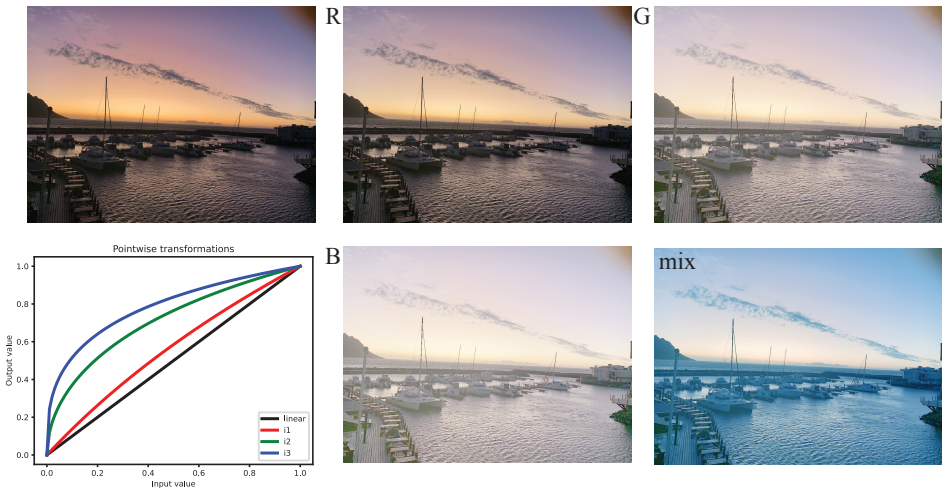


FIGURE 3.1: *A number of pointwise image transformations applied to the image on the* **top left**. *The* **bottom left** *shows plots of the function applied; on the* **right**, *results of applying these functions to that image. These transformations tend to spread out the dark values, and squash the brighter values.* **R**, **G** *and* **B** *respectively show the red, green and blue functions applied to each of the color channels of the image.* **Mix** *shows the result of applying the red function to the red channel, the green function to the green channel, and the blue function to the blue channel. Darker pixels tend to shift to the blue in the mix result, and brighter pixels have a less pronounced color shift.* Image credit: *Figure shows my photograph of a sunset at Gordon's Bay.*

CHAPTER 3

# Geometric Image Transformations

## 3.1 POINTWISE IMAGE TRANSFORMATIONS

Linear image sensors present problems. The *dynamic range* (ratio of largest value to smallest value) of spectral energy fields can be startlingly large (1e6: 1 is often cited). Simple consumer cameras report 8 bits (256 levels) of intensity per channel. A picture from a linear camera that reports 8 bits per channel will look strange, because even relatively simple scenes have a higher dynamic range than 255. One can build cameras that can report significantly higher dynamic ranges, but this takes work (Chapter 41.2). If the camera has a linear response and a dynamic range of 255, either a lot of the image will be too dark to be resolved, or much of the image will be at the highest value, or both will happen. This is usually fixed by ensuring that the number digitized by the camera *isn't* linearly related to

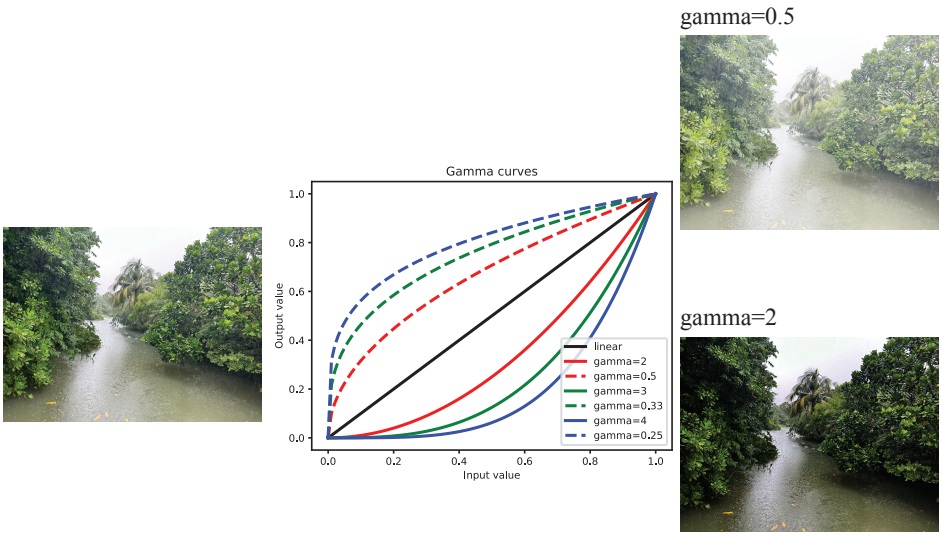


FIGURE 3.2: *Many imaging and rendering devices have a response that is a power of the input, so that output* $= C input^{\gamma}$*, where* $\gamma$ *is a parameter of the device. One can simulate this effect by applying a transform like those shown in the* **center** *(curves for several values of* $\gamma$*). Note that you can remove the effect of such a transform* – gamma correct *the image – by applying another such transform with an appropriately chosen* $\gamma$*. The image on the* **left** *is transformed to the two examples on the* **right** *with different* $\gamma$ *values.* Image credit: *Figure shows my photograph of a river in Singapore.*

brightness. Internal electronics ensures that the *camera response function* mapping the intensity arriving at the sensor to the reported pixel value looks something like Figure **??**. This increases the response to dark values, and reduces it to light values, so that the overall distribution of pixel values is familiar. Typically, the function used approximates the response of film (which isn't linear) because people are familiar with that. A camera response function is one example of a *pointwise image transformation*.

Most such transformations occur *after* the image has been digitized. You take the array of pixels and apply some function to each pixel value. Simple, but useful, examples include: forming a negative (map $x$ to $1-x$); contrast adjustment (choose a function that makes dark pixels darker and light pixels lighter, Figure 3.1); and gamma correction (using a function that corrects for a quirk of image encoding, Figure 3.2).

## 3.2 GEOMETRIC TRANSFORMATIONS

There are a number of important and useful geometric transformations of the plane that can be applied to images. Image transformations are implemented in the same way as subsampling: by scanning the pixels of the target and modifying them using interpolates of pixels from the source. This means it is important

that transformations are invertible. Adopt the convention that a point $\mathbf{x} = (x, y)$ is mapped by a transformation to the point $\mathbf{u} = (u, v) = (u(x, y), v(x, y))$, and $\mathbf{u} = (u, v)$ is mapped to $\mathbf{x} = (x, y)$ by the inverse. In vector notation, $\mathbf{x}$ is mapped to $\mathbf{u}$, and so on. Write $\mathbf{A}$ for a $2 \times 2$ matrix, whose $i$, $j$'th component is $a_{ij}$.

**Translation** maps the point $(x, y)$ to the point $(u, v) = (u(x, y), v(x, y)) = (x + t_x, y + t_y)$ for two constants $t_x$ and $t_y$. Here $(x, y) = (u - t_x, v - t_y)$. In vector notation, $\mathbf{u} = \mathbf{x}+\mathbf{t}$ and $\mathbf{x} = \mathbf{u}-\mathbf{t}$. Translation preserves lengths and angles. Choose two points $\mathbf{x}_1$ and $\mathbf{x}_2$. The squared distance from $\mathbf{x}_1$ to $\mathbf{x}_2$ is $(\mathbf{x}_1 - \mathbf{x}_2)^T(\mathbf{x}_1 - \mathbf{x}_2)$; but for a translation $(\mathbf{u}_1 - \mathbf{u}_2) = (\mathbf{x}_1 - \mathbf{x}_2)$. A similar argument shows that angles are preserved (**exercises** ).

**Rotation** takes the point $(x, y)$ to the point $(u, v) = (u(x, y), v(x, y)) = x\cos\theta - y\sin\theta, x\sin\theta + y\cos\theta$. Here $\theta$ is the angle of rotation, rotation is anticlockwise, and $(x, y) = u\cos\theta + v\sin\theta, -u\sin\theta + v\cos\theta$. Write $\mathcal{R}$ for a rotation matrix (a matrix where $\mathcal{R}^T\mathcal{R} = \mathcal{I}$ and $\det(\mathcal{R}) = 1$); then $\mathbf{u} = \mathcal{R}\mathbf{x}$ and $\mathbf{x} = \mathcal{R}^{-1}\mathbf{u} = \mathcal{R}^T\mathbf{u}$. Rotation preserves lengths and angles. Choose two points $\mathbf{x}_1$ and $\mathbf{x}_2$. The squared distance from $\mathbf{x}_1$ to $\mathbf{x}_2$ is $(\mathbf{x}_1 - \mathbf{x}_2)^T(\mathbf{x}_1 - \mathbf{x}_2)$; but for a rotation $(\mathbf{u}_1 - \mathbf{u}_2) = \mathcal{R}(\mathbf{x}_1 - \mathbf{x}_2)$ and $\mathcal{R}^T\mathcal{R} = \mathcal{I}$. A similar argument shows that angles are preserved (**exercises** ).

**A Euclidean transformation** is a rotation and translation, so $(u(x, y), v(x, y)) = (x\cos\theta - y\sin\theta + t_x, v(x, y) = x\sin\theta + y\cos\theta + t_y)$. Euclidean transformations preserve lengths and angles (and so areas) and are sometimes referred to as rigid body transformations. Here $(x, y) = ((u - t_x)\cos\theta + (v - t_y)\sin\theta, -(u - t_x)\sin\theta + (v - t_y)\cos\theta)$. In vector notation, $\mathbf{u} = \mathcal{R}\mathbf{x} + \mathbf{t}$ and $\mathbf{x} = \mathcal{R}^{-1}(\mathbf{u} - \mathbf{t}) = \mathcal{R}^T(\mathbf{u} - \mathbf{t})$. Euclidean transformations preserve lengths and angles (you can think of a Euclidean transformation as a rotation followed by a translation).

**Uniform scaling** where $(u, v) = (sx, sy)$ for $s > 0$. Here $(x, y) = (1/su, 1/sv)$. In vector notation, $\mathbf{u} = s\mathbf{x}$ and $\mathbf{x} = (1/s)\mathbf{u}$. Uniform scaling preserves angles, but not lengths (**exercises** ).

**Non-uniform scaling** where $(u, v) = (sx, ty)$ for $s$ and $t$ both positive, and so $(x, y) = (1/su, 1/tv)$. Write $\mathrm{diag}(s, t)$ for the matrix with $s$ and $t$ on the diagonal. In vector notation, $\mathbf{u} = \mathrm{diag}(s, t)\mathbf{x}$ and $\mathbf{x} = \mathrm{diag}(1/s, 1/t)\mathbf{u}$. Non-uniform scaling will usually change both lengths and angles.

**Affine transformations** are better written in vector notation. Write $\mathcal{A}$ for a $2 \times 2$ matrix which is invertible, and $\mathbf{t}$ for some constant vector. Here $\mathbf{u} = \mathcal{A}\mathbf{x} + \mathbf{t}$ and $\mathbf{x} = \mathcal{A}^{-1}(\mathbf{u} - \mathbf{t})$. Affine transformations will usually change both lengths and angles.

**Projective transformations** involve quite inefficient notation if one does not know homogenous coordinates (Section **??**), and writing them in vector form is clumsy. Write $p_{ij}$ for the $i$, $j$'th component of a $3 \times 3$ matrix $\mathcal{P}$ that is invertible. Then

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} \frac{p_{11}x+p_{12}y+p_{13}}{p_{31}x+p_{32}y+p_{33}} \\ \frac{p_{21}x+p_{22}y+p_{23}}{p_{31}x+p_{32}y+p_{33}} \end{bmatrix}.$$

The inverse transformation is obtained by applying the inverse of $\mathcal{P}$ to $\mathbf{u}$ according to the recipe above. Notice that all the classes of transformation described are a case of a projective transformation (**exercises** ). For a vector representation,

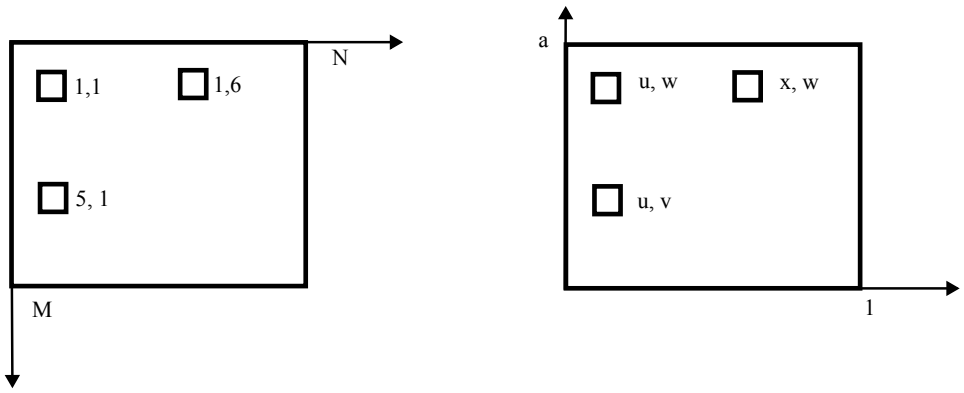


FIGURE 3.3: *The most common coordinate system for images, on the* **left**. *The origin is at the top left corner, and we count in pixels. This is an $M \times N$ image. I will use the convention $\mathcal{I}_{ij}$ for points in this coordinate system, so the top right pixel is $\mathcal{I}_{1M}$. It is usual for pixel locations to be indexed starting at 1 (so $1 \leq i \leq M$ and $1 \leq j \leq N$). In some environments (notoriously, Python), the index starts at 0. Keep track of this point, or you will lose some pixels. On the* **right**, *the origin is at the bottom left, and the coordinate axes are more familiar. It is a good idea to use a range from $0 - 1$ (rather than $0 - M$) in this coordinate system, but if the image is not square one direction will run from 0 to a. Converting from one coordinate system to the other is straightforward,* but *not being consistent about the coordinate system you are working in is an important source of simple, annoying errors. I will always work in the coordinate system shown on the* **left**.

write

$$\mathcal{P} = \begin{bmatrix} \mathbf{p}_1^T & p_{13} \\ \mathbf{p}_2^T & p_{23} \\ \mathbf{p}_3^T & p_{33} \end{bmatrix}$$

for a $3 \times 3$ array with inverse $\mathcal{Q}$. Then

$$\mathbf{u} = \begin{bmatrix} \frac{\mathbf{p}_1^T\mathbf{x}+p_{13}}{\mathbf{p}_3^T\mathbf{x}+p_{33}} \\ \frac{\mathbf{p}_2^T\mathbf{x}+p_{23}}{\mathbf{p}_3^T\mathbf{x}+p_{33}} \end{bmatrix} \text{ and } \mathbf{x} = \begin{bmatrix} \frac{\mathbf{q}_1^T\mathbf{u}+q_{13}}{\mathbf{q}_3^T\mathbf{u}+q_{33}} \\ \frac{\mathbf{q}_2^T\mathbf{u}+q_{23}}{\mathbf{q}_3^T\mathbf{u}+q_{33}} \end{bmatrix}$$

Notice also that if $\mathcal{P} = \lambda\mathcal{Q}$ for some $\lambda \neq 0$, then $\mathcal{P}$ and $\mathcal{Q}$ implement the same projective transformation.

## 3.3 GEOMETRIC TRANSFORMATIONS OF IMAGES

Write $\mathcal{S}$ for a source image and $\mathcal{T}$ for a target image. The pixel values for the source image $\mathcal{S}_{ij}$ are known for $i$, $j$ integer points where $1 \leq i \leq s_1$ and $1 \leq j \leq s_2$. The target image has pixel locations on the $i$, $j$ integer points where $1 \leq j \leq t_1$ and $1 \leq j \leq t_2$. As in Section 2.2, the correct procedure is to scan the pixels of $\mathcal{T}$ and then modify them using interpolates of pixels from $\mathcal{S}$. This means it is important that transformations are invertible, and both $(u(x, y), v(x, y))$ and $(x(u, v), y(u, v))$ are known.

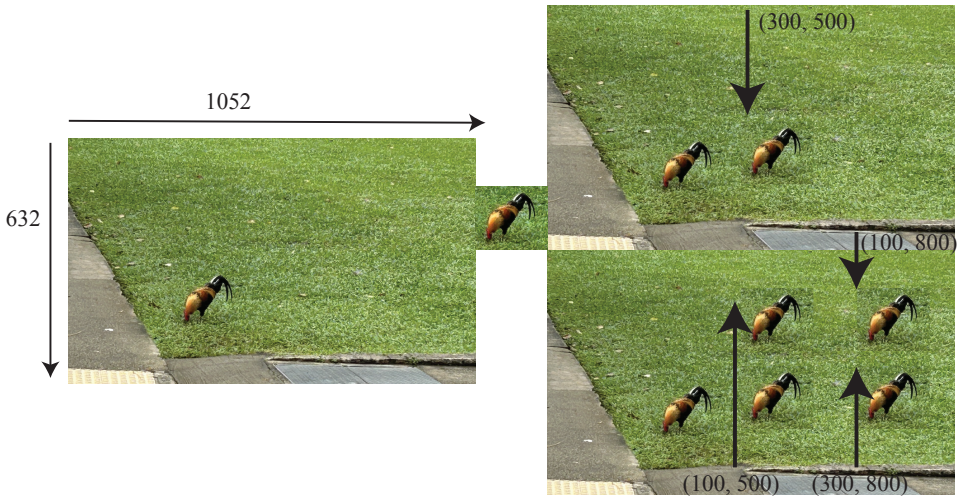


FIGURE 3.4: *The chicken in the* **left image** *has been cropped to yield the* **center image** *(which is* $162 \times 187$*), then translated and pasted to various points in the left image, to yield the images on the* **right**. *Note the choice of coordinate system strongly affects the value of translation. The chicken's origin is at the top left hand corner, yielding the translations shown in the overlay (left image scales are shown for reference). You should check you agree the translations indicated yield the chickens shown.* Image credit: *Figure shows my photograph of jungle fowl in Singapore.*

**Coordinate systems:** The most common convention for image coordinate systems is strange at first glance. This coordinate system is shown in Figure 3.3 on the left. The inversion of the $y$-axis and of the order of coordinates is an annoying leftover from the way matrices are indexed. It is quite usual to use this coordinate system, and I will do so in what follows. Readers should be aware that there are a variety of alternative conventions, and the choice of coordinate system has a significant effect on the expressions used to describe image transformations.

**Notation:** I will need to refer to image values both at sample points – which I will write $\mathcal{S}_{ij}$ – and at points that are possibly not sample points – $\mathcal{S}(x, y)$. For points that are not sample points, care is required. If $1 \leq x \leq s_1$ and $1 \leq y \leq s_2$, then $\mathcal{S}(x, y)$ can be obtained by interpolation; otherwise, there is no meaning to the expression (and this should never occur). Transformations always take a source image $\mathcal{S}$ which is $s_M \times s_N$ to a target image $\mathcal{T}$ which is $t_M \times t_N$.

### 3.3.1 Cropping, Translation, Pasting and Blending

**Cropping** creates a smaller target image from a source image. One specifies a crop window in the $s_M \times s_N$ source image by $1 \leq x_n, x_x \leq M$ and $1 \leq y_n, y_x \leq N$. Here the vertices of the window are integers, and there is no interpolation. The target image is an $(x_x - x_n) \times (y_x - y_n)$ image. For $1 \leq i \leq x_x - x_n$ and $1 \leq j \leq y_n - y_x$,

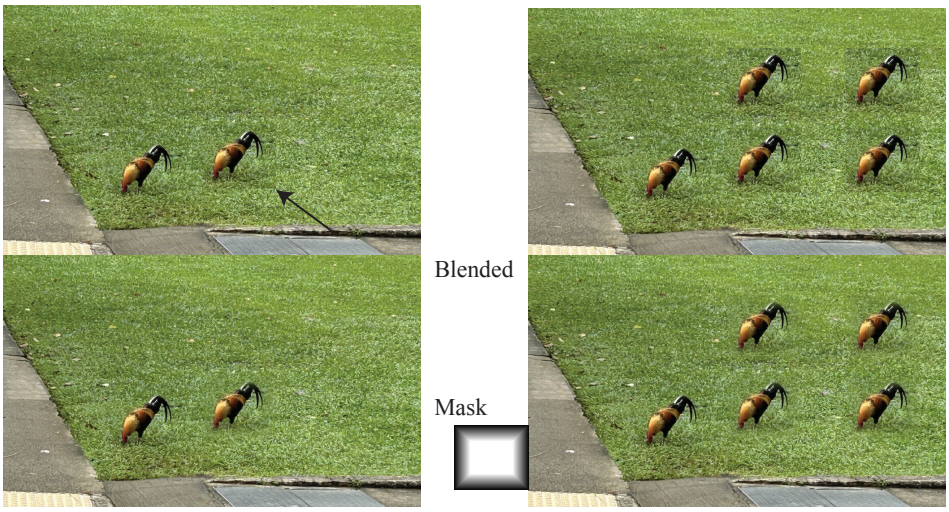


FIGURE 3.5: *The chickens of Figure 3.4 are simply pasted in the* **top row** *(as in that figure, reproduced here for comparison; the arrow on the* **left** *shows a problem with pasting not identified in that figure). In the* **bottom row**, *the chickens have been blended using the blending mask shown. Note the pasting is much less obvious.* Image credit: *Figure shows my photograph of jungle fowl in Singapore.*

we have

$$\mathcal{T}_{ij} = \mathcal{S}_{i-x_n, j-y_n}$$

For more complicated transformations of an image, the range of pixels in the target image has an important effect. If the transformation you apply to the source image places the result outside the window of the target image, you will not see anything. The range of pixels in the source image is also important. For pixels outside the range of the image, the value of an image isn't known and can't be interpolated.

**Translation** as a geometric transformation moves the location of each source pixel value. Write $c_x, c_y$ for the translation – equivalently, the top left corner of the source image will go to $c_x + 1, c_y + 1$ in the target image – to get

$$\mathcal{T}_{ij} \leftarrow \begin{cases} \mathcal{S}_{i-c_x, j-c_y} & \text{If } 1 \leq i - c_x \leq s_M \text{ and } 1 \leq j - c_y \leq s_N \\ \mathcal{T}_{ij} & \text{Otherwise} \end{cases}$$

where $1 \leq i \leq t_M$ and $1 \leq j \leq t_M$. You should check what happens for all values of $c_x, c_y$, and that the result is always $t_M \times t_N$ (**exercises** ). There are really two cases here. If you start with $\mathcal{T}$ that contains only zeros, then translation creates an image which has $\mathcal{S}$ at some offset, possibly cropped to fit $\mathcal{T}$. If $\mathcal{T}$ is a more conventional image, then translation will *paste* $\mathcal{S}$ into $\mathcal{T}$ at the given location.

Translation can yield quite convincing composite images (Figure **??**). However, close scrutiny of the multi-chicken image shows boundaries of the window where the translated chicken was pasted. These boundaries can be spotted because

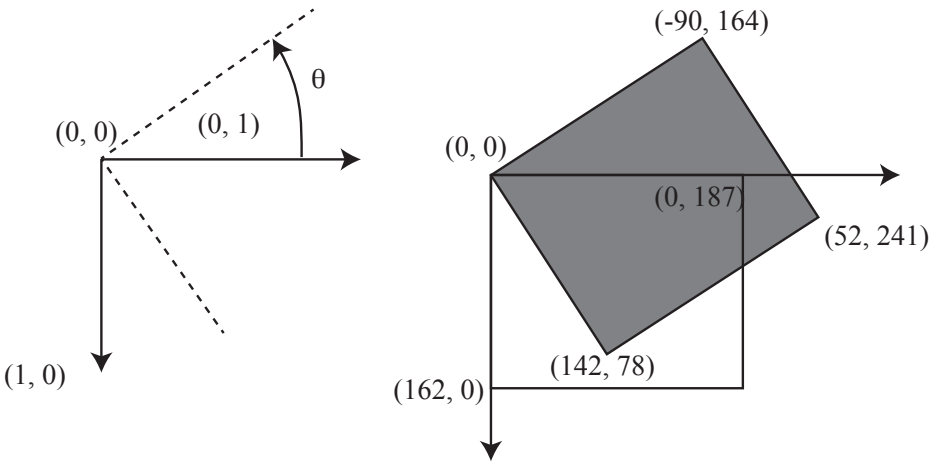


FIGURE 3.6: **Left** *shows the image coordinate system for reference, together with the result of rotating coordinate axes clockwise by $\theta$ (which in this example is 0.5 radians, about $30^0$). Notice that a significant chunk of the source image ends up with negative coordinates.* **Right** *shows the original source rectangle from the cropped chicken of 3.4 (recall this is $162 \times 187$) as an open rectangle, and the rotated source rectangle in gray. The target image is then set up to enclose the whole result (implicitly translating the rotated source image) and pixels are then scanned into the target.*

the grass on the left of the chicken is a little darker than the grass on which it was placed. Notice that translation can result in the source image ending up outside the window of the target image (**exercises** ).

**Blending** can help to alleviate problems with boundaries. Blending requires a mask the same size as the source image, with values in the range $[0, 1]$. Write $\mathcal{M}_{ij}$ for the $i$, $j$'th mask pixel. Then

$$\mathcal{T}_{ij} \leftarrow \begin{cases} \mathcal{M}_{ij}\mathcal{S}_{ij} + (1 - \mathcal{M}_{ij})\mathcal{T}_{ij} & \text{If } 1 \leq x \leq s_1 \text{ and } 1 \leq y \leq s_2 \\ \mathcal{T}_{ij} & \text{Otherwise} \end{cases}$$

Of course, one can translate and blend rather than translating and pasting (**exercises** ). As Figure 3.5 indicates, blending can suppress problems at boundaries fairly effectively. The choice of blending mask can get interesting (**exercises** ).

### 3.3.2 Rotation, Scaling, Affine and Projective Transformations

**Rotation** rotates the source image, then pastes it into the target image. Rotation presents a problem. Imagine rotating the grid of positive integer points by $180^0$ anti-clockwise around the origin – all the grid points are still integer, but they are now all negative. This means that you can rotate an image and have an empty result (because all the pixels in the rotated image are outside the span of the target image). Usually, this is fixed by translating the image as well as rotating

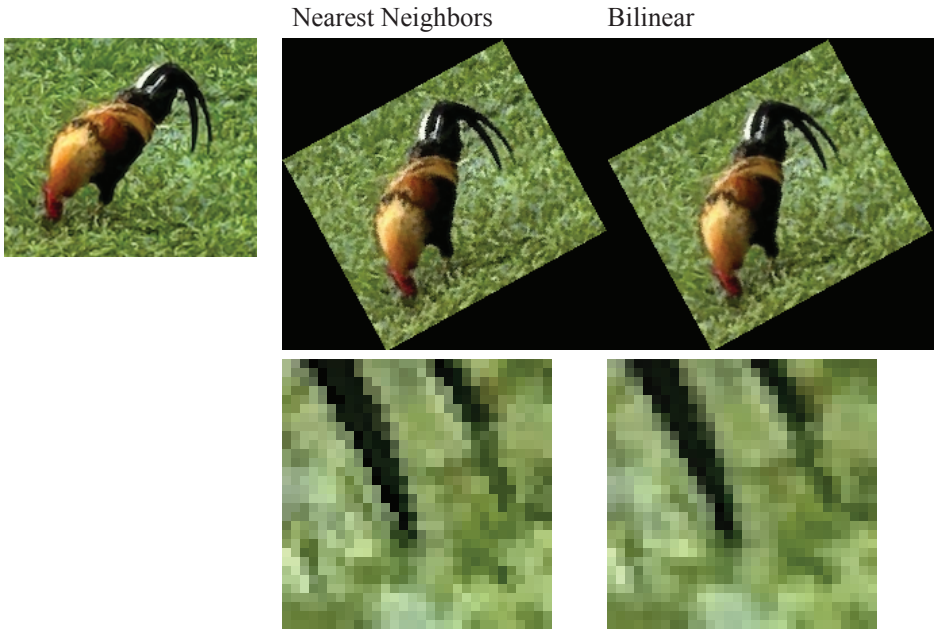


FIGURE 3.7: *The chicken of Figure 3.4, rotated by 0.5 radians as in Figure 3.6, showing the effect of different choices of interpolation. I have zoomed in on a section of the tail feathers to make the difference more apparent.* Image credit: *Figure shows my photograph of jungle fowl in Singapore.*

it. Another problem is caused by the fact that the rotated image usually spans more pixels in the coordinate directions than the source image (Figure 3.6). As a result, several reasonable choices of translation are possible, and most APIs support many different choice. One natural choice creates a target image whose horizontal and vertical spans are big enough to contain the rotated image, and translates the source image to be centered in that target. Informally, one determines how big the rotated image will be; constructs a target image that will span that; then scans the target, picking up pixels from the source image using the inverse transformation. In detail, to construct this result for a rotation $\mathcal{R}$:

- Write $\mathbf{x}_1 = (1, 1)^T$, $\mathbf{x}_2 = (s_1, 1)^T$, $\mathbf{x}_3 = (s_1, s_2)^T$ and $\mathbf{x}_4 = (1, s_2)^T$ for the four vertices of the source image.

- Compute $\mathbf{u}_i = \mathcal{R}\mathbf{x}_i$ for the result of rotating these vertices by $\mathcal{R}$. Now write $u_n$, $u_x$ for the smallest (resp. largest) value of the first component of these points; similarly, $v_n$, $v_x$ for the smallest (resp. largest) value of the second component of these points.

- $\mathcal{T}$ is now a ceil$(u_x - u_n) + 1 \times$ ceil$(v_x - v_n) + 1$ image. Write $\rho_{mn}$ for the $m$, $n$'th component of $\mathcal{R}^{-1}$; $x(i, j) = \rho_{11}(i - 1 - u_n) + \rho_{12}(j - 1 - v_n)$; and

$y(i,j) = \rho_{21}(i - 1 - u_n) + \rho_{22}(j - 1 - v_n)$ Now

$$\mathcal{T}_{ij} \leftarrow \begin{cases} \mathcal{S}(x(i,j), y(i,j)) & \text{If } 1 \leq x \leq s_1 \text{ and } 1 \leq y \leq s_2 \\ \mathcal{T}_{ij} & \text{otherwise} \end{cases}$$

The choice of interpolate has a real effect (Figure 3.7).

**Uniform scaling** involves two cases. Section 2.2 dealt with the case $s > 1$. Uniform scaling for $s < 1$ is downsampling, and has been dealt with in some detail in Section 2.3.

**Non-uniform scaling** presents a combination of problems. If, say $s > 1$ and $t < 1$, we are upsampling in one direction and downsampling in the other. If $t$ is relatively close to 1 (so there is not much downsampling), it is usually sufficient to ignore the upsampling, apply a gaussian smoother to the source, then resample with interpolation. If the downsampling is very aggressive, it may be better to smooth in one direction only, which is beyond scope.

**Affine** transformations follow the recipe for the rotation. One determines how big the transformed image will be; constructs a target image that will span that; then scans the target, picking up pixels from the source image using the inverse transformation. Finding the size of the transformed image is straightforward (**exercises** ). However, an affine transformation may involve a component of scaling, which might be non-uniform. One way to see this is to apply a singular value decomposition to $\mathcal{A}$ which will yield

$$\mathcal{A} = \mathcal{U}\Sigma\mathcal{V}^T$$

where $\mathcal{U}$ and $\mathcal{V}$ are rotations. But $\Sigma$ is diagonal, and may be non-uniform. As long as the values on the diagonal of $\Sigma$ are not too different, and the smallest is not too small, then one can apply a gaussian smoother to the source, and resample with interpolation. A robust smoothing strategy is firmly beyond scope, however.

**Projective transformations** follow the same general recipe as rotations, but smoothing is now tricky. One determines how big the transformed image will be; constructs a target image that will span that; then scans the target, picking up pixels from the source image using the inverse transformation. For a general projective transformation, there might be singular points, caused by a divide-by-zero. For geometric reasons, these projective transformations do not arise in cases interesting to us (Section 41.2), and should be seen as evidence of a problem elsewhere. Nasty smoothing problems occur because at some pixels a projective transformation may upsample an image and at different pixels downsample the image. For this effect, look at Figure 3.9 and consider what happens if the transform scales the image as well (the **exercises** do the details). It is relatively straightforward to predict at a given pixel whether downsampling is occuring, and the degree of downsampling (**exercises** ), meaning a gaussian pyramid is useful. At a pixel in the target image, predict which location in the source image will be used; estimate the degree of smoothing required; then look at the relevant layer of the gaussian pyramid. This strategy is sometimes referred to as *MIP-mapping*.

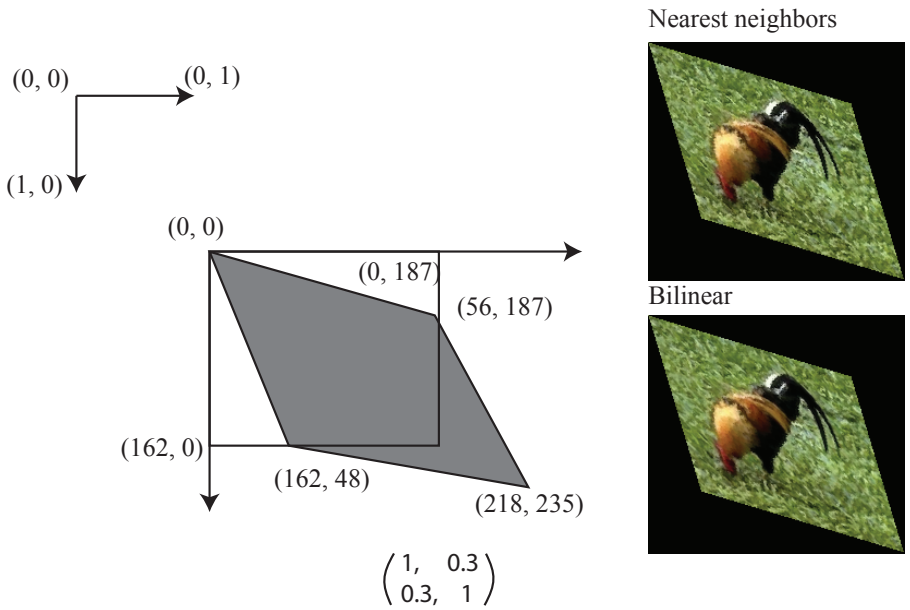


FIGURE 3.8: **Top left** *shows the image coordinate system for reference. In this coordinate system, the affine transformation whose matrix is shown at the* **bottom** *is applied to the original chicken crop of Figure 3.4 (recall this is* $162 \times 187$*; open rectangle). The gray diamond indicates the result. The target image is then set up to enclose the whole result, and pixels scanned into the target. In this case, the source image was not smoothed, because there is relatively little downsampling (the diamond is not much smaller than the open rectangle).* **Top right** *shows the result using nearest neighbors interpolation, and* **bottom right** *shows the result using bilinear interpolation. Look closely at the tail feathers to see the difference.*

## 3.4 APPLICATIONS

### 3.4.1 Aligning Color Separations

Simple geometric transformations can be extremely useful. One application comes from early color photography. Color photography is usually dated to the 1930's when it first became available to the public. In fact, James Clerk Maxwell described a method to capture a color photograph in an 1855 paper. The procedure likely looks straightforward to you: obtain three color filters, and take a picture of the scene through each of these filters. Capturing these *color separations* presented a number of technical challenges, and the first color photograph was taken by Thomas Sutton in 1861. Actually displaying pictures obtained like this was tricky. One had to pass red light through the red separation, green through the green, and blue through the blue, then ensure all three resulting images lay on top of one another on screen. Turning them into the image files we are familiar with is also tricky, because each layer of the separation is typically a bit offset from the others (the

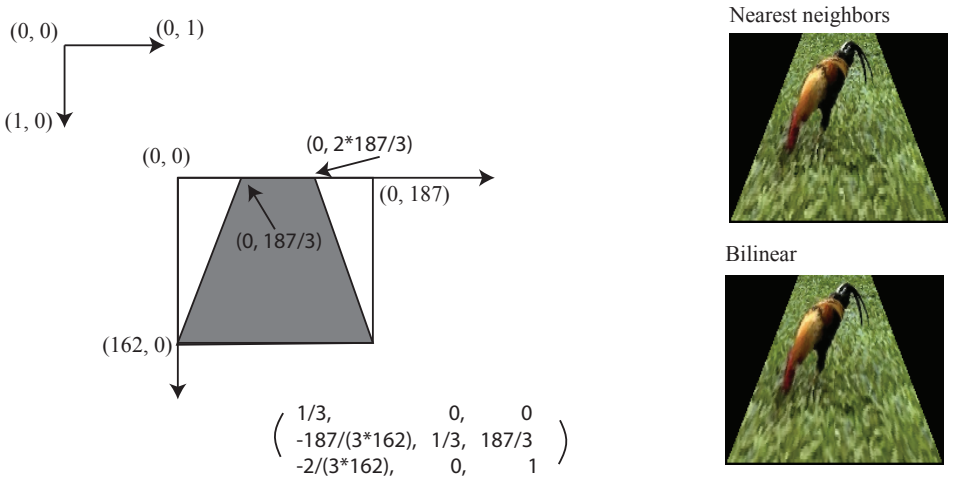


FIGURE 3.9: **Top left** *shows the image coordinate system for reference. In this coordinate system, the projective transformation whose matrix is shown at the* **bottom** *is applied to the original chicken crop of Figure 3.4 (recall this is* $162 \times 187$*; open rectangle). The gray region indicates the result. Note that the projective transformation has taken the rectangular source to a shape that is not even a parallelogram. The target image is then set up to enclose the whole result, and pixels scanned into the target. In this case, the source image was not smoothed, because there is relatively little downsampling (the gray region is not much smaller than the open rectangle).* **Top right** *shows the result using nearest neighbors interpolation, and* **bottom right** *shows the result using bilinear interpolation. Look closely at the tail feathers to see the difference.*

camera moved slightly between photographs), and each layer has aged and been damaged slightly differently.

Separations are *in register* if they lie over one another exactly and so form a color image. If they are out of register, objects will have slight, odd color halos. Early color separations tend not to be in register. A class assignment, now hallowed by tradition in computer vision, but likely to have originated with A. Efros in 2010, uses the pictures of Sergei Mikhailovich Prokudin-Gorskii (1863-1944). Prokudin-Gorskii traveled the Russian empire and took color photographs of many scenes. He left Russia in 1918. His negatives survived and ended up in the Library of Congress. A digitized version of the collection is available online. The assignment asks students to register the color separations for some of these images.

There is a natural strategy: write a function that is smallest when the G (respectively B) separation is in register with the R separation; now search for the best value of the cost function obtained by small translations of the G (respectively B) separation.

The search is easy when the separations are at relatively low resolution. The offsets will be relatively small (a few pixels or so). It is then practical to simply evaluate the cost function at a grid of translations, and choose the best (fussier

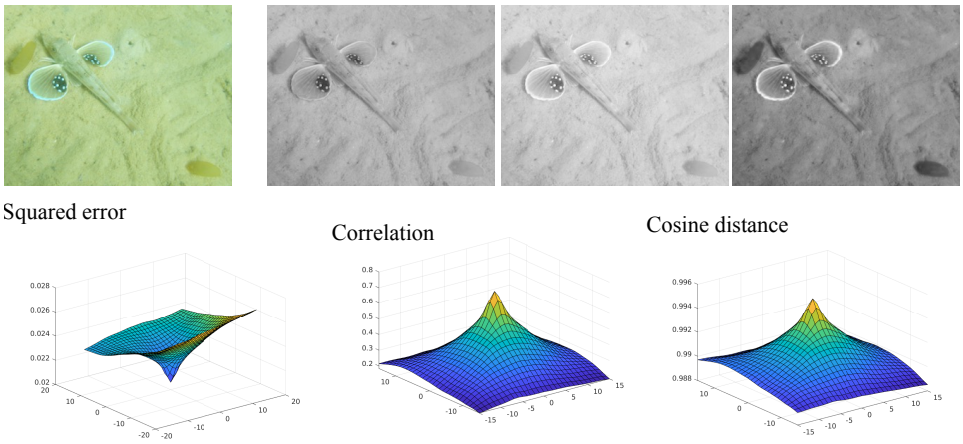


FIGURE 3.10: **Top left** *shows a Gurnard, flashing its pectoral fins in alarm.* **Top rest** *shows the color separations of this image (in red, green, blue order). The image is slightly blue-green (taken at about 5 meters depth, where water absorbs red light), and this shows as a darker red separation.* **Bottom** *shows how various cost functions react to registering red to blue. The correct alignment is at 0, 0 and the images are 257 by 323. Notice that: all the extrema are in the right place, but the correlation and cosine distance must be maximized, and the squared error minimized; the squared error changes relatively little from the best to the worst, because the blue image is rather unlike the red; both cosine distance and correlation are much more sensitive than SSD – they fall off much more quickly than the SSD rises.* Image credit: *Figure shows my photograph of a Gurnard, at Long Beach in Cape Town.*

readers might interpolate, exercises). The remaining issue is the cost function. Section 3.4.2 describes a number of possible cost functions.

This assignment requires care when one works with the high resolution version of the scans. These are quite big, and there can be moderately large offsets. Simply looking at each offset in turn will be hideously expensive (dealt with in Section 13.1.2).

### 3.4.2 Scoring an Overlap with a Cost Function

The *sum of squared differences* or *SSD* scores the similarity between the overlapping parts of two separations $\mathcal{R}$ and $\mathcal{B}$. Given an offset $m, n$, the SSD is

$$C_{\text{reg}}(m, n; \mathcal{R}, \mathcal{B}) = \frac{1}{N_o} \sum_{\text{overlap}} \left(\mathcal{R}_{ij} - \mathcal{G}_{i-m,j-n}\right)^2 .$$

Here overlap is the rectangle of pixel locations with meaningful values for both $\mathcal{R}$ and $\mathcal{G}$ and $N_o$ is the number of pixels in that rectangle. Notice that overlap and so $N_o$ change with $m$ and $n$, so we must compare overlaps of different sizes for different offsets. This means it is important that $C_{\text{reg}}$ is an average.

The SSD assumes that the images to be registered are very close to the same when they are aligned. But the separations do not agree exactly when they overlap – if they did, the image would be a monochrome image. It is useful to have alternative cost functions that (a) will tend to be minimized or maximized when the images are correctly registered and (b) change quite quickly when they are not.

Quite widely used alternatives are:

- The *cosine distance*, given by:

$$C_{cos}(m,n) = \sum_{\text{overlap}} \frac{(\mathcal{A}_{ij} * \mathcal{B}_{i-m,j-n})}{\sqrt{\sum_{\text{overlap}} \mathcal{A}_{ij}^2} \sqrt{\sum_{\text{overlap}} \mathcal{B}_{i-m,j-n}^2}}.$$

  Annoyingly, this cost function is largest when best, even though it's called a distance. Some authors subtract this distance from one (its largest value) to fix this.

- The *correlation coefficient*, given by:

$$C_{corr}(m,n) = \sum_{\text{overlap}} \frac{(\mathcal{A}_{ij} - \mu_A) * (\mathcal{B}_{i-m,j-n} - \mu_B)}{\sqrt{\sum_{\text{overlap}} \mathcal{A}_{ij}^2} \sqrt{\sum_{\text{overlap}} \mathcal{B}_{i-m,j-n}^2}}$$

$$\text{where } \mu_A = \frac{1}{N_O} \sum_{overlap} \mathcal{A}_{ij} \text{ and}$$

$$\text{where } \mu_B = \frac{1}{N_O} \sum_{overlap} \mathcal{B}_{ij}.$$

  This is big for the best alignment. Notice how this corrects for the mean of the overlap in each window.

Each is in the range $-1$ to 1, and neither scales with the size of the overlap neighborhood. Terminology in this area is severely confused. The cosine distance isn't a distance; it is sometimes referred to as *normalized correlation*; and sometimes as *correlation*. Several functions similar to correlation are referred to as correlation. Figure 3.10 shows how these cost functions behave when trying to register the red and blue separations of an image. These separations will be fairly similar, but not exactly the same.

### 3.4.3 Elementary Object Detection, or Find the Chicken

*Object detection* is the problem of determining whether an object appears in an image *and* where it is if it is there. There are a wide range of variants, explored in much greater detail in Chapter 41.2; differences hinge on how one interprets the word "object", an alarmingly rich question.

A very simple object detector can be built out of the mosaic procedure. Assume $\mathcal{A}$ is an image which might contain an object, and $\mathcal{B}$ is a *template* – an example image of the object to be detected. For every offset $m, n$ where $\mathcal{B}$ lies inside $\mathcal{A}$, compute the cost function and store values in an array (the *score array*). Notice that if the values are small, then at that offset, the overlapping bits of $\mathcal{B}$ and $\mathcal{A}$

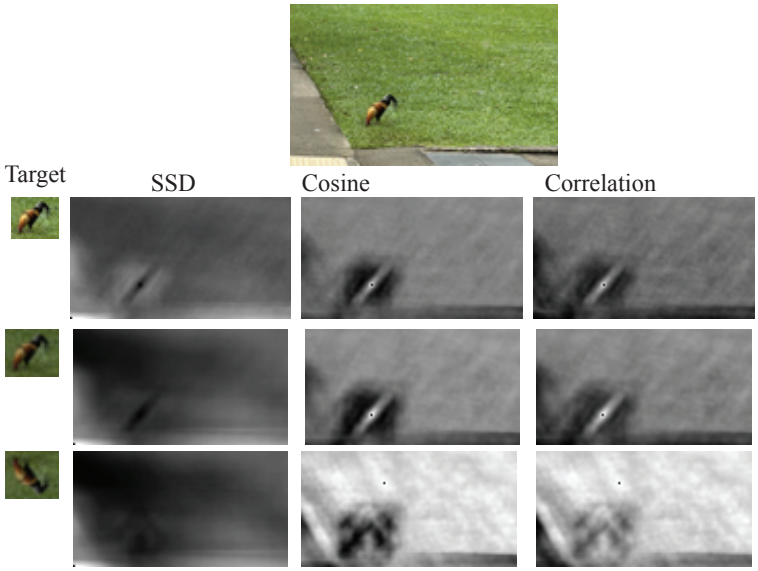


FIGURE 3.11: *Translation and an image matching cost function yield an elementary detector. Model the object – here, the chicken – using an image window (***leftmost column***), then translate this window to each location in the image (***top***) and compute the cost of the overlap. If the underlying image looks a lot like the chicken, you will get a good value of the cost function (***other columns***. For SSD, a good value is small – and so dark – for others it is large – and so light. This elementary detector has serious problems. In the* **second row***, the chicken template is darker than the original image, and so SSD matches are not particularly good. Cosine distance and correlation are less affected. But chickens don't stay in a fixed configuration, and if the chicken moves* **third row***, all scores fall off.* Image credit: *Figure shows my photograph of jungle fowl in Singapore.*

"look like" one another. If they "look like" one another sufficiently (test the cost function against a threshold), declare that the object is present. Figure 3.11 shows what the arrays look like for a variety of cost functions.

This detector will tend to overcount objects rather significantly. Shifting a template by one or two pixels will not tend to change the cost function by much. This means if the cost function is below threshold at $m, n$, it is likely to be below threshold at neighboring points in the score array, too. This could mean you find many instances of the object nearly on top of one another. A straightforward procedure called *non-maximum suppression* deals with this. Find the smallest below threshold value in the score array. Record an object present at that location, then suppress that location and all nearby values (nearby might mean, for example, all values in a $k \times k$ window centered on the current best value in score array) by setting all to a large value. Repeat this procedure until there are no more below threshold values in the score array.

There are other good reasons this isn't a good object detector. Look at Fig-

ure 3.11. The detector will only find chickens if they are in the same configuration as the template, and on a grass background, and with the same lighting. Some of this can be fixed with straightforward procedures. For some specialized applications, where very little computing is available, and where relatively few pixels lie on the object, a detector built like this can be useful, but outside these applications different procedures are used. A large family of modern detectors are built on this framework, with some crucial modifications: the cost function for evaluating the match between an image window and the concept "chicken" is much more sophisticated than just comparing image pixels with template pixels and the search procedure is more elaborate and more efficient (Chapter **??**).