21 Geometric Operations

Common to all the filters and point operations described so far is the fact that they may change the intensity function of an image but the position of each pixel, and thus the geometry of the image, remains the same. The purpose of geometric operations, which are discussed in this chapter, is to deform an image by altering its geometry. Typical examples are shifting, rotating, or scaling images, as shown in Fig. 21.1. Geometric operations are frequently needed in practical applications, for example, in virtually any modern graphical computer interface. Today we take for granted that windows and images in graphic or video applications can be zoomed continuously to arbitrary size. Geometric image operations are also important in computer graphics where textures, which are usually raster images, are deformed to be mapped onto the corresponding 3D surfaces, possibly in real time. Of course, geometric operations are not as simple as their commonality may suggest. While it is obvious, for example, that an image could be enlarged by some integer factor n simply by replicating each pixel $n \times n$ times, the results would probably not be appealing, and it also gives us no immediate idea how to handle continuous scaling, rotating images, or other image deformations. In general, geometric operations that achieve high-quality results are not trivial to implement and are also computationally demanding, even on today's fast computers.

In principle, a geometric operation transforms a given image I to a new image I' by modifying the *coordinates* of image pixels,

$$I'(x', y') \leftarrow I(x, y), \tag{21.1}$$

that is, the value of the image function I at the original location (x, y) moves to the new position (x', y') in the transformed image I'. Thus (at least in the continuous case) the *values* of the image elements do not change but only their *positions*.

To model this process, we first need a 2D transformation function or *geometric mapping* T, for example, in the form

$$T : \mathbb{R}^2 \to \mathbb{R}^2, \tag{21.2}$$

© Springer-Verlag London 2016
W. Burger, M.J. Burge, *Digital Image Processing*, Texts in Computer Science,
DOI 10.1007/978-1-4471-6684-9_21

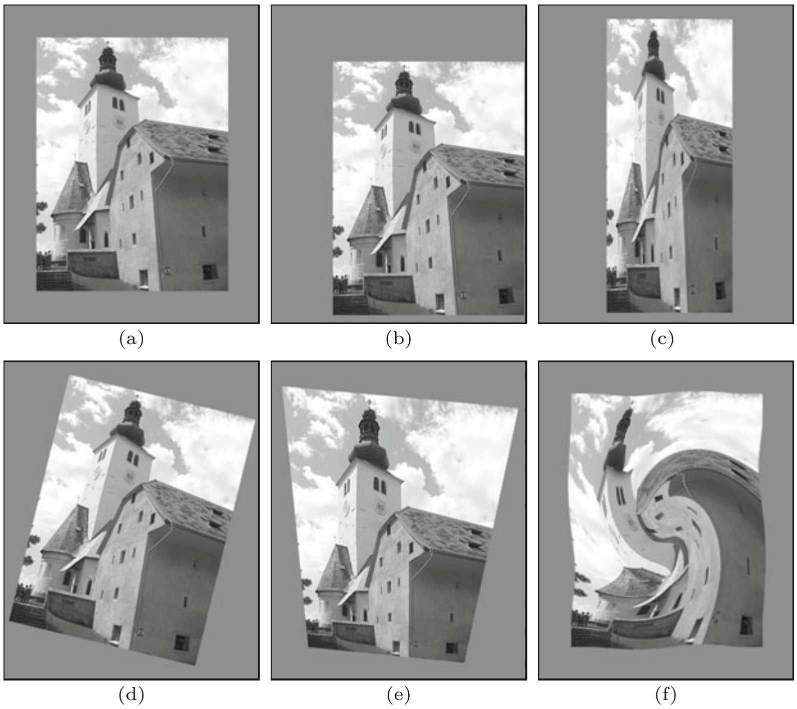


Fig. 21.1
Typical examples for geometric operations: original image (a), translation (b), scaling (contracting or stretching) in x and y directions (c), rotation about the center (d), projective transformation (e), and nonlinear distortion (f).

that specifies for each original 2D coordinate point $\boldsymbol{x} = (x, y)$ the corresponding target point $\boldsymbol{x}' = (x', y')$ in the new image I',

$$(x', y') = T(x, y). \tag{21.3}$$

Notice that the coordinates (x, y) and (x', y') specify points in the *continuous* image plane $\mathbb{R} \times \mathbb{R}$. The main problem in transforming digital images is that the pixels $I(u, v)$ are defined not on a continuous plane but on a *discrete* raster $\mathbb{Z} \times \mathbb{Z}$. Obviously, a transformed coordinate $(u', v') = T(u, v)$ produced by the mapping function $T()$ will, in general, no longer fall onto a discrete raster point. The solution to this problem is to compute intermediate pixel values for the transformed image by a process called *interpolation* (see Ch. 22), which is the second essential element in any geometric operation.

21.1 2D Coordinate Transformations

The mapping function $T()$ in Eqn. (21.3) is an arbitrary continuous function that for reasons of simplicity is often specified as two separate functions,

$$x' = T_{\mathrm{x}}(x, y) \qquad \text{and} \qquad y' = T_{\mathrm{y}}(x, y) \tag{21.4}$$

for the x and y components, respectively.

21.1.1 Simple Geometric Mappings

The simple mapping functions include translation, scaling, shearing, and rotation, defined as follows:

Translation (shift) by a vector (d_x, d_y):

$$\begin{array}{l} T_{\mathrm{x}} : x' = x + d_x \\ T_{\mathrm{y}} : y' = y + d_y \end{array} \quad \text{or} \quad \begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} d_x \\ d_y \end{pmatrix}. \tag{21.5}$$

Scaling (contracting or stretching) along the x or y axis by the factor s_x or s_y, respectively:

$$\begin{array}{l} T_{\mathrm{x}} : x' = s_x \cdot x \\ T_{\mathrm{y}} : y' = s_y \cdot y \end{array} \quad \text{or} \quad \begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} s_x & 0 \\ 0 & s_y \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix}. \tag{21.6}$$

Shearing along the x and y axis by the factor b_x and b_y, respectively (for shearing in only one direction, the other factor is set to zero):

$$\begin{array}{l} T_{\mathrm{x}} : x' = x + b_x \cdot y \\ T_{\mathrm{y}} : y' = y + b_y \cdot x \end{array} \quad \text{or} \quad \begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} 1 & b_x \\ b_y & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix}. \tag{21.7}$$

Rotation by an angle α, with the coordinate origin being the center of rotation:

$$\begin{array}{l} T_{\mathrm{x}} : x' = x \cdot \cos\alpha \, - \, y \cdot \sin\alpha \\ T_{\mathrm{y}} : y' = x \cdot \sin\alpha \, + \, y \cdot \cos\alpha \end{array} \quad \text{or} \tag{21.8}$$

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix}. \tag{21.9}$$

Rotating the image by an angle α around an *arbitrary center point* $\boldsymbol{x}_c = (x_c, y_c)$ is accomplished by first translating the image by $(-x_c, -y_c)$, such that $\boldsymbol{x}_c$ coincides with the origin, then rotating the image about the origin (as in Eqn. (21.9)), and finally shifting the image back by (x_c, y_c). The resulting composite transformation is

$$\begin{array}{l} T_{\mathrm{x}} : x' \; = \; x_c + (x - x_c) \cdot \cos\alpha \; - \; (y - y_c) \cdot \sin\alpha \\ T_{\mathrm{y}} : y' \; = \; y_c + (x - x_c) \cdot \sin\alpha \; + \; (y - y_c) \cdot \cos\alpha \end{array} \tag{21.10}$$

or

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} x_c \\ y_c \end{pmatrix} + \begin{pmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{pmatrix} \cdot \begin{pmatrix} x - x_c \\ y - y_c \end{pmatrix}. \tag{21.11}$$

The combination of the operations listed in Eqns. (21.5)–(21.9) constitute the important class of "affine" transformations or *affine mappings* (see also Sec. 21.1.3).

21.1.2 Homogeneous Coordinates

To simplify the concatenation of linear mappings, it is advantageous to specify all operations in the form of vector-matrix multiplications, as in Eqns. (21.6)–(21.9). Note that pure translation Eqn. (21.5), which corresponds to a vector addition, cannot be formulated as a vector-matrix multiplication. Fortunately, this difficulty can be elegantly resolved with so-called *homogeneous coordinates* (see, e.g., [75, p. 204]).[1]

[1] See also Sec. B.5 in the Appendix.

To turn an "ordinary" (i.e., *Cartesian*) coordinate into a homogeneous coordinate, the original vector is simply extended by an additional element with constant value 1. For example, a 2D Cartesian point $\boldsymbol{x} = (x, y)^\mathsf{T}$ converts to a 3D vector,

$$\operatorname{hom}(\boldsymbol{x}) = \operatorname{hom}\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \underline{\boldsymbol{x}}. \tag{21.12}$$

Note that the homogeneous representation is not unique, but any scaled vector $s \cdot \underline{\boldsymbol{x}}$ is an equivalent homogeneous representation of the Cartesian coordinate $\boldsymbol{x}$, that is

$$\boldsymbol{x} = \operatorname{hom}^{-1}(\underline{\boldsymbol{x}}) = \operatorname{hom}^{-1}(s \cdot \underline{\boldsymbol{x}}), \tag{21.13}$$

for any nonzero $s \in \mathbb{R}$. For example, the homogeneous coordinates $\underline{\boldsymbol{x}}_1 = (3, 2, 1)^\mathsf{T}$, $\underline{\boldsymbol{x}}_2 = (-6, -4, -2)^\mathsf{T}$, and $\underline{\boldsymbol{x}}_3 = (30, 20, 10)^\mathsf{T}$ are all equivalent representations of the same Cartesian coordinate $\boldsymbol{x} = (3, 2)^\mathsf{T}$.

The reverse mapping from a 3D homogeneous coordinate $\underline{\boldsymbol{x}} = (\underline{x}, \underline{y}, \underline{z})^\mathsf{T}$ to the corresponding 2D Cartesian coordinate is denoted

$$\operatorname{hom}^{-1}(\underline{\boldsymbol{x}}) = \operatorname{hom}^{-1}\begin{pmatrix} \underline{x} \\ \underline{y} \\ \underline{z} \end{pmatrix} = \frac{1}{\underline{z}} \cdot \begin{pmatrix} \underline{x} \\ \underline{y} \end{pmatrix} = \boldsymbol{x} \tag{21.14}$$

With the help of homogeneous coordinates, we can now define a 2D *translation* (Eqn. (21.5)) as a vector-matrix product in the form

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \operatorname{hom}^{-1}\Bigl[\begin{pmatrix} 1 & 0 & d_x \\ 0 & 1 & d_y \\ 0 & 0 & 1 \end{pmatrix} \cdot \operatorname{hom}\begin{pmatrix} x \\ y \end{pmatrix}\Bigr] \tag{21.15}$$

$$= \begin{pmatrix} 1 & 0 & d_x \\ 0 & 1 & d_y \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} x + d_x \\ y + d_y \end{pmatrix}, \tag{21.16}$$

which had been our motive for introducing homogeneous coordinates in the first place. As we shall see in the following sections, homogeneous coordinates allow us to write many common 2D coordinate transformations in the form

$$\underline{\boldsymbol{x}}' = \mathbf{A} \cdot \underline{\boldsymbol{x}}\,, \tag{21.17}$$

where $\mathbf{A}$ is a 3×3 matrix. Note that (due to the relation in Eqn. (21.13)) multiplying the matrix $\mathbf{A}$ by some scalar factor s yields the same transformation in terms of Cartesian coordinates, that is,

$$\boldsymbol{x}' = \operatorname{hom}^{-1}[\mathbf{A} \cdot \underline{\boldsymbol{x}}] = \operatorname{hom}^{-1}[s \cdot (\mathbf{A} \cdot \underline{\boldsymbol{x}})] = \operatorname{hom}^{-1}[(s \cdot \mathbf{A}) \cdot \underline{\boldsymbol{x}}], \tag{21.18}$$

for any nonzero $s \in \mathbb{R}$.

21.1.3 Affine (Three-Point) Mapping

In general, and analogous to Eqn. (21.16), we can express any combination of 2D translation, scaling, and rotation as vector-matrix multiplication in homogeneous coordinates in the form

$$\underline{\boldsymbol{x}}' = \mathbf{A}_{\text{affine}} \cdot \underline{\boldsymbol{x}} \tag{21.19}$$

or $\boldsymbol{x}' = \text{hom}^{-1}[\mathbf{A}_{\text{affine}} \cdot \text{hom}(\boldsymbol{x})]$ in Cartesian coordinates, that is,

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \text{hom}^{-1}\left[\begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}\right] = \begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}. \tag{21.20}$$

This 2D coordinate transformation is called an "affine mapping" with the six parameters $a_{00}, \ldots, a_{12}$, where a_{02}, a_{12} specify the translation (equivalent to d_x, d_y in Eqn. (21.5)) and a_{00}, a_{01}, a_{10}, a_{11} aggregate the scaling, shearing, and rotation coefficients (see Eqns. (21.6)–(21.9)). For example, the affine transformation matrix for a rotation about the origin by an angle α is specified by the matrix

$$\mathbf{A}_{\text{rot}} = \begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix}. \tag{21.21}$$

In this way, compound transformations can be constructed easily by consecutive matrix multiplications (from right to left). For example, the transformation matrix for a rotation by α about a given center point $\boldsymbol{x}_c = (x_c, y_c)^\intercal$ (see Eqn. (21.11)), composed by a translation to the origin followed by a rotation and another translation, is

$$\mathbf{A} = \underbrace{\begin{pmatrix} 1 & 0 & x_c \\ 0 & 1 & y_c \\ 0 & 0 & 1 \end{pmatrix}}_{\substack{\text{translation by} \\ (x_c, y_c)^\intercal}} \cdot \underbrace{\begin{pmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix}}_{\substack{\text{rotation by } \alpha \\ \text{(about the origin)}}} \cdot \underbrace{\begin{pmatrix} 1 & 0 & -x_c \\ 0 & 1 & -y_c \\ 0 & 0 & 1 \end{pmatrix}}_{\substack{\text{translation by} \\ (-x_c, -y_c)^\intercal}} \tag{21.22}$$

$$= \begin{pmatrix} 1 & 0 & x_c \\ 0 & 1 & y_c \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & x_c \\ 0 & 1 & y_c \\ 0 & 0 & 1 \end{pmatrix}^{-1} \tag{21.23}$$

$$= \begin{pmatrix} \cos\alpha & -\sin\alpha & x_c \cdot (1-\cos\alpha) + y_c \cdot \sin\alpha \\ \sin\alpha & \cos\alpha & y_c \cdot (1-\cos\alpha) - x_c \cdot \sin\alpha \\ 0 & 0 & 1 \end{pmatrix}. \tag{21.24}$$

Of course, the result is the same as in Eqn. (21.10).

Note that multiplying two affine transformation matrices always yields another affine transformation. Also, an affine transformation maps straight lines to straight lines, triangles to triangles, and rectangles to parallelograms, as illustrated in Fig. 21.2. The distance ratio between points on a straight line remains unchanged by this type of mapping function.

Affine transformation parameters from three point pairs

The six parameters of the 2D affine mapping (Eqn. (21.20)) are uniquely determined by three pairs of corresponding points $(\boldsymbol{x}_0, \boldsymbol{x}'_1)$, $(\boldsymbol{x}_1, \boldsymbol{x}'_1)$, $(\boldsymbol{x}_2, \boldsymbol{x}'_2)$, with the first point $\boldsymbol{x}_i = (x_i, y_i)$ of each pair located in the original image and the corresponding point $\boldsymbol{x}'_i = (x'_i, y'_i)$ located in the target image. From these six coordinate values, the

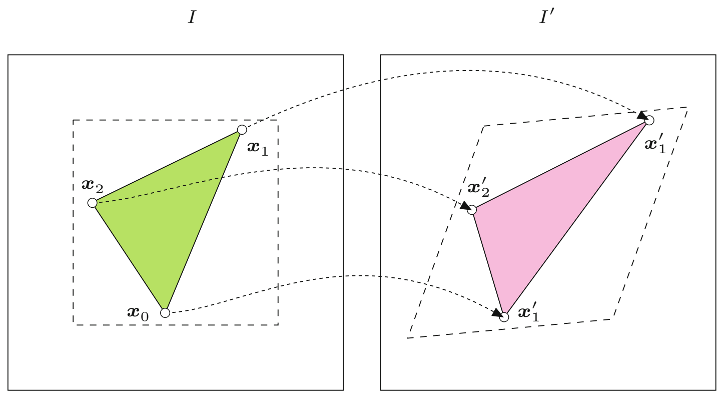


Fig. 21.2
Affine mapping. An affine 2D transformation is uniquely specified by three pairs of corresponding points; for example, $(\boldsymbol{x}_0, \boldsymbol{x}'_1)$, $(\boldsymbol{x}_1, \boldsymbol{x}'_1)$, and $(\boldsymbol{x}_2, \boldsymbol{x}'_2)$.

six transformation parameters $a_{00}, \ldots, a_{12}$ are derived by solving the system of linear equations

$$\begin{aligned} x'_0 &= a_{00} \cdot x_0 + a_{01} \cdot y_0 + a_{02}, & y'_0 &= a_{10} \cdot x_0 + a_{11} \cdot y_0 + a_{12}, \\ x'_1 &= a_{00} \cdot x_1 + a_{01} \cdot y_1 + a_{02}, & y'_1 &= a_{10} \cdot x_1 + a_{11} \cdot y_1 + a_{12}, \\ x'_2 &= a_{00} \cdot x_2 + a_{01} \cdot y_2 + a_{02}, & y'_2 &= a_{10} \cdot x_2 + a_{11} \cdot y_2 + a_{12}, \end{aligned} \tag{21.25}$$

provided that the points (vectors) $\boldsymbol{x}_0$, $\boldsymbol{x}_1$, $\boldsymbol{x}_2$ are linearly independent (i.e., that they do not lie on a common straight line). Since Eqn. (21.25) consists of two independent sets of linear 3×3 equations for x'_i and y'_i, the solution can be written in closed form as

$$\begin{aligned} a_{00} &= \tfrac{1}{d} \cdot [y_0(x'_1 - x'_2) + y_1(x'_2 - x'_0) + y_2(x'_0 - x'_1)], \\ a_{01} &= \tfrac{1}{d} \cdot [x_0(x'_2 - x'_1) + x_1(x'_0 - x'_2) + x_2(x'_1 - x'_0)], \\ a_{10} &= \tfrac{1}{d} \cdot [y_0(y'_1 - y'_2) + y_1(y'_2 - y'_0) + y_2(y'_0 - y'_1)], \\ a_{11} &= \tfrac{1}{d} \cdot [x_0(y'_2 - y'_1) + x_1(y'_0 - y'_2) + x_2(y'_1 - y'_0)], \\ a_{02} &= \tfrac{1}{d} \cdot [x_0(y_2 x'_1 - y_1 x'_2) + x_1(y_0 x'_2 - y_2 x'_0) + x_2(y_1 x'_0 - y_0 x'_1)], \\ a_{12} &= \tfrac{1}{d} \cdot [x_0(y_2 y'_1 - y_1 y'_2) + x_1(y_0 y'_2 - y_2 y'_0) + x_2(y_1 y'_0 - y_0 y'_1)], \end{aligned} \tag{21.26}$$

with $d = x_0(y_2 - y_1) + x_1(y_0 - y_2) + x_2(y_1 - y_0)$.

Inverse affine mapping

The inverse of the affine transformation, which is often required in practice (see Sec. 21.2.2), can be calculated by simply applying the inverse of the transformation matrix $\mathbf{A}_{\text{affine}}$ (Eqn. (21.20)) in homogeneous coordinate space, that is,

$$\underline{\boldsymbol{x}} = \mathbf{A}_{\text{affine}}^{-1} \cdot \underline{\boldsymbol{x}}' \tag{21.27}$$

or $\boldsymbol{x} = \text{hom}^{-1}\left[\mathbf{A}_{\text{affine}}^{-1} \cdot \text{hom}(\boldsymbol{x}')\right]$ in Cartesian coordinates, that is,

$$\begin{pmatrix} x \\ y \end{pmatrix} = \mathrm{hom}^{-1}\Bigl[\begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ 0 & 0 & 1 \end{pmatrix}^{-1} \cdot \begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix}\Bigr] \tag{21.28}$$

$$= \mathrm{hom}^{-1}\Bigl[\underbrace{\frac{1}{a_{00}a_{11} - a_{01}a_{10}} \cdot \begin{pmatrix} a_{11} & -a_{01} & a_{01}a_{12} - a_{02}a_{11} \\ -a_{10} & a_{00} & a_{02}a_{10} - a_{00}a_{12} \\ 0 & 0 & a_{00}a_{11} - a_{01}a_{10} \end{pmatrix}}_{\mathbf{A}_{\text{affine}}^{-1}} \cdot \begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix}\Bigr] \tag{21.29}$$

$$= \frac{1}{a_{00}a_{11} - a_{01}a_{10}} \cdot \begin{pmatrix} a_{11} & -a_{01} & a_{01}a_{12} - a_{02}a_{11} \\ -a_{10} & a_{00} & a_{02}a_{10} - a_{00}a_{12} \end{pmatrix} \cdot \begin{pmatrix} x' \\ y' \end{pmatrix}. \tag{21.30}$$

Since the bottom row of $\mathbf{A}_{\text{affine}}^{-1}$ in Eqn. (21.29) consists of the elements $(0, 0, 1)$, the inverse mapping is again an affine transformation. Of course, the inverse of the affine mapping can also be found directly (i.e., without inverting the transformation matrix) from the given point coordinates $(\boldsymbol{x}_i, \boldsymbol{x}'_i)$ by using Eqns. (21.25) and (21.26) with *interchanged* source and target coordinates.

21.1.4 Projective (Four-Point) Mapping

In contrast to the affine transformation, which provides a mapping between arbitrary triangles, the projective transformation is a linear mapping between arbitrary *quadrilaterals* (Fig. 21.3). This is particularly useful for deforming images controlled by mesh partitioning, as described in Sec. 21.1.7. To map from an arbitrary sequence of four 2D points ($\boldsymbol{x}_0$, $\boldsymbol{x}_1$, $\boldsymbol{x}_2$, $\boldsymbol{x}_3$) to a set of corresponding points ($\boldsymbol{x}'_0$, $\boldsymbol{x}'_1$, $\boldsymbol{x}'_2$, $\boldsymbol{x}'_3$), the transformation requires eight degrees of freedom, two more than needed for the affine transformation. Analogous to the affine transformation (Eqn. (21.20)), a projective transformation can be expressed as a linear mapping in homogeneous coordinates,

$$\underline{\boldsymbol{x}}' = \mathbf{A}_{\text{proj}} \cdot \underline{\boldsymbol{x}} \tag{21.31}$$

or $\boldsymbol{x}' = \mathrm{hom}^{-1}\bigl[\mathbf{A}_{\text{proj}} \cdot \mathrm{hom}(\boldsymbol{x})\bigr]$ in Cartesian coordinates, that is,

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \mathrm{hom}^{-1}\Bigl[\begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}\Bigr] \tag{21.32}$$

$$= \frac{1}{a_{20} \cdot x + a_{21} \cdot y + 1} \cdot \begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}, \tag{21.33}$$

with the two additional elements (parameters) a_{20} and a_{21} in the transformation matrix $\mathbf{A}_{\text{proj}}$. Because x, y appear in the denominator of the fraction in Eqn. (21.33), the projective mapping is generally *nonlinear* in Cartesian coordinates. Despite this nonlinearity, straight lines are preserved under this transformation. In fact, this is the most general transformation that maps straight lines to straight lines in 2D, and it actually maps any Nth-order algebraic curve onto another Nth-order algebraic curve. In particular, circles and ellipses always transform into other second-order curves (i.e., conic sections). Unlike the affine transformation, however, parallel lines do not generally map to parallel lines under a projective transformation (cf. Fig.

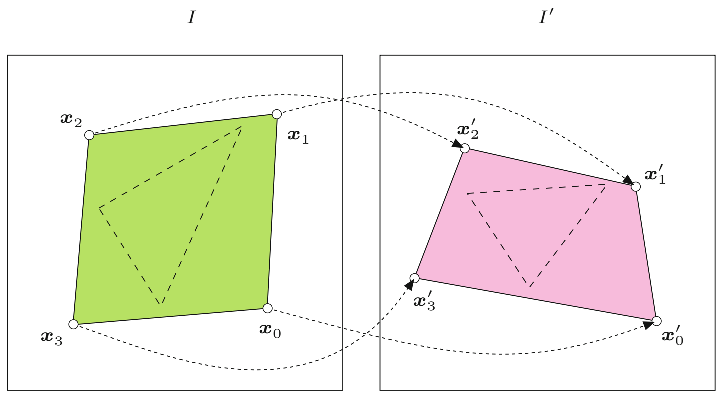


Fig. 21.3
Projective mapping. Four pairs of corresponding 2D points, $(\boldsymbol{x}_0, \boldsymbol{x}'_0)$, $(\boldsymbol{x}_1, \boldsymbol{x}'_1)$, $(\boldsymbol{x}_2, \boldsymbol{x}'_2)$, $(\boldsymbol{x}_3, \boldsymbol{x}'_3)$ uniquely specify a projective transformation. Straight lines are again mapped to straight lines, and a rectangle is mapped to some quadrilateral. In general, neither parallelism between straight lines nor the distance ratio is preserved.

21.3) and the distance ratios between points on a line are not preserved. The projective mapping is therefore sometimes referred to as "perspective" or "pseudo-perspective".

Projective transformation parameters from four point pairs

Given four pairs of corresponding 2D points, $(\boldsymbol{x}_0, \boldsymbol{x}'_0), \ldots, (\boldsymbol{x}_3, \boldsymbol{x}'_3)$, with one point $\boldsymbol{x}_i = (x_i, y_i)^\mathsf{T}$ in the source image and the second point $\boldsymbol{x}'_i = (x'_i, y'_i)^\mathsf{T}$ in the target image, the eight unknown transformation parameters $a_{00}, \ldots, a_{21}$ can be found by solving a system of linear equations. Multiplying Eqn. (21.33) by the common denominator on the right hand side gives

$$\begin{aligned} x' \cdot (a_{20} \cdot x + a_{21} \cdot y + 1) &= a_{00} \cdot x + a_{01} \cdot y + a_{02}, \\ y' \cdot (a_{20} \cdot x + a_{21} \cdot y + 1) &= a_{10} \cdot x + a_{11} \cdot y + a_{12}, \end{aligned} \tag{21.34}$$

and thus

$$\begin{aligned} a_{20} \cdot x \cdot x' + a_{21} \cdot y \cdot x' + x' &= a_{00} \cdot x + a_{01} \cdot y + a_{02}, \\ a_{20} \cdot x \cdot y' + a_{21} \cdot y \cdot y' + y' &= a_{10} \cdot x + a_{11} \cdot y + a_{12}, \end{aligned} \tag{21.35}$$

for any pair of corresponding points $\boldsymbol{x} = (x, y)^\mathsf{T}$ and $\boldsymbol{x}' = (x', y')^\mathsf{T}$. By slightly rearranging Eqn. (21.35) and inserting the (known) source and target point coordinates (x_i, y_i) and (x'_i, y'_i), respectively, we obtain one such pair of linear equations

$$\begin{aligned} x'_i &= a_{00} \cdot x_i + a_{01} \cdot y_i + a_{02} - a_{20} \cdot x_i \cdot x'_i - a_{21} \cdot y_i \cdot x'_i, \\ y'_i &= a_{10} \cdot x_i + a_{11} \cdot y_i + a_{12} - a_{20} \cdot x_i \cdot y'_i - a_{21} \cdot y_i \cdot y'_i, \end{aligned} \tag{21.36}$$

for each point pair $i = 0, \ldots, 3$ and the eight unknowns $a_{00}, \ldots, a_{21}$. Combining the resulting eight equations in the usual matrix notation yields

$$\begin{pmatrix} x'_0 \\ y'_0 \\ x'_1 \\ y'_1 \\ x'_2 \\ y'_2 \\ x'_3 \\ y'_3 \end{pmatrix} = \begin{pmatrix} x_0 & y_0 & 1 & 0 & 0 & 0 & -x_0 x'_0 & -y_0 x'_0 \\ 0 & 0 & 0 & x_0 & y_0 & 1 & -x_0 y'_0 & -y_0 y'_0 \\ x_1 & y_1 & 1 & 0 & 0 & 0 & -x_1 x'_1 & -y_1 x'_1 \\ 0 & 0 & 0 & x_1 & y_1 & 1 & -x_1 y'_1 & -y_1 y'_1 \\ x_2 & y_2 & 1 & 0 & 0 & 0 & -x_2 x'_2 & -y_2 x'_2 \\ 0 & 0 & 0 & x_2 & y_2 & 1 & -x_2 y'_2 & -y_2 y'_2 \\ x_3 & y_3 & 1 & 0 & 0 & 0 & -x_3 x'_3 & -y_3 x'_3 \\ 0 & 0 & 0 & x_3 & y_3 & 1 & -x_3 y'_3 & -y_3 y'_3 \end{pmatrix} \cdot \begin{pmatrix} a_{00} \\ a_{01} \\ a_{02} \\ a_{10} \\ a_{11} \\ a_{12} \\ a_{20} \\ a_{21} \end{pmatrix}, \tag{21.37}$$

or

$$\boldsymbol{b} \;=\; \mathbf{M} \cdot \boldsymbol{a}\,. \tag{21.38}$$

Note that all elements of the vector $\boldsymbol{b} = (x'_0, \ldots, y'_3)^\mathsf{T}$ and the matrix $\mathbf{M}$ are obtained from the specified point coordinates and are thus constants. The unknown parameters $\boldsymbol{a} = (a_{00}, \ldots, a_{21})^\mathsf{T}$ can be found by solving the system of linear equations in Eqn. (21.38) with standard numerical methods, for example, the Gauss algorithm [35, p. 276]. It is recommended to use proven numerical software for this purpose.[2]

If we want to use *more than four* corresponding point pairs to recover the eight parameters of a projective transformation, the system of linear equations in Eqn. (21.37) becomes overdetermined, that is, the system has more equations than unknowns. In general, n pairs of corresponding points yield a stack of $2n$ equations, so the vector $\boldsymbol{b}$ in Eqn. (21.37) has the length $2n$ and the matrix $\mathbf{M}$ is of size $2n{\times}8$ (vector $\boldsymbol{a}$ remains the same). Overdetermined systems like this can be solved in a least-squares sense (minimizing $\|\mathbf{M}\cdot\boldsymbol{a}-\boldsymbol{b}\|$), for example, using the singular-value (SVD) or QR decomposition of $\mathbf{M}$ [96,145].[3] Other solutions for the multi-point case are discussed later in this section (see p. 524).

Inverse projective mapping

In general, any *linear* transformation of the form $\underline{\boldsymbol{x}}' = \mathbf{A}\cdot\underline{\boldsymbol{x}}$ (in homogeneous coordinates $\underline{\boldsymbol{x}}$, $\underline{\boldsymbol{x}}'$) can be inverted by applying the inverse of the matrix $\mathbf{A}$, that is,

$$\underline{\boldsymbol{x}} = \mathbf{A}^{-1}\cdot\underline{\boldsymbol{x}}' \tag{21.39}$$

provided that $\mathbf{A}$ is nonsingular ($\det(\mathbf{A}) \neq 0$). The inverse of a 3×3 matrix $\mathbf{A}$ is comparatively easy to find in closed form using the relation

$$\mathbf{A}^{-1} = \frac{1}{\det(\mathbf{A})}\cdot \operatorname{adj}(\mathbf{A}), \tag{21.40}$$

with the determinant $\det(\mathbf{A})$ and the *adjugate* matrix $\operatorname{adj}(\mathbf{A})$ (see, e.g., [35, pp. 251, 260], [145, p. 219]). In particular, for a real-valued 3×3 matrix

$$\mathbf{A} = \begin{pmatrix} a_{00} & a_{01} & a_{02}\\ a_{10} & a_{11} & a_{12}\\ a_{20} & a_{21} & a_{22}\end{pmatrix}, \tag{21.41}$$

the determinant can be calculated as

$$\begin{aligned}\det(\mathbf{A}) = {} & a_{00}\,a_{11}\,a_{22} + a_{01}\,a_{12}\,a_{20} + a_{02}\,a_{10}\,a_{21}\\ & - a_{00}\,a_{12}\,a_{21} - a_{01}\,a_{10}\,a_{22} - a_{02}\,a_{11}\,a_{20},\end{aligned} \tag{21.42}$$

and the 3×3 adjugate matrix is

$$\operatorname{adj}(\mathbf{A}) = \begin{pmatrix} a_{11}a_{22}-a_{12}a_{21} & a_{02}a_{21}-a_{01}a_{22} & a_{01}a_{12}-a_{02}a_{11}\\ a_{12}a_{20}-a_{10}a_{22} & a_{00}a_{22}-a_{02}a_{20} & a_{02}a_{10}-a_{00}a_{12}\\ a_{10}a_{21}-a_{11}a_{20} & a_{01}a_{20}-a_{00}a_{21} & a_{00}a_{11}-a_{01}a_{10}\end{pmatrix}. \tag{21.43}$$

[2] See Sec. B.7.1 in the Appendix.

[3] See Sec. B.7.2 in the Appendix.

In the special case of a projective mapping, the coefficient $a_{22} = 1$ (Eqn. (21.32)), which slightly simplifies the calculation.

Since scalar multiples of homogeneous vectors are all equivalent in Cartesian space (see Eqn. (21.18)), the multiplication by the constant factor $1/\det(\mathbf{A})$ in Eqn. (21.40) can be omitted. Thus, to invert a linear 2D transformation specified by a 3×3 matrix $\mathbf{A}$, we only need to multiply the homogeneous coordinate vector with the adjugate matrix $\operatorname{adj}(\mathbf{A})$, that is,

$$\underline{\boldsymbol{x}} = \mathbf{A}^{-1} \cdot \underline{\boldsymbol{x}}' \;\equiv\; \operatorname{adj}(\mathbf{A}) \cdot \underline{\boldsymbol{x}}'. \tag{21.44}$$

Returning to Cartesian coordinates, the inverse transformation can be written as

$$\boldsymbol{x} = \hom^{-1}[\operatorname{adj}(\mathbf{A}) \cdot \hom(\boldsymbol{x}')]\,. \tag{21.45}$$

This method can be used to invert any linear transformation in 2D, including the affine and projective mapping functions described already. Consequently, the inversion of the *affine* transformation shown earlier (see Eqn. (21.29)) is only a special case of this general method.

Of course, matrix inversion may also be implemented with standard linear algebra software, which is not only less error-prone but also offers better numerical stability (see also Sec. B.1 in the Appendix).

Projective mapping via the unit square

An alternative method for finding the projective mapping parameters for a given set of image points is to use a two-stage mapping through the unit square $\mathcal{S}_1$, which avoids iteratively solving a system of equations [256, p. 55] [105]. The projective mapping, shown in Fig. 21.4, from the four corner points of the unit square $\mathcal{S}_1$ to an arbitrary quadrilateral $\mathcal{Q} = (\boldsymbol{x}'_0, \ldots, \boldsymbol{x}'_3)$ with

$$\begin{aligned} (0,0) &\mapsto \boldsymbol{x}'_0, & (1,1) &\mapsto \boldsymbol{x}'_2, \\ (1,0) &\mapsto \boldsymbol{x}'_1, & (0,1) &\mapsto \boldsymbol{x}'_3, \end{aligned} \tag{21.46}$$

reduces the system of equations in Eqn. (21.37) to

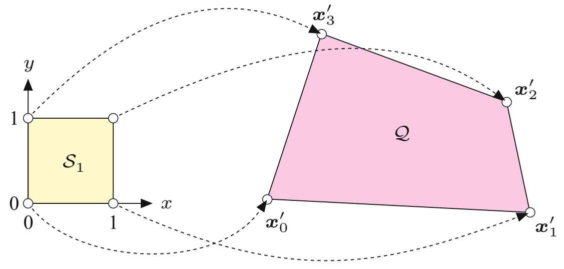


Fig. 21.4 Projective mapping from the unit square $\mathcal{S}_1$ to an arbitrary quadrilateral $\mathcal{Q} = (\boldsymbol{x}'_0, \ldots, \boldsymbol{x}'_3)$.

$$\begin{aligned}
x'_0 &= a_{02},\\
y'_0 &= a_{12},\\
x'_1 &= a_{00} + a_{02} - a_{20} \cdot x'_1,\\
y'_1 &= a_{10} + a_{12} - a_{20} \cdot y'_1,\\
x'_2 &= a_{00} + a_{01} + a_{02} - a_{20} \cdot x'_2 - a_{21} \cdot x'_2,\\
y'_2 &= a_{10} + a_{11} + a_{12} - a_{20} \cdot y'_2 - a_{21} \cdot y'_2,\\
x'_3 &= a_{01} + a_{02} - a_{21} \cdot x'_3,\\
y'_3 &= a_{11} + a_{12} - a_{21} \cdot y'_3.
\end{aligned} \tag{21.47}$$

This set of equations has the following closed-form solution for the eight unknown transformation parameters $a_{00}, a_{01}, \ldots, a_{21}$:

$$a_{20} = \frac{(x'_0 - x'_1 + x'_2 - x'_3)\cdot(y'_3 - y'_2) - (y'_0 - y'_1 + y'_2 - y'_3)\cdot(x'_3 - x'_2)}{(x'_1 - x'_2)\cdot(y'_3 - y'_2) \;-\; (x'_3 - x'_2)\cdot(y'_1 - y'_2)}, \tag{21.48}$$

$$a_{21} = \frac{(y'_0 - y'_1 + y'_2 - y'_3)\cdot(x'_1 - x'_2) - (x'_0 - x'_1 + x'_2 - x'_3)\cdot(y'_1 - y'_2)}{(x'_1 - x'_2)\cdot(y'_3 - y'_2) \;-\; (x'_3 - x'_2)\cdot(y'_1 - y'_2)} \tag{21.49}$$

and

$$a_{00} = x'_1 - x'_0 + a_{20}\, x'_1, \quad a_{01} = x'_3 - x'_0 + a_{21}\, x'_3, \quad a_{02} = x'_0, \tag{21.50}$$

$$a_{10} = y'_1 - y'_0 + a_{20}\, y'_1, \quad a_{11} = y'_3 - y'_0 + a_{21}\, y'_3, \quad a_{12} = y'_0. \tag{21.51}$$

By calculating the inverse of the corresponding 3×3 transformation matrix (Eqn. (21.40)), the mapping may be *reversed* to transform an arbitrary quadrilateral to the unit square. A mapping T between two arbitrary quadrilaterals,

$$\mathcal{Q} \xrightarrow{T} \mathcal{Q}',$$

can thus be implemented by combining a reversed mapping and a forward mapping via the unit square. As illustrated in Fig. 21.5, the transformation of an arbitrary quadrilateral $\mathcal{Q} = (\boldsymbol{x}_0, \boldsymbol{x}_1, \boldsymbol{x}_2, \boldsymbol{x}_3)$ to a second quadrilateral $\mathcal{Q}' = (\boldsymbol{x}'_0, \boldsymbol{x}'_1, \boldsymbol{x}'_2, \boldsymbol{x}'_3)$ is accomplished in two steps involving the linear transformations T_1 and T_2 between the two quadrilaterals and the unit square $\mathcal{S}_1$, that is,

$$\mathcal{Q} \xleftarrow{T_1} \mathcal{S}_1 \xrightarrow{T_2} \mathcal{Q}'. \tag{21.52}$$

The parameters for the projective transformations T_1 and T_2 are obtained by inserting the corresponding point coordinates of $\mathcal{Q}$ and $\mathcal{Q}'$ ($\boldsymbol{x}_i$ and $\boldsymbol{x}'_i$, respectively) into Eqns. (21.48)–(21.51). The complete transformation T is then the concatenation of the two transformations T_1^{-1} and T_2, that is,

$$\boldsymbol{x}' \;=\; T(\boldsymbol{x}) \;=\; T_2\big(T_1^{-1}(\boldsymbol{x})\big), \tag{21.53}$$

or, expressed in matrix notation (using homogeneous coordinates),

$$\underline{\boldsymbol{x}}' \;=\; \mathbf{A} \cdot \underline{\boldsymbol{x}} \;=\; \mathbf{A}_2 \cdot \mathbf{A}_1^{-1} \cdot \underline{\boldsymbol{x}}\,. \tag{21.54}$$

Of course, the matrix $\mathbf{A} = \mathbf{A}_2 \cdot \mathbf{A}_1^{-1}$ needs to be calculated only once for a particular transformation and can then be used repeatedly for mapping any other image points $\boldsymbol{x}_i$.

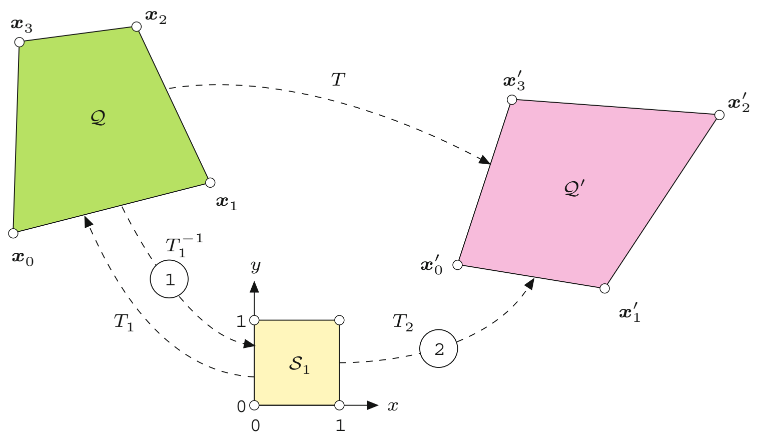


Fig. 21.5 Two-step projective transformation between arbitrary quadrilaterals. In the first step, quadrilateral $\mathcal{Q}$ is transformed to the unit square $\mathcal{S}_1$ by the inverse mapping function T_1^{-1}. In the second step, T_2 transforms the square $\mathcal{S}_1$ to the target quadrilateral $\mathcal{Q}'$. The complete mapping T results from the concatenation of the mappings T_1^{-1} and T_2.

Example

The source and the target quadrilaterals $\mathcal{Q}$ and $\mathcal{Q}'$, respectively, are specified by the following coordinate points:

$$\begin{array}{lllll} \mathcal{Q}: & \boldsymbol{x}_0 = (2,5), & \boldsymbol{x}_1 = (4,6), & \boldsymbol{x}_2 = (7,9), & \boldsymbol{x}_3 = (5,9); \\ \mathcal{Q}': & \boldsymbol{x}'_0 = (4,3), & \boldsymbol{x}'_1 = (5,2), & \boldsymbol{x}'_2 = (9,3), & \boldsymbol{x}'_3 = (7,5). \end{array}$$

Using Eqns. (21.48)–(21.51), the transformation parameters (matrices) for the projective mappings from the unit $\mathcal{S}_1$ square to the quadrilaterals $\mathbf{A}_1 : \mathcal{S}_1 \mapsto \mathcal{Q}$ and $\mathbf{A}_2 : \mathcal{S}_1 \mapsto \mathcal{Q}'$ are obtained as

$$\mathbf{A}_1 = \begin{pmatrix} 3.3\dot{3} & 0.50 & 2.00 \\ 3.00 & -0.50 & 5.00 \\ 0.3\dot{3} & -0.50 & 1.00 \end{pmatrix} \quad \text{and} \quad \mathbf{A}_2 = \begin{pmatrix} 1.00 & -0.50 & 4.00 \\ -1.00 & -0.50 & 3.00 \\ 0.00 & -0.50 & 1.00 \end{pmatrix}.$$

Concatenating the inverse mapping $\mathbf{A}_1^{-1}$ with $\mathbf{A}_2$ (by matrix multiplication), we get the complete mapping $\mathbf{A} = \mathbf{A}_2 \cdot \mathbf{A}_1^{-1}$ with

$$\mathbf{A}_1^{-1} = \begin{pmatrix} 0.60 & -0.45 & 1.05 \\ -0.40 & 0.80 & -3.20 \\ -0.40 & 0.55 & -0.95 \end{pmatrix} \quad \text{and} \quad \mathbf{A} = \begin{pmatrix} -0.80 & 1.35 & -1.15 \\ -1.60 & 1.70 & -2.30 \\ -0.20 & 0.15 & 0.65 \end{pmatrix}.$$

The library method `makeMapping()` in class `ProjectiveMapping` (see Sec. 21.3) is an implementation of this two-step technique.

Projective transformation parameters from *more than four* point pairs

The projective transformation in Eqn. (21.32) describes a mapping between pairs of arbitrary quadrilaterals in the 2D plane. This geometric relation is also known under the terms *projective isomorphism* or *homography*. The concept is frequently encountered in computer vision, because the transformations between two views of a planar 3D point set can be modeled as a homography (with only 8 degrees of freedom) in the 2D image plane, which is important, for example, for camera calibration, and 3D surface reconstruction. In this context, it is often necessary to estimate the homography parameters from a larger set of 2D point matches, for example, from multiple points assumed to be located on a planar 3D surface. This is the same problem as finding the projective mapping between sets of $n > 4$ corresponding point pairs in 2D.

Several approaches to "homography estimation" exist, including linear and (iterative) nonlinear methods. The simplest and most common is the direct linear transform (DLT) method [56,103], which requires solving a system of $2n$ homogenous linear equations, typically done by singular value decomposition (SVD).

21.1.5 Bilinear Mapping

Similar to the projective transformation (Eqn. (21.32)), the bilinear mapping function

$$\begin{aligned} T_\mathrm{x}: \quad x' &= a_0 \cdot x + a_1 \cdot y + a_2 \cdot x \cdot y + a_3, \\ T_\mathrm{y}: \quad y' &= b_0 \cdot x + b_1 \cdot y + b_2 \cdot x \cdot y + b_3, \end{aligned} \tag{21.55}$$

is specified with four pairs of corresponding points and has eight parameters $(a_0, \ldots, a_3, b_0, \ldots, b_3)$. The transformation is nonlinear because of the mixed term $x \cdot y$ and cannot be described by a linear transformation, even with homogeneous coordinates. In contrast to the projective transformation, the straight lines are not preserved in general but map onto quadratic curves. Similarly, circles are not mapped to ellipses by a bilinear transform.

A bilinear mapping is uniquely specified by four corresponding pairs of 2D points $(\boldsymbol{x}_0, \boldsymbol{x}'_0), \ldots, (\boldsymbol{x}_3, \boldsymbol{x}'_3)$. In the general case, for a bilinear mapping between arbitrary quadrilaterals, the coefficients $a_0, \ldots, a_3, b_0, \ldots, b_3$ (Eqn. (21.55)) are found as the solution of two separate systems of equations, each with four unknowns:

$$\begin{pmatrix} x'_0 \\ x'_1 \\ x'_2 \\ x'_3 \end{pmatrix} = \begin{pmatrix} x_0 & y_0 & x_0 \cdot y_0 & 1 \\ x_1 & y_1 & x_1 \cdot y_1 & 1 \\ x_2 & y_2 & x_2 \cdot y_2 & 1 \\ x_3 & y_3 & x_3 \cdot y_3 & 1 \end{pmatrix} \cdot \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{pmatrix} \quad \text{or} \quad \boldsymbol{x} = \mathbf{M} \cdot \boldsymbol{a}, \tag{21.56}$$

$$\begin{pmatrix} y'_0 \\ y'_1 \\ y'_2 \\ y'_3 \end{pmatrix} = \begin{pmatrix} x_0 & y_0 & x_0 \cdot y_0 & 1 \\ x_1 & y_1 & x_1 \cdot y_1 & 1 \\ x_2 & y_2 & x_2 \cdot y_2 & 1 \\ x_3 & y_3 & x_3 \cdot y_3 & 1 \end{pmatrix} \cdot \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix} \quad \text{or} \quad \boldsymbol{y} = \mathbf{M} \cdot \boldsymbol{b}. \tag{21.57}$$

These equations can again be solved using standard numerical methods. In the special case of bilinearly mapping the unit square $\mathcal{S}_1$ to an arbitrary quadrilateral $\mathcal{Q} = (\boldsymbol{x}'_0, \ldots, \boldsymbol{x}'_3)$, the parameters $a_0, \ldots, a_3$ and $b_0, \ldots, b_3$ are found as

$$a_0 = x'_1 - x'_0, \qquad b_0 = y'_1 - y'_0, \tag{21.58}$$

$$a_1 = x'_3 - x'_0, \qquad b_1 = y'_3 - y'_0, \tag{21.59}$$

$$a_2 = x'_0 - x'_1 + x'_2 - x'_3, \qquad b_2 = y'_0 - y'_1 + y'_2 - y'_3, \tag{21.60}$$

$$a_3 = x'_0, \qquad b_3 = y'_0. \tag{21.61}$$

Figure 21.6 shows results of the affine, projective, and bilinear transformations applied to a simple test pattern. The affine transformation (Fig. 21.6(b)) is specified by mapping to the triangle 1-2-3, while the four points of the quadrilateral 1-2-3-4 define the projective and the bilinear transforms (Fig. 21.6(c, d)).

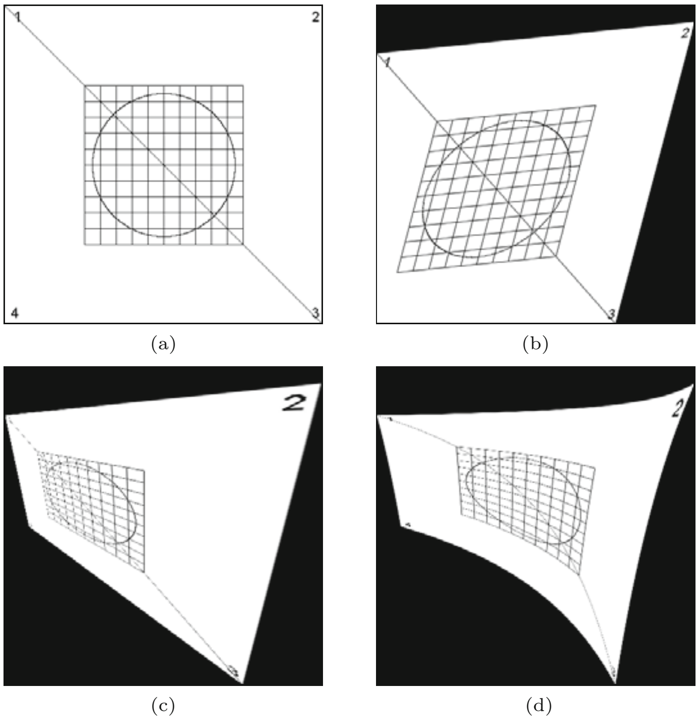


Fig. 21.6
Geometric transformations compared: original image (a), affine transformation with respect to the triangle 1-2-3 (b), projective transformation (c), and bilinear transformation (d).

21.1.6 Other Nonlinear Image Transformations

The bilinear transformation discussed in the previous section is only one example of a nonlinear mapping in 2D that cannot be expressed as a simple matrix-vector multiplication in homogeneous coordinates. Many other types of nonlinear deformations exist; for example, to implement various artistic effects for creative imaging. This type of image deformation is often called "image warping". Depending on the type of transformation used, the derivation of the *inverse* transformation function—which is required for the practical computation of the mapping using *target-to-source mapping* (see Sec. 21.2.2)—is not always easy or may even be impossible. In the following three examples, we therefore look straight at the inverse maps

$$\boldsymbol{x} = T^{-1}(\boldsymbol{x}') \qquad (21.62)$$

without really bothering about the corresponding *forward* transformations.

"Twirl" transformation

The twirl mapping causes the image to be rotated around a given anchor point $\boldsymbol{x}_c = (x_c, y_c)$ with a space-variant rotation angle, which has a fixed value α at the center $\boldsymbol{x}_c$ and decreases linearly with the radial distance from the center. The image remains unchanged outside the limiting radius $r_{\max}$. The associated (*inverse*) mapping is defined as

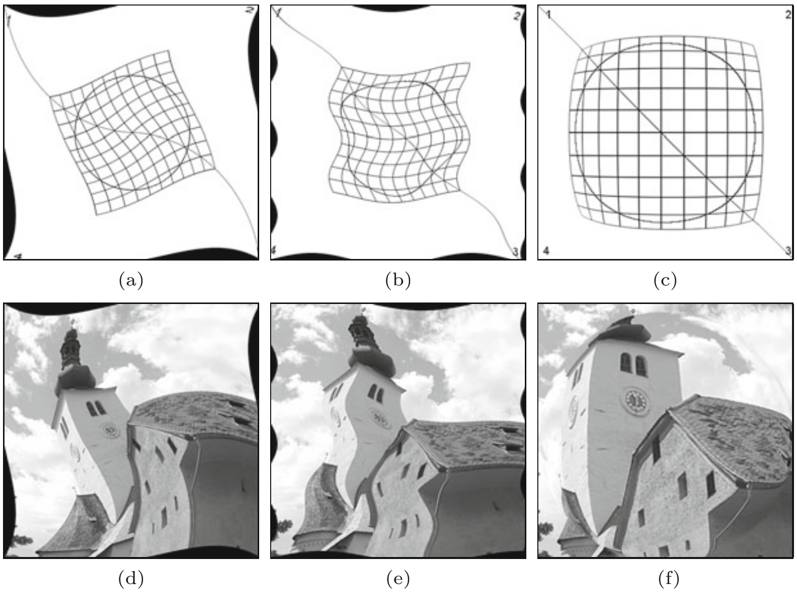


Fig. 21.7
Various nonlinear image deformations: *twirl* (a, d), *ripple* (b, e), and *sphere* (c, f) transformations. The size of the original images is 400 × 400 pixels.

$$T_x^{-1}\colon x = \begin{cases} x_c + r\cdot\cos(\beta) & \text{for } r \le r_{\max}, \\ x' & \text{for } r > r_{\max}, \end{cases} \tag{21.63}$$

$$T_y^{-1}\colon y = \begin{cases} y_c + r\cdot\sin(\beta) & \text{for } r \le r_{\max}, \\ y' & \text{for } r > r_{\max}, \end{cases} \tag{21.64}$$

with

$$r = \sqrt{d_x^2 + d_y^2}, \qquad d_x = x' - x_c, \tag{21.65}$$

$$\beta = \mathrm{ArcTan}(d_x, d_y) + \alpha\cdot\left(\tfrac{r_{\max}-r}{r_{\max}}\right), \qquad d_y = y' - y_c. \tag{21.66}$$

Figure 21.7(a, d) shows a twirl mapping with the anchor point $\boldsymbol{x}_c$ placed at the image center. The limiting radius $r_{\max}$ is half the length of the image diagonal, and the rotation angle is $\alpha = 43°$ at the center.

“Ripple” transformation

The ripple transformation causes a local wavelike displacement of the image along both the x and y directions. The parameters of this mapping function are the period lengths $\tau_x, \tau_y \neq 0$ (in pixels) and the corresponding amplitude values a_x, a_y for the displacement in both directions:

$$T_x^{-1}\colon x = x' + a_x\cdot\sin\left(\tfrac{2\pi\cdot y'}{\tau_x}\right), \tag{21.67}$$

$$T_y^{-1}\colon y = y' + a_y\cdot\sin\left(\tfrac{2\pi\cdot x'}{\tau_y}\right). \tag{21.68}$$

An example for the ripple mapping with $\tau_x = 120$, $\tau_y = 250$, $a_x = 10$, and $a_y = 15$ is shown in Fig. 21.7(b, e).

Spherical transformation

The spherical deformation imitates the effect of viewing the image through a transparent hemisphere or lens placed on top of the image.

The parameters of this transformation are the position $\boldsymbol{x}_c = (x_c, y_c)$ of the lens center, the radius of the lens $r_{\max}$ and its refraction index ρ. The corresponding mapping functions are defined as

$$T_x^{-1}\colon x = x' - \begin{cases} z \cdot \tan(\beta_x) & \text{for } r \le r_{\max}, \\ 0 & \text{for } r > r_{\max}, \end{cases} \tag{21.69}$$

$$T_y^{-1}\colon y = y' - \begin{cases} z \cdot \tan(\beta_y) & \text{for } r \le r_{\max}, \\ 0 & \text{for } r > r_{\max}, \end{cases} \tag{21.70}$$

with

$$\begin{aligned} r &= \sqrt{d_x^2 + d_y^2}\,, & \beta_x &= \bigl(1-\tfrac{1}{\rho}\bigr) \cdot \sin^{-1}\bigl(\tfrac{d_x}{\sqrt{(d_x^2+z^2)}}\bigr), & d_x &= x'-x_c, \\ z &= \sqrt{r_{\max}^2 - r^2}\,, & \beta_y &= \bigl(1-\tfrac{1}{\rho}\bigr) \cdot \sin^{-1}\bigl(\tfrac{d_y}{\sqrt{(d_y^2+z^2)}}\bigr), & d_y &= y'-y_c. \end{aligned} \tag{21.71}$$

Figure 21.7(c, f) shows a spherical transformation with the lens positioned at the image center. The lens radius $r_{\max}$ is set to half of the image width, and the refraction index is $\rho = 1.8$.

See Exercise 21.4 for additional examples of nonlinear geometric tarnsformations.

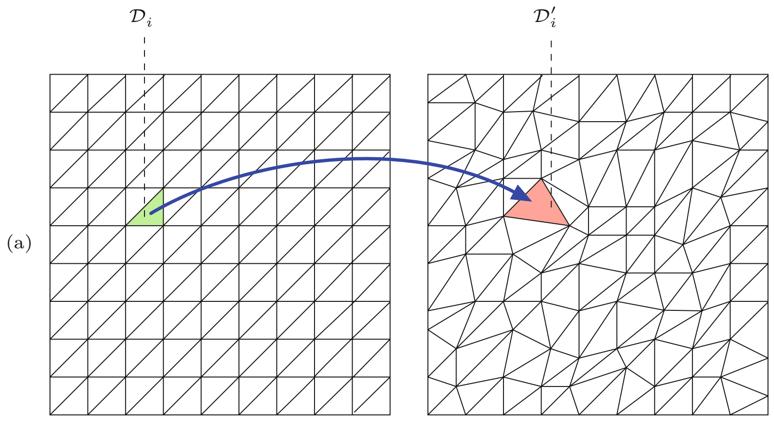
21.1.7 Piecewise Image Transformations

All the geometric transformations discussed so far are *global* (i.e., the same mapping function is applied to all pixels in the given image). It is often necessary to deform an image such that a larger number of n original image points $\boldsymbol{x}_0, \ldots, \boldsymbol{x}_n$ are precisely mapped onto a given set of target points $\boldsymbol{x}'_0, \ldots, \boldsymbol{x}'_n$. For $n = 3$, this problem can be solved with an affine mapping (see Sec. 21.1.3), and for $n = 4$ we could use a projective or bilinear mapping (see Secs. 21.1.4 and 21.1.5). A precise global mapping of $n > 4$ points requires a more complicated function $T(\boldsymbol{x})$ (e.g., a 2D nth-order polynomial or a spline function).

An alternative is to use *local* or *piecewise* transformations, where the image is partitioned into disjoint patches that are transformed separately, applying an individual mapping function to each patch. In practice, it is common to partition the image into a *mesh* of triangles or quadrilaterals, as illustrated in Fig. 21.8.

For a *triangular* mesh partitioning (Fig. 21.8(a)), the transformation between each pair of triangles $\mathcal{D}_i \rightarrow \mathcal{D}'_i$ could be accomplished with an *affine* mapping, whose parameters must be computed individually for every patch. Similarly, the *projective* transformation would be suitable for mapping each patch in a mesh partitioning composed of *quadrilaterals* $\mathcal{Q}_i$ (Fig. 21.8(b)). Since both the affine and the projective transformations preserve the straightness of lines, we can be certain that no holes or overlaps will arise and the deformation will appear continuous between adjacent mesh patches.

Local transformations of this type are frequently used; for example, to register aerial and satellite images or to undistort images for panoramic stitching. In computer graphics, similar techniques are used to map texture images onto polygonal 3D surfaces in the rendered 2D image. Another popular application of this technique is



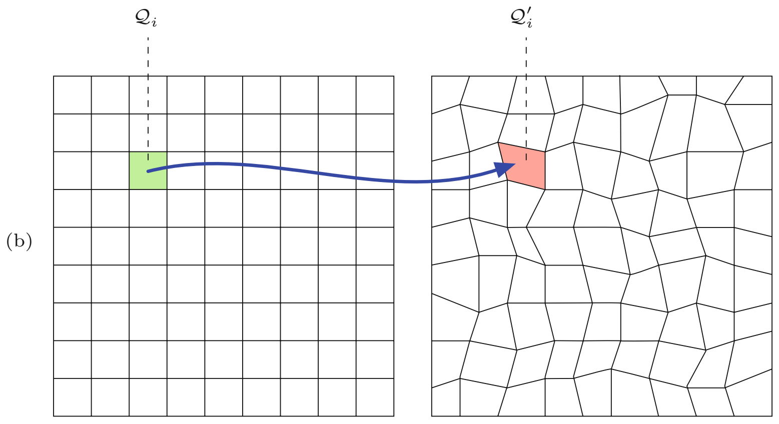


Fig. 21.8
Mesh partitioning examples. Almost arbitrary image deformations can be implemented by partitioning the image plane into nonoverlapping triangles $\mathcal{D}_i, \mathcal{D}'_i$ (a) or quadrilaterals $\mathcal{Q}_i, \mathcal{Q}'_i$ (b) and applying simple local transformations. Every patch in the resulting mesh is transformed separately with the required transformation parameters derived from the corresponding three or four corner points, respectively.

"morphing" [256], which performs a stepwise geometric transformation from one image to another while simultaneously blending their intensity (or color) values.[4]

21.2 Resampling the Image

In the discussion of geometric transformations, we have so far considered the 2D image coordinates as being continuous (i.e., real-valued). In reality, the picture elements in digital images reside at discrete (i.e., integer-valued) coordinates, and thus transferring a discrete image into another discrete image without introducing significant losses in quality is a nontrivial subproblem in the implementation of geometric transformations.

Based on the original image $I(u, v)$ and some (continuous) geometric transformations $T(x, y)$, the aim is to create a transformed image $I'(u', v')$ where all coordinates are discrete (i.e., $u, v \in \mathbb{Z}$ and

[4] Image morphing has also been implemented in ImageJ as a plugin (*iMorph*) by Hajime Hirase (http://rsb.info.nih.gov/ij/plugins/morph.html).

$u', v' \in \mathbb{Z}$).[5] This can be accomplished in one of two ways, which differ by the mapping direction and are commonly referred to as *source-to-target* or *target-to-source* mapping, respectively.

21.2.1 Source-to-Target Mapping

In this approach, which appears quite natural at first sight, we compute for every pixel (u, v) of the original (*source*) image I the corresponding transformed position

$$(x', y') = T(u, v) \tag{21.72}$$

in the target image I'. In general, the result will *not* coincide with any of the raster points, as illustrated in Fig. 21.9. Subsequently, we would have to decide in which pixel in the target image I' the original intensity or color value from $I(u, v)$ should be stored. We could perhaps even think of somehow distributing this value onto all adjacent pixels.

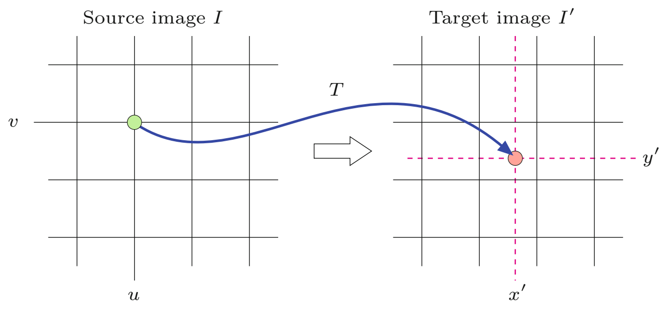


Fig. 21.9
Source-to-target mapping. For each discrete pixel position (u, v) in the source image I, the corresponding (continuous) target position (x', y') is found by applying the geometric transformation $T(u, v)$. In general, the target position (x', y') does not coincide with any discrete raster point. The source pixel value $I(u, v)$ is subsequently transferred to one of the adjacent target pixels.

The problem with the source-to-target method is that, depending on the geometric transformation T, some elements in the target image I' may never be "hit" at all (i.e., never receive a source pixel value)! This happens, for example, when the image is enlarged (even slightly) by the geometric transformation. The resulting holes in the target image would be difficult to close in a subsequent processing step. Conversely, one would have to consider (e.g., when the image is shrunk) that a single element in the target image I' may be hit by multiple source pixels and thus image content may get lost. In the light of all these complications, source-to-target mapping is not really the method of choice.

21.2.2 Target-to-Source Mapping

This method avoids most difficulties encountered in the source-to-target mapping by simply reversing the image generation process. For every discrete pixel position (u', v') in the *target* image, we determine the corresponding (continuous) point

[5] Remark on notation: We mostly use (u, v) or (u', v') to denote *discrete* (integer) coordinates and (x, y) or (x', y') for *continuous* (real-valued) coordinates.

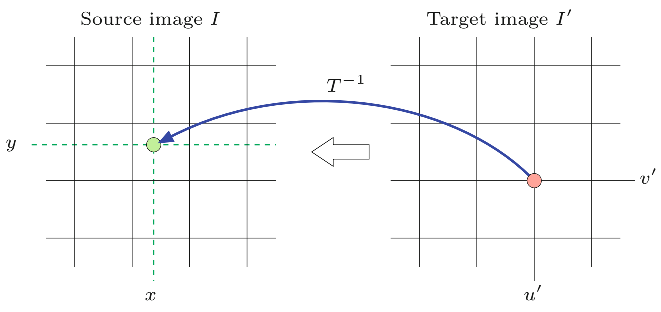


Fig. 21.10
Target-to-source mapping. For each discrete pixel position (u', v') in the target image I', the corresponding continuous source position (x, y) is found by applying the inverse mapping function $T^{-1}(u', v')$. The new pixel value $I'(u', v')$ is determined by interpolating the pixel values in the source image within some neighborhood of (x, y).

$$(x, y) = T^{-1}(u', v') \tag{21.73}$$

in the source image plane using the inverse geometric transformation T^{-1}. Of course, the coordinate (x, y) again does not fall onto a raster point in general (Fig. 21.10) and thus we have to decide from which of the neighboring source pixels to extract the resulting target pixel value. This problem of interpolating among intensity values is discussed in detail in Chapter 22.

The major advantage of the target-to-source method is that all pixels in the target image I' (and only these) are computed and filled exactly once such that no holes or multiple hits can occur. This, however, requires the *inverse* geometric transformation T^{-1} to be available, which is no disadvantage in most cases since the forward transformation T itself is never really needed. Due to its simplicity, which is also demonstrated in Alg. 21.1, *target-to-source* mapping is the common method for geometrically transforming 2D images.

```
1: TransformImage $(I, T)$
       Input: $I$, source image; $T$, continuous mapping $\mathbb{R}^2 \mapsto \mathbb{R}^2$.
       Returns the transformed image.
2:     $(M, N) \leftarrow$ Size$(I)$
3:     $I' \leftarrow$ duplicate$(I)$                      ▷ create the target image
4:     for all $(u, v) \in M \times N$ do                  ▷ loop over all target pixels
5:         $(x, y) \leftarrow T^{-1}(u, v)$
6:         $I'(u, v) \leftarrow$ GetInterpolatedValue$(I, x, y)$
7:     return $I'$
```

Alg. 21.1
Geometric image transformation using target-to-source mapping. Given are the original (source) image I and the continuous coordinate transformation T. GetInterpolatedValue(I, x, y) returns the interpolated value of the source image I at the continuous position (x, y).

21.3 Java Implementation

In plain ImageJ, only a few simple geometric operations are provided as methods for the `ImageProcessor` class, such as rotation and flipping.[6] This section describes the implementation of the transformations described in this chapter, which is openly available as part of the `imagingbook` library.[7]

[6] Additional operations, including affine transformations, are available as plugin classes as part of the optional `TransformJ` package [162].
[7] Package `imagingbook.pub.geometry.mappings`.

21.3.1 General Mappings (Class `Mapping`)

The abstract class `Mapping` is the superclass for all subsequent transformations. All subclasses of `Mapping` are required to implement the method `applyTo(double[] pnt)`, which applies the associated transformation to a given coordinate point and returns the transformed point. The actual transformations are implemented by its concrete sub-classes. The `applyTo()` method is defined in multiple versions with different signatures:

`double[] applyTo (double[] pnt)`
: Applies this transformation to the 2D point (of type `double[]`) and returns the transformed coordinate.

`Point2D applyTo (Point2D pnt)`
: Applies this transformation to the 2D point (of type `Point2D`) and returns the transformed coordinate.

`Point2D[] applyTo (Point2D[] pnts)`
: Applies this transformation to a sequence of the 2D points (of type `Point2D`) and returns a sequence of transformed coordinates.

In addition, the `Mapping` class can also be used to transform entire images:

`double[] applyTo (ImageProcessor source, ImageProcessor target, PixelInterpolator.Method im)`
: Transforms the input image `source` onto the output image `target` by target-to-source mapping, using the pixel interpolation method `im`.

`double[] applyTo (ImageProcessor ip, PixelInterpolator.Method im)`
: Transforms the input image `ip` destructively, using the pixel interpolation method `im`.

`double[] applyTo (ImageInterpolator source, ImageProcessor target)`
: Transforms the input image (specified by the interpolator `source`) onto the output image `target` by target-to-source mapping.

Other methods defined by class `Mapping`:

`Mapping duplicate ()`
: Returns a copy of this mapping.

`Mapping getInverse ()`
: Returns the inverse of this mapping if available. Otherwise an `UnsupportedOperationException` is thrown.

21.3.2 Linear Mappings

Linear transformations are implemented by class `LinearMapping`,[8] with sub-classes including

`AffineMapping`,	`Scaling`,
`ProjectiveMapping`,	`Shear`,
`Rotation`,	`Translation`.

[8] Package `imagingbook.pub.geometry.mappings.linear`.

21.3.3 Nonlinear Mappings

Selected nonlinear transformations are implemented by the following subclasses of `Mapping`:[9]

`BilinearMapping`,	`ShereMapping`,
`RippleMapping`,	`TwirlMapping`.

21.3.4 Sample Applications

The following two ImageJ plugins show two simple examples of the use of the classes in Secs. 21.3.2 and 21.3.3 for implementing geometric operations and pixel interpolation (see Ch. 22 for details). Note that these plugins can be applied to any type of image.

Example 1: image rotation

The example in Prog. 21.1 shows a plugin (`Transform_Rotate`) to rotate an image by 15°. First (in line 16) the geometric mapping object (`map`) is created as an instance of class `Rotation`, with the supplied angle being converted from degrees to radians. The actual transformation of the image is performed by invoking the method `applyTo()` in line 17.

Prog. 21.1
Image rotation example using the `Rotation` class (ImageJ plugin).

```java
 1 import ij.ImagePlus;
 2 import ij.plugin.filter.PlugInFilter;
 3 import ij.process.ImageProcessor;
 4 import imagingbook.pub.geometry.interpolators.pixel.
       PixelInterpolator;
 5 import imagingbook.pub.geometry.mappings.Mapping;
 6 import imagingbook.pub.geometry.mappings.linear.Rotation;
 7
 8 public class Transform_Rotate implements PlugInFilter {
 9   static double angle = 15; // rotation angle (in degrees)
10
11     public int setup(String arg, ImagePlus imp) {
12         return DOES_ALL;
13     }
14
15     public void run(ImageProcessor ip) {
16       Mapping map = new Rotation((2*Math.PI*angle)/360);
17       map.applyTo(ip, PixelInterpolator.Method.Bicubic);
18     }
19 }
```

Example 2: projective transformation

The second example in Prog. 21.2 illustrates the implementation of a projective transformation. The geometric mapping T is defined by two corresponding quadrilaterals `P` = `p0`, ..., `p3` and `Q` = `q0`, ..., `q3`, respectively. In a real application, these points would probably be specified interactively or given as the result of a mesh partitioning.

[9] Package `imagingbook.pub.geometry.mappings.nonlinear`.

Prog. 21.2 Projective image transformation example using the `ProjectiveMapping` class (ImageJ plugin).

```
 1 import ij.ImagePlus;
 2 import ij.plugin.filter.PlugInFilter;
 3 import ij.process.ImageProcessor;
 4 import imagingbook.pub.geometry.interpolators.pixel.
       PixelInterpolator;
 5 import imagingbook.pub.geometry.mappings.Mapping;
 6 import imagingbook.pub.geometry.mappings.linear.
       ProjectiveMapping;
 7 import java.awt.Point;
 8 import java.awt.geom.Point2D;
 9
10 public class Transform_Projective implements PlugInFilter {
11
12     public int setup(String arg, ImagePlus imp) {
13         return DOES_ALL;
14     }
15
16     public void run(ImageProcessor ip) {
17       Point2D p0 = new Point(0, 0);
18       Point2D p1 = new Point(400, 0);
19       Point2D p2 = new Point(400, 400);
20       Point2D p3 = new Point(0, 400);
21
22       Point2D q0 = new Point(0, 60);
23       Point2D q1 = new Point(400, 20);
24       Point2D q2 = new Point(300, 400);
25       Point2D q3 = new Point(30, 200);
26
27       Mapping map = new
28         ProjectiveMapping(p0, p1, p2, p3, q0, q1, q2, q3);
29
30       map.applyTo(ip, PixelInterpolator.Method.Bilinear);
31     }
32 }
```

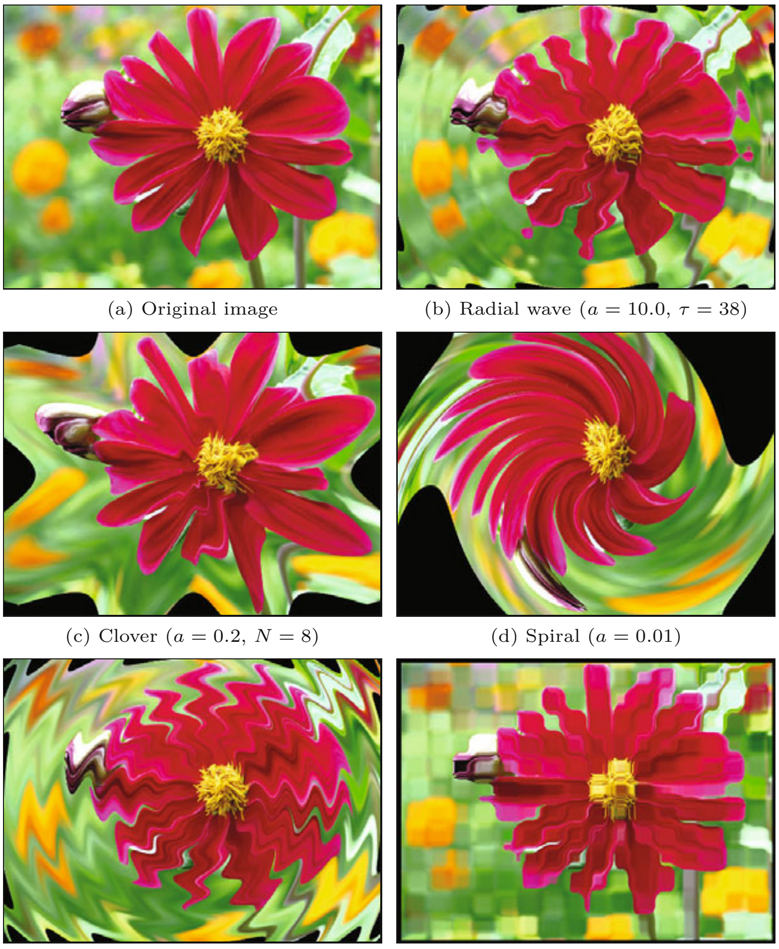
The transformation object `map` (representing the forward transformation T) is created by calling the associated constructor `ProjectiveMapping()` in line 28. The mapping is applied to the input image (line 30), as in the previous example, except for the use of *bilinear* pixel interpolation.

21.4 Exercises

Exercise 21.1. Show that a straight line $y = kx+d$ in 2D is mapped to another straight line under a projective transformation, as defined in Eqn. (21.32).

Exercise 21.2. Show that parallel lines remain parallel under affine transformation (Eqn. (21.20)).

Exercise 21.3. Design a nonlinear geometric transformation similar to the ripple transformation (Eqn. (21.67)) that uses a *sawtooth* function instead of a sinusoid for the distortions in the horizontal



(a) Original image (b) Radial wave ($a = 10.0$, $\tau = 38$) (c) Clover ($a = 0.2$, $N = 8$) (d) Spiral ($a = 0.01$) (e) Angular wave ($a = 0.1$, $\tau = 38$) (f) Tapestry ($a = 5.0$, $\tau_x = \tau_y = 30$)

Fig. 21.11 Examples of the nonlinear geometric transformations defined in Exercise 21.4. The reference point $\boldsymbol{x}_c$ is always taken at the image center.

and vertical directions. Use the class `TwirlMapping` as a template for your implementation.

Exercise 21.4. Implement one or more of the following nonlinear geometric transformations (see Fig. 21.11):

A. **Radial wave** transformation: This transformation simulates an omni-directional wave which originates from a fixed center point $\boldsymbol{x}_c$ (see Fig. 21.11(b)). The inverse transformation (applied to a target image point $\boldsymbol{x}' = (x', y')$) is

$$T^{-1}\colon \boldsymbol{x} = \begin{cases} \boldsymbol{x}_c & \text{for } r = 0, \\ \boldsymbol{x}_c + \frac{r+\delta}{r} \cdot (\boldsymbol{x}' - \boldsymbol{x}_c) & \text{for } r > 0, \end{cases} \tag{21.74}$$

with $r = \|\boldsymbol{x}' - \boldsymbol{x}_c\|$ and $\delta = a \cdot \sin(2\pi r/\tau)$. Parameter a specifies the *amplitude* (strength) of the distortion and τ is the *period* (width) of the radial wave (in pixel units).

B. **Clover** transformation: This transformation distorts the image in the form of a N-leafed clover shape (see Fig. 21.11(c)). The associated inverse transformation is the same as in Eqn. (21.74) but uses

$$\delta = a \cdot r \cdot \cos(N \cdot \alpha), \quad \text{with } \alpha = \angle(\boldsymbol{x}' - \boldsymbol{x}_{\mathrm{c}}) \tag{21.75}$$

instead. Again $r = \|\boldsymbol{x}' - \boldsymbol{x}_{\mathrm{c}}\|$ is the radius of the target image point $\boldsymbol{x}'$ from the designated center point $\boldsymbol{x}_{\mathrm{c}}$. Parameter a specifies the amplitude of the distortion and N is the number of radial "leaves".

C. **Spiral** transformation: This transformation (see Fig. 21.11(d)) is similar to the *twirl* transformation in Eqns. (21.63)–(21.64), defined by the inverse transformation

$$T^{-1}\colon \boldsymbol{x} = \boldsymbol{x}_{\mathrm{c}} + r \cdot \begin{pmatrix} \cos(\beta) \\ \sin(\beta) \end{pmatrix}, \tag{21.76}$$

with $\beta = \angle(\boldsymbol{x}' - \boldsymbol{x}_{\mathrm{c}}) + a \cdot r$ and $r = \|\boldsymbol{x}' - \boldsymbol{x}_{\mathrm{c}}\|$ denoting the distance from the target point $\boldsymbol{x}'$ and the center point $\boldsymbol{x}_{\mathrm{c}}$. The angle β increases linearly with r; parameter a specifies the "velocity" of the spiral.

D. **Angular wave** transformation: This is another variant of the *twirl* transformation in Eqns. (21.63)–(21.64). Its inverse transformation is the same as for the spiral mapping in Eqn. (21.76), but in this case

$$\beta = \angle(\boldsymbol{x}' - \boldsymbol{x}_{\mathrm{c}}) + a \cdot \sin\left(\tfrac{2\pi r}{\tau}\right). \tag{21.77}$$

Thus the angle β is modified by a sine function with amplitude a (see Fig. 21.11(e)).

E. **Tapestry** transformation: In this case the inverse transformation of a target point $\boldsymbol{x}' = (x', y')$ is

$$T^{-1}\colon \boldsymbol{x} = \boldsymbol{x}' + a \cdot \begin{pmatrix} \sin\left(\frac{2\pi}{\tau_x} \cdot (x' - x_{\mathrm{c}})\right) \\ \sin\left(\frac{2\pi}{\tau_y} \cdot (y' - y_{\mathrm{c}})\right) \end{pmatrix}, \tag{21.78}$$

again with the center point $\boldsymbol{x}_{\mathrm{c}} = (x_{\mathrm{c}}, y_{\mathrm{c}})$. Parameter a specifies the distortion's amplitude and τ_x, τ_y are the wavelengths (measured in pixel units) along the x and y axis, respectively (see Fig. 21.11(f)).

Exercise 21.5. Implement an interactive program (plugin) that performs projective rectification (see Sec. 21.1.4) of a selected quadrilateral, as shown in Fig. 21.12. Make your program perform the following steps:

1. Let the user mark the source quad in the source image I as a polygon-shaped *region of interest* (ROI) with at least four points $\boldsymbol{x}_0, \ldots, \boldsymbol{x}_3$. In ImageJ this is easily done with the built-in polygon selection tool (see Prog. 21.3 for handling ROI points).
2. Create an output image I' of fixed size (i.e., proportional to A4 or Letter paper size).
3. The target rectangle is defined by the four corners $\boldsymbol{x}'_0, \ldots, \boldsymbol{x}'_3$ of the output image. The source and target points are associated 1:1, that is, the four corresponding point pairs are $\langle \boldsymbol{x}_0, \boldsymbol{x}'_0 \rangle, \ldots, \langle \boldsymbol{x}_3, \boldsymbol{x}'_3 \rangle$.

4. From the four point pairs, create an instance of `Projective-Mapping`, as demonstrated in Prog. 21.2.
5. Test the obtained mapping by applying **A** to the specified source points $\boldsymbol{x}_0, \ldots, \boldsymbol{x}_3$. Make sure they project exactly to the specified target points $\boldsymbol{x}'_0, \ldots, \boldsymbol{x}'_3$.
6. Apply the obtained mapping from the source to the target image using the method[10]

```
    void applyTo(ImageProcessor source,
       ImageProcessor target, InterpolationMethod im).
```

7. Show the resulting output image.

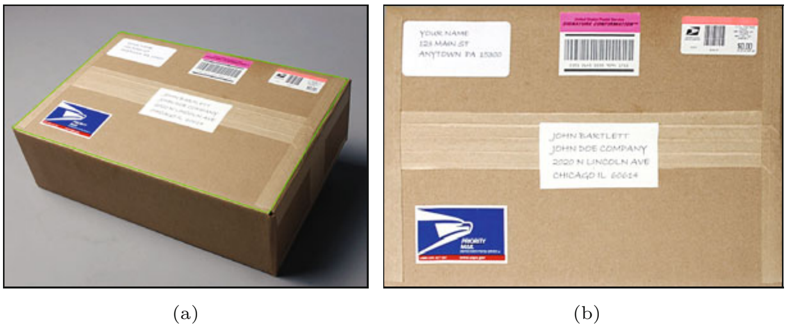


Fig. 21.12
Projective rectification example (see Exercise 21.5). Source image and user-defined selection (a); transformed output image (b).

[10] Defined in class `imagingbook.pub.geometry.mappings.Mapping`.

Prog. 21.3
ImageJ plugin demonstrating the extraction of vertex points from a user-selected polygon-ROI (region of interest). Notice that (in line 21) the region of interest (ROI) is obtained from the associated `ImagePlus` instance (to which a reference is kept in line 16) and not from the supplied `ImageProcessor` object. ImageJ's ROI coordinates are integer positions in general.

```java
 1 import java.awt.Point;
 2 import java.awt.Polygon;
 3 import java.awt.geom.Point2D;
 4
 5 import ij.ImagePlus;
 6 import ij.gui.PolygonRoi;
 7 import ij.gui.Roi;
 8 import ij.plugin.filter.PlugInFilter;
 9 import ij.process.ImageProcessor;
10
11 public class Get_Roi_Points implements PlugInFilter {
12
13   ImagePlus im = null;
14
15   public int setup(String args, ImagePlus im) {
16     this.im = im;    // keep a reference to im
17     return DOES_ALL + ROI_REQUIRED;
18   }
19
20   public void run(ImageProcessor source) {
21     Roi roi = im.getRoi();
22     if (!(roi instanceof PolygonRoi)) {
23       IJ.error("Polygon selection required!");
24       return;
25     }
26
27     Polygon poly = roi.getPolygon();
28
29     // copy polygon vertices to a point array:
30     Point2D[] pts = new Point2D[poly.npoints];
31     for (int i = 0; i < poly.npoints; i++) {
32       pts[i] = new Point(poly.xpoints[i], poly.ypoints[i]);
33     }
34
35     ... // use the ROI points in pts
36
37   }
38
39 }
```