CHAPTER 7

GEOMETRIC OPERATIONS

WHAT WILL WE LEARN?

- What do geometric operations do to an image and what are they used for?
- What are the techniques used to enlarge/reduce a digital image?
- What are the main interpolation methods used in association with geometric operations?
- What are affine transformations and how can they be performed using MATLAB?
- How can I rotate, flip, crop, or resize images in MATLAB?
- What is image registration and where is it used?

7.1 INTRODUCTION

Geometric operations modify the geometry of an image by repositioning pixels in a constrained way. In other words, rather than changing the pixel values of an image (as most techniques studied in Part I of this book do), they modify the spatial relationships between groups of pixels representing features or objects of interest within the image. Figure 7.1 shows examples of typical geometric operations whose details will be presented later in this chapter.

Practical Image and Video Processing Using MATLAB®. By Oge Marques.

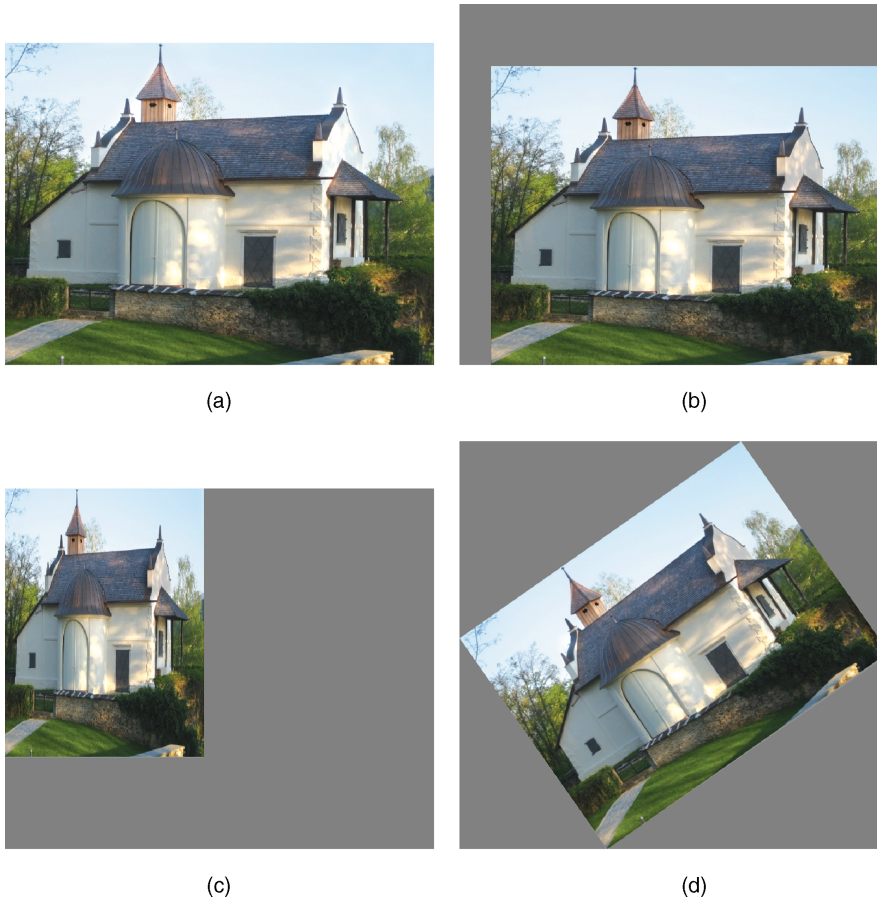

FIGURE 7.1 Examples of typical geometric operations: (a) original image; (b) translation (shifting); (c) scaling (resizing); (d) rotation.

Geometric operations can be used to accomplish different goals, such as the following:

- Correcting geometric distortions introduced during the image acquisition process (e.g., due to the use of a fish-eye lens).
- Creating special effects on existing images, such as twirling, bulging, or squeezing a picture of someone's face.
- As part of *image registration*—the process of matching the common features of two or more images of the same scene, acquired from different viewpoints or using different equipment.

Most geometric operations consist of two basic components:

1. *Mapping Function*: This is typically specified using a set of *spatial transformation equations* (and a procedure to solve them) (Section 7.2).
2. *Interpolation Methods*: These are used to compute the new value of each pixel in the spatially transformed image (Section 7.3).

7.2 MAPPING AND AFFINE TRANSFORMATIONS

A geometric operation can be described mathematically as the process of transforming an input image $f(x, y)$ into a new image $g(x', y')$ by modifying the *coordinates* of image pixels:

$$f(x, y) \rightarrow g(x', y') \tag{7.1}$$

that is, the pixel value originally located at coordinates (x, y) will be relocated to coordinates (x', y') in the output image.

To model this process, a *mapping function* is needed. The mapping function specifies the new coordinates (in the output image) for each pixel in the input image:

$$(x', y') = T(x, y) \tag{7.2}$$

This mapping function is an arbitrary 2D function. It is often specified as two separate functions, one for each dimension:

$$x' = T_x(x, y) \tag{7.3}$$

and

$$y' = T_y(x, y) \tag{7.4}$$

where T_x and T_y are usually expressed as polynomials in x and y. The case where T_x and T_y are linear combinations of x and y is called *affine transformation* (or *affine mapping*):

$$x' = a_0x + a_1y + a_2 \tag{7.5}$$

$$y' = b_0x + b_1y + b_2 \tag{7.6}$$

Equations (7.5) and (7.6) can also be expressed in matrix form as follows:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} a_0 & a_1 & a_2 \\ b_0 & b_1 & b_2 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{7.7}$$

Affine mapping transforms straight lines to straight lines, triangles to triangles, and rectangles to parallelograms. Parallel lines remain parallel and the distance ratio

TABLE 7.1 Summary of Transformation Coefficients for Selected Affine Transformations

Transformation	a_0	a_1	a_2	b_0	b_1	b_2
Translation by Δ_x, Δ_y	1	0	Δ_x	0	1	Δ_y
Scaling by a factor $[s_x, s_y]$	s_x	0	0	0	s_y	0
Counterclockwise rotation by angle θ	$\cos\theta$	$\sin\theta$	0	$-\sin\theta$	$\cos\theta$	0
Shear by a factor $[\mathrm{sh}_x, \mathrm{sh}_y]$	1	sh_y	0	sh_x	1	0

between points on a straight line does not change. Four of the most common geometric operations—translation, scaling, rotation, and shearing—are all special cases of equations (7.5) and (7.6), as summarized in Table 7.1.

The six parameters of the 2D affine mapping (equation (7.7)) are uniquely determined by three pairs of corresponding points. Given the coordinates of relevant points before and after the transformation, one can write n equations in x and y and solve them to find the n transformation coefficients. Figure 7.2 shows an example of a triangle (three vertices, $n = 6$), before and after the affine transformation.

In MATLAB

The IPT has two functions associated with affine transforms: `maketform` and `imtransform`. The `maketform` function is used to define the desired 2D spatial transformation. It creates a MATLAB structure (called a `TFORM`) that contains all the parameters required to perform the transformation. In addition to affine transformations, `maketform` also supports the creation of projective and custom transformations. After having defined the desired transformation, it can be applied to an input image using function `imtransform`. In Tutorial 7.2 (page 142), you will have a chance to learn more about these functions.

■ EXAMPLE 7.1

Generate the affine transformation matrix for each of the operations below: (a) rotation by 30°; (b) scaling by a factor 3.5 in both dimensions; (c) translation by [25, 15] pixels;

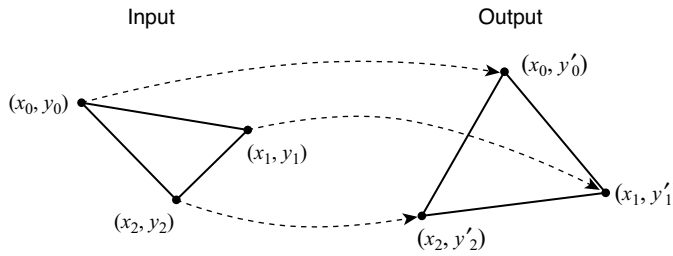

FIGURE 7.2 Mapping one triangle onto another by an affine transformation.

(d) shear by a factor [2, 3]. Use MATLAB to apply the resulting matrices to an input image of your choice.

Solution

Plugging the values into Table 7.1, we obtain the following:
(a) Since $\cos 30^\circ = 0.866$ and $\sin 30^\circ = 0.500$:

$$\begin{bmatrix} 0.866 & -0.500 & 0 \\ 0.500 & 0.866 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

(b)

$$\begin{bmatrix} 3.5 & 0 & 0 \\ 0 & 3.5 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

(c)

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 25 & 15 & 1 \end{bmatrix}$$

(d)

$$\begin{bmatrix} 1 & 3 & 0 \\ 2 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

MATLAB code:

```
filename = 'any image of your choice'
I =  imread(filename);

% Rotation
Ta = maketform('affine', ...
     [cosd(30) -sind(30) 0; sind(30) cosd(30) 0; 0 0 1]');
Ia = imtransform(I,Ta);

%Scaling
Tb = maketform('affine',[3.5 0 0; 0 3.5 0; 0 0 1]');
Ib = imtransform(I,Tb);
```

```
% Translation
xform = [1 0 25; 0 1 15; 0 0 1]';
Tc = maketform('affine',xform);
Ic = imtransform(I,Tc, 'XData', ...
     [1 (size(I,2)+xform(3,1))], 'YData', ...
     [1 (size(I,1)+xform(3,2))],'FillValues', 128 );

% Shearing
Td = maketform('affine',[1 3 0; 2 1 0; 0 0 1]');
Id = imtransform(I,Td);
```

7.3 INTERPOLATION METHODS

7.3.1 The Need for Interpolation

After a geometric operation has been performed on the original image, the resulting value for each pixel can be computed in two different ways. The first one is called *forward mapping*—also known as *source-to-target mapping* (Figure 7.3)—and consists of iterating over every pixel of the input image, computing its new coordinates, and copying the value to the new location. This approach has a number of problems, such as the following:

- Many coordinates calculated by the transformation equations are not integers, and need to be rounded off to the closest integer to properly index a pixel in the output image.
- Many coordinates may lie out of bounds (e.g., negative values).

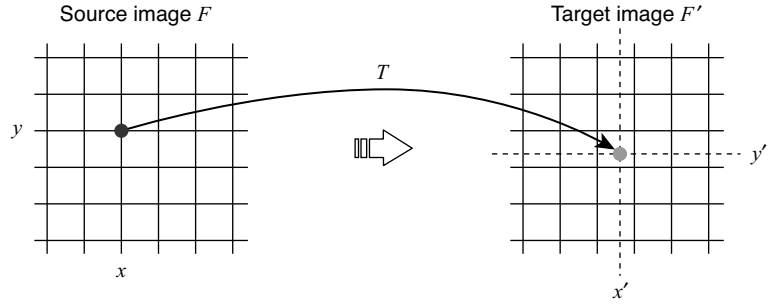

FIGURE 7.3 Forward mapping: for each pixel position in the input image, the corresponding (continuous) target position—resulting from applying a geometric transformation T—is found in the output image. In general, the target position (x', y') does not coincide with any discrete raster point, and the value of the pixel in the input image is copied to one of the adjacent target pixels. Redrawn from [BB08].

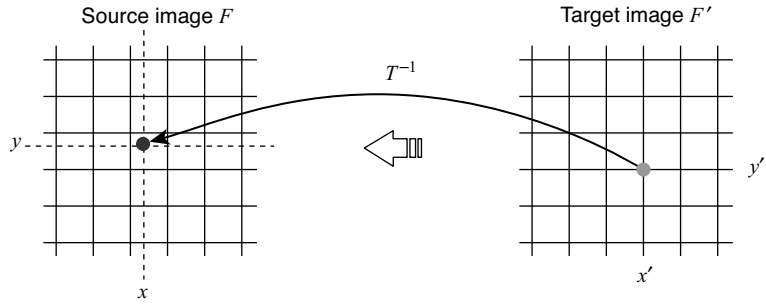

FIGURE 7.4 Backward mapping: for each discrete pixel position in the output image, the corresponding continuous position in the input image (x, y) is found by applying the inverse mapping function T^{-1}. The new pixel value is found by interpolation among the neighbors of (x, y) in the input image. Redrawn from [BB08].

- Many output pixels' coordinates are addressed several times during the calculations (which is wasteful) and some are not addressed at all (which leads to "holes" in the output image, meaning that no pixel value was computed for that coordinate pair).

The solution to the limitations listed above usually comes in the form of a *backward mapping*—also known as *target-to-source mapping* (Figure 7.4)—approach, which consists of visiting every pixel in the output image and applying the *inverse* transformation to determine the coordinates in the input image from which a pixel value must be sampled. Since this backward mapping process often results in coordinates outside the sampling grid in the original image, it usually requires some type of *interpolation* to compute the best value for that pixel.

7.3.2 A Simple Approach to Interpolation

If you were asked to write code to enlarge or reduce an image by a certain factor (e.g., a factor of 2 in both directions), you would probably deal with the problem of removing pixels (in the case of shrinking) by subsampling the original image by a factor of 2 in both dimensions, that is, skipping every other pixel along each row and column. Conversely, for the task of enlarging the image by a factor of 2 in both dimensions, you would probably opt for copying each original pixel to an $n \times n$ block in the output image. These simple interpolation schemes (*pixel removal* and *pixel duplication*, respectively) are fast and easy to understand, but suffer from several limitations, such as the following:

- The "blockiness" effect that may become noticeable when enlarging an image.
- The possibility of removing essential information in the process of shrinking an image.

- The difficulty in extending these approaches to arbitrary, noninteger, resizing factors.

Other simplistic methods—such as using the mean (or median) value of the original $n \times n$ block in the input image to determine the value of each output pixel in the shrunk image—also produce low-quality results and are bound to fail in some cases. These limitations call for improved interpolation methods, which will be briefly described next.

7.3.3 Zero-Order (Nearest-Neighbor) Interpolation

This baseline interpolation scheme rounds off the calculated coordinates (x', y') to their nearest integers. Zero-order (or *nearest-neighbor*) interpolation is simple and computationally fast, but produces low-quality results, with artifacts such as blockiness effects—which are more pronounced at large-scale factors—and jagged straight lines—particularly after rotations by angles that are not multiples of 90° (see Figure 7.5b).

7.3.4 First-Order (Bilinear) Interpolation

First-order (or bilinear) interpolation calculates the gray value of the interpolated pixel (at coordinates (x', y')) as a weighted function of the gray values of the four pixels surrounding the reference pixel in the input image. Bilinear interpolation produces visually better results than the nearest-neighbor interpolation at the expense of additional CPU time (see Figure 7.5c).

7.3.5 Higher Order Interpolations

Higher order interpolations are more sophisticated—and computationally expensive—methods for interpolating the gray value of a pixel. The third-order interpolation scheme implemented in several MATLAB functions is also known as *bicubic interpolation*. It takes into account the 4×4 neighborhood around the reference pixel and computes the resulting gray level of the interpolated pixel by performing the convolution of the 4×4 neighborhood with a cubic function.

Figure 7.5 shows the results of using different interpolation schemes to rotate an image by 35°. The jagged edge effect of the zero-order interpolation is visible (in part b), but there is little—if any—perceived difference between the bipolar (part c) and bicubic (part d) results.

7.4 GEOMETRIC OPERATIONS USING MATLAB

In this section, we will present a summary of typical geometric operations involving digital images that can easily be implemented using MATLAB and the IPT.

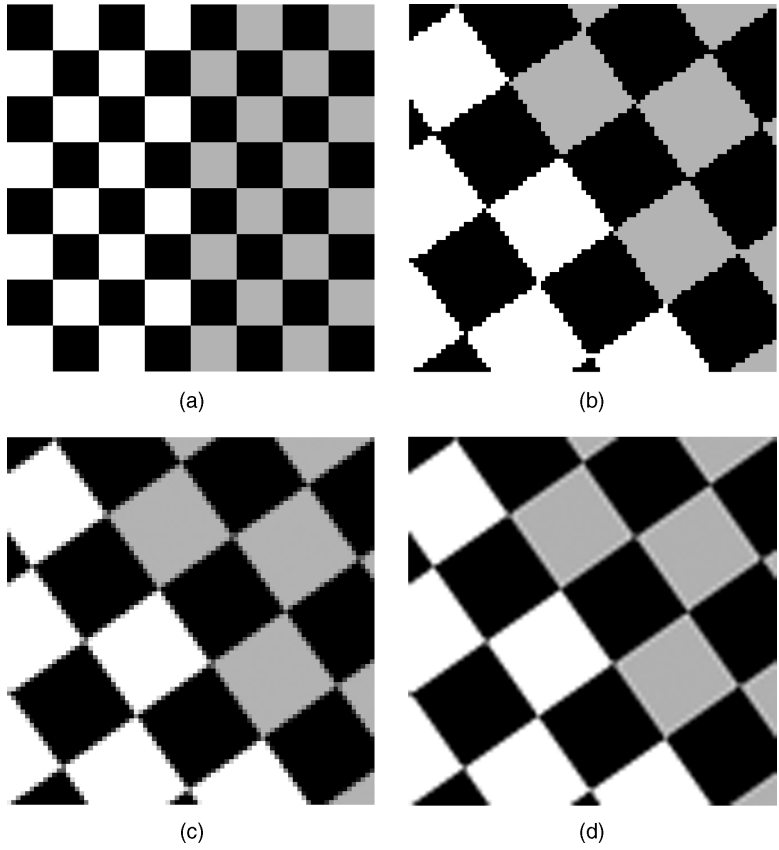

FIGURE 7.5 Effects of different interpolation techniques on rotated images: (a) original image; zoomed-in versions of rotated (35°) image using (b) zero-order (nearest-neighbor) interpolation; (c) first-order (bilinear) interpolation; (d) third-order (bicubic) interpolation.

7.4.1 Zooming, Shrinking, and Resizing

One of the most common geometric operations is the resize operation. In this book, we distinguish between *true image resizing*—where the resulting image size (in pixels) is changed—and *resizing an image for human viewing*—which we will refer to as *zooming (in)* and *shrinking* (or *zooming out*). They are both useful image processing operations and often rely on the same underlying algorithms. The main difference lies in the fact that zooming and shrinking are usually performed interactively (with a tool such as IPT's `imshow` or `imtool`) and their results last for a brief moment, whereas resizing is typically accomplished in a noninteractive way (e.g., as part of a MATLAB script) and its results are stored for longer term use.

The IPT has a function for resizing images, `imresize`. The `imresize` function allows the user to specify the interpolation method used (nearest-neighbor, bilinear, or

bicubic interpolation—the default method). It is a rather sophisticated function, which also allows specification of an interpolation kernel and additional parameter/value pairs. In Tutorial 7.1 (page 138), you will use this function and explore some of its options.

7.4.2 Translation

Translation of an input image $f(x, y)$ with respect to its Cartesian origin to produce an output image $g(x', y')$ where each pixel is displaced by $[\Delta_x, \Delta_y]$ (i.e., $x' = x + \Delta_x$ and $y' = y + \Delta_y$) consists of a special case of affine transform (as discussed in Section 7.2). In Tutorial 7.2 (page 142), you will use `maketform` and `imtransform` to perform image translation.

7.4.3 Rotation

Rotation of an image constitutes another special case of affine transform (as discussed in Section 7.2). Consequently, image rotation can also be accomplished using `maketform` and `imtransform`.

The IPT also has a specialized function for rotating images, `imrotate`. Similar to `imresize`, `imrotate` allows the user to specify the interpolation method used: nearest-neighbor (the default method), bilinear, or bicubic. It also allows specification of the size of the output image. In Tutorials 7.1 (page 138), and 7.2 (page 142), you will explore the `imrotate` function.

7.4.4 Cropping

The IPT has a function for cropping images, `imcrop`, which crops an image to a specified rectangle. The crop rectangle can be specified interactively (with the mouse) or its coordinates be passed as parameters to the function. In Tutorial 7.1 (page 138), you will experiment with both options for using this function.

7.4.5 Flipping

The IPT has two functions for flipping matrices (which can also be used for raster images, of course): `flipud`—which flips a matrix up to down—and `fliplr`—which flips a matrix left to right. In Tutorial 7.1 (page 138), you will experiment with both functions.

7.5 OTHER GEOMETRIC OPERATIONS AND APPLICATIONS

7.5.1 Warping

Warping can be defined as the "transformation of an image by reparameterization of the 2D plane" [FDHF+05]. Warping techniques are sometimes referred to as *rubber*

sheet transformations, because they resemble the process of applying an image to a sheet of rubber and stretching it according to a predefined set of rules.

The *quadratic warp* is a particular case of *polynomial warping*, where the transformed coordinates (x', y') for a pixel whose original coordinates are (x, y) are given by the following equations:

$$x' = a_0x^2 + a_1y^2 + a_2xy + a_3x + a_4y + a_5 \tag{7.8}$$

$$y' = b_0x^2 + b_1y^2 + b_2xy + b_3x + b_4y + b_5 \tag{7.9}$$

where the coefficients $a_0, ..., a_5, b_0, ..., b_5$ are typically chosen to introduce more complex distortions into an image, for example, turning straight lines into curves. One practical application of the quadratic warping method is to compensate for lens distortions, particularly the *barrel* and *pincushion* distortions discussed in Chapter 5. A variant of equations (7.8) and (7.9), using third-degree polynomials and 20 coefficients, is called a *cubic warp*.

Warping operations of order 2 or higher are usually specified using control points in the source image and mapping them to specified locations in the destination image. Control points are usually associated with key locations and features in the image, such as corners of objects. It is possible to specify more than the minimally required number of control points; in these cases, a least-square method is used to determine the coefficients that best match the desired displacements.

Piecewise warping is an alternative to polynomial warping. It allows the desired warping to be specified with the help of a control grid on top of the input image. The user specifies which control points should be moved by dragging the intersections of the gridlines to new locations using the mouse.

7.5.2 Nonlinear Image Transformations

Nonlinear image transformations usually involve a conversion from rectangular to polar coordinates followed by a deliberate distortion of the resulting points.

Twirling The twirl transformation causes an image to be rotated around an anchor point of coordinates (x_c, y_c) with a space-variant rotation angle: the angle has a value of α at the anchor point and decreases linearly with the radial distance from the center. The effect is limited to a region within the maximum radius r_{max}. All pixels outside this region remain unchanged.

Since this transformation uses backward mapping, we are interested in the equations for the *inverse* mapping function:

$$T_x{}^{-1} : x = \begin{cases} x_c + r\cos(\theta) & \text{for } r \le r_{max} \\ x' & \text{for } r > r_{max} \end{cases} \tag{7.10}$$

and

$$T_y^{-1} : y = \begin{cases} y_c + r\sin(\theta) & \text{for } r \le r_{\max} \\ y' & \text{for } r > r_{\max} \end{cases} \tag{7.11}$$

where

$$d_x = x' - x_c, \quad r = \sqrt{{d_x}^2 + {d_y}^2}$$
$$d_y = y' - y_c, \quad \theta = \arctan(d_x, d_y) + \alpha \cdot \left(\frac{r_{\max} - r}{r_{\max}}\right)$$

Rippling The ripple transformation causes a local wave-like displacement of the image along both directions, x and y. The parameters for this mapping function are the (nonzero) period lengths L_x, L_y (in pixels) and the associated amplitude values A_x, A_y. The inverse transformation function is given by the following:

$$T_x^{-1} : x = x' + A_x \cdot \sin\left(\frac{2\pi \cdot y'}{L_x}\right) \tag{7.12}$$

$$T_y^{-1} : y = y' + A_y \cdot \sin\left(\frac{2\pi \cdot x'}{L_y}\right) \tag{7.13}$$

Twirling, rippling, and many other nonlinear transformations are often used to create an artistic (or humorous) effect on an image by causing controllable deformations (Figure 7.6).

FIGURE 7.6 Image deformation effects using *Photo Booth*.

7.5.3 Morphing

Morphing is a geometric transformation technique that gradually converts an image into another. The idea is to produce a visible *metamorphosis* effect as intermediate images are being displayed. Image morphing was quite popular in TV, movies, and advertisements in the 1980s and 1990s, but has lost impact since then.

Morphing can be seen as a modified version of piecewise warping, in which the user specifies control points in both the initial and final images. These control points are then used to generate two meshes (one from each image). Affine transformations relate the resulting meshes. An important aspect of morphing is that the warp is computed incrementally, one small step at a time, in combination with a dissolve effect from the initial image to the final one.

An alternative method for image morphing, *fields-based morphing*, originally proposed by Beier and Neely [BN92], does not use meshes. It relies on pairs of reference lines drawn on both images and computes the perpendicular distance between each pixel and each control line. It then uses distance and relative position to determine the correct position where a pixel should be placed in the final image. In this method, all control lines influence, to some extent, the outcome for a certain pixel: the closer the line, the stronger the influence.

7.5.4 Seam Carving

Seam carving [AS07] is a recently proposed—but already very popular—image operator for content-aware image resizing. The basic idea of the algorithm is to find *seams*[1] in the original image and use that information to either (i) reduce the image size by removing ("carving out") the seams that contribute the least to the image's contents or (ii) enlarge the image by inserting additional seams. By applying these operators in both directions, the image can be retargeted to a new size with very little loss of meaningful contents. Figure 7.7 shows an example of seam carving for content-aware image resizing.

7.5.5 Image Registration

Image registration is the process of aligning two or more images of the same scene. First, each input image is compared with a *reference image*—also known as *base image*. Next, a spatial transformation is applied to the input image in order to align it with the base image. The key step in image registration is determining the parameters of the spatial transformation required to bring the images into alignment. This is a complex and fascinating topic for which many books have been written (see "Learn More About It" section at the end of the chapter).

[1] A seam is an "optimal 8-connected path of pixels on a single image from top to bottom, or left to right, where optimality is defined by an image energy function" [AS07].

(a) (b)

FIGURE 7.7 Using seam carving for content-aware resizing: (a) original image (334 × 500 pixels); (b) cropped image (256 × 256 pixels). Original image from Flickr. Seam carving results were obtained using the publicly available implementation by Mathias Lux: http://code.google.com/p/java-imageseams/.

The IPT contains a *Control Point Selection* tool for interactive image registration (Figure 7.10). You will learn how to use that tool to perform basic image registration tasks in Tutorial 7.2 (page 142).

7.6 TUTORIAL 7.1: IMAGE CROPPING, RESIZING, FLIPPING, AND ROTATION

Goal

The goal of this tutorial is to learn how to crop, resize, and rotate digital images.

Objectives

- Learn how to crop an image using the `imcrop` function.
- Learn how to resize an image using the `imresize` function.
- Learn how to flip an image upside down and left–right using `flipud` and `fliplr`.
- Learn how to rotate an image using the `imrotate` function.
- Explore interpolation methods for resizing and rotating images.

Procedure

In the first part of this tutorial, you will learn how to crop an image. Cropping in MATLAB can be done interactively—using the *Crop Image* option in the Image Tool (`imtool`) toolbar—or programmatically—using the `imcrop` function.

1. Open the `cameraman` image and use the *Crop Image* option in the Image Tool (`imtool`) toolbar to crop it such that only the portion of the image containing the tallest building in the background is selected to become the cropped image. Pay attention to (and write down) the coordinates of the top left and bottom right corners as you select the rectangular area to be cropped. You will need this information for the next step.
2. Double-click inside the selected area to complete the cropping operation.
3. Save the resulting image using the **File > Save as...** option in the `imtool` menu. Call it `cropped_building.png`.

```
I = imread('cameraman.tif');
imtool(I)
```

Question 1 Which numbers did you record for the top left and bottom right coordinates and what do they mean? *Hint*: Pay attention to the convention used by the **Pixel info** status bar at the bottom of the `imtool` main window. The IPT occasionally uses a so-called *spatial coordinate system*, whereas y represents rows and x represents columns. This does *not* correspond to the image axis coordinate system defined in Chapter 2.

4. Open and display the cropped image.

```
I2 = imread('cropped_building.png');
imshow(I2)
```

5. We shall now use the coordinates recorded earlier to perform a similar cropping from a script.
6. The `imcrop` function expects the crop rectangle—a four-element vector `[xmin ymin width height]`—to be passed as a parameter.
7. Perform the steps below replacing my values for `x1`, `y1`, `x2`, and `y2` with the values you recorded earlier.

```
x1 = 186;  x2 = 211;  y1 = 105;  y2 = 159;
xmin = x1;  ymin = y1;  width = x2-x1;  height = y2-y1;
I3 = imcrop(I, [xmin ymin width height]);
imshow(I3)
```

Resizing an image consists of enlarging or shrinking it, using nearest-neighbor, bilinear, or bicubic interpolation. Both resizing procedures can be executed using the `imresize` function. Let us first explore enlarging an image.

8. Enlarge the `cameraman` image by a scale factor of 3. By default, the function uses bicubic interpolation.

```
I_big1 = imresize(I,3);
figure, imshow(I), title('Original Image');
figure, imshow(I_big1), ...
    title('Enlarged Image w/ bicubic interpolation');
```

As you have seen in Chapter 4, the IPT function `imtool` can be used to inspect the pixel values of an image.[2] The `imtool` function provides added functionality to visual inspection of images, such as zooming and pixel inspection.

9. Use the `imtool` function to inspect the resized image, `I_big1`.

```
imtool(I_big1)
```

10. Scale the image again using nearest-neighbor and bilinear interpolations.

```
I_big2 = imresize(I,3,'nearest');
I_big3 = imresize(I,3,'bilinear');
figure, imshow(I_big2), ...
    title('Resized w/ nearest-neighbor interpolation');
figure, imshow(I_big3), ...
    title('Resized w/ bilinear interpolation');
```

Question 2 Visually compare the three resized images. How do they differ?

One way to shrink an image is by simply deleting rows and columns of the image.

11. Close any open figures.
12. Reduce the size of the cameraman image by a factor of 0.5 in both dimensions.

```
I_rows = size(I,1);
I_cols = size(I,2);
I_sm1 = I(1:2:I_rows, 1:2:I_cols);
figure, imshow(I_sm1);
```

Question 3 How did we scale the image?

Question 4 What are the limitations of this technique?

Although the technique above is computationally efficient, its limitations may require us to use another method. Just as we used the `imresize` function for enlarging, we can just as well use it for shrinking. When using the `imresize` function, a scale factor larger than 1 will produce an image larger than the original, and a scale factor smaller than 1 will result in an image smaller than the original.

[2]Note that this function is available only in MATLAB Version 7 and above.

13. Shrink the image using the `imresize` function.

```
I_sm2 = imresize(I,0.5,'nearest');
I_sm3 = imresize(I,0.5,'bilinear');
I_sm4 = imresize(I,0.5,'bicubic');
figure, subplot(1,3,1), imshow(I_sm2), ...
    title('Nearest-neighbor Interpolation');
subplot(1,3,2), imshow(I_sm3), title('Bilinear Interpolation');
subplot(1,3,3), imshow(I_sm4), title('Bicubic Interpolation');
```

Note that in the case of shrinking using either bilinear or bicubic interpolation, the `imresize` function automatically applies a low-pass filter to the image (whose default size is 11×11), slightly blurring it before the image is interpolated. This helps to reduce the effects of aliasing during resampling (see Chapter 5).

Flipping an image upside down or left–right can be easily accomplished using the `flipud` and `fliplr` functions.

14. Close all open figures and clear all workspace variables.
15. Flip the `cameraman` image upside down.
16. Flip the `cameraman` image from left to right.

```
I =  imread('cameraman.tif');
J = flipud(I);
K = fliplr(I);
subplot(1,3,1), imshow(I), title('Original image')
subplot(1,3,2), imshow(J), title('Flipped upside-down')
subplot(1,3,3), imshow(K), title('Flipped left-right')
```

Rotating an image is achieved through the `imrotate` function.

17. Close all open figures and clear all workspace variables.
18. Rotate the `eight` image by an angle of 35°.

```
I = imread('eight.tif');
I_rot = imrotate(I,35);
imshow(I_rot);
```

Question 5 Inspect the size (number of rows and columns) of `I_rot` and compare it with the size of `I`. Why are they different?

Question 6 The previous step rotated the image counterclockwise. How would you rotate the image 35° clockwise?

We can also use different interpolation methods when rotating the image.

19. Rotate the same image using bilinear interpolation.

```
I_rot2 = imrotate(I,35,'bilinear');
figure, imshow(I_rot2)
```

Question 7 How did bilinear interpolation affect the output of the rotation? *Hint:* The difference is noticeable between the two images near the edges of the rotated image and around the coins.

20. Rotate the same image, but this time crop the output.

```
I_rot3 = imrotate(I,35,'bilinear','crop');
figure, imshow(I_rot3)
```

Question 8 How did the `crop` setting change the size of our output?

7.7 TUTORIAL 7.2: SPATIAL TRANSFORMATIONS AND IMAGE REGISTRATION

In this tutorial, we will explore the IPT's functionality for performing spatial transformations (using the `imtransform`, `maketform`, and other related functions). We will also show a simple example of selecting control points (using the IPT's *Control Point Selection* tool) and using spatial transformations in the context of image registration.

In the first part of this tutorial, you will use `imtransform` and `maketform` to implement affine transformations (see Table 7.1), apply them to a test image, and inspect the results.

1. Open the `cameraman` image.
2. Use `maketform` to make an affine transformation that resizes the image by a factor $[s_x, s_y]$. The `maketform` function can accept transformation matrices of various sizes for N-dimensional transformations. But since `imtransform` only performs 2D transformations, you can only specify 3×3 transformation matrices. For affine transformations, the first two columns of the 3×3 matrices will have the values $a_0, a_1, a_2, b_0, b_1, b_2$ from Table 7.1, whereas the last column must contain `0 0 1`.
3. Use `imtransform` to apply the affine transformation to the image.
4. Compare the resulting image with the one you had obtained using `imresize`.

```
I1 = imread('cameraman.tif');
sx = 2; sy = 2;
```

```
T = maketform('affine',[sx 0 0; 0 sy 0; 0 0 1]');
I2 = imtransform(I1,T);
imshow(I2), title('Using affine transformation')
I3 = imresize(I1, 2);
figure, imshow(I3), title('Using image resizing')
```

Question 1 Compare the two resulting images (`I2` and `I3`). Inspect size, gray-level range, and visual quality. How are they different? Why?

5. Use `maketform` to make an affine transformation that rotates an image by an angle θ.
6. Use `imtransform` to apply the affine transformation to the image.
7. Compare the resulting image with the one you had obtained using `imrotate`.

```
I1 = imread('cameraman.tif');
theta = 35*pi/180
xform = [cos(theta) sin(theta) 0; -sin(theta) cos(theta) 0; 0 0 1]'
T = maketform('affine',xform);
I4 = imtransform(I1, T);
imshow(I4), title('Using affine transformation')
I5 = imrotate(I1, 35);
figure, imshow(I5), title('Using image rotating')
```

Question 2 Compare the two resulting images (`I4` and `I5`). Inspect size, gray-level range, and visual quality. How are they different? Why?

8. Use `maketform` to make an affine transformation that translates an image by Δ_x, Δ_y.
9. Use `imtransform` to apply the affine transformation to the image and use a fill color (average gray in this case) to explicitly indicate the translation.
10. Display the resulting image.

```
I1 = imread('cameraman.tif');
delta_x = 50;
delta_y = 100;
xform = [1 0 delta_x; 0 1 delta_y; 0 0 1]'
tform_translate = maketform('affine',xform);
I6 = imtransform(I1, tform_translate,...
     'XData', [1 (size(I1,2)+xform(3,1))],...
     'YData', [1 (size(I1,1)+xform(3,2))],...
     'FillValues', 128 );
figure, imshow(I6)
```

Question 3 Compare the two images (`I1` and `I6`). Inspect size, gray-level range, and visual quality. How are they different? Why?

11. Use `maketform` to make an affine transformation that performs shearing by a factor [sh_x, sh_y] on an input image.
12. Use `imtransform` to apply the affine transformation to the image.
13. Display the resulting image.

```
I = imread('cameraman.tif');
sh_x = 2; sh_y = 1.5;
xform = [1 sh_y 0; sh_x 1 0; 0 0 1]'
T = maketform('affine',xform);
I7 = imtransform(I1, T);
imshow(I7)
```

Image Registration

In the last part of the tutorial, you will learn how to use spatial transformations in the context of image registration. The main steps are illustrated in a block diagram format in Figure 7.8.

14. Open the base image (Figure 7.9a) and the unregistered image (Figure 7.9b).

```
base = imread('klcc_a.png');
unregistered = imread('klcc_b.png');
```

15. Specify control points in both images using `cpselect` (Figure 7.10). This is an interactive process that is explained in detail in the IPT online documentation. For the purpose of this tutorial, we will perform the following:
    - Open the *Control Point Selection* tool.
    - Choose a zoom value that is appropriate and lock the ratio.
    - Select the *Control Point Selection* tool in the toolbar.
    - Select a total of 10 control points per image, making sure that after we select a point in one image with click on the corresponding point in the other image, thereby establishing a match for that point. See Figure 7.11 for the points I chose.
    - Save the resulting control points using the **File > Export Points to Workspace** option in the menu.

```
cpselect(unregistered, base);
```

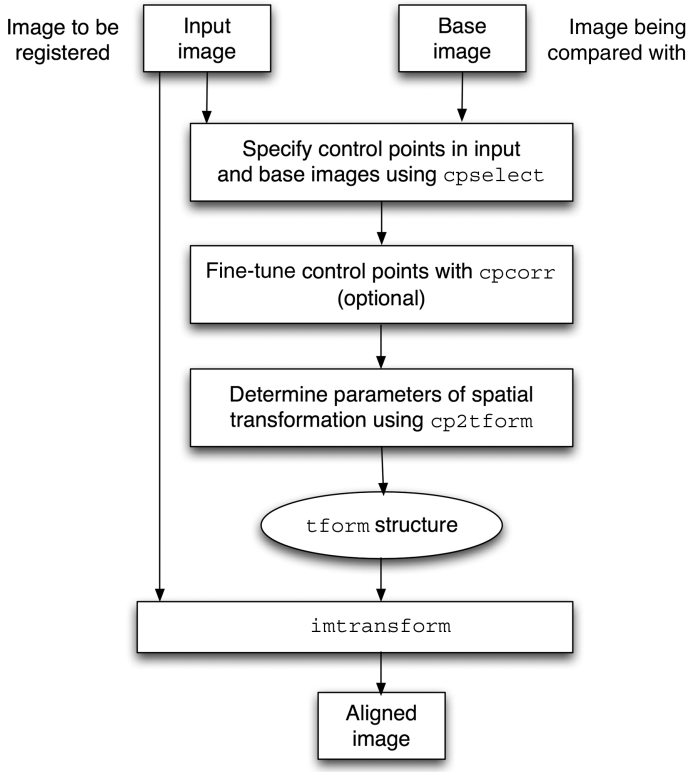

FIGURE 7.8 Image registration using MATLAB and the IPT.

(a) (b)

FIGURE 7.9 Interactive image registration: (a) base image; (b) unregistered image.

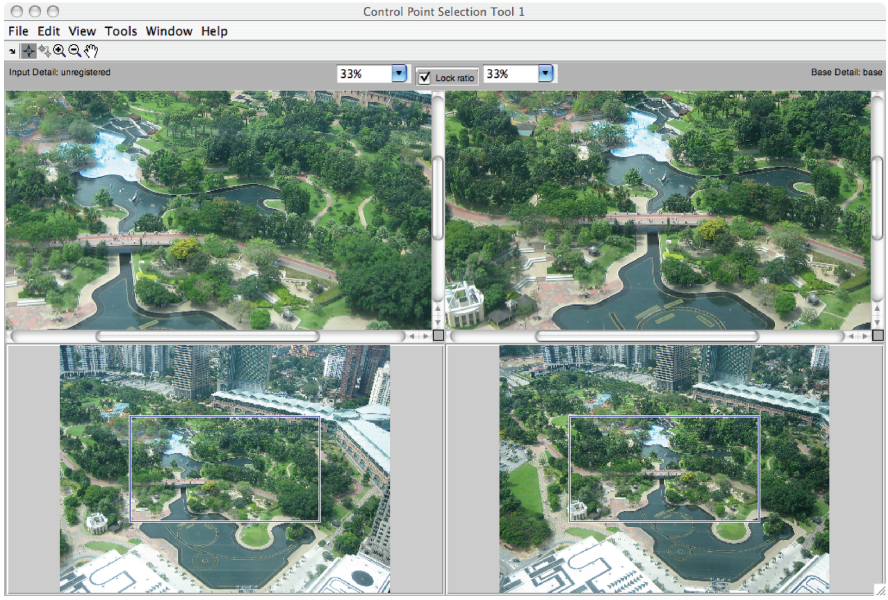

FIGURE 7.10 The *Control Point Selection* tool.

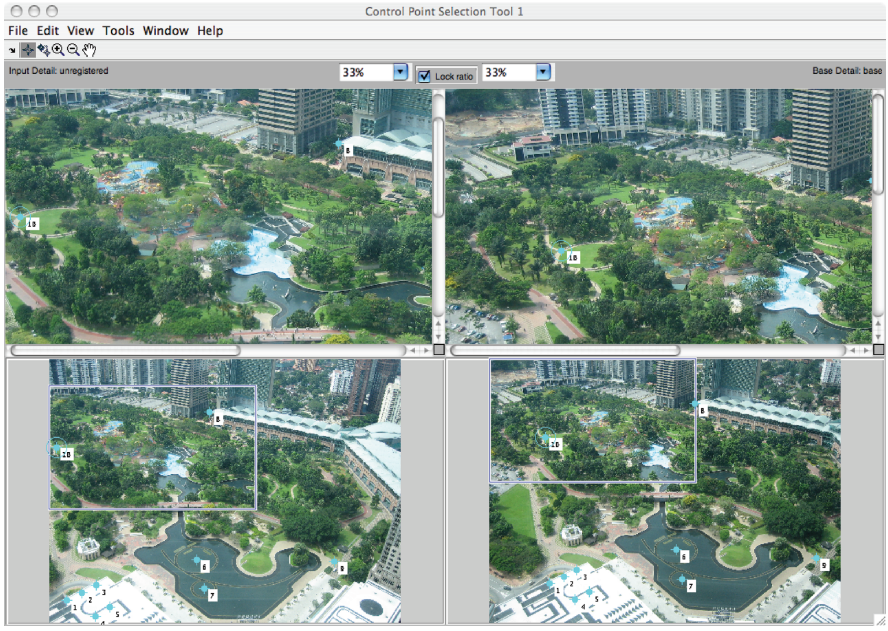

FIGURE 7.11 Selected points.

16. Inspect the coordinates of the selected control points.

```
base_points
input_points
```

17. Use `cpcorr` to fine-tune the selected control points.

```
input_points_adj = cpcorr(input_points,base_points,...
                          unregistered(:,:,1),base(:,:,1))
```

Question 4 Compare the values for `input_points_adj` with that for `input_points`. Did you notice any changes? Why (not)?

18. This is a critical step. We need to specify the type of transformation we want to apply to the unregistered image based on the type of distortion that it contains. In this case, since the distortion appears to be a combination of translation, rotation, and scaling, we shall use the `'nonreflective similarity'` transformation type. This type requires only two pairs of control points.
19. Once we have selected the type of transformation, we can determine its parameters using `cp2tform`.
20. Use the resulting `tform` structure to align the unregistered image (using `imtransform`).

```
% Select the type of transformation
mytform1 = cp2tform(input_points,base_points,...
            'nonreflective similarity');

% Transform the unregistered image
info = imfinfo('klcc_a.png');
registered = imtransform(unregistered,mytform1,...
               'XData',[1 info.Width], 'YData',[1 info.Height]);
```

21. Display the registered image overlaid on top of the base image.

```
figure, imshow(registered);
hold on
h = imshow(base);
set(h, 'AlphaData', 0.6)
```

Question 5 Are you happy with the results? If you had to do it again, what would you do differently?

WHAT HAVE WE LEARNED?

- Geometric operations modify the geometry of an image by repositioning pixels in a constrained way. They can be used to remove distortions in the image acquisition process or to deliberately introduce a distortion that matches an image with another (e.g., *morphing*).
- Enlarging or reducing a digital image can be done with two different purposes in mind: (1) to actually change the image's dimensions (in pixels), which can be accomplished in MATLAB by function `imresize`; (2) to temporarily change the image size for viewing purposes, through zooming in/out operations, which can be accomplished in MATLAB as part of the functionality of image display primitives such as `imtool` and `imshow`.
- The main interpolation methods used in association with geometric operations are zero-order (or *nearest-neighbor*) interpolation (simple and fast, but leads to low-quality results), first-order (or bilinear) interpolation, and higher-order (e.g., bicubic) interpolation (more sophisticated—and computationally expensive—but leads to best results).
- Affine transformations are a special class of geometric operations, such that once applied to an image, straight lines are preserved and parallel lines remain parallel. Translation, rotation, scaling, and shearing are all special cases of affine transformations. MATLAB's IPT has two functions associated with affine transformations: `maketform` and `imtransform`.
- Image rotation can be performed using the IPT `imrotate` function.
- An image can be flipped horizontally or vertically in MATLAB using simple linear algebra and matrix manipulation instructions.
- The IPT has a function for cropping images, `imcrop`, which crops an image to a specified rectangle and which can be specified either interactively (with the mouse) or via parameter passing.
- Image warping is a technique by which an image's geometry is changed according to a template.
- Image morphing is a geometric transformation technique that converts an image into another in an incremental way. It was popular in TV, movies, and advertisements in the 1980s and 1990s, but has lost impact since then.

LEARN MORE ABOUT IT

- Chapter 5 of [GWE04] contains a MATLAB function for visualizing affine transforms using grids.
- For a deeper coverage of (advanced) interpolation methods, we recommend Chapter 16 of [BB08] and Sections 10.5 and 10.6 of [Jah05].
- Zitová and Flusser [ZF03] have published a survey of image registration methods. For a book-length treatment of the topic, refer to [Gos05].

- The book by Wolberg [Wol90] is a historical reference for image warping.
- For more on warping and morphing, refer to [GDCV99].

ON THE WEB

- MATLAB Image Warping—E. Meyers (MIT)
  http://web.mit.edu/emeyers/www/warping/warp.html
- Image warping using MATLAB GUI (U. of Sussex, England)
  http://www.lifesci.sussex.ac.uk/research/cuttlefish/image_warping_software.htm
- Image morphing with MATLAB
  http://www.stephenmullens.co.uk/image_morphing/

7.8 PROBLEMS

7.1 Use `imrotate` to rotate an image by an arbitrary angle (not multiple of 90°) using the three interpolation methods discussed in Section 7.3. Compare the visual quality of the results obtained with each method and their computational cost (e.g., using MATLAB functions `tic` and `toc`).

7.2 Consider the MATLAB snippet below (assume `X` is a gray-level image) and answer this question: Will `X` and `Z` be identical? Explain.

```
Y = imresize(X,0.5,'nearest');
Z = imresize(Y,2.0,'nearest');
```

7.3 Consider the MATLAB snippet below. It creates an 80×80 black-and-white image (`B`) and uses a simple approach to image interpolation (described in Section 7.3.2) to reduce it to a 40×40 pixel equivalent (`C`). Does the method accomplish its goal? Explain.

```
A = eye(80,80);
A(26:2:54,:)=1;
B = imcomplement(A);
C = B(1:2:end, 1:2:end);
```