



**Figure 3.1** Some common image processing operations: (a) original image; (b) increased contrast; (c) change in hue; (d) “posterized” (quantized colors); (e) blurred; (f) rotated.

Now that we have seen how images are formed through the interaction of 3D scene elements, lighting, and camera optics and sensors, let us look at the first stage in most computer vision applications, namely the use of image processing to preprocess the image and convert it into a form suitable for further analysis. Examples of such operations include exposure correction and color balancing, the reduction of image noise, increasing sharpness, or straightening the image by rotating it (Figure 3.1). While some may consider image processing to be outside the purview of computer vision, most computer vision applications, such as computational photography and even recognition, require care in designing the image processing stages in order to achieve acceptable results.

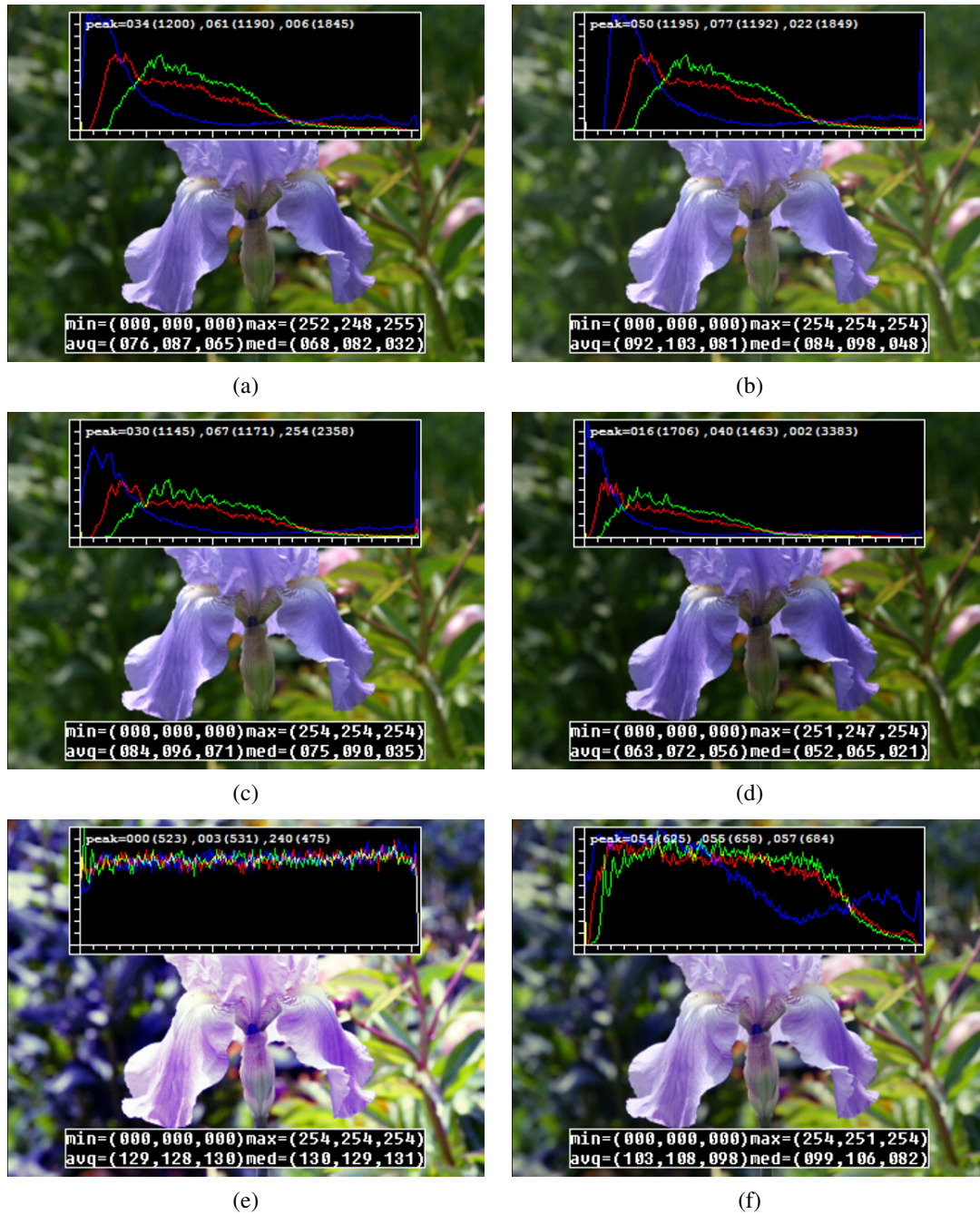
In this chapter, we review standard image processing operators that map pixel values from one image to another. Image processing is often taught in electrical engineering departments as a follow-on course to an introductory course in signal processing (Oppenheim and Schaffer 1996; Oppenheim, Schaffer, and Buck 1999). There are several popular textbooks for image processing (Crane 1997; Gomes and Velho 1997; Jähne 1997; Pratt 2007; Russ 2007; Burger and Burge 2008; Gonzales and Woods 2008).

We begin this chapter with the simplest kind of image transforms, namely those that manipulate each pixel independently of its neighbors (Section 3.1). Such transforms are often called *point operators* or *point processes*. Next, we examine *neighborhood* (area-based) operators, where each new pixel's value depends on a small number of neighboring input values (Sections 3.2 and 3.3). A convenient tool to analyze (and sometimes accelerate) such neighborhood operations is the *Fourier Transform*, which we cover in Section 3.4. Neighborhood operators can be cascaded to form *image pyramids* and *wavelets*, which are useful for analyzing images at a variety of resolutions (scales) and for accelerating certain operations (Section 3.5). Another important class of global operators are *geometric transformations*, such as rotations, shears, and perspective deformations (Section 3.6). Finally, we introduce *global optimization* approaches to image processing, which involve the minimization of an energy functional or, equivalently, optimal estimation using Bayesian *Markov random field* models (Section 3.7).

## 3.1 Point operators

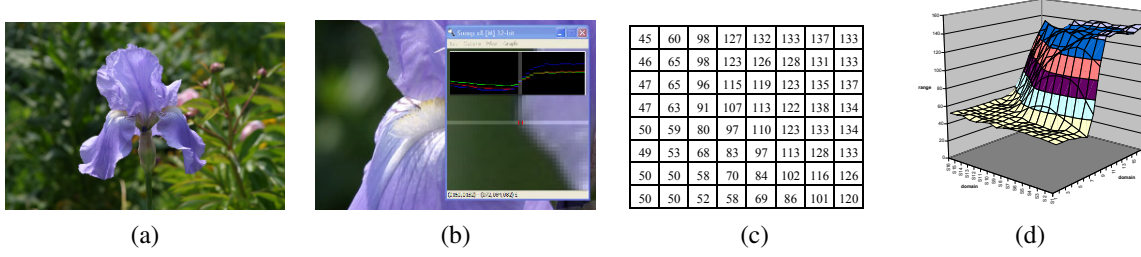
The simplest kinds of image processing transforms are *point operators*, where each output pixel's value depends on only the corresponding input pixel value (plus, potentially, some globally collected information or parameters). Examples of such operators include brightness and contrast adjustments (Figure 3.2) as well as color correction and transformations. In the image processing literature, such operations are also known as *point processes* (Crane 1997).

We begin this section with a quick review of simple point operators such as brightness scaling and image addition. Next, we discuss how colors in images can be manipulated. We then present *image compositing* and *matting* operations, which play an important role in computational photography (Chapter 10) and computer graphics applications. Finally, we describe the more global process of *histogram equalization*. We close with an example application that manipulates *tonal values* (exposure and contrast) to improve image appearance.



**Figure 3.2** Some local image processing operations: (a) original image along with its three color (per-channel) histograms; (b) brightness increased (additive offset,  $b = 16$ ); (c) contrast increased (multiplicative gain,  $a = 1.1$ ); (d) gamma (partially) linearized ( $\gamma = 1.2$ ); (e) full histogram equalization; (f) partial histogram equalization.





**Figure 3.3** Visualizing image data: (a) original image; (b) cropped portion and scanline plot using an image inspection tool; (c) grid of numbers; (d) surface plot. For figures (c)–(d), the image was first converted to grayscale.

### 3.1.1 Pixel transforms

A general image processing *operator* is a function that takes one or more input images and produces an output image. In the continuous domain, this can be denoted as

$$g(\mathbf{x}) = h(f(\mathbf{x})) \text{ or } g(\mathbf{x}) = h(f_0(\mathbf{x}), \dots, f_n(\mathbf{x})), \quad (3.1)$$

where  $\mathbf{x}$  is in the  $D$ -dimensional *domain* of the functions (usually  $D = 2$  for images) and the functions  $f$  and  $g$  operate over some *range*, which can either be scalar or vector-valued, e.g., for color images or 2D motion. For discrete (sampled) images, the domain consists of a finite number of *pixel locations*,  $\mathbf{x} = (i, j)$ , and we can write

$$g(i, j) = h(f(i, j)). \quad (3.2)$$

Figure 3.3 shows how an image can be represented either by its color (appearance), as a grid of numbers, or as a two-dimensional function (surface plot).

Two commonly used point processes are multiplication and addition with a constant,

$$g(\mathbf{x}) = af(\mathbf{x}) + b. \quad (3.3)$$

The parameters  $a > 0$  and  $b$  are often called the *gain* and *bias* parameters; sometimes these parameters are said to control *contrast* and *brightness*, respectively (Figures 3.2b–c).<sup>1</sup> The bias and gain parameters can also be spatially varying,

$$g(\mathbf{x}) = a(\mathbf{x})f(\mathbf{x}) + b(\mathbf{x}), \quad (3.4)$$

e.g., when simulating the *graded density filter* used by photographers to selectively darken the sky or when modeling vignetting in an optical system.

Multiplicative gain (both global and spatially varying) is a *linear* operation, since it obeys the *superposition principle*,

$$h(f_0 + f_1) = h(f_0) + h(f_1). \quad (3.5)$$

(We will have more to say about linear shift invariant operators in Section 3.2.) Operators such as image squaring (which is often used to get a local estimate of the *energy* in a band-pass filtered signal, see Section 3.5) are not linear.

<sup>1</sup> An image's luminance characteristics can also be summarized by its *key* (average luminance) and *range* (Kopf, Uyttendaele, Deussen *et al.* 2007).

Another commonly used *dyadic* (two-input) operator is the *linear blend* operator,

$$g(\mathbf{x}) = (1 - \alpha)f_0(\mathbf{x}) + \alpha f_1(\mathbf{x}). \quad (3.6)$$

By varying  $\alpha$  from  $0 \rightarrow 1$ , this operator can be used to perform a temporal *cross-dissolve* between two images or videos, as seen in slide shows and film production, or as a component of image *morphing* algorithms (Section 3.6.3).

One highly used non-linear transform that is often applied to images before further processing is *gamma correction*, which is used to remove the non-linear mapping between input radiance and quantized pixel values (Section 2.3.2). To invert the gamma mapping applied by the sensor, we can use

$$g(\mathbf{x}) = [f(\mathbf{x})]^{1/\gamma}, \quad (3.7)$$

where a gamma value of  $\gamma \approx 2.2$  is a reasonable fit for most digital cameras.

### 3.1.2 Color transforms

While color images can be treated as arbitrary vector-valued functions or collections of independent bands, it usually makes sense to think about them as highly correlated signals with strong connections to the image formation process (Section 2.2), sensor design (Section 2.3), and human perception (Section 2.3.2). Consider, for example, brightening a picture by adding a constant value to all three channels, as shown in Figure 3.2b. Can you tell if this achieves the desired effect of making the image look brighter? Can you see any undesirable side-effects or artifacts?

In fact, adding the same value to each color channel not only increases the apparent *intensity* of each pixel, it can also affect the pixel's *hue* and *saturation*. How can we define and manipulate such quantities in order to achieve the desired perceptual effects?

As discussed in Section 2.3.2, chromaticity coordinates (2.104) or even simpler color ratios (2.116) can first be computed and then used after manipulating (e.g., brightening) the luminance  $Y$  to re-compute a valid RGB image with the same hue and saturation. Figure 2.32g–i shows some color ratio images multiplied by the middle gray value for better visualization.

Similarly, color balancing (e.g., to compensate for incandescent lighting) can be performed either by multiplying each channel with a different scale factor or by the more complex process of mapping to XYZ color space, changing the nominal white point, and mapping back to RGB, which can be written down using a linear  $3 \times 3$  *color twist* transform matrix. Exercises 2.9 and 3.1 have you explore some of these issues.

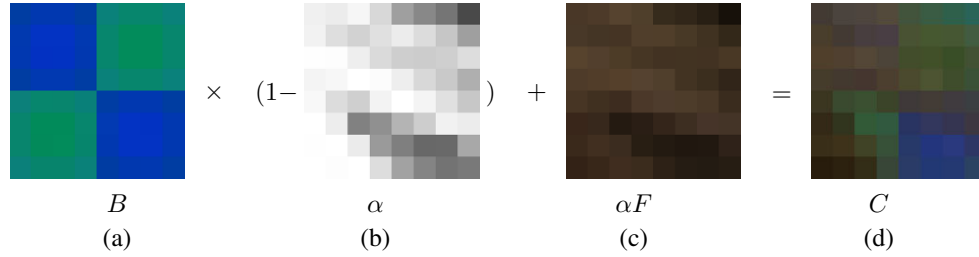
Another fun project, best attempted after you have mastered the rest of the material in this chapter, is to take a picture with a rainbow in it and enhance the strength of the rainbow (Exercise 3.29).

### 3.1.3 Compositing and matting

In many photo editing and visual effects applications, it is often desirable to cut a *foreground* object out of one scene and put it on top of a different *background* (Figure 3.4). The process of extracting the object from the original image is often called *matting* (Smith and Blinn



**Figure 3.4** Image matting and compositing (Chuang, Curless, Salesin *et al.* 2001) © 2001 IEEE: (a) source image; (b) extracted foreground object  $F$ ; (c) alpha matte  $\alpha$  shown in grayscale; (d) new composite  $C$ .



**Figure 3.5** Compositing equation  $C = (1 - \alpha)B + \alpha F$ . The images are taken from a close-up of the region of the hair in the upper right part of the lion in Figure 3.4.

1996), while the process of inserting it into another image (without visible artifacts) is called *compositing* (Porter and Duff 1984; Blinn 1994a).

The intermediate representation used for the foreground object between these two stages is called an *alpha-matted color image* (Figure 3.4b–c). In addition to the three color RGB channels, an alpha-matted image contains a fourth *alpha* channel  $\alpha$  (or A) that describes the relative amount of *opacity* or *fractional coverage* at each pixel (Figures 3.4c and 3.5b). The opacity is the opposite of the *transparency*. Pixels within the object are fully opaque ( $\alpha = 1$ ), while pixels fully outside the object are transparent ( $\alpha = 0$ ). Pixels on the boundary of the object vary smoothly between these two extremes, which hides the perceptual visible *jaggies* that occur if only binary opacities are used.

To composite a new (or foreground) image on top of an old (background) image, the *over operator*, first proposed by Porter and Duff (1984) and then studied extensively by Blinn (1994a; 1994b), is used,

$$C = (1 - \alpha)B + \alpha F. \quad (3.8)$$

This operator *attenuates* the influence of the background image  $B$  by a factor  $(1 - \alpha)$  and then adds in the color (and opacity) values corresponding to the foreground layer  $F$ , as shown in Figure 3.5.

In many situations, it is convenient to represent the foreground colors in *pre-multiplied* form, i.e., to store (and manipulate) the  $\alpha F$  values directly. As Blinn (1994b) shows, the pre-multiplied RGBA representation is preferred for several reasons, including the ability to blur or resample (e.g., rotate) alpha-matted images without any additional complications (just treating each RGBA band independently). However, when matting using local color consistency (Ruzon and Tomasi 2000; Chuang, Curless, Salesin *et al.* 2001), the pure un-



**Figure 3.6** An example of light reflecting off the transparent glass of a picture frame (Black and Anandan 1996) © 1996 Elsevier. You can clearly see the woman's portrait inside the picture frame superimposed with the reflection of a man's face off the glass.

multiplied foreground colors  $F$  are used, since these remain constant (or vary slowly) in the vicinity of the object edge.

The over operation is not the only kind of compositing operation that can be used. Porter and Duff (1984) describe a number of additional operations that can be useful in photo editing and visual effects applications. In this book, we concern ourselves with only one additional, commonly occurring case (but see Exercise 3.2).

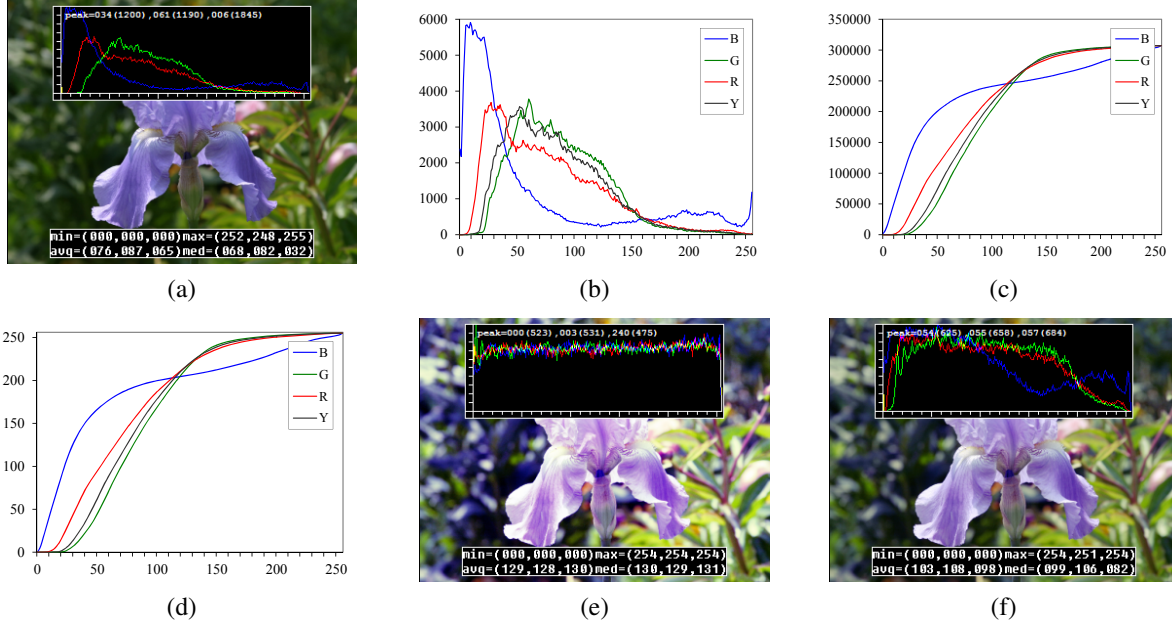
When light reflects off clean transparent glass, the light passing through the glass and the light reflecting off the glass are simply added together (Figure 3.6). This model is useful in the analysis of *transparent motion* (Black and Anandan 1996; Szeliski, Avidan, and Anandan 2000), which occurs when such scenes are observed from a moving camera (see Section 8.5.2).

The actual process of *matting*, i.e., recovering the foreground, background, and alpha matte values from one or more images, has a rich history, which we study in Section 10.4. Smith and Blinn (1996) have a nice survey of traditional *blue-screen matting* techniques, while Toyama, Krumm, Brumitt *et al.* (1999) review *difference matting*. More recently, there has been a lot of activity in computational photography relating to *natural image matting* (Ruzon and Tomasi 2000; Chuang, Curless, Salesin *et al.* 2001; Wang and Cohen 2007a), which attempts to extract the mattes from a single natural image (Figure 3.4a) or from extended video sequences (Chuang, Agarwala, Curless *et al.* 2002). All of these techniques are described in more detail in Section 10.4.

### 3.1.4 Histogram equalization

While the brightness and gain controls described in Section 3.1.1 can improve the appearance of an image, how can we automatically determine their best values? One approach might be to look at the darkest and brightest pixel values in an image and map them to pure black and pure white. Another approach might be to find the *average* value in the image, push it towards middle gray, and expand the *range* so that it more closely fills the displayable values (Kopf, Uyttendaele, Deussen *et al.* 2007).

How can we visualize the set of lightness values in an image in order to test some of



**Figure 3.7** Histogram analysis and equalization: (a) original image (b) color channel and intensity (luminance) histograms; (c) cumulative distribution functions; (d) equalization (transfer) functions; (e) full histogram equalization; (f) partial histogram equalization.

these heuristics? The answer is to plot the *histogram* of the individual color channels and luminance values, as shown in Figure 3.7b.<sup>2</sup> From this distribution, we can compute relevant statistics such as the minimum, maximum, and average intensity values. Notice that the image in Figure 3.7a has both an excess of dark values and light values, but that the mid-range values are largely under-populated. Would it not be better if we could simultaneously brighten some dark values and darken some light values, while still using the full extent of the available dynamic range? Can you think of a mapping that might do this?

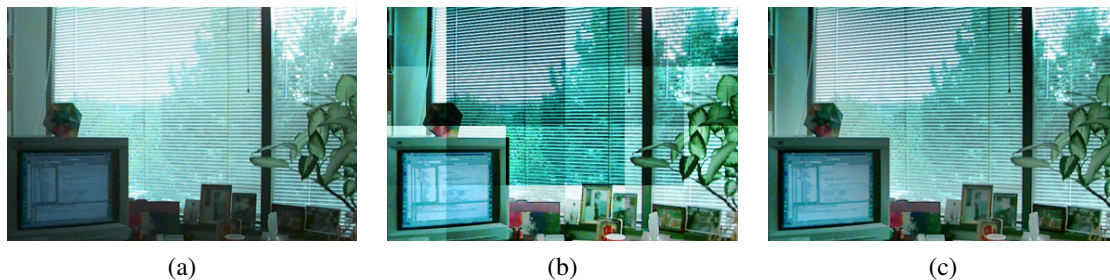
One popular answer to this question is to perform *histogram equalization*, i.e., to find an intensity mapping function  $f(I)$  such that the resulting histogram is flat. The trick to finding such a mapping is the same one that people use to generate random samples from a *probability density function*, which is to first compute the *cumulative distribution function* shown in Figure 3.7c.

Think of the original histogram  $h(I)$  as the distribution of grades in a class after some exam. How can we map a particular grade to its corresponding *percentile*, so that students at the 75% percentile range scored better than  $3/4$  of their classmates? The answer is to integrate the distribution  $h(I)$  to obtain the cumulative distribution  $c(I)$ ,

$$c(I) = \frac{1}{N} \sum_{i=0}^I h(i) = c(I-1) + \frac{1}{N} h(I), \quad (3.9)$$

<sup>2</sup> The histogram is simply the *count* of the number of pixels at each gray level value. For an eight-bit image, an accumulation table with 256 entries is needed. For higher bit depths, a table with the appropriate number of entries (probably fewer than the full number of gray levels) should be used.





**Figure 3.8** Locally adaptive histogram equalization: (a) original image; (b) block histogram equalization; (c) full locally adaptive equalization.

where  $N$  is the number of pixels in the image or students in the class. For any given grade or intensity, we can look up its corresponding percentile  $c(I)$  and determine the final value that pixel should take. When working with eight-bit pixel values, the  $I$  and  $c$  axes are rescaled from  $[0, 255]$ .

Figure 3.7d shows the result of applying  $f(I) = c(I)$  to the original image. As we can see, the resulting histogram is flat; so is the resulting image (it is “flat” in the sense of a lack of contrast and being muddy looking). One way to compensate for this is to only *partially* compensate for the histogram unevenness, e.g., by using a mapping function  $f(I) = \alpha c(I) + (1 - \alpha)I$ , which is a linear blend between the cumulative distribution function and the identity transform (a straight line). As you can see in Figure 3.7e, the resulting image maintains more of its original grayscale distribution while having a more appealing balance.

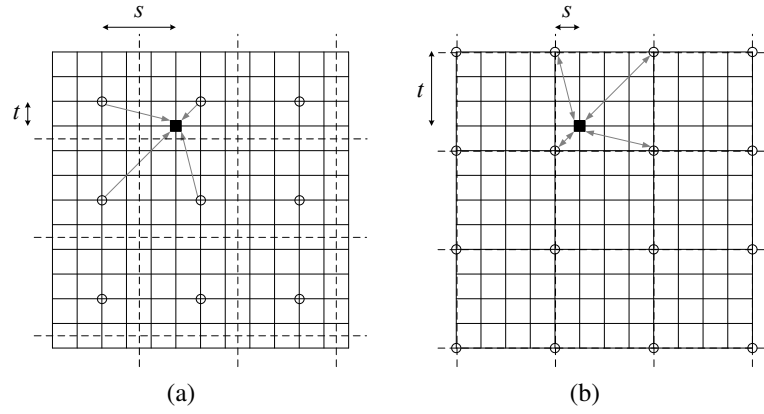
Another potential problem with histogram equalization (or, in general, image brightening) is that noise in dark regions can be amplified and become more visible. Exercise 3.6 suggests some possible ways to mitigate this, as well as alternative techniques to maintain contrast and “punch” in the original images (Larson, Rushmeier, and Piatko 1997; Stark 2000).

### Locally adaptive histogram equalization

While global histogram equalization can be useful, for some images it might be preferable to apply different kinds of equalization in different regions. Consider for example the image in Figure 3.8a, which has a wide range of luminance values. Instead of computing a single curve, what if we were to subdivide the image into  $M \times M$  pixel blocks and perform separate histogram equalization in each sub-block? As you can see in Figure 3.8b, the resulting image exhibits a lot of blocking artifacts, i.e., intensity discontinuities at block boundaries.

One way to eliminate blocking artifacts is to use a *moving window*, i.e., to recompute the histogram for every  $M \times M$  block centered at each pixel. This process can be quite slow ( $M^2$  operations per pixel), although with clever programming only the histogram entries corresponding to the pixels entering and leaving the block (in a raster scan across the image) need to be updated ( $M$  operations per pixel). Note that this operation is an example of the *non-linear neighborhood operations* we study in more detail in Section 3.3.1.

A more efficient approach is to compute non-overlapped block-based equalization functions as before, but to then smoothly interpolate the transfer functions as we move between blocks. This technique is known as *adaptive histogram equalization* (AHE) and its contrast-



**Figure 3.9** Local histogram interpolation using relative  $(s, t)$  coordinates: (a) block-based histograms, with block centers shown as circles; (b) corner-based “spline” histograms. Pixels are located on grid intersections. The black square pixel’s transfer function is interpolated from the four adjacent lookup tables (gray arrows) using the computed  $(s, t)$  values. Block boundaries are shown as dashed lines.

limited (gain-limited) version is known as CLAHE (Pizer, Amburn, Austin *et al.* 1987).<sup>3</sup> The weighting function for a given pixel  $(i, j)$  can be computed as a function of its horizontal and vertical position  $(s, t)$  within a block, as shown in Figure 3.9a. To blend the four lookup functions  $\{f_{00}, \dots, f_{11}\}$ , a *bilinear* blending function,

$$f_{s,t}(I) = (1-s)(1-t)f_{00}(I) + s(1-t)f_{10}(I) + (1-s)tf_{01}(I) + stf_{11}(I) \quad (3.10)$$

can be used. (See Section 3.5.2 for higher-order generalizations of such *spline* functions.) Note that instead of blending the four lookup tables for each output pixel (which would be quite slow), we can instead blend the results of mapping a given pixel through the four neighboring lookups.

A variant on this algorithm is to place the lookup tables at the *corners* of each  $M \times M$  block (see Figure 3.9b and Exercise 3.7). In addition to blending four lookups to compute the final value, we can also *distribute* each input pixel into four adjacent lookup tables during the histogram accumulation phase (notice that the gray arrows in Figure 3.9b point both ways), i.e.,

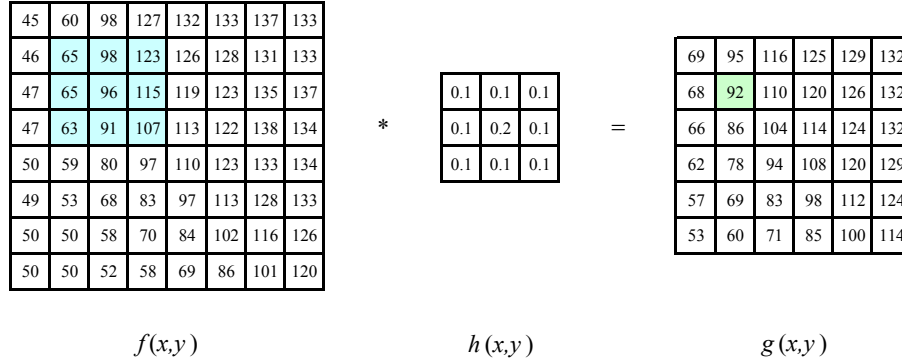
$$h_{k,l}(I(i, j)) += w(i, j, k, l), \quad (3.11)$$

where  $w(i, j, k, l)$  is the bilinear weighting function between pixel  $(i, j)$  and lookup table  $(k, l)$ . This is an example of *soft histogramming*, which is used in a variety of other applications, including the construction of SIFT feature descriptors (Section 4.1.3) and vocabulary trees (Section 14.3.2).

### 3.1.5 Application: Tonal adjustment

One of the most widely used applications of point-wise image processing operators is the manipulation of contrast or *tone* in photographs, to make them look either more attractive or

<sup>3</sup>This algorithm is implemented in the MATLAB `adapt_hist` function.



**Figure 3.10** Neighborhood filtering (convolution): The image on the left is convolved with the filter in the middle to yield the image on the right. The light blue pixels indicate the source neighborhood for the light green destination pixel.

more interpretable. You can get a good sense of the range of operations possible by opening up any photo manipulation tool and trying out a variety of contrast, brightness, and color manipulation options, as shown in Figures 3.2 and 3.7.

Exercises 3.1, 3.5, and 3.6 have you implement some of these operations, in order to become familiar with basic image processing operators. More sophisticated techniques for tonal adjustment (Reinhard, Ward, Pattanaik *et al.* 2005; Bae, Paris, and Durand 2006) are described in the section on high dynamic range tone mapping (Section 10.2.1).

## 3.2 Linear filtering

Locally adaptive histogram equalization is an example of a *neighborhood operator* or *local operator*, which uses a collection of pixel values in the vicinity of a given pixel to determine its final output value (Figure 3.10). In addition to performing local tone adjustment, neighborhood operators can be used to *filter* images in order to add soft blur, sharpen details, accentuate edges, or remove noise (Figure 3.11b–d). In this section, we look at *linear* filtering operators, which involve weighted combinations of pixels in small neighborhoods. In Section 3.3, we look at non-linear operators such as morphological filters and distance transforms.

The most commonly used type of neighborhood operator is a *linear filter*, in which an output pixel’s value is determined as a weighted sum of input pixel values (Figure 3.10),

$$g(i, j) = \sum_{k, l} f(i + k, j + l) h(k, l). \quad (3.12)$$

The entries in the weight *kernel* or *mask*  $h(k, l)$  are often called the *filter coefficients*. The above *correlation* operator can be more compactly notated as

$$g = f \otimes h. \quad (3.13)$$