CHAPTER

2 Images and imaging operations

Images are at the core of vision, and there are many ways—from simple to sophisticated—for processing and analyzing them. This chapter concentrates on simple algorithms, which nevertheless need to be treated carefully as there are important subtleties to be learnt. Above all, it aims to show that quite a lot can be achieved with such algorithms, which can readily be programmed and tested by the reader.

Look out for:

- the different types of image—binary, gray scale, and color
- a compact notation for presenting image processing operations
- basic pixel operations—clearing, copying, inverting, and thresholding
- basic window operations—shifting, shrinking, and expanding
- grayscale brightening and contrast-stretching operations
- binary edge location and noise removal operations
- multiimage and convolution operations
- the distinction between sequential and parallel operations, and complications that can arise in the sequential case
- problems that arise around the edge of the image.

Although elementary, this chapter actually provides basic methodology for the whole of Part 1 and much of Part 2 of the book, and its importance should neither be underestimated nor the subtleties be ignored. Full understanding at this stage will save many complications later, when programming more sophisticated algorithms.

pĭx′ĕllātėd, a. picture broken into a regular tiling
pĭx′ĭlātėd, a. pixie-like, crazy, deranged

2.1 INTRODUCTION

This chapter is concerned with images and simple image processing operations. It is intended to lead on to more advanced image analysis operations that are of use for machine vision in an industrial environment. Perhaps the main purpose of this

Computer Vision. DOI: http://dx.doi.org/10.1016/B978-0-12-809284-2.00002-2

chapter is to introduce the reader to some basic techniques and notation that will be of use throughout the book. However, the image processing algorithms introduced here are of value in their own right in disciplines ranging from remote sensing to medicine and from forensic to military and scientific applications.

This chapter deals with images that have already been obtained from suitable sensors: the latter are covered in a later chapter. Typical of such images is that shown in Fig. 2.1A. This is a gray-tone image that at first sight appears to be a normal "black and white" photograph. However, closer inspection shows that it is composed of a large number of individual picture cells or "pixels." In fact, the image is a 128 × 128 array of pixels. To get a better feel for the limitations

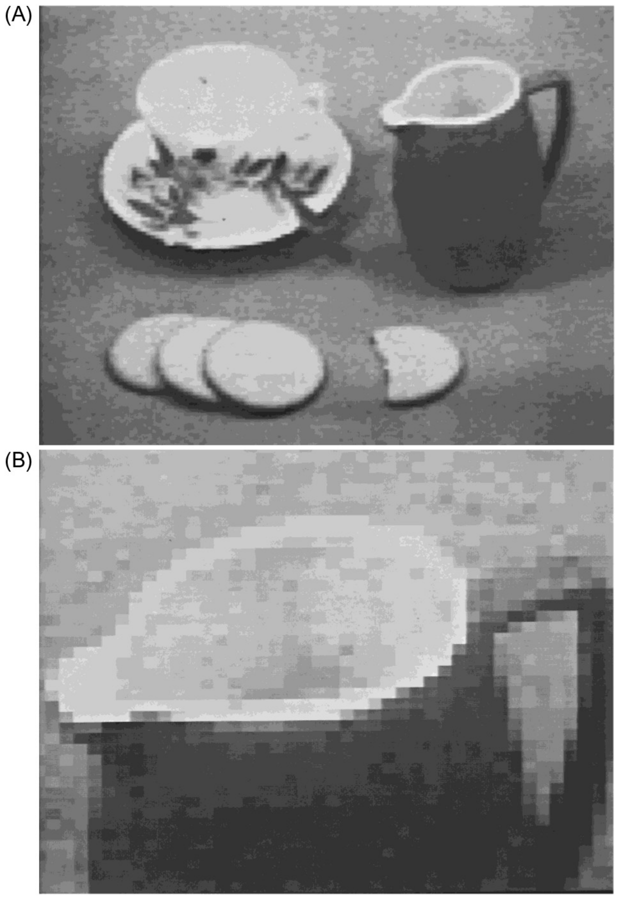

FIGURE 2.1

Typical grayscale image: (A) grayscale image digitized into a 128 × 128 array of pixels; (B) section of image shown in (A) subjected to 3-fold linear magnification: the individual pixels are now clearly visible.

of such a digitized image, Fig. 2.1B shows a 42 × 42 section that has been subjected to a 3-fold magnification so that the pixels can be examined individually.

It is not easy to see that these gray-tone images are digitized into a gray scale containing just 64 gray levels. To some extent, high spatial resolution compensates for lack of grayscale resolution, and as a result it is difficult to see the difference between an individual shade of gray and the shade it would have had in an ideal picture. In addition, when we look at the magnified section of image in Fig. 2.1B, it is difficult to understand the significance of the individual pixel intensities—the whole is becoming lost in a mass of small parts. Early television cameras typically gave a grayscale resolution that was accurate only to about one part in 50, corresponding to about 6 bits of useful information per pixel. Modern solid-state cameras commonly give less noise and may allow 8 or even 9 bits of information per pixel. However, there are many occasions when it is not worthwhile to aim for such high grayscale resolutions, particularly when the result will not be visible to the human eye or when there is an enormous amount of other data that a robot can use to locate objects within the field of view. Note that if the human eye can see an object in a digitized image of particular spatial and grayscale resolution, it is in principle possible to devise a computer algorithm to do the same thing.

Nevertheless, there is a range of applications for which it is valuable to retain a good grayscale resolution, so that highly accurate measurements can be made from a digital image. This is the case in many robotic applications, where high-accuracy checking of components is critical. More details will be shared about this later. In addition, it will be seen in Part 2 that certain techniques for locating components efficiently require local edge orientation to be estimated to better than 1°, and this can be achieved only if at least 6 bits of grayscale information are available per pixel.

2.1.1 GRAY SCALE VERSUS COLOR

Returning now to the image of Fig. 2.1A, we might reasonably ask whether it would be better to replace the gray scale with color, using an RGB color camera and three digitizers for the three main colors. There are two aspects of color that are important for the present discussion. One is the *intrinsic value* of color in machine vision: the other is the *additional storage and processing penalty* if might bring. It is tempting to say that the latter aspect is of no great importance given the cheapness of modern computers that have both high storage and high speed. On the other hand, high-resolution images can arrive from a collection of CCTV cameras at huge data rates, and it will be many years before it will be possible to analyze *all* the data arriving from such sources as they come in. Hence if color adds substantially to the storage and processing load, this will need to be justified.

Against this, the *potential* of color for helping with many aspects of inspection, surveillance, control, and a wide variety of other applications including medicine (color playing a crucial role in images taken during surgery) is enormous. This is illustrated with regard to robot navigation and driving in Figs. 2.2 and 2.3;

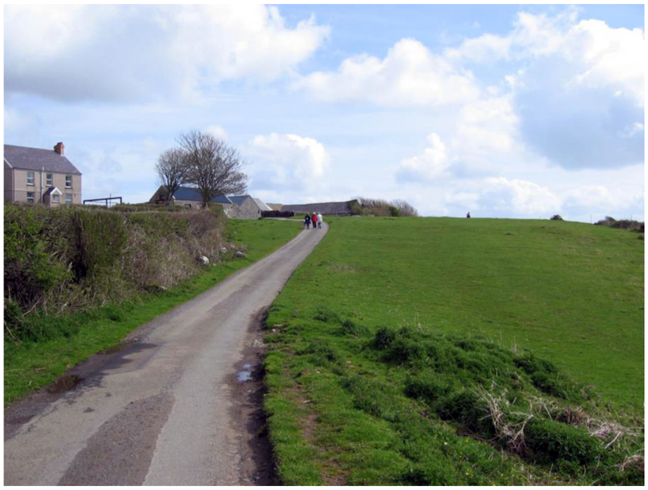

FIGURE 2.2

Value of color for segmentation and recognition. In natural outdoor scenes such as this, color helps with segmentation and with recognition. While it may have been important to the early human when discerning sources of food in the wild, robot drones may benefit by using color to aid navigation.

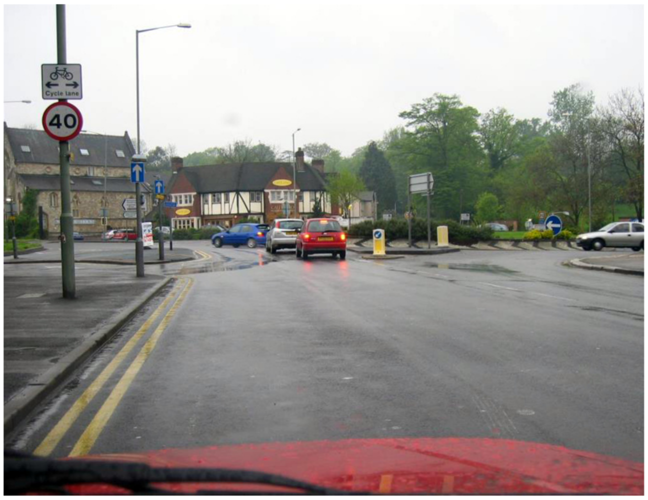

FIGURE 2.3

Value of color in the built environment. Color plays an important role for the human in managing the built environment. In a vehicle, a plethora of bright lights, road signs, and markings (such as yellow lines) are coded to help the driver; they may likewise help a robot to drive more safely by the provision of crucial information.

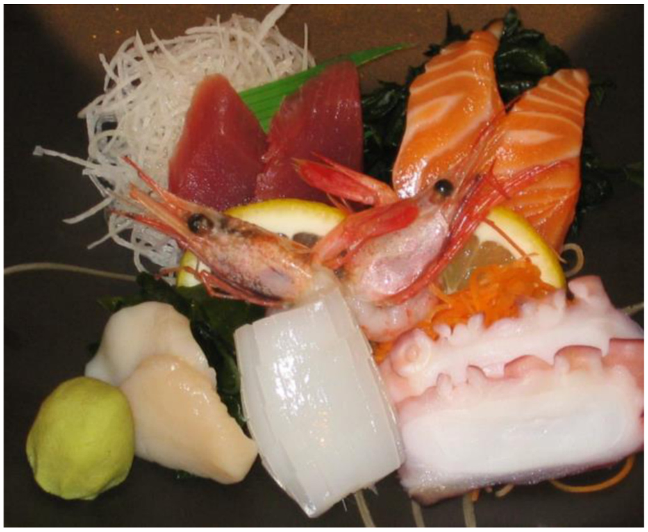

FIGURE 2.4

Value of color for food inspection. Much food is brightly colored, as with this Japanese meal. While this may be attractive to the human, it could also help the robot to check quickly for foreign bodies or toxic substances.

and for food inspection in Figs. 2.4 and 2.5; for color filtering, see Figs. 3.12 and 3.13. Notice that some of these images almost have color for color's sake (especially in Figs. 2.4 and 2.5), although none of them are artificially generated. In others the color is more subdued (Fig. 2.3), and in Fig. 2.5 (excluding the tomatoes) it is quite subtle. The point to be made here is that for color to be useful it need not be garish, but can be subtle as long as it brings the right sort of information to bear on the task in hand. Suffice it to say that in some of the simpler inspection applications, where mechanical components are scrutinized on a conveyor or workbench, it is quite likely to be the shape that is in question rather than the color of the object or its parts. On the other hand, if an automatic fruit picker is to be devised, it will probably be more crucial to check color than specific shape. We leave it to the reader to imagine when and where color is particularly useful or merely an unnecessary luxury.

Next, it is useful to consider the processing aspect of color. In many cases good color discrimination is required to separate and segment two types of object from each other. Typically this will mean not using one or other specific color channel, but subtracting two or combining three in such a way as to foster discrimination. [We use the term "channel" not just to refer to the red, green, or blue channel, but any derived channel obtained by combining the colors into a single color dimension.] In the worst case of combining three color channels by simple

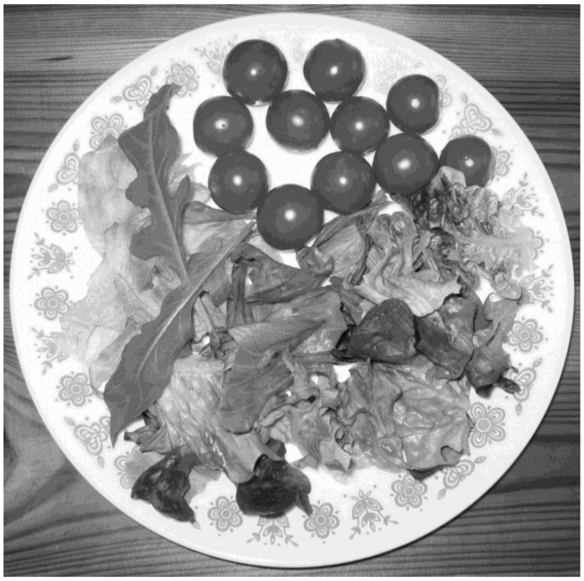

FIGURE 2.5

Subtle shades of color in food inspection. While much food is brightly colored, as for the tomatoes in this picture, green salad leaves show much more subtle combinations of color, and may indeed provide the only reliable means of identification. This could be important for inspection both of the raw product and its state when it reaches the warehouse or the supermarket.

arithmetic processing in which each pixel is treated identically, the processing load will be very light. In contrast, the amount of processing required to *determine* the optimal means of combining the data from the color channels and to carry out different operations dynamically on different parts of the image may be far from negligible, and some care will be needed in the analysis. These problems arise because color signals are inhomogeneous: this contrasts with the situation for grayscale images, where the bits representing the gray scale are of the same type and take the form of a number representing the pixel intensity: they can thus be processed as a single entity on a digital computer.

2.2 IMAGE PROCESSING OPERATIONS

In what follows the images of Figs. 2.1A and 2.7A are considered in some detail, examining some of the many image processing operations that can be performed on them. The resolution of these images reveals a considerable amount of detail and at the same time shows how it relates to the more "meaningful" global information. This should help to make it clear how simple imaging operations contribute to image interpretation.

When performing image processing operations, we start with an image in one storage area and generate a new processed image in another storage area. In practice these storage areas may either be in a special hardware unit called a frame store that is interfaced to the computer or they may be in the main memory of the computer or on one of its discs. In the past a special frame store was required to store images, since each image contains a good fraction of a megabyte of information and this amount of space was not available for normal users in the computer main memory. Nowadays this is less of a problem, but for image acquisition and display a frame store is still required. However, we shall not worry about such details here. Instead it will be assumed that all images are inherently visible and that they are stored in various image "spaces" P, Q, R, etc. Thus we might start with an image in space P and copy it to space Q, for example.

2.2.1 SOME BASIC OPERATIONS ON GRAYSCALE IMAGES

In the following sections we assume a certain degree of familiarity with a language such as C++: readers who are unfamiliar with C++, or Java, which is similar at the required level of programming, should refer to books such as Stroustrup (1991) and Schildt (1995).

Perhaps the simplest of imaging operations is that of clearing an image or setting the contents of a given image space to a constant level. We need some way of arranging this, and accordingly the following C++ routine may be written for implementing it:

```
for (j = 0; j <= 127; j++)
  for (i = 0; i <= 127; i++)                (2.1)
    P[j][i] = alpha;
```

In this routine the local pixel intensity value is expressed as P[*j*][*i*], since P-space is taken to be a 2-D array of intensity values (Table 2.1). In what follows it will be advantageous to rewrite such routines in the more succinct form:

```
for all pixels in image do {P0 = alpha; }    (2.2)
```

as this will aid understanding by removing irrelevant programming detail. The reason for calling the pixel intensity P0 will become clear later.

Another simple imaging operation is to copy an image from one space to another. This is achieved, without changing the contents of the original space P, by the routine:

```
for all pixels in image do {Q0 = P0; }    (2.3)
```

Table 2.1 C++ Notation.

Notation	Meaning
`++`	Increment the preceding variable
`[ ]`	Add array index after a variable
`[ ][ ]`	Add two array indices after a variable: the last is the faster running index
`(int)`	Changes the following variable to integer type
`(float)`	Changes the following variable to floating point
`{ }`	Encloses a sequence of instructions
`if ( ) { };`	Basic conditional statement: ( ) encloses the condition; { } encloses the instructions to be executed
`if ( ) { }; else if ( ) { }; ... ; else { };`	The most general type of conditional statement
`while ( ) { }`	Common type of iterated loop
`do { } while ( );`	Another common type of iterated loop
`do { } until ( );`	"until" means the same as "while not." This is often a convenient notation, although it is not strict C++.
`for ( ; ; ) { };`	Here the conditional statement ( ) has three arguments separated by semicolons: they are the initial condition; the terminating condition; and the incrementation operation, respectively
=	Forces equality (literally: "takes the value")
==	Tests for equality in a conditional expression
<=	$\leq$
>=	$\geq$
!=	$\neq$
!	Logical NOT
&&	Logical AND
||	Logical OR
//	Indicates that the remainder of the line is a comment
/* ... */	Brackets enclosing a comment
$\left.\begin{array}{l}\text{A0...A8}\\ \text{B0...B8}\\ \text{C0...C8}\end{array}\right\}$	Bit image variables in 3 × 3 window[a]
$\left.\begin{array}{l}\text{P0...P8}\\ \text{Q0...Q8}\\ \text{R0...R8}\end{array}\right\}$	Byte image variables in 3 × 3 window[a]
P[0], ...	Equivalent to P0, ...

The purpose of this table is to show what is meant by the various C++ commands and instructions used in this book. It is not intended to be comprehensive: The aim is merely to be helpful to the reader. In general, only notation that differs between C++ and other commonly used languages such as Pascal is included, in order to eliminate possible ambiguity or confusion.

[a]*These predefined variables denote special syntax not available in C++, but useful for simplifying the image processing algorithms presented in Chapter 2 et seq.*

A more interesting operation is that of inverting the image, as in the process of converting a photographic negative to a positive. This process is represented as follows:

$$\text{for all pixels in image do } \{Q0 = 255 - P0;\} \tag{2.4}$$

In this case it is assumed that pixel intensity values lie within the range 0–255, as is commonly true for frame stores that represent each pixel as one byte of information. Note that such intensity values are commonly unsigned and this is assumed generally in what follows.

There are many operations of these types. Some other simple operations are those that shift the image left, right, up, down, or diagonally. They are easy to implement if the new local intensity is made identical to that at a neighboring location in the original image. It is evident how this would be expressed in the double suffix notation used in the original C++ routine. In the new shortened notation, it is necessary to name neighboring pixels in some convenient way, and we here employ the following simple scheme:

P4	P3	P2
P5	P0	P1
P6	P7	P8

with a similar scheme for other image spaces. With this notation, it is easy to express a left shift of an image as follows:

$$\text{for all pixels in image do } \{Q0 = P1;\} \tag{2.5}$$

Similarly, a shift down to the bottom right is expressed as:

$$\text{for all pixels in image do } \{Q0 = P4;\} \tag{2.6}$$

It will now be clear why P0 and Q0 were chosen for the basic notation of pixel intensity: the "0" denotes the central pixel in the "neighborhood" or "window," and corresponds to zero shift when copying from one space to another. However, the type of window operation presented above is much more powerful than single pixel operations, and we shall see many examples of it in what follows. Meanwhile, note that it can give rise to difficulties around the boundaries of the image: we shall return to this point in Section 2.4.

There is a whole range of possible operations associated with modifying images in such a way as to match them to the requirements of a human viewer. For example, adding a constant intensity makes the image brighter:

$$\text{for all pixels in image do } \{Q0 = P0 + \text{beta};\} \tag{2.7}$$

and the image can be made darker in the same way. A more interesting operation is to stretch the contrast of a dull image:

$$\text{for all pixels in image do } \{Q0 = P0^{*}\text{gamma} + \text{beta};\} \tag{2.8}$$

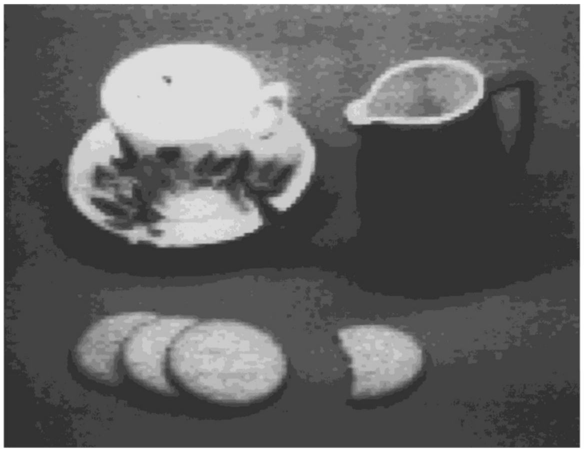

FIGURE 2.6

Contrast stretching: Effect of increasing the contrast in the image of Fig. 2.1A by a factor of two and adjusting the mean intensity level appropriately. The interior of the jug can now be seen more easily. Note, however, that there is no additional information in the new image.

where gamma $>$ 1. In practice (as for Fig. 2.6), it is necessary to ensure that intensities do not result that are outside the normal range, e.g., by using an operation of the form:

```
for all pixels in image do {
    QQ = P0*gamma + beta;
    if (QQ < 0) Q0 = 0;
    else if (QQ > 255) Q0 = 255;                (2.9)
    else Q0 = QQ;
}
```

Most practical situations demand more sophisticated transfer functions—either nonlinear or piecewise linear—but such complexities are ignored here.

In what follows we clarify the discussion by adopting a simple set of notations: the first few letters of the alphabet (A, B, C, . . .) are used consistently to denote binary image spaces, and later letters (P, Q, R, . . .) to denote grayscale images (Table 2.1). In software these variables are assumed to be predeclared, and in hardware (e.g., frame store) terms they are taken to refer to dedicated memory spaces containing only the necessary 1 or 8 bits per pixel. The intricacies of data transfer between variables of different types are important considerations, which are not addressed in detail here: it is sufficient to assume that both A0 = P0 and P0 = A0 correspond to a single-bit transfer, except that in the latter case the top 7 bits are assigned the value 0.

The next operation we shall consider is that of thresholding grayscale images to convert them to binary images. This topic is covered in more detail later, since it is widely used to detect objects in images. However, our purpose here is to

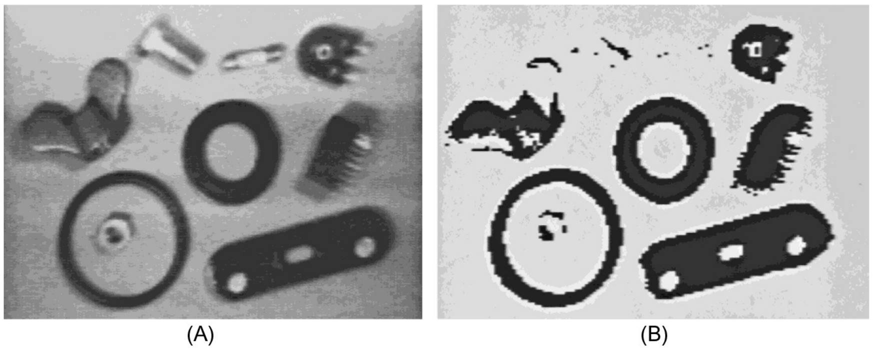

FIGURE 2.7

Thresholding of grayscale images: (A) 128 × 128 pixel grayscale image of a collection of parts; (B) effect of thresholding the image.

look on it as another basic imaging operation. It can be implemented using the routine

```
for all pixels in image do {
    if (P0 > thresh) A0 = 1; else A0 = 0;        (2.10)
}
```

If, as very often happens, objects appear as dark objects on a light background, it is easier to visualize the subsequent binary processing operations by inverting the thresholded image using a routine such as:

```
for all pixels in image do {A0 = 1 − A0; }        (2.11)
```

However, it would be more usual to combine the two operations into a single routine of the form:

```
for all pixels in image do {
    if (P0 > thresh) A0 = 0;  else A0 = 1;        (2.12)
}
```

To display the resulting image in a form as close as possible to the original, it can be reinverted and given the full range of intensity values (intensity values 0 and 1 being scarcely visible):

```
for all pixels in image do {R0 = 255*(1 − A0);}        (2.13)
```

Fig. 2.7 shows the effect of these two operations.

2.2.2 BASIC OPERATIONS ON BINARY IMAGES

Once the image has been thresholded, a wide range of binary imaging operations become possible. Only a few such operations are covered here, with the aim of

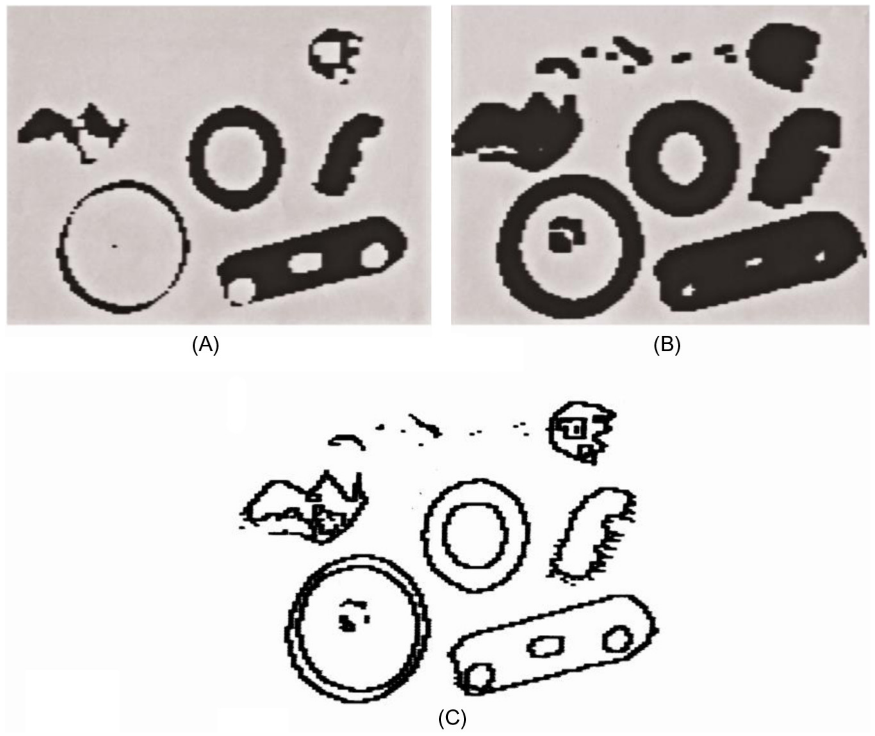

FIGURE 2.8

Simple operations applied to binary images: (A) effect of shrinking the dark-thresholded objects appearing in Fig. 2.7B; (B) effect of expanding these dark objects; (C) result of applying an edge location routine. Note that the shrink, expand, and edge routines are applied to the *dark* objects: this implies that the intensities are initially inverted as part of the thresholding operation and then reinverted as part of the display operation (see text).

being instructive rather than comprehensive. With this in mind, a routine may be written for shrinking dark thresholded objects (Fig. 2.8A), which is here represented by a set of 1s in a background of 0s:

```
for all pixels in image do {
    sigma = A1 + A2 + A3 + A4 + A5 + A6 + A7 + A8;
    if (A0 == 0) B0 = 0;
    else if (sigma < 8) B0 = 0;                          (2.14)
    else B0 = 1;
}
```

In fact, the logic of this routine can be simplified to give the following more compact version:

```
for all pixels in image do {
    sigma = A1 + A2 + A3 + A4 + A5 + A6 + A7 + A8;      (2.15)
    if (sigma < 8) B0 = 0; else B0 = A0;
}
```

Note that the process of shrinking dark objects also expands light objects, including the light background. It also expands holes in dark objects. The opposite process, that of expanding dark objects (or shrinking light ones), is achieved (Fig. 2.8B) with the routine:

```
for all pixels in image do {
    sigma = A1 + A2 + A3 + A4 + A5 + A6 + A7 + A8;
    if (sigma > 0) B0 = 1; else B0 = A0;
}                                                        (2.16)
```

The processes of shrinking and expanding are also widely known by the respective terms "erosion" and "dilation" (see also Chapter 7: Texture Analysis). Each of the above routines using them employs the same technique for interrogating neighboring pixels in the original image: as will be apparent on numerous occasions in this book, the sigma value is a useful and powerful descriptor for 3 × 3 pixel neighborhoods. Thus "if (sigma > 0)" can be taken to mean "if next to a dark object" and the consequence can be read as "then expand it." Similarly, "if (sigma < 8)" can be taken to mean "if next to a light object" or "if next to light background," and the consequence can be read as "then expand the light background into the dark object."

The process of finding the edge of a binary object has several possible interpretations. Clearly, it can be assumed that an edge point has a sigma value in the range 1–7 inclusive. However, it may be defined as being within the object, within the background, or in either position. Taking the definition that the edge of an object has to lie within the object (Fig. 2.8C), the following edge-finding routine for binary images results:

```
for all pixels in image do {
    sigma = A1 + A2 + A3 + A4 + A5 + A6 + A7 + A8;
    if (sigma == 8) B0 = 0; else B0 = A0;
}                                                        (2.17)
```

This strategy amounts to canceling out object pixels that are not on the edge. For this and a number of other algorithms (including the shrink and expand algorithms already encountered), a thorough analysis of exactly which pixels should be set to 1 and 0 (or which should be retained and which eliminated), involves drawing up tables of the form:

		Sigma	
		0 to 7	8
A0	0	0	0
	1	1	0

This reflects the fact that algorithm specification includes a recognition phase and an action phase, i.e., it is necessary first to locate situations within an image where (for example) edges are to be marked or noise eliminated, and then action must be taken to implement the change.

Another function that can usefully be performed on binary images is the removal of "salt and pepper" noise, i.e., noise that appears as a light spot on a dark background or a dark spot on a light background. The first problem to be solved is that of recognizing such noise spots; the second is the simpler one of correcting the intensity value. For the first of these tasks the sigma value is again useful. To remove salt noise (which has binary value 0 in our convention), we arrive at the following routine:

```
for all pixels in image do {
    sigma = A1 + A2 + A3 + A4 + A5 + A6 + A7 + A8;
    if (sigma == 8) B0 = 1;  else B0 = A0;                    (2.18)
}
```

which can be read as leaving the pixel intensity unchanged unless it is proven to be a salt noise spot. The corresponding routine for removing pepper noise (binary value 1) is:

```
for all pixels in image do {
    sigma = A1 + A2 + A3 + A4 + A5 + A6 + A7 + A8;
    if (sigma == 0) B0 = 0;  else B0 = A0;                    (2.19)
}
```

Combining these two routines into one operation (Fig. 2.9A) gives:

```
for all pixels in image do {
    sigma = A1 + A2 + A3 + A4 + A5 + A6 + A7 + A8;
    if (sigma == 0) B0 = 0;
    else if (sigma == 8) B0 = 1;                              (2.20)
    else B0 = A0;
}
```

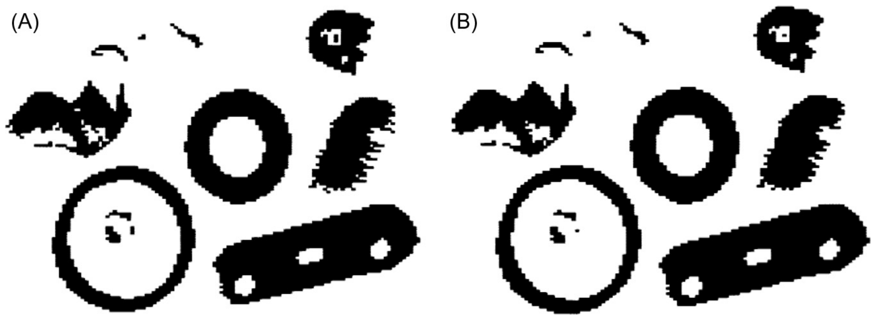

FIGURE 2.9

Simple binary noise removal operations: (A) result of applying a "salt and pepper" noise removal operation to the thresholded image in Fig. 2.7B; (B) result of applying a less stringent noise removal routine: this is effective in cutting down the jagged spurs that appear on some of the objects.

The routine can be made less stringent in its specification of noise pixels, so that it removes spurs on objects and background: this is achieved (Fig. 2.9B) by a variant such as:

```
for all pixels in image do {
    sigma = A1 + A2 + A3 + A4 + A5 + A6 + A7 + A8;
    if (sigma < 2) B0 = 0;
    else if (sigma > 6) B0 = 1;                          (2.21)
    else B0 = A0;
}
```

As before, if there is any doubt about the algorithm, its specification should be set up rigorously—as shown in the following table.

		Sigma		
		0 or 1	**2 to 6**	**7 or 8**
A0	**0**	0	0	1
	1	0	1	1

There are many other simple operations that can usefully be applied to binary images and some of them are dealt with in Chapter 8, Binary Shape Analysis.

2.3 CONVOLUTIONS AND POINT SPREAD FUNCTIONS

Convolution is a powerful and widely used technique in image processing and other areas of science. It appears in many applications throughout this book, and it is therefore useful to introduce it at an early stage. We start by defining the convolution of two functions $f(x)$ and $g(x)$ as the integral:

$$f(x) \otimes g(x) = \int_{-\infty}^{\infty} f(u)g(x-u)\mathrm{d}u \tag{2.22}$$

The action of this integral is normally described as the result of applying a point spread function (PSF) $g(x)$ to all points of a function $f(x)$ and accumulating the contributions at every point. It is significant that if the PSF is very narrow—ideally, a delta function—then the convolution is identical to the original function $f(x)$. This makes it natural to think of the function $f(x)$ as having been spread out under the influence of $g(x)$. This argument may give the impression that convolution necessarily blurs the original function but this is not always so if, for example, the PSF has a distribution of positive and negative values.

When convolution is applied to digital images, the above formulation changes in two ways: (1) a double integral must be used in respect of the two dimensions and (2) integration must be changed into discrete summation. The new form of the convolution is:

$$F(x,y) = f(x,y) \otimes g(x,y) = \sum_i \sum_j f(i,j) g(x-i, y-j) \tag{2.23}$$

where g is now referred to as a spatial convolution mask. The fact that the mask has to be inverted before it is applied is inconvenient for visualizing the process of convolution—particularly when matching operations are involved, e.g., for corner location (see Chapter 6: Corner, Interest Point, and Invariant Feature Detection). In this book we therefore present only preinverted masks of the form:

$$h(x,y) = g(-x, -y) \tag{2.24}$$

Convolution can then be calculated using the more intuitive formula:

$$F(x,y) = \sum_i \sum_j f(x+i, y+j) h(i,j) \tag{2.25}$$

which involves multiplying corresponding values in the modified mask and the neighborhood under consideration. Reexpressing this result for a 3 × 3 neighborhood and writing the mask coefficients in the form:

$$\begin{bmatrix} h4 & h3 & h2 \\ h5 & h0 & h1 \\ h6 & h7 & h8 \end{bmatrix}$$

the algorithm can be obtained in terms of our earlier notation:

```
for all pixels in image do {
    Q0 = P0*h0 + P1*h1 + P2*h2 + P3*h3 + P4*h4
            + P5*h5 + P6*h6 + P7*h7 + P8*h8;                (2.26)
}
```

We are now in a position to apply convolution to a real situation. At this stage we attempt to suppress noise by averaging over nearby pixels. A simple way of achieving this is to use the convolution mask:

$$\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

where the number in front of the mask weights all the coefficients in the mask and is inserted to ensure that applying the convolution does not alter the mean intensity in the image. As hinted above, this particular convolution has the effect of blurring the image as well as reducing the noise level (Fig. 2.10). More details will be shared about this in Chapter 3, Image Filtering and Morphology.

The above discussion makes it clear that convolutions are linear operators. In fact, they are the most general spatially invariant linear operators that can be applied to a signal such as an image. Note that linearity is often of interest in that it permits mathematical analysis to be performed that would otherwise be intractable.

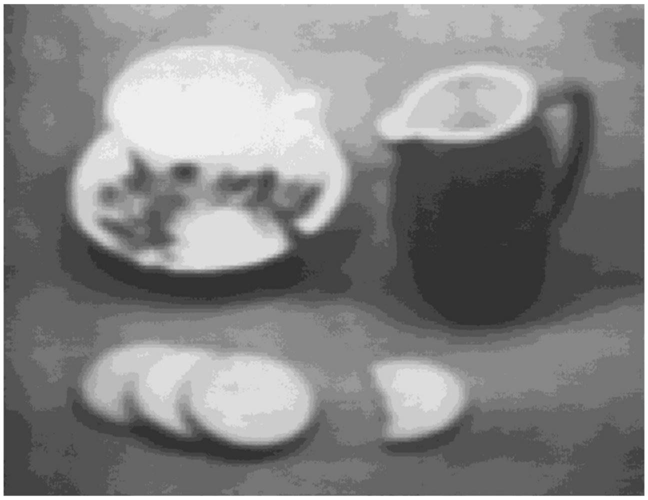

FIGURE 2.10

Noise suppression by neighborhood averaging achieved by convolving the original image of Fig. 2.1A with a uniform mask within a 3 × 3 neighborhood. Note that noise is suppressed only at the expense of introducing significant blurring.

2.4 SEQUENTIAL VERSUS PARALLEL OPERATIONS

It will be noticed that most of the operations defined so far have started with an image in one space and finished with an image in a different space. Unfortunately, many of the operations will not work satisfactorily if we do not use separate input and output spaces in this way. This is because they are inherently "parallel processing" routines. This term is used, as these are the types of process that would be performed by a parallel computer possessing a number of processing elements equal to the number of pixels in the image, so that all the pixels are processed simultaneously. If a serial computer is to *simulate* the operation of a parallel computer, then it must have separate input and output image spaces and rigorously work in such a way that it uses the original image values to compute the output pixel values. This means that an operation such as the following cannot be an ideal parallel process:

```
for all pixels in image do {
   sigma = A1 + A2 + A3 + A4 + A5 + A6 + A7 + A8;
   if (sigma < 8) A0 = 0; else A0 = A0;                  (2.27)
}
```

This is so because, when the operation is half completed, the output pixel intensity will depend not only on some of the unprocessed pixel values but also on some that have already been processed. For example, if the computer makes a normal (forward) TV raster scan through the image, the situation at a general point in the scan will be

$$\boxed{\begin{array}{ccc}\surd & \surd & \surd \\ \surd & \times & \times \\ \times & \times & \times\end{array}}$$

where the ticked pixels have already been processed and the others have not. As a result, the above operation will shrink all objects to nothing!

A much simpler illustration of this is obtained by attempting to shift an image to the right using the following routine:

$$\text{for all pixels in image do } \{\text{P0} = \text{P5};\} \tag{2.28}$$

In fact, all this achieves is to fill up the image with values corresponding to those off its left edge, whatever they are assumed to be. Thus we have shown that the shifting process is inherently parallel. (Note that whenever the computer is performing a 3 × 3 (or larger) window operation, it has to assume some value for off-image pixel intensities: usually whatever value is selected will be inaccurate, and so the final processed image will contain a border that is also inaccurate. This will be so whether the off-image pixel addresses are trapped in software or in specially designed circuitry in the frame store.)

It will be seen later that there are some processes that are inherently sequential—i.e., the processed pixel has to be returned immediately to the *original* image space. Meanwhile, note that not all of the routines described so far need to be restricted rigorously to parallel processing. In particular, all single-pixel routines (essentially, those that only refer to the single pixel in a 1 × 1 neighborhood) can validly be performed as if they were sequential in nature. Such routines include the following intensity adjustment and thresholding operations:

$$\text{for all pixels in image do } \{\text{P0} = \text{P0}^{*}\text{gamma} + \text{beta};\} \tag{2.29}$$

$$\text{for all pixels in image do } \left\{\text{if (P0} > \text{thresh) P0} = 1;\ \text{else P0} = 0;\right\} \tag{2.30}$$

These remarks are intended to act as a warning. In general, it is safest to design algorithms that are exclusively parallel processes unless there is a definite need to make them sequential. It will be seen later how this need can arise.

2.5 CONCLUDING REMARKS

This chapter has introduced a compact notation for representing imaging operations and has demonstrated some basic parallel processing routines. Chapter 3, Image Filtering and Morphology, extends this work to see how noise suppression can be achieved in grayscale images. This leads on to more advanced image analysis work that is directly relevant to machine vision applications. In particular, Chapter 4, The Role of Thresholding, studies in more detail the thresholding of grayscale images, building on the work of Section 2.2.1, while Chapter 8, Binary Shape Analysis, studies object shape analysis in binary images.

Pixel—pixel operations can be used to make radical changes in digital images. However, this chapter has shown that window—pixel operations are far more powerful, and capable of performing all manner of size- and-shape changing operations, as well as eliminating noise. But caveat emptor—sequential operations can have some odd effects if adventitiously applied.

2.6 BIBLIOGRAPHICAL AND HISTORICAL NOTES

Since the aim of this chapter was not to cover the most recent material but to provide a succinct overview of basic techniques, it will not be surprising that most of the topics discussed were developed well over 20 years ago and have been used by a large number of workers in many areas. For example, thresholding of grayscale images was first reported at least as long ago as 1960, while shrinking and expanding of binary picture objects dates from a similar period. Discussion of the origins of other techniques is curtailed: for further details, the reader is referred to the texts by (for example) Gonzalez and Woods (2008), Nixon and Aguado (2008), Petrou and Petrou (2010), and Sonka et al. (2007). We also refer to two texts that cover programming aspects of image processing in some depth: Parker (1994) that covers C programming and Whelan and Molloy (2001) that covers Java programming. More specialized texts will be referred to in the subsequent chapters.

2.7 PROBLEMS

1. Derive an algorithm for finding the edges of binary picture objects by applying a shrink operation and combining the result with the original image. Is the result the same as that obtained using the edge-finding routine (2.17)? Prove your statement rigorously by drawing up suitable algorithm tables as in Section 2.2.2.
2. In a certain frame store, each off-image pixel can be taken to have either the value 0 or the intensity for the nearest image pixel. Which will give the more meaningful results for (1) shrinking, (2) expanding, and (3) a blurring convolution?
3. Suppose the noise elimination routines of Eqs. (2.20) and (2.21) were reimplemented as sequential algorithms. Show that the action of the first would be unchanged, whereas the second would produce very odd effects on some binary images.