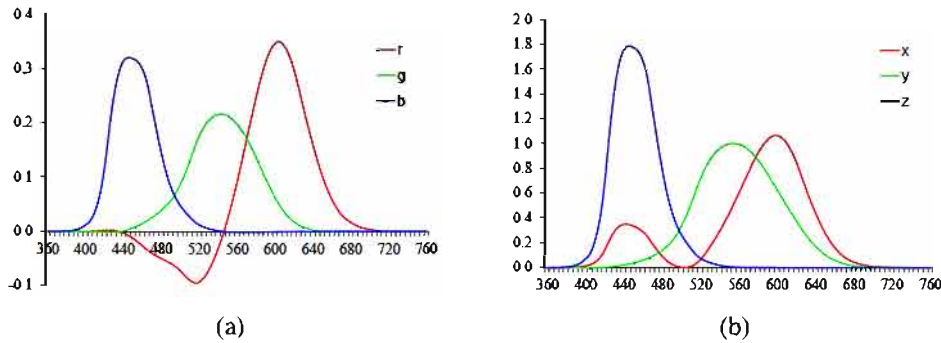
Through what process is it possible for two different colors, such as red and green, to interact to produce a third color like yellow? Are the wavelengths somehow mixed up to produce a new wavelength?

You probably know that the correct answer has nothing to do with physically mixing wavelengths. Instead, the existence of three primaries is a result of the *tri-stimulus* (or *tri-chromatic*) nature of the human visual system, since we have three different kinds of cone, each of which responds selectively to a different portion of the color spectrum (Glassner 1995; Wyszecki and Stiles 2000; Fairchild 2005; Reinhard, Ward, Pattanaik *et al.* 2005; Livingstone 2008).<sup>18</sup> Note that for machine vision applications, such as remote sensing and terrain classification, it is preferable to use many more wavelengths. Similarly, surveillance applications can often benefit from sensing in the near-infrared (NIR) range.

### CIE RGB and XYZ

To test and quantify the tri-chromatic theory of perception, we can attempt to reproduce all *monochromatic* (single wavelength) colors as a mixture of three suitably chosen primaries.

<sup>18</sup> See also Mark Fairchild's Web page, [http://www.cis.rit.edu/fairchild/WhyIsColor/books\\_links.html](http://www.cis.rit.edu/fairchild/WhyIsColor/books_links.html).



**Figure 2.28** Standard CIE color matching functions: (a)  $\bar{r}(\lambda)$ ,  $\bar{g}(\lambda)$ ,  $\bar{b}(\lambda)$  color spectra obtained from matching pure colors to the R=700.0nm, G=546.1nm, and B=435.8nm primaries; (b)  $\bar{x}(\lambda)$ ,  $\bar{y}(\lambda)$ ,  $\bar{z}(\lambda)$  color matching functions, which are linear combinations of the  $(\bar{r}(\lambda), \bar{g}(\lambda), \bar{b}(\lambda))$  spectra.

(Pure wavelength light can be obtained using either a prism or specially manufactured color filters.) In the 1930s, the Commission Internationale d’Eclairage (CIE) standardized the RGB representation by performing such *color matching* experiments using the primary colors of red (700.0nm wavelength), green (546.1nm), and blue (435.8nm).

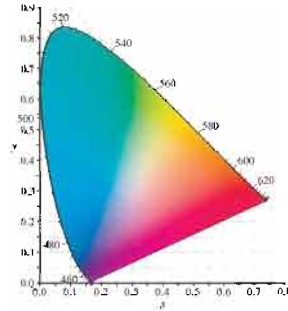
Figure 2.28 shows the results of performing these experiments with a *standard observer*, i.e., averaging perceptual results over a large number of subjects. You will notice that for certain pure spectra in the blue–green range, a *negative* amount of red light has to be added, i.e., a certain amount of red has to be added to the color being matched in order to get a color match. These results also provided a simple explanation for the existence of *metamers*, which are colors with different spectra that are perceptually indistinguishable. Note that two fabrics or paint colors that are metamers under one light may no longer be so under different lighting.

Because of the problem associated with mixing negative light, the CIE also developed a new color space called XYZ, which contains all of the pure spectral colors within its positive octant. (It also maps the Y axis to the *luminance*, i.e., perceived relative brightness, and maps pure white to a diagonal (equal-valued) vector.) The transformation from RGB to XYZ is given by

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \frac{1}{0.17697} \begin{bmatrix} 0.49 & 0.31 & 0.20 \\ 0.17697 & 0.81240 & 0.01063 \\ 0.00 & 0.01 & 0.99 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}. \quad (2.103)$$

While the official definition of the CIE XYZ standard has the matrix normalized so that the Y value corresponding to pure red is 1, a more commonly used form is to omit the leading





**Figure 2.29** CIE chromaticity diagram, showing colors and their corresponding  $(x, y)$  values. Pure spectral colors are arranged around the outside of the curve.

fraction, so that the second row adds up to one, i.e., the RGB triplet  $(1, 1, 1)$  maps to a  $Y$  value of 1. Linearly blending the  $(\bar{r}(\lambda), \bar{g}(\lambda), \bar{b}(\lambda))$  curves in Figure 2.28a according to (2.103), we obtain the resulting  $(\bar{x}(\lambda), \bar{y}(\lambda), \bar{z}(\lambda))$  curves shown in Figure 2.28b. Notice how all three spectra (color matching functions) now have only positive values and how the  $\bar{y}(\lambda)$  curve matches that of the luminance perceived by humans.

If we divide the XYZ values by the sum of  $X+Y+Z$ , we obtain the *chromaticity coordinates*

$$x = \frac{X}{X+Y+Z}, \quad y = \frac{Y}{X+Y+Z}, \quad z = \frac{Z}{X+Y+Z}, \quad (2.104)$$

which sum up to 1. The chromaticity coordinates discard the absolute intensity of a given color sample and just represent its pure color. If we sweep the monochromatic color  $\lambda$  parameter in Figure 2.28b from  $\lambda = 380\text{nm}$  to  $\lambda = 800\text{nm}$ , we obtain the familiar *chromaticity diagram* shown in Figure 2.29. This figure shows the  $(x, y)$  value for every color value perceivable by most humans. (Of course, the CMYK reproduction process in this book does not actually span the whole gamut of perceivable colors.) The outer curved rim represents where all of the pure monochromatic color values map in  $(x, y)$  space, while the lower straight line, which connects the two endpoints, is known as the *purple line*.

A convenient representation for color values, when we want to tease apart luminance and chromaticity, is therefore  $Yxy$  (luminance plus the two most distinctive chrominance components).

### **L\*a\*b\* color space**

While the XYZ color space has many convenient properties, including the ability to separate luminance from chrominance, it does not actually predict how well humans perceive *differences* in color or luminance.

Because the response of the human visual system is roughly logarithmic (we can perceive *relative* luminance differences of about 1%), the CIE defined a non-linear re-mapping of the XYZ space called  $L^*a^*b^*$  (also sometimes called CIELAB), where differences in luminance or chrominance are more perceptually uniform.<sup>19</sup>

The  $L^*$  component of *lightness* is defined as

$$L^* = 116f\left(\frac{Y}{Y_n}\right), \quad (2.105)$$

where  $Y_n$  is the luminance value for nominal white (Fairchild 2005) and

$$f(t) = \begin{cases} t^{1/3} & t > \delta^3 \\ t/(3\delta^2) + 2\delta/3 & \text{else,} \end{cases} \quad (2.106)$$

is a finite-slope approximation to the cube root with  $\delta = 6/29$ . The resulting 0...100 scale roughly measures equal amounts of lightness perceptibility.

In a similar fashion, the  $a^*$  and  $b^*$  components are defined as

$$a^* = 500 \left[ f\left(\frac{X}{X_n}\right) - f\left(\frac{Y}{Y_n}\right) \right] \quad \text{and} \quad b^* = 200 \left[ f\left(\frac{Y}{Y_n}\right) - f\left(\frac{Z}{Z_n}\right) \right], \quad (2.107)$$

where again,  $(X_n, Y_n, Z_n)$  is the measured white point. Figure 2.32i–k show the  $L^*a^*b^*$  representation for a sample color image.

### Color cameras

While the preceding discussion tells us how we can uniquely describe the perceived tri-stimulus description of any color (spectral distribution), it does not tell us how RGB still and video cameras actually work. Do they just measure the amount of light at the nominal wavelengths of red (700.0nm), green (546.1nm), and blue (435.8nm)? Do color monitors just emit exactly these wavelengths and, if so, how can they emit negative red light to reproduce colors in the cyan range?

In fact, the design of RGB video cameras has historically been based around the availability of colored phosphors that go into television sets. When standard-definition color television was invented (NTSC), a mapping was defined between the RGB values that would drive the three color guns in the cathode ray tube (CRT) and the XYZ values that unambiguously define perceived color (this standard was called ITU-R BT.601). With the advent of HDTV and newer monitors, a new standard called ITU-R BT.709 was created, which specifies the XYZ

<sup>19</sup> Another perceptually motivated color space called  $L^*u^*v^*$  was developed and standardized simultaneously (Fairchild 2005).



values of each of the color primaries,

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} 0.412453 & 0.357580 & 0.180423 \\ 0.212671 & 0.715160 & 0.072169 \\ 0.019334 & 0.119193 & 0.950227 \end{bmatrix} \begin{bmatrix} R_{709} \\ G_{709} \\ B_{709} \end{bmatrix}. \quad (2.108)$$

In practice, each color camera integrates light according to the *spectral response function* of its red, green, and blue sensors,

$$\begin{aligned} R &= \int L(\lambda)S_R(\lambda)d\lambda, \\ G &= \int L(\lambda)S_G(\lambda)d\lambda, \\ B &= \int L(\lambda)S_B(\lambda)d\lambda, \end{aligned} \quad (2.109)$$

where  $L(\lambda)$  is the incoming spectrum of light at a given pixel and  $\{S_R(\lambda), S_G(\lambda), S_B(\lambda)\}$  are the red, green, and blue *spectral sensitivities* of the corresponding sensors.

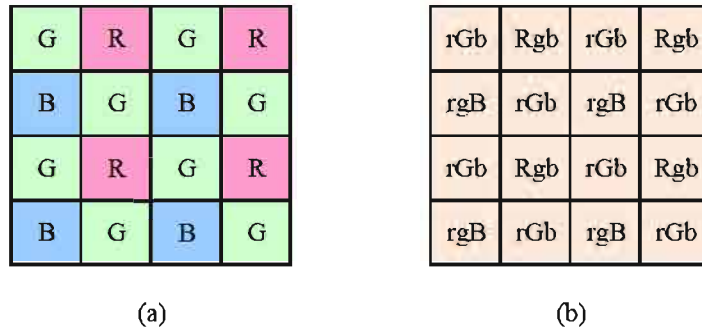
Can we tell what spectral sensitivities the cameras actually have? Unless the camera manufacturer provides us with this data or we observe the response of the camera to a whole spectrum of monochromatic lights, these sensitivities are *not* specified by a standard such as BT.709. Instead, all that matters is that the tri-stimulus values for a given color produce the specified RGB values. The manufacturer is free to use sensors with sensitivities that do not match the standard XYZ definitions, so long as they can later be converted (through a linear transform) to the standard colors.

Similarly, while TV and computer monitors are supposed to produce RGB values as specified by Equation (2.108), there is no reason that they cannot use digital logic to transform the incoming RGB values into different signals to drive each of the color channels. Properly calibrated monitors make this information available to software applications that perform *color management*, so that colors in real life, on the screen, and on the printer all match as closely as possible.

### Color filter arrays

While early color TV cameras used three *vidicons* (tubes) to perform their sensing and later cameras used three separate RGB sensing chips, most of today's digital still and video cameras use a *color filter array* (CFA), where alternating sensors are covered by different colored filters.<sup>20</sup>

<sup>20</sup> A newer chip design by Foveon (<http://www.foveon.com>) stacks the red, green, and blue sensors beneath each other, but it has not yet gained widespread adoption.



**Figure 2.30** Bayer RGB pattern: (a) color filter array layout; (b) interpolated pixel values, with unknown (guessed) values shown as lower case.

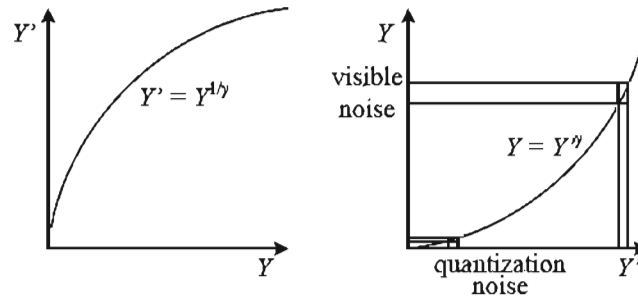
The most commonly used pattern in color cameras today is the *Bayer pattern* (Bayer 1976), which places green filters over half of the sensors (in a checkerboard pattern), and red and blue filters over the remaining ones (Figure 2.30). The reason that there are twice as many green filters as red and blue is because the luminance signal is mostly determined by green values and the visual system is much more sensitive to high frequency detail in luminance than in chrominance (a fact that is exploited in color image compression—see Section 2.3.3). The process of *interpolating* the missing color values so that we have valid RGB values for all the pixels is known as *demosaicing* and is covered in detail in Section 10.3.1.

Similarly, color LCD monitors typically use alternating stripes of red, green, and blue filters placed in front of each liquid crystal active area to simulate the experience of a full color display. As before, because the visual system has higher resolution (acuity) in luminance than chrominance, it is possible to digitally pre-filter RGB (and monochrome) images to enhance the perception of crispness (Betrissey, Blinn, Dresevic *et al.* 2000; Platt 2000).

### Color balance

Before encoding the sensed RGB values, most cameras perform some kind of *color balancing* operation in an attempt to move the white point of a given image closer to pure white (equal RGB values). If the color system and the illumination are the same (the BT.709 system uses the daylight illuminant  $D_{65}$  as its reference white), the change may be minimal. However, if the illuminant is strongly colored, such as incandescent indoor lighting (which generally results in a yellow or orange hue), the compensation can be quite significant.

A simple way to perform color correction is to multiply each of the RGB values by a different factor (i.e., to apply a diagonal matrix transform to the RGB color space). More complicated transforms, which are sometimes the result of mapping to XYZ space and back,



**Figure 2.31** Gamma compression: (a) The relationship between the input signal luminance  $Y$  and the transmitted signal  $Y'$  is given by  $Y' = Y^{1/\gamma}$ . (b) At the receiver, the signal  $Y'$  is exponentiated by the factor  $\gamma$ ,  $\hat{Y} = Y'^{\gamma}$ . Noise introduced during transmission is squashed in the dark regions, which corresponds to the more noise-sensitive region of the visual system.

actually perform a *color twist*, i.e., they use a general  $3 \times 3$  color transform matrix.<sup>21</sup> Exercise 2.9 has you explore some of these issues.

### Gamma

In the early days of black and white television, the phosphors in the CRT used to display the TV signal responded non-linearly to their input voltage. The relationship between the voltage and the resulting brightness was characterized by a number called *gamma* ( $\gamma$ ), since the formula was roughly

$$B = V^{\gamma}, \quad (2.110)$$

with a  $\gamma$  of about 2.2. To compensate for this effect, the electronics in the TV camera would pre-map the sensed luminance  $Y$  through an inverse gamma,

$$Y' = Y^{\frac{1}{\gamma}}, \quad (2.111)$$

with a typical value of  $\frac{1}{\gamma} = 0.45$ .

The mapping of the signal through this non-linearity before transmission had a beneficial side effect: noise added during transmission (remember, these were analog days!) would be reduced (after applying the gamma at the receiver) in the darker regions of the signal where it was more visible (Figure 2.31).<sup>22</sup> (Remember that our visual system is roughly sensitive to relative differences in luminance.)

<sup>21</sup> Those of you old enough to remember the early days of color television will naturally think of the *hue* adjustment knob on the television set, which could produce truly bizarre results.

<sup>22</sup> A related technique called *companding* was the basis of the Dolby noise reduction systems used with audio tapes.

When color television was invented, it was decided to separately pass the red, green, and blue signals through the same gamma non-linearity before combining them for encoding. Today, even though we no longer have analog noise in our transmission systems, signals are still quantized during compression (see Section 2.3.3), so applying inverse gamma to sensed values is still useful.

Unfortunately, for both computer vision and computer graphics, the presence of gamma in images is often problematic. For example, the proper simulation of radiometric phenomena such as shading (see Section 2.2 and Equation (2.87)) occurs in a linear radiance space. Once all of the computations have been performed, the appropriate gamma should be applied before display. Unfortunately, many computer graphics systems (such as shading models) operate directly on RGB values and display these values directly. (Fortunately, newer color imaging standards such as the 16-bit sRGB use a linear space, which makes this less of a problem (Glassner 1995).)

In computer vision, the situation can be even more daunting. The accurate determination of surface normals, using a technique such as photometric stereo (Section 12.1.1) or even a simpler operation such as accurate image deblurring, require that the measurements be in a linear space of intensities. Therefore, it is imperative when performing detailed quantitative computations such as these to first undo the gamma and the per-image color re-balancing in the sensed color values. Chakrabarti, Scharstein, and Zickler (2009) develop a sophisticated 24-parameter model that is a good match to the processing performed by today's digital cameras; they also provide a database of color images you can use for your own testing.<sup>23</sup>

For other vision applications, however, such as feature detection or the matching of signals in stereo and motion estimation, this linearization step is often not necessary. In fact, determining whether it is necessary to undo gamma can take some careful thinking, e.g., in the case of compensating for exposure variations in image stitching (see Exercise 2.7).

If all of these processing steps sound confusing to model, they are. Exercise 2.10 has you try to tease apart some of these phenomena using empirical investigation, i.e., taking pictures of color charts and comparing the RAW and JPEG compressed color values.

### Other color spaces

While RGB and XYZ are the primary color spaces used to describe the spectral content (and hence tri-stimulus response) of color signals, a variety of other representations have been developed both in video and still image coding and in computer graphics.

The earliest color representation developed for video transmission was the YIQ standard developed for NTSC video in North America and the closely related YUV standard developed for PAL in Europe. In both of these cases, it was desired to have a *luma* channel Y (so called

---

<sup>23</sup> <http://vision.middlebury.edu/color/>.

since it only roughly mimics true luminance) that would be comparable to the regular black-and-white TV signal, along with two lower frequency *chroma* channels.

In both systems, the Y signal (or more appropriately, the Y' luma signal since it is gamma compressed) is obtained from

$$Y'_{601} = 0.299R' + 0.587G' + 0.114B', \quad (2.112)$$

where R'G'B' is the triplet of gamma-compressed color components. When using the newer color definitions for HDTV in BT.709, the formula is

$$Y'_{709} = 0.2125R' + 0.7154G' + 0.0721B'. \quad (2.113)$$

The UV components are derived from scaled versions of  $(B' - Y')$  and  $(R' - Y')$ , namely,

$$U = 0.492111(B' - Y') \text{ and } V = 0.877283(R' - Y'), \quad (2.114)$$

whereas the IQ components are the UV components rotated through an angle of  $33^\circ$ . In composite (NTSC and PAL) video, the chroma signals were then low-pass filtered horizontally before being modulated and superimposed on top of the Y' luma signal. Backward compatibility was achieved by having older black-and-white TV sets effectively ignore the high-frequency chroma signal (because of slow electronics) or, at worst, superimposing it as a high-frequency pattern on top of the main signal.

While these conversions were important in the early days of computer vision, when frame grabbers would directly digitize the composite TV signal, today all digital video and still image compression standards are based on the newer YCbCr conversion. YCbCr is closely related to YUV (the  $C_b$  and  $C_r$  signals carry the blue and red color difference signals and have more useful mnemonics than UV) but uses different scale factors to fit within the eight-bit range available with digital signals.

For video, the Y' signal is re-scaled to fit within the [16...235] range of values, while the Cb and Cr signals are scaled to fit within [16...240] (Gomes and Velho 1997; Fairchild 2005). For still images, the JPEG standard uses the full eight-bit range with no reserved values,

$$\begin{bmatrix} Y' \\ C_b \\ C_r \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.168736 & -0.331264 & 0.5 \\ 0.5 & -0.418688 & -0.081312 \end{bmatrix} \begin{bmatrix} R' \\ G' \\ B' \end{bmatrix} + \begin{bmatrix} 0 \\ 128 \\ 128 \end{bmatrix}, \quad (2.115)$$

where the R'G'B' values are the eight-bit gamma-compressed color components (i.e., the actual RGB values we obtain when we open up or display a JPEG image). For most applications, this formula is not that important, since your image reading software will directly

provide you with the eight-bit gamma-compressed R'G'B' values. However, if you are trying to do careful image deblocking (Exercise 3.30), this information may be useful.

Another color space you may come across is *hue, saturation, value* (HSV), which is a projection of the RGB color cube onto a non-linear chroma angle, a radial saturation percentage, and a luminance-inspired value. In more detail, value is defined as either the mean or maximum color value, saturation is defined as scaled distance from the diagonal, and hue is defined as the direction around a color wheel (the exact formulas are described by Hall (1989); Foley, van Dam, Feiner *et al.* (1995)). Such a decomposition is quite natural in graphics applications such as color picking (it approximates the Munsell chart for color description). Figure 2.32–n shows an HSV representation of a sample color image, where saturation is encoded using a gray scale (saturated = darker) and hue is depicted as a color.

If you want your computer vision algorithm to only affect the value (luminance) of an image and not its saturation or hue, a simpler solution is to use either the  $Yxy$  (luminance + chromaticity) coordinates defined in (2.104) or the even simpler *color ratios*,

$$r = \frac{R}{R + G + B}, \quad g = \frac{G}{R + G + B}, \quad b = \frac{B}{R + G + B} \quad (2.116)$$

(Figure 2.32e–h). After manipulating the luma (2.112), e.g., through the process of histogram equalization (Section 3.1.4), you can multiply each color ratio by the ratio of the new to old luma to obtain an adjusted RGB triplet.

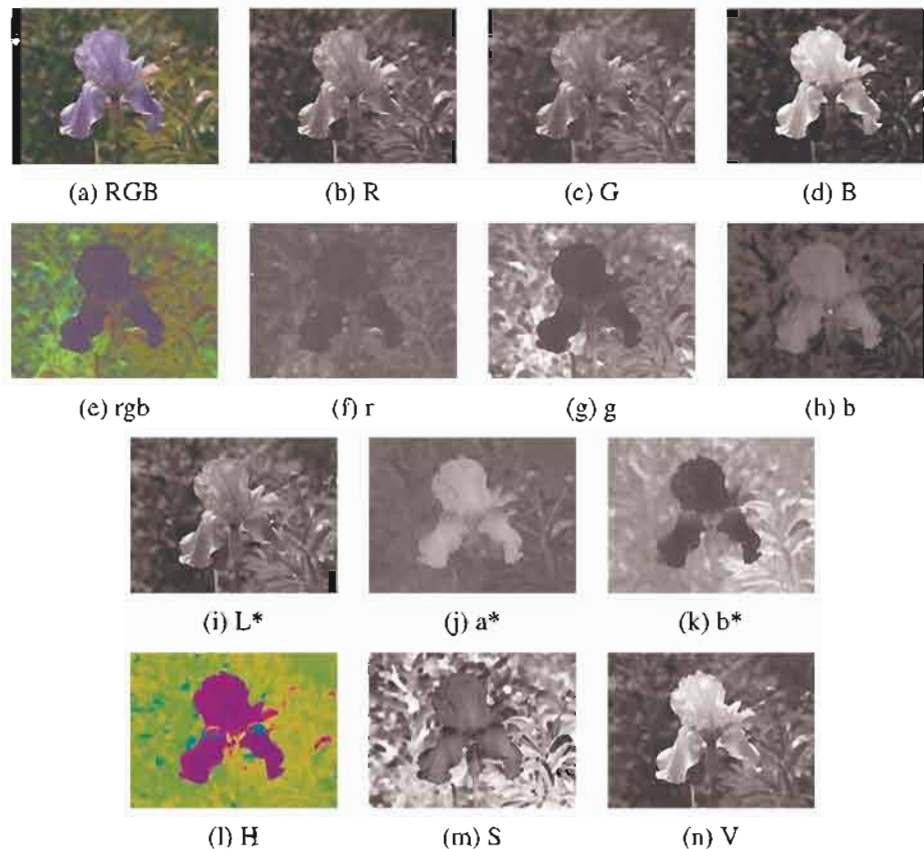
While all of these color systems may sound confusing, in the end, it often may not matter that much which one you use. Poynton, in his *Color FAQ*, <http://www.poynton.com/ColorFAQ.html>, notes that the perceptually motivated L\*a\*b\* system is qualitatively similar to the gamma-compressed R'G'B' system we mostly deal with, since both have a fractional power scaling (which approximates a logarithmic response) between the actual intensity values and the numbers being manipulated. As in all cases, think carefully about what you are trying to accomplish before deciding on a technique to use.<sup>24</sup>

### 2.3.3 Compression

The last stage in a camera's processing pipeline is usually some form of image compression (unless you are using a lossless compression scheme such as camera RAW or PNG).

All color video and image compression algorithms start by converting the signal into YCbCr (or some closely related variant), so that they can compress the luminance signal with higher fidelity than the chrominance signal. (Recall that the human visual system has poorer

<sup>24</sup> If you are at a loss for questions at a conference, you can always ask why the speaker did not use a perceptual color space, such as L\*a\*b\*. Conversely, if they did use L\*a\*b\*, you can ask if they have any concrete evidence that this works better than regular colors.



**Figure 2.32** Color space transformations: (a–d) RGB; (e–h) rgb. (i–k)  $L^*a^*b^*$ ; (l–n) HSV. Note that the rgb,  $L^*a^*b^*$ , and HSV values are all re-scaled to fit the dynamic range of the printed page.

frequency response to color than to luminance changes.) In video, it is common to subsample Cb and Cr by a factor of two horizontally; with still images (JPEG), the subsampling (averaging) occurs both horizontally and vertically.

Once the luminance and chrominance images have been appropriately subsampled and separated into individual images, they are then passed to a *block transform* stage. The most common technique used here is the *discrete cosine transform* (DCT), which is a real-valued variant of the discrete Fourier transform (DFT) (see Section 3.4.3). The DCT is a reasonable approximation to the Karhunen–Loève or eigenvalue decomposition of natural image patches, i.e., the decomposition that simultaneously packs the most energy into the first coefficients and diagonalizes the joint covariance matrix among the pixels (makes transform coefficients





**Figure 2.33** Image compressed with JPEG at three quality settings. Note how the amount of block artifact and high-frequency aliasing (“mosquito noise”) increases from left to right.

statistically independent). Both MPEG and JPEG use  $8 \times 8$  DCT transforms (Wallace 1991; Le Gall 1991), although newer variants use smaller  $4 \times 4$  blocks or alternative transformations, such as wavelets (Taubman and Marcellin 2002) and lapped transforms (Malvar 1990, 1998, 2000) are now used.

After transform coding, the coefficient values are quantized into a set of small integer values that can be coded using a variable bit length scheme such as a Huffman code or an arithmetic code (Wallace 1991). (The DC (lowest frequency) coefficients are also adaptively predicted from the previous block’s DC values. The term “DC” comes from “direct current”, i.e., the non-sinusoidal or non-alternating part of a signal.) The step size in the quantization is the main variable controlled by the *quality* setting on the JPEG file (Figure 2.33).

With video, it is also usual to perform block-based *motion compensation*, i.e., to encode the difference between each block and a *predicted* set of pixel values obtained from a shifted block in the previous frame. (The exception is the *motion-JPEG* scheme used in older DV camcorders, which is nothing more than a series of individually JPEG compressed image frames.) While basic MPEG uses  $16 \times 16$  motion compensation blocks with integer motion values (Le Gall 1991), newer standards use adaptively sized block, sub-pixel motions, and the ability to reference blocks from older frames. In order to recover more gracefully from failures and to allow for random access to the video stream, predicted P frames are interleaved among independently coded I frames. (Bi-directional B frames are also sometimes used.)

The quality of a compression algorithm is usually reported using its *peak signal-to-noise ratio* (PSNR), which is derived from the average *mean square error*,

$$MSE = \frac{1}{n} \sum_{\mathbf{x}} [I(\mathbf{x}) - \hat{I}(\mathbf{x})]^2, \quad (2.117)$$

where  $I(\mathbf{x})$  is the original uncompressed image and  $\hat{I}(\mathbf{x})$  is its compressed counterpart, or equivalently, the *root mean square error* (RMS error), which is defined as

$$RMS = \sqrt{MSE}. \quad (2.118)$$



The PSNR is defined as

$$PSNR = 10 \log_{10} \frac{I_{\max}^2}{MSE} = 20 \log_{10} \frac{I_{\max}}{RMS}, \quad (2.119)$$

where  $I_{\max}$  is the maximum signal extent, e.g., 255 for eight-bit images.

While this is just a high-level sketch of how image compression works, it is useful to understand so that the artifacts introduced by such techniques can be compensated for in various computer vision applications.

## 2.4 Additional reading

As we mentioned at the beginning of this chapter, it provides but a brief summary of a very rich and deep set of topics, traditionally covered in a number of separate fields.

A more thorough introduction to the geometry of points, lines, planes, and projections can be found in textbooks on multi-view geometry (Hartley and Zisserman 2004; Faugeras and Luong 2001) and computer graphics (Foley, van Dam, Feiner *et al.* 1995; Watt 1995; OpenGL-ARB 1997). Topics covered in more depth include higher-order primitives such as quadrics, conics, and cubics, as well as three-view and multi-view geometry.

The image formation (synthesis) process is traditionally taught as part of a computer graphics curriculum (Foley, van Dam, Feiner *et al.* 1995; Glassner 1995; Watt 1995; Shirley 2005) but it is also studied in physics-based computer vision (Wolff, Shafer, and Healey 1992a).

The behavior of camera lens systems is studied in optics (Möller 1988; Hecht 2001; Ray 2002).

Some good books on color theory have been written by Healey and Shafer (1992); Wyszecki and Stiles (2000); Fairchild (2005), with Livingstone (2008) providing a more fun and informal introduction to the topic of color perception. Mark Fairchild's page of color books and links<sup>25</sup> lists many other sources.

Topics relating to sampling and aliasing are covered in textbooks on signal and image processing (Crane 1997; Jähne 1997; Oppenheim and Schaffer 1996; Oppenheim, Schaffer, and Buck 1999; Pratt 2007; Russ 2007; Burger and Burge 2008; Gonzales and Woods 2008).

## 2.5 Exercises

**A note to students:** This chapter is relatively light on exercises since it contains mostly background material and not that many usable techniques. If you really want to understand

<sup>25</sup> [http://www.cis.rit.edu/fairchild/WhyIsColor/books\\_links.html](http://www.cis.rit.edu/fairchild/WhyIsColor/books_links.html).

multi-view geometry in a thorough way, I encourage you to read and do the exercises provided by Hartley and Zisserman (2004). Similarly, if you want some exercises related to the image formation process, Glassner's (1995) book is full of challenging problems.

**Ex 2.1: Least squares intersection point and line fitting—advanced** Equation (2.4) shows how the intersection of two 2D lines can be expressed as their cross product, assuming the lines are expressed as homogeneous coordinates.

1. If you are given more than two lines and want to find a point  $\tilde{\mathbf{x}}$  that minimizes the sum of squared distances to each line,

$$D = \sum_i (\tilde{\mathbf{x}} \cdot \tilde{\mathbf{l}}_i)^2, \quad (2.120)$$

how can you compute this quantity? (Hint: Write the dot product as  $\tilde{\mathbf{x}}^T \tilde{\mathbf{l}}_i$  and turn the squared quantity into a *quadratic form*,  $\tilde{\mathbf{x}}^T \mathbf{A} \tilde{\mathbf{x}}$ .)

2. To fit a line to a bunch of points, you can compute the *centroid* (mean) of the points as well as the *covariance matrix* of the points around this mean. Show that the line passing through the centroid along the major axis of the covariance ellipsoid (largest eigenvector) minimizes the sum of squared distances to the points.
3. These two approaches are fundamentally different, even though projective duality tells us that points and lines are interchangeable. Why are these two algorithms so apparently different? Are they actually minimizing different objectives?

**Ex 2.2: 2D transform editor** Write a program that lets you interactively create a set of rectangles and then modify their “pose” (2D transform). You should implement the following steps:

1. Open an empty window (“canvas”).
2. Shift drag (rubber-band) to create a new rectangle.
3. Select the deformation mode (motion model): translation, rigid, similarity, affine, or perspective.
4. Drag any corner of the outline to change its transformation.

This exercise should be built on a set of pixel coordinate and transformation classes, either implemented by yourself or from a software library. Persistence of the created representation (save and load) should also be supported (for each rectangle, save its transformation).

**Ex 2.3: 3D viewer** Write a simple viewer for 3D points, lines, and polygons. Import a set of point and line commands (primitives) as well as a viewing transform. Interactively modify the object or camera transform. This viewer can be an extension of the one you created in (Exercise 2.2). Simply replace the viewing transformations with their 3D equivalents.

(Optional) Add a z-buffer to do hidden surface removal for polygons.

(Optional) Use a 3D drawing package and just write the viewer control.

**Ex 2.4: Focus distance and depth of field** Figure out how the focus distance and depth of field indicators on a lens are determined.

1. Compute and plot the focus distance  $z_o$  as a function of the distance traveled from the focal length  $\Delta z_i = f - z_i$  for a lens of focal length  $f$  (say, 100mm). Does this explain the hyperbolic progression of focus distances you see on a typical lens (Figure 2.20)?
2. Compute the depth of field (minimum and maximum focus distances) for a given focus setting  $z_o$  as a function of the circle of confusion diameter  $c$  (make it a fraction of the sensor width), the focal length  $f$ , and the f-stop number  $N$  (which relates to the aperture diameter  $d$ ). Does this explain the usual depth of field markings on a lens that bracket the in-focus marker, as in Figure 2.20a?
3. Now consider a zoom lens with a varying focal length  $f$ . Assume that as you zoom, the lens stays in focus, i.e., the distance from the rear nodal point to the sensor plane  $z_i$  adjusts itself automatically for a fixed focus distance  $z_o$ . How do the depth of field indicators vary as a function of focal length? Can you reproduce a two-dimensional plot that mimics the curved depth of field lines seen on the lens in Figure 2.20b?

**Ex 2.5: F-numbers and shutter speeds** List the common f-numbers and shutter speeds that your camera provides. On older model SLRs, they are visible on the lens and shutter speed dials. On newer cameras, you have to look at the electronic viewfinder (or LCD screen/indicator) as you manually adjust exposures.

1. Do these form geometric progressions; if so, what are the ratios? How do these relate to exposure values (EVs)?
2. If your camera has shutter speeds of  $\frac{1}{60}$  and  $\frac{1}{125}$ , do you think that these two speeds are exactly a factor of two apart or a factor of  $125/60 = 2.083$  apart?
3. How accurate do you think these numbers are? Can you devise some way to measure exactly how the aperture affects how much light reaches the sensor and what the exact exposure times actually are?

**Ex 2.6: Noise level calibration** Estimate the amount of noise in your camera by taking repeated shots of a scene with the camera mounted on a tripod. (Purchasing a remote shutter release is a good investment if you own a DSLR.) Alternatively, take a scene with constant color regions (such as a color checker chart) and estimate the variance by fitting a smooth function to each color region and then taking differences from the predicted function.

1. Plot your estimated variance as a function of level for each of your color channels separately.
2. Change the ISO setting on your camera; if you cannot do that, reduce the overall light in your scene (turn off lights, draw the curtains, wait until dusk). Does the amount of noise vary a lot with ISO/gain?
3. Compare your camera to another one at a different price point or year of make. Is there evidence to suggest that “you get what you pay for”? Does the quality of digital cameras seem to be improving over time?

**Ex 2.7: Gamma correction in image stitching** Here’s a relatively simple puzzle. Assume you are given two images that are part of a panorama that you want to stitch (see Chapter 9). The two images were taken with different exposures, so you want to adjust the RGB values so that they match along the seam line. Is it necessary to undo the gamma in the color values in order to achieve this?

**Ex 2.8: Skin color detection** Devise a simple skin color detector (Forsyth and Fleck 1999; Jones and Rehg 2001; Vezhnevets, Sazonov, and Andreeva 2003; Kakumanu, Makrogiannis, and Bourbakis 2007) based on chromaticity or other color properties.

1. Take a variety of photographs of people and calculate the *xy chromaticity values* for each pixel.
2. Crop the photos or otherwise indicate with a painting tool which pixels are likely to be skin (e.g. face and arms).
3. Calculate a color (chromaticity) distribution for these pixels. You can use something as simple as a mean and covariance measure or as complicated as a mean-shift segmentation algorithm (see Section 5.3.2). You can optionally use non-skin pixels to model the *background distribution*.
4. Use your computed distribution to find the skin regions in an image. One easy way to visualize this is to paint all non-skin pixels a given color, such as white or black.
5. How sensitive is your algorithm to color balance (scene lighting)?

6. Does a simpler chromaticity measurement, such as a color ratio (2.116), work just as well?

**Ex 2.9: White point balancing—tricky** A common (in-camera or post-processing) technique for performing white point adjustment is to take a picture of a white piece of paper and to adjust the RGB values of an image to make this a neutral color.

1. Describe how you would adjust the RGB values in an image given a sample “white color” of  $(R_w, G_w, B_w)$  to make this color neutral (without changing the exposure too much).
2. Does your transformation involve a simple (per-channel) scaling of the RGB values or do you need a full  $3 \times 3$  color twist matrix (or something else)?
3. Convert your RGB values to XYZ. Does the appropriate correction now only depend on the XY (or xy) values? If so, when you convert back to RGB space, do you need a full  $3 \times 3$  color twist matrix to achieve the same effect?
4. If you used pure diagonal scaling in the direct RGB mode but end up with a twist if you work in XYZ space, how do you explain this apparent dichotomy? Which approach is correct? (Or is it possible that neither approach is actually correct?)

If you want to find out what your camera *actually* does, continue on to the next exercise.

**Ex 2.10: In-camera color processing—challenging** If your camera supports a RAW pixel mode, take a pair of RAW and JPEG images, and see if you can infer what the camera is doing when it converts the RAW pixel values to the final color-corrected and gamma-compressed eight-bit JPEG pixel values.

1. Deduce the pattern in your color filter array from the correspondence between co-located RAW and color-mapped pixel values. Use a color checker chart at this stage if it makes your life easier. You may find it helpful to split the RAW image into four separate images (subsampling even and odd columns and rows) and to treat each of these new images as a “virtual” sensor.
2. Evaluate the quality of the demosaicing algorithm by taking pictures of challenging scenes which contain strong color edges (such as those shown in in Section 10.3.1).
3. If you can take the same exact picture after changing the color balance values in your camera, compare how these settings affect this processing.
4. Compare your results against those presented by Chakrabarti, Scharstein, and Zickler (2009) or use the data available in their database of color images.<sup>26</sup>

---

<sup>26</sup> <http://vision.middlebury.edu/color/>.

