Chapter 5
Transforming Image Locations

5.1 Location Transformations

Numerically, a digital image has a width W px, a height H px and an extent E px. Each pixel has a location (X, Y) within the extent and a colour (R, G, B) within the colour space. A *location transformation* is a systematic change of all the pixel locations of an image, without change of colour.

Elementary location transformations are as follows:

Cropping,
Framing,
Dilating,
Translating,
Reflecting,
Rotating,
Shearing and
Inverting.

Computationally, the preferred methods for dilating, translating, reflecting, rotating and shearing are by matrix multiplication [2], Chap. 7. Efficient implementations of these methods are taken from the Python Imaging Library [3].

To do a location transformation, the steps are as follows:

Select the source image file in storage.
Open it.
Select the transformation wanted.
Set the transformation parameters.
Do the transformation.
Save the transformed image to storage.

A. Parkin, *Computing Colour Image Processing*,
https://doi.org/10.1007/978-3-319-74076-8_5

5.2 Cropping

Cropping an image removes one or more pixels from left, top, right and bottom of the array. We can do this in an image editor, such as Microsoft Paint, where the options are Home - Select - Rectangular selection - (Set rectangle) - Cut to clipboard - File - New - (Don't save old) - Paste from clipboard - Save-as - (New filename and extension).

Or we can write a script to do the same thing. The script is listed below, and Fig. 5.1 shows a simple image before and after running this script:

```
# pycrop: Python program to crop a .bmp image.
# Written by Alan Parkin 2017.

from PIL import Image
import os, sys

# Get source image and show it.
# Enter any .bmp filemame in working directory.
imagefilename = raw_input("Image filename? ")
#Open it, show it, and print W H and E.
im = Image.open(imagefilename)
im.show()
width = im.size[0]
height = im.size[1]
extent = width * height
print "SOURCE IMAGE"
print "width W", width
print "height H", height
print "extent E", extent
print "--------"

# Enter crop box wanted.
print "Enter left, top, right, bottom coordinates of box wanted, as px "
lefte = raw_input("Left edge? ")
tope = raw_input("Top edge? ")
righte = raw_input("Right edge? ")
bottome = raw_input("Bottom edge? ")
lint = int(lefte)
tint = int(tope)
rint = int(righte)
bint = int(bottome)

# Do crop and show. Save under new filename in show.
imcropped = im.crop((lint, tint, rint, bint))
imcropped.show()
```

Figure 5.2 shows a camera image cropped by this script.

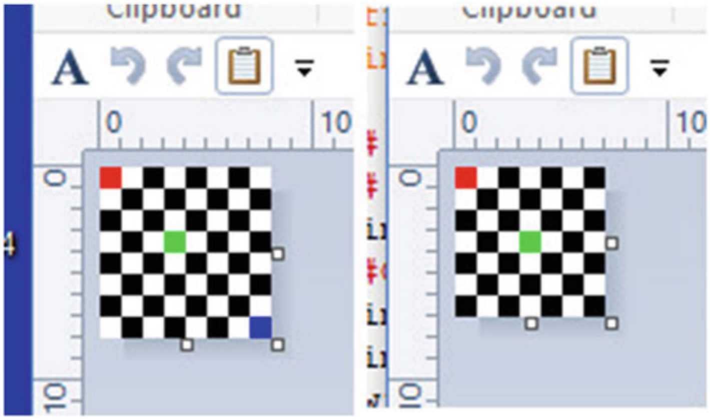

Fig. 5.1 Cropping an image. **a** Simple image 8 × 8 px, magnified in a Paint window. **b** Image cropped to 7 × 7 px, without interpolation

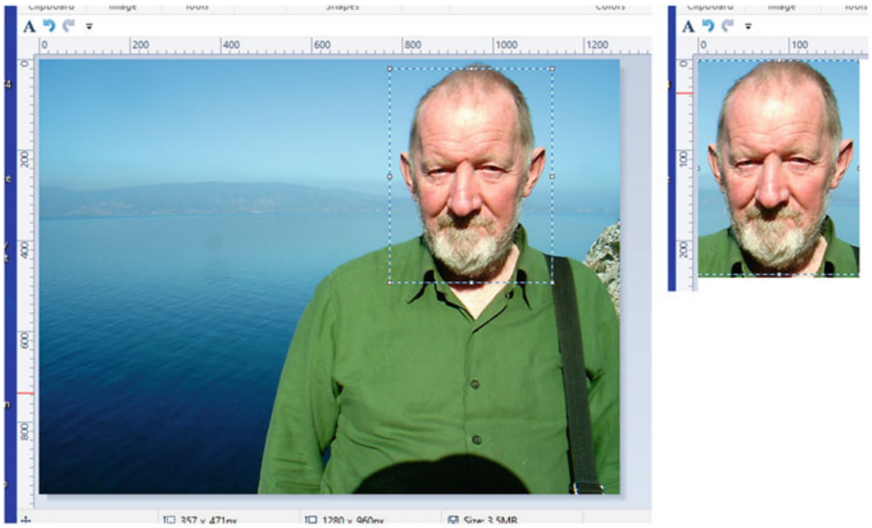

Fig. 5.2 Cropping an image. **a** Camera image 1280 × 960 px, with crop rectangle marked. **b** Cropped image 180 × 240 px

5.3 Framing

Framing an image adjoins none or more pixels to left, top, right and bottom of the array: the inverse of cropping. We can do this in an image editor, such as Microsoft Paint, where the options are File - Open image (note width and height) - Image - Select all - Clipboard - Copy - File - New - Edit colour (select frame colour) - Shapes (select rectangle) - (draw rectangle of frame width and height) - Clipboard - Paste -

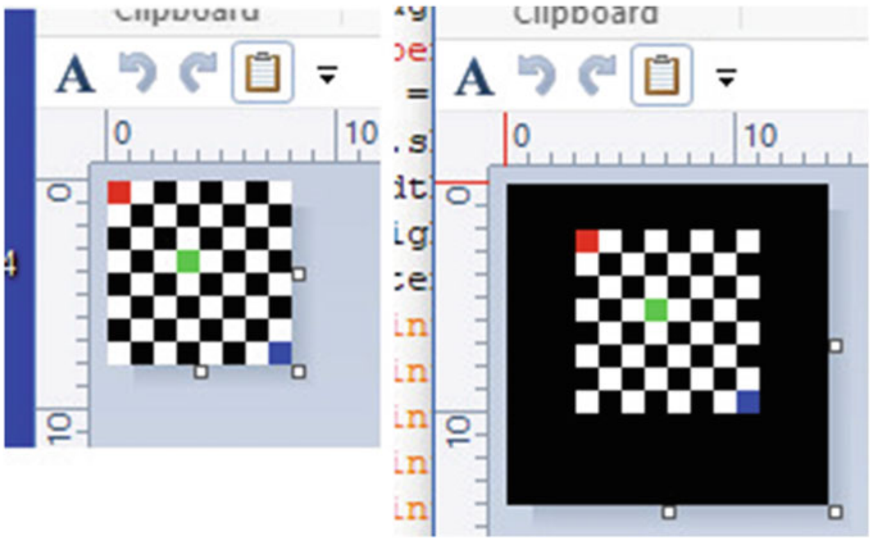

Fig. 5.3 Framing an image. **a** Simple image 8 × 8 px, magnified in a Paint window. **b** The imaged framed, to 14 × 14 px

(move image within frame). For example, Fig. 5.3 shows a simple image before and after framing.

Or we can write a script to do the same thing. The script is listed below, and Fig. 5.3 shows a simple image before and after running this script:

```
# pyframe: Python script to frame a .bmp image.
# Written by Alan Parkin 2017.

from PIL import Image
import os, sys

# Get source image and show it.
# Enter any .bmp filemame in working directory.
imagefilename = raw_input("Image filename? ")
#Open it, show it, and print W H and E.
im = Image.open(imagefilename)
im.show()
width = im.size[0]
height = im.size[1]
extent = width * height
print "SOURCE IMAGE"
print "width W", width
print "height H", height
print "extent E", extent
print "--------"

# Enter frame thicknesses wanted.
print "Enter top, left, right, bottom frame thicknesses wanted, in px "
topt = raw_input("Top thickness? ")
leftt = raw_input("Left thickness? ")
rightt = raw_input("Right thickness? ")
bottomt = raw_input("Bottom thickness? ")
lint = int(leftt)
tint = int(topt)
rint = int(rightt)
```

```
bint = int(bottomt)
framewidth = lint + width + rint
print "framewidth", framewidth
frameheight = tint + height + bint
print "frameheight", frameheight

# Do frame and show. Save under new filename in show.
frame = Image.new("RGB", (framewidth, frameheight))
frame.paste(im, (lint, tint), 0)
frame.show()
```

5.4 Dilating

In dilating (often called resizing or stretching/shrinking) , an image multiplies the width or height or both by a chosen factor F. Dilating down removes columns and rows of pixels from an image. To dilate an image down from $W \times H$ px to $W' \times H'$ px we remove $W - W'$ columns and $H - H'$ rows of pixels. For example, Fig. 5.4 shows a simple image 8 × 8 px dilated down to 7 × 7 px by removing 8−7 = 1 column and 1 row. Much work has gone into introducing an interpolation process into simple digital dilation, in pursuit of a smoothed “photographic” result (which we may think is rather un-digital in spirit). Interpolation averages the colour of each pixel between two or four neighbours. Figure 5.4 shows dilations without interpolation, and with three different interpolations.

Dilating up repeats columns and rows of pixels in an image: to dilate an image up from $W \times H$ px to $W' \times H'$ px, we repeat $W' - W$ columns and $H' - H$ rows of pixels. For example, Fig. 5.4 shows the simple image 8 × 8 px dilated up to 9 × 9 px by repeating 9−8 = 1 column and 1 row without interpolation, and with three interpolations.

Dilating can be applied in the X-direction or the Y-direction separately, or in both directions together.

We can do a dilation in an image editor, such as Microsoft Paint, where the options are Home - Select - Rectangular selection - (Set rectangle) - Cut to clipboard - File - New - (Don’t save old) - Paste from clipboard - Save-as - (New filename and extension). Or we can write a script to do the same thing, as shown below:

```
# pydilate: Python program to dilate a .bmp image.
# Written by Alan Parkin 2017.

from PIL import Image
import os, sys

# Get source image and show it.
# Enter any .bmp filemame in working directory.
imagefilename = raw_input("Image filename? ")
#Open it, show it, and print W H and E.
im = Image.open(imagefilename)
im.show()
width = im.size[0]
height = im.size[1]
extent = width * height
print "SOURCE IMAGE"
print "width W", width
print "height H", height
print "extent E", extent

# Enter dilations and filter wanted.
print "--------"
print "Enter dilations wanted, as percents"
xpc = raw_input("In the x-direction? ")
ypc = raw_input("In the y-direction? ")
rsfilter = raw_input("Enter 0 for no filter, 2 for bicubic, 3 for antialias? ")
xfloat = float(xpc)
xfactor = float(xfloat / 100)
xrecip = 1 / xfactor
a = xrecip
newwidth = int(width * xfactor)
yfloat = float(ypc)
yfactor = float(yfloat / 100)
yrecip = 1 / yfactor
e = yrecip
newheight = int(height * yfactor)
rsf = int(rsfilter)

# Print new width and height.
print "--------"
print "new width ", newwidth
print "new height ", newheight

# Do dilate and show. Save under new filename in show.
imdilate = im.transform((newwidth, newheight), 0, (a,0,0,0,e,0),rsf)
imdilate.show()
```

Unlike the temporary diminution and magnification provided in an image editor, dilation makes a permanent change to the image, so it is usually returned to storage under a new filename.

Dilating down from extent E to extent E', $E > E'$, changes location resolution *LOCRES* from $1/E$ to a coarser $1/E'$. Dilating up from extent E to extent E', $E < E'$, changes location resolution from $1/E$ to a coarser $(E/E')/E'$, which appears as an increase of pixel size relative to extent. For example, Fig. 5.5a shows a camera image 222 $\times$ 290 px, with location resolution 1/64380. (b) shows the image after dilating down by a factor of 2–50 percent, then dilating up again by a factor of 2 to the original 100 percent. The apparent pixels are now the merge of four of the original pixels; the apparent extent is 111 $\times$ 145 = 16095 px, and the apparent location resolution is 1/16095. (c) shows the image after dilating down by a factor of 4–25 percent, then dilating up again by a factor of 4 to the original 100 percent. The apparent pixels are now the merge of 16 of the original pixels; the apparent extent is 56 $\times$ 73 = 4032 px, and the apparent location resolution is 1/4032. (d) shows the image after dilating down by a factor of 8–12.5 percent, then dilating up again by a factor of 8 to the original 100 percent. The apparent pixels are now the merge of 64 of the original pixels; the apparent extent is 28 $\times$ 36 = 1008 px, and the apparent location resolution is 1/1008.

The script pydownup listed below does a dilation down and a dilation up to the original extent, using an interpolated resize:

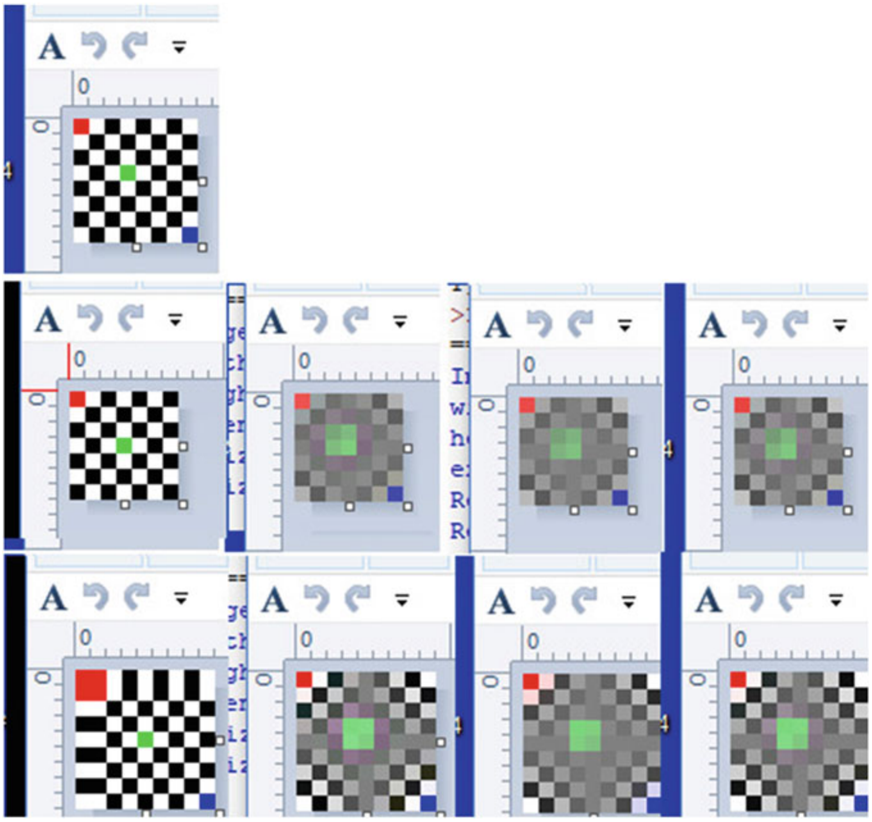

Fig. 5.4 Dilating an image. Top row: **a** Source image 8 × 8 px, magnified in Paint window. Middle row: **b** Dilated down to 7 × 7 px without interpolation. **c** With nearest neighbour interpolation. **d** With bilinear 2 × 2 interpolation. **e** With bicubic 4 × 4 interpolation. Bottom row: **f** Dilated up to 9 × 9 px without interpolation. **g–i** With interpolation as above

```
# pydownup: Python program to dilate a .bmp image
# down then up.
# Written by Alan Parkin 2017.

from PIL import Image
import os, sys

#Get filename of .bmp image which is in working directory
imagefilename = raw_input("Image filename? ")
#Open it and show it
im = Image.open(imagefilename)
im.show()
width = im.size[0]
height = im.size[1]
extent = width * height
print "SOURCE IMAGE"
print "width W", width
print "height H", height
print "extent E", extent
print "--------"
```

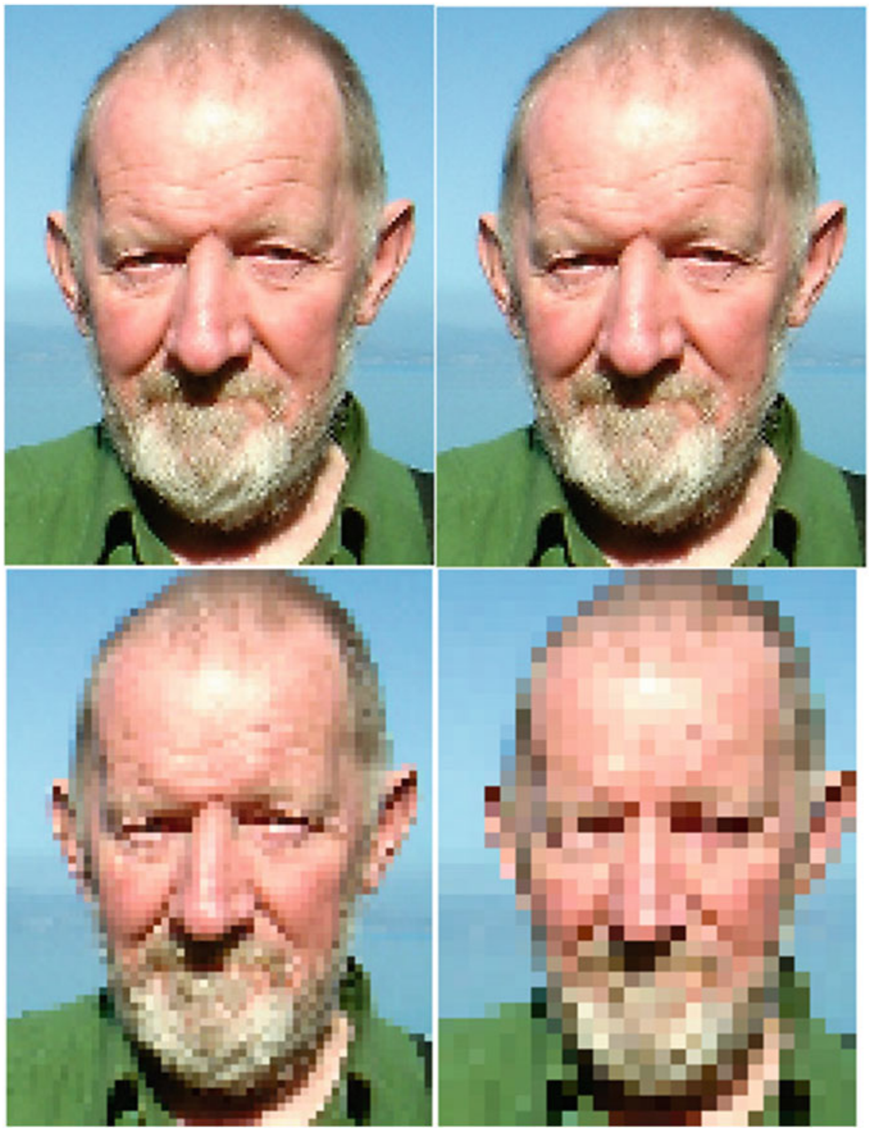

Fig. 5.5 Dilating down then up. **a** Source image 222 × 290 px (100 percent). **b** After dilating (**a**) down by factor 2–50 percent, then dilating up again by factor 2–100 percent. **c** After dilating (**a**) down by factor 4–25 percent, then dilating up again to 100 percent. **d** After dilating (**a**) down by factor 8–12.5 percent, then dilating up again to 100 percent

```
# Enter factor to resize down then up.
downupfactor = raw_input("Factor to resize down and up (integer)? ")
dufac = int(downupfactor)

# Do resize down.
downwidth = width/dufac
downheight = height/dufac
imdownsize = im.resize((downwidth, downheight), 0)

# Do resize up.
imupsize = imdownsize.resize((width, height), 0)
imupsize.show()
```

5.5 Translating

Translating moves an image leftwards or upwards or rightwards or downwards, without rotating, so that it lies partly or wholly outside its original frame. For example, Fig. 5.6 shows a simple image translated rightwards and downwards, then leftwards and upwards.

We can do a translation in an image editor, such as Microsoft Paint, where the options are Home - Select - Rectangular selection - Move to new position - Save-as - (New filename and extension).

Or we can write a script to do the translation, as shown below:

```
# pytranslate: Python program to translate
# a .bmp image.
#Written by Alan Parkin 2017.

from PIL import Image
#from PIL import ImageFilter
import os, sys

# Get source image and show it.
# Enter any .bmp filemame in working directory.
imagefilename = raw_input("Image filename? ")
#Open it, show it, and print W H and E.
im = Image.open(imagefilename)
im.show()
width = im.size[0]
height = im.size[1]
extent = width * height
print "width W", width
print "height H", height
print "extent E", extent

# Enter matrix element c to move image to the right,
# -c to the left. Enter f to move image down,
# -f up. Original extent is the containing window.
matc = raw_input("Move image left/right by -c/+c pixels ? ")
matf = raw_input("Moveimage up/down by -f/+f pixels? ")
matcint = -int(matc)
print "matcint ", matcint
matfint = -int(matf)
print "matfint ", matfint

# Do translate and show. Save under new filename in show.
imtranslate = im.transform((width, height), 0, (1,0,matcint,0,1,matfint))
imtranslate.show()
```

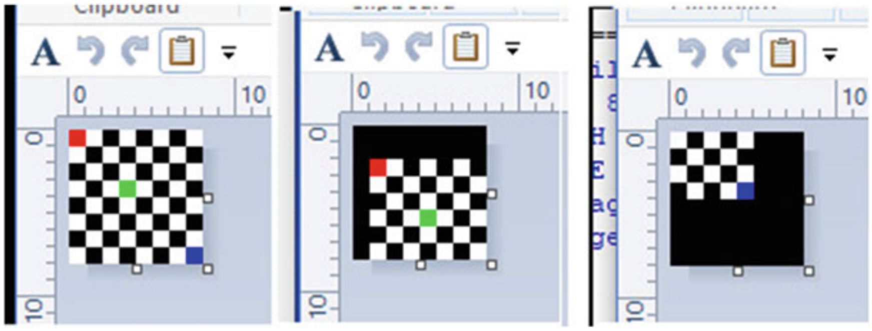

Fig. 5.6 Translating an image. **a** Simple image in original position. **b** Translated 1 px rightwards and 2 px downwards. **c** Translated 3 px leftwards and 4 px upwards

5.6 Reflecting

Reflecting switches left and right in an image, as if in a vertical mirror through its centre, or switches top and bottom, as if in a horizontal mirror through its centre. For example, Fig. 5.7 shows the simple image from Fig. 5.6 reflected left/right, top/bottom and both left/right and top/bottom (this last is the same as rotated 180 degrees).

We can do a reflection in an image editor, such as Microsoft Paint, where the options are Home - Select - Select all - Rotate/flip - Flip vertical or Flip horizontal - File - Save-as (New filename and extension).

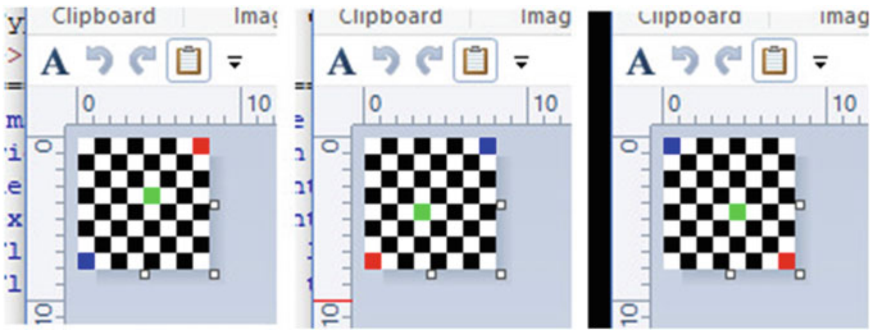

Fig. 5.7 Reflecting an image. **a** The simple image from Fig. 5.6 reflected left/right. **b** Reflected top/bottom. **c** Reflected left/right and top/bottom

Or we can write a script to do the reflection, as shown below:

```
# pyreflect: Python program to reflect a .bmp image
# left/right or top/bottom or both.
# Written by Alan Parkin 2017.

from PIL import Image
import os, sys

# Get source image and show it.
# Enter any .bmp filemame in working directory.
imagefilename = raw_input("Image filename? ")
#Open it, show it, and print W H and E.
im = Image.open(imagefilename)
im.show()
width = im.size[0]
height = im.size[1]
extent = width * height
print "width W", width
print "height H", height
print "extent E", extent

# Enter left-right or top-bottom flip.
xrefl = raw_input("Flip left-right (y/n) ? ")
yrefl = raw_input("Flip top-bottom (y/n) ? ")

# Do left-right or top-bottom or both transposes.
# Show result and save in Paint.
if xrefl == "y" and yrefl == "n":
    imlr = im.transpose(0)
    imlr.show()

elif xrefl == "n" and yrefl == "y":
    imtb = im.transpose(1)
    imtb.show()

elif xrefl == "y" and yrefl == "y":
    imlr = im.transpose(0)
    imboth = imlr.transpose(1)
    imboth.show()
```

5.7 Rotating

Rotating turns an image clockwise or counterclockwise about a pole at its centre. For example, Fig. 5.8 shows a simple image rotated counterclockwise by 15, 30 and 45 degrees, without and with interpolation. Rotating takes part of an image outside its original rectangular frame: in these examples, the new frame contains the whole rotated image. Figure 5.9 shows a camera image rotated for pictorial reasons.

We can do 90-degree rotations in Microsoft Paint, where the options are Home - Select - Select all - Rotate/flip - rotate left 90/right 90/1eft 90/rotate 180 - File - Save-as (New filename and extension). Some editors such as Microsoft Paint.net offer rotation by any degrees (Fig. 5.9).

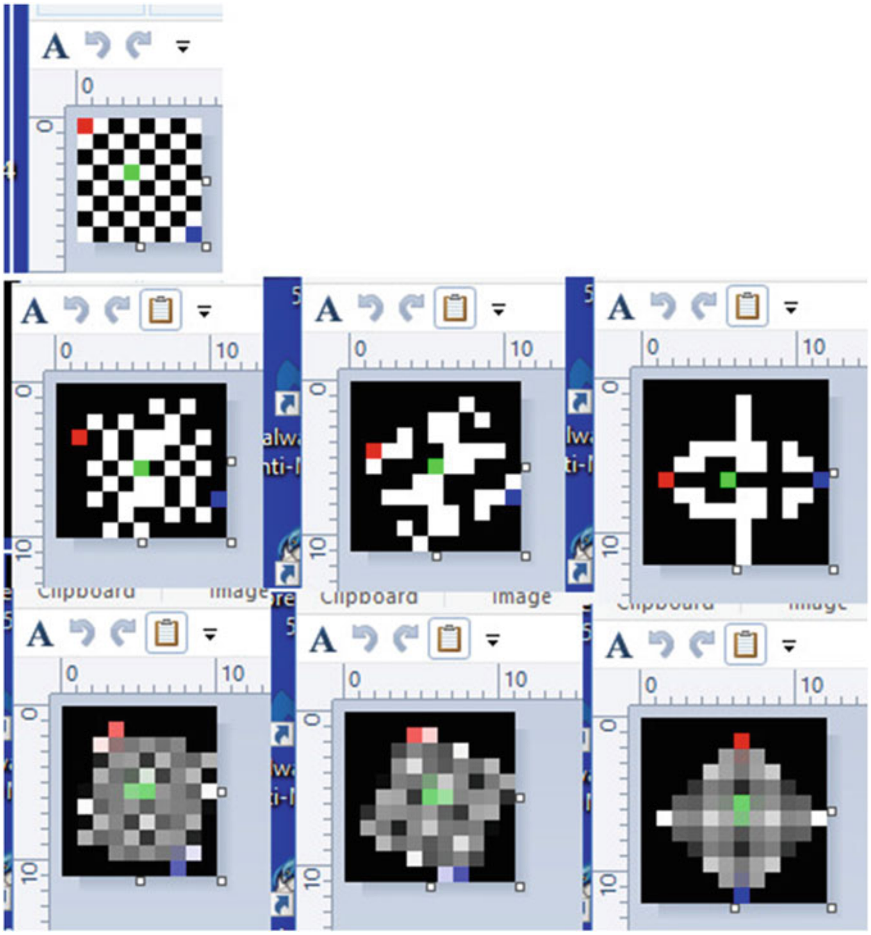

Fig. 5.8 Rotating a simple image. Top row: **a** Simple image 8 × 8 px. Middle row, without interpolation. **b** Rotated 15 degrees counterclockwise; frame expanded to 11 × 10 px. **c** Rotated by 30 degrees counterclockwise; frame expanded to 11 × 11 px. **d** Rotated by 45 degrees counterclockwise; frame expanded to 12 × 12 px. Bottom row: with interpolation: **e** Rotated by 15 degrees clockwise; frame expanded to 10 × 11 px. **d** Rotated by 45 degrees counterclockwise; frame expanded to 12 × 12 px. **e** Rotated by 15 degrees clockwise; frame expanded to 10 × 11 px.**f** Rotated by 30 degrees clockwise; frame expanded to 11 × 11 px. **g** Rotated by 45 degrees clockwise; frame expanded to 12 × 12 px

Or we can write a script to do rotating, as shown below:

```
# pyrotate: Python program to rotate a .bmp image.
# Written by Alan Parkin 2017.

from PIL import Image
from math import radians, cos, sin
import os, sys
```

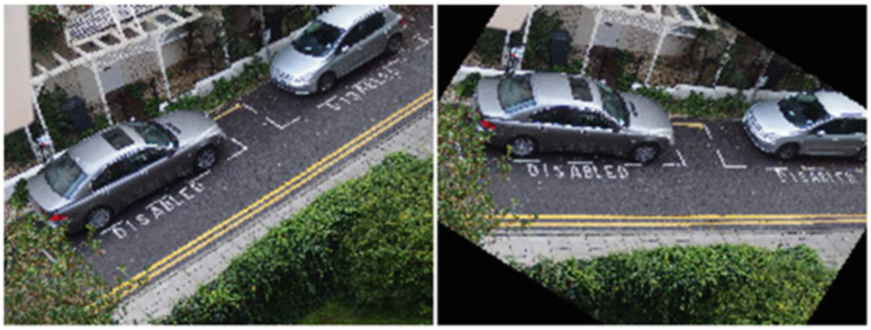

Fig. 5.9 Rotating a camera image. **a** Source image. **b** Rotated so that the double yellow line is horizontal

```
# Get source image and show it.
# Enter any .bmp filemame in working directory.
imagefilename = raw_input("Image filename? ")
#Open it, show it, and print W H and E.
im = Image.open(imagefilename)
#im.show()
width = im.size[0]
height = im.size[1]
extent = width * height
print "width W", width
print "height H", height
print "extent E", extent

# Enter degrees to rotate counter-clockwise about image centre.
deg = raw_input("Enter degrees to rotate counter-clockwise ? ")
filtercode = raw_input("Enter 0 for no interpolation, 2 for bicubic, 3 for lanczos ? ")
clipexpand = raw_input("Enter 0 to clip or 1 to expand the extent ? ")
degint = int(deg)
fico = int(filtercode)
clex = int(clipexpand)

# Do rotate and show. Save image under new filemame in show.
imro = im.rotate(degint, fico, clex)
imro.show()
```

5.8 Shearing

Shearing (also called skewing) slants a rectangular image to a parallelogram, away from the X-axis, or the Y-axis, or both. Figure 5.10 shows a simple image sheared away from the Y-axis, the X-axis and both axes, without and with interpolation. Shearing takes part of an image outside its original rectangular frame: in these examples, the new frame contains the whole rotated image. Figure 5.11 shows a camera image sheared.

We can do shearing in Microsoft Paint, where the options are Home - Image - Select For example, Fig. 5.10- Select all - Resize and Skew - Skew horizontal/vertical degrees - File - Save-as (New filename and extension).

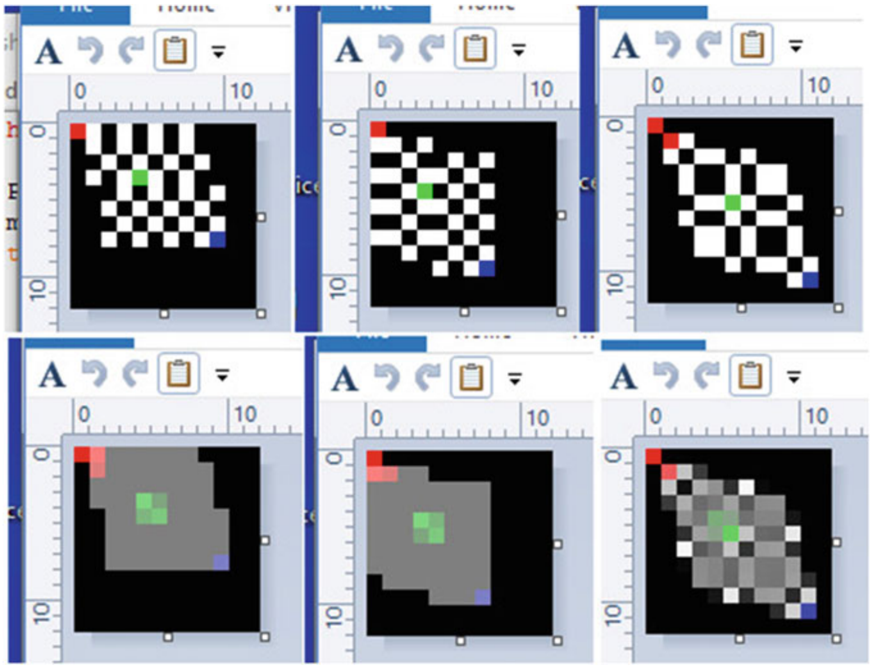

Fig. 5.10 Shearing a simple image. Top row, without interpolation: **a** Simple image 8 × 8 px sheared away from the Y-axis by 15 degrees. **b** Sheared away from the X-axis by 15 degrees. **c** Sheared away from both axes by 15 degrees. Bottom row: **d** As **a** with interpolation. **e** As **b** with interpolation. **f** As **c** with interpolation. The new frames are all 12 × 12 px

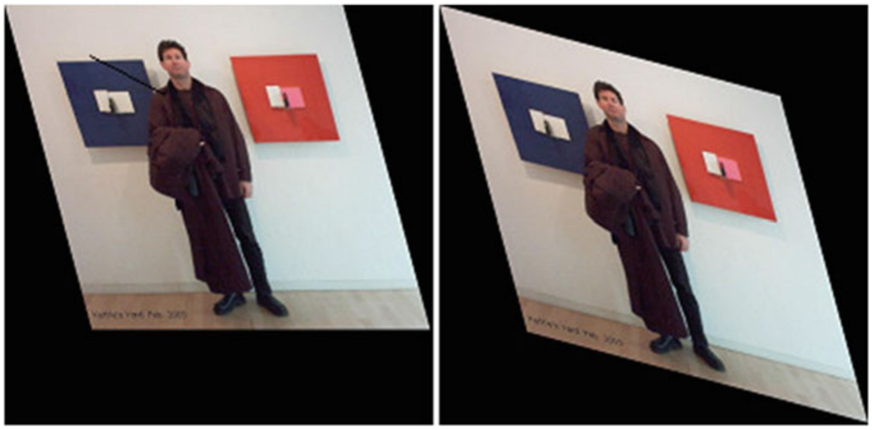

Fig. 5.11 Shearing a camera image. **a** Camera image sheared away from the Y-axis by 15 degrees. **b** Sheared away from both axes by 15 degrees

Or we can write a script to do shearing, as shown below:

```
# pyshear: Python program to shear a .bmp image.
# Written by Alan Parkin 2017.

from PIL import Image
from math import radians, tan
import os, sys

# Get source image and show it.
# Enter any .bmp filemame in working directory.
imagefilename = raw_input("Image filename? ")
#Open it, show it, and print W H and E.
im = Image.open(imagefilename)
#im.show()
width = im.size[0]
height = im.size[1]
extent = width * height
print "width W", width
print "height H", height
print "extent E", extent

# Enter degrees to shear from y-axis and x-axis.
xdeg = raw_input(" + degrees to shear left, - right, from y-axis ? ")
xdegint = int(xdeg)
xradangle = radians(xdegint)
xtanangle = tan(xradangle)
ydeg = raw_input("+ degrees to shear up, - down, from x-axis ? ")
ydegint = int(ydeg)
yradangle = radians(ydegint)
ytanangle = tan(yradangle)
interpol = raw_input("0 for no interpolation, 2 for bicubic ? ")
interp =int(interpol)
newwidth = int(width * 1.5)
newheight = int(height * 2.5)

# Do shear and show. Save under new filename in show.
imshear = im.transform((newwidth,newheight),0,(1,xtanangle,0,ytanangle,1,0),interp)
imshear.show()
```

5.9 Inverting

Inverting reflects an image in a fixed circle [5]. Given a circle with centre C and radius K, inverting moves each pixel Q of an image inside the circle to a new location Q' outside the circle, and each Q' to Q, so that

$$CQ * CQ' = K^2 \tag{5.1}$$

as shown in Fig. 5.12. Centre C remains fixed; parallel straight lines become circles through C; angles are preserved but sense is reversed. Inverting an image produces a great range of results according to the placing and radius of the circle, and is well worth careful investigation. For example, Fig. 5.13 shows a chequer image with five inversions about various circles, and a camera image with two inversions.

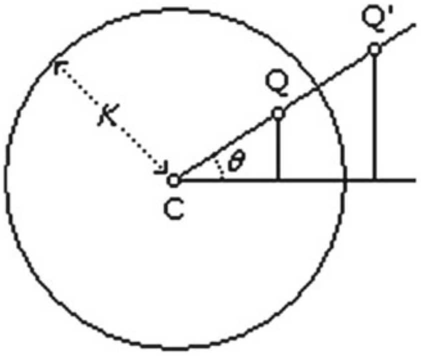

Fig. 5.12 The geometry of inverting. CQ * CQ' = K^2

Inversion is not offered by the ordinary image editors. We can write a script to do inversions, as shown below:

```
# pyinvert: Invert .bmp image in circle.
# Written by Alan Parkin 2017.

from PIL import Image
from math import tan, atan, cos, sin
import os, sys

# Get source image and show it: enter any .bmp
# filemame in working directory.
imagefilename = raw_input("Image filename? ")
#Open it, show it, and print W H and E.
im = Image.open(imagefilename)
im.show()
width = im.size[0]
height = im.size[1]
extent = width * height
print "SOURCE IMAGE"
print "width W", width
print "height H", height
print "extent E", extent
# Call PIL load to get source image as xy array.
souxy = im.load()

# Create output xy array.
outim = Image.new("RGB", (width,height), "white")
outxy = outim.load()

# Enter circle centre and radius.
x0str = raw_input("Enter circle centre x coord? ")
y0str = raw_input("Enter circle centre y coord? ")
kstr = raw_input("Enter circle radius in px? ")
x0 = float(x0str)
y0 = float(y0str)
k = float(kstr)
print "Centre x0", x0
print "Centre y0", y0
print "Radius k", k

# for each source pixel P find inverse pixel Q.
for y in range(height):
    for x in range(width):
        a = x - x0
```

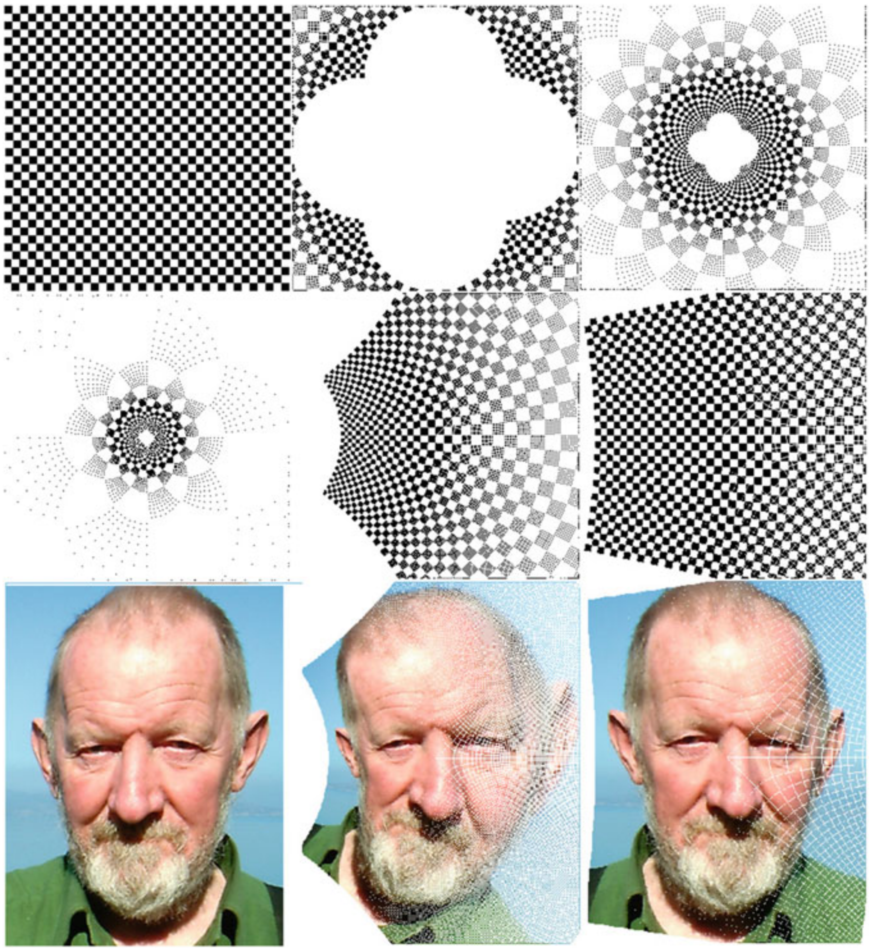
Fig. 5.13 Inverting an image. Top row: **a** Source image 288 × 288 px. **b** C = (144, 144), K = 144. **c** C = (144, 144), K = 72. Middle row: **d** C = (144, 144), K = 36. **e** C = (−144, 144), K = 288. **f** C = (−288, 144), K = 576. Bottom row: **g** Camera source image 375 × 480 px. **h** C = (−375, 240), K = 555 (compare **e** above). **i** C = (−3000, 240), K = 3187 (compare **f** above)

```
b = a * a
c = y - y0
d = c * c
e = b + d
f = (k * k) * a
if e == 0:
    e = 0.0001
g = f / e
xinv = x0 + g
h = (k * k) * c
i = h / e
yinv = y0 + i
if xinv < 0:
    xinv = 0
```

```
        if xinv > width - 1:
            xinv = width - 1

        if yinv < 0:
            yinv = 0

        if yinv > height - 1:
            yinv = height - 1

        # Set pixel Q with rgb of pixel P.
        outxy[xinv, yinv] = souxy[x, y]

outim.show()
```

References

1. Microsoft paint. https://en.wikipedia.org/wiki/Microsoft_Paint
2. Parkin A (2016) Digital imaging primer. Springer, Heidelberg
3. Fredrik Lundh effbot. http://effbot.org
4. Digital camera. https://en.wikipedia.org/wiki/Digital_camera
5. Morley F, Morley F V (1933) Inversive Geometry. Ginn, Boston MA. Dover reprint: http://store.doverpublications.com/0486493393.html