Alan Parkin

Computing Colour Image Processing

Digital Colour Primer

Springer

Computing Colour Image Processing

Alan Parkin

Computing Colour Image Processing

Digital Colour Primer

Springer

Alan Parkin
London
UK

ISBN 978-3-319-74075-1 ISBN 978-3-319-74076-8 (eBook)
https://doi.org/10.1007/978-3-319-74076-8

Library of Congress Control Number: 2018933504

Printed on acid-free paper

This Springer imprint is published by the registered company Springer International Publishing AG part of Springer Nature
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

For Lily 1926–2008

Preface

This book is based on observations and opinions formed over many years of professional practice in graphic design, commercial data processing and amateur painting.

The first observation is that colour in digital imaging is a distinct field which shares some of the knowledge and traditions of the vast spread of other colour activities, and diverges from them in other ways.

The second observation is that sRGB colour standards, implemented in generally available equipment, are a suitable environment for further study.

The third observation is that the complete numerical representation of colour in any digital image brings radically new opportunities for processing colour by program.

The fourth observation is that some of the formal properties of colour schemes can be analysed and measured computationally, in ways not possible hitherto.

The fifth observation is that any digital image can be computationally brought to a norm of neutral balance: an idea with good traditional antecedents.

The first opinion is that the general principles of economy of materials for purposes, and fitness of means to ends, found in most traditional arts and crafts, should be applied in digital imaging.

The second opinion is that digital images are often created and used at levels of resolution of location and colour which are unnecessary and inconvenient. The specific character and charm of digital images lie at the threshold where pixel structure and colour gradation steps are just about visible at the intended viewing distance.

The third opinion is that commonly available commercial software for colour manipulation is inappropriately based on photographic conventions. Scripting in Python offers a better way to explore and develop digital colour.

The fourth opinion is that digital imaging is an interesting field for the exercise of curiosity, skill, luck, discrimination and taste, over and above its undoubted usefulness in business, technology and science. It can move towards art and connoisseurship.

The fifth opinion is that preparing a digital image, simple or complex, is like preparing food. It begins with growing and harvesting the ingredients (by GUI or program), or hunting and killing (by camera or scanner or download). It proceeds by peeling, skinning, chopping up and cooking the ingredients (by the transformations); and finally serving up the finished dish (as a display or printout).

This book is a second attempt to put these observations and opinions into practice. The first attempt (*Digital Imaging Primer*, Parkin A., Springer, 2016) used BASIC programming to explain and illustrate. This book uses Python scripting to explain and illustrate some 20 elementary tools. For serious use, these scripts can be freely improved and expanded, and can be wrapped into full GUI Tkinter applications.

Let us honour the universities, institutions, commercial enterprises and independent enthusiasts, many indicated in the chapter references, who have made digital imaging available to all. Special thanks are due to Dr. Claus Ascheron and his team at Springer for bringing this book into being, and for their personal kindness throughout.

And may you, gentle reader, enjoy a happy lifetime among the coloured pixels.

London, UK and Hydra, Greece Alan Parkin

Contents

Chapter 1
Colour Environments

1.1 The Many Meanings of Colour

Colour has a vast spread of competing meanings and treatments. A preliminary task is to articulate this spread, in order to identify the corner which we wish to investigate further. One way to do this is by distinguishing typical environments in which colour occurs.

1.2 Everyday Seeing

Figure 1.1 shows the environment of everyday seeing. In this and the following flowcharts:

A rectangular box shows a *process*.
An arrow shows a *flow* of something from one process to another.
A *black/white inversion* shows the boundary of an entity.
A sloped box shows an *input or output* process at an entity boundary.
A dished box shows *storage* of some kind.

Phenomenologically, as each of us moves about and looks around, we have a continuously changing perception of the nearby external world. We know the difference between movement of objects out there and movement of ourselves. We can articulate the continuous perception by turning our attention to various aspects of the scene: what objects are there, how many, what shapes and colours they have and so on. We can remember perceptions from the immediate past, and from further back. Sometimes, we talk or write about what we see.

Philosophically, perception has been a contentious topic for centuries [1]. In recent times, the treatment has often centred on language, language games and private languages [2, 3].

A. Parkin, *Computing Colour Image Processing*,
https://doi.org/10.1007/978-3-319-74076-8_1

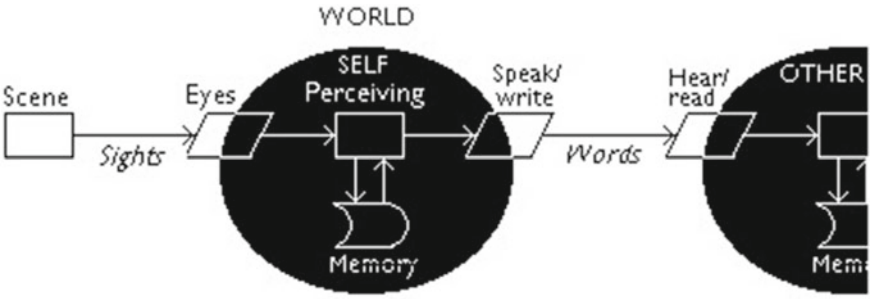


Fig. 1.1 The everyday seeing environment. In the world, a changing scene is available to those that have eyes to see. Each waking self takes in a continuously changing stream of sights from its continuously changing cone of view. This stream is processed in ways which are little understood, to emerge in consciousness as perceptions of a stable outer world. Some aspects of these perceptions pass through short-term memory, and some sink into long-term memory. Linguistic communication with others is important in this environment

Fig. 1.2 The eleven colour names in common use in English: white, black, red, yellow, green, blue, brown, purple, pink, orange and grey

Linguistically, the variety of basic colour terms in different natural languages has attracted much interest. It is known that in English and many other languages about eleven colour names are in common use [4, 5], together with a few modifiers such as 'light', 'dark', 'very' and '-ish'. Figure 1.2 shows the English eleven colours.

In the everyday seeing environment, we may say that colour means the competent use of common colour terms in describing perceptions to oneself and to others.

1.3 The Science of Seeing

Figure 1.3 shows the scientific environment of seeing.

Scientifically, a great deal is known about the causal chain on the input side of seeing [6, 7]. Physically, the behaviour of light sources and of objects reflecting, transmitting and absorbing light is well understood. Physiologically, the forming of an image on the retina of the eye and the transducing of the image to nerve impulses are also well understood [8]. Neurophysiologically, there is steady progress

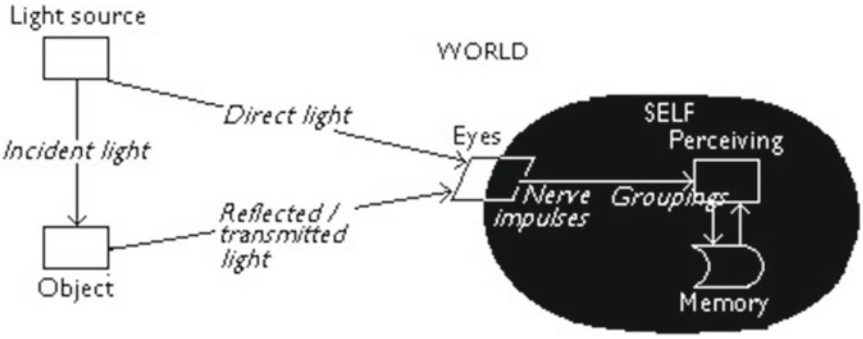


Fig. 1.3 The scientific seeing environment. In the world, light from various sources falling on various objects constitutes a scene. Eyes receive light directly, or reflected or transmitted by objects, form optical images on the retinas and transduce the momentary images to binary nerve impulses. The brain somehow groups the nerve impulses to eventuate as stable perceptions of the scene, which can be selectively stored in memory

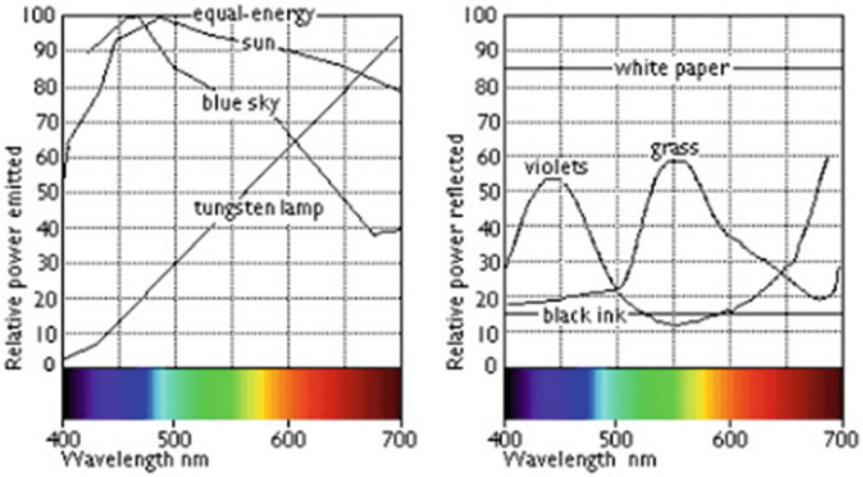


Fig. 1.4 Physics of light. **a** Spectral power distributions of three light sources. **b** Spectral power distributions of four reflecting surfaces

in understanding the further brain processes in seeing. But the causal chain is so far unable to cover the leap from nerve impulses to experiences of perception, attention and memory [9].

Figure 1.4 shows typical physical plots of colours as spectral power distributions for some light sources and some reflecting objects.

In the scientific environment, colour means the spectral power distribution of light in the external world.

1.4 Measuring Colour

Figure 1.5 shows the main components of measuring colour.

Physically, radiometry [10] measures the very wide spectrum of electromagnetic radiation (EMR). The fundamental measuring device is some form of bolometer which converts heat energy to electrical energy. Light is the visible spectrum of EMR, where the wavelengths are between 400 and 700 nm.

Psychophysically, photometry [11] measures total light intensity arriving at and leaving a surface, as sensed by a normal observer's eye. The measuring device is some form of photometer which brings a test sample of light up against an adjustable calibrated standard light. An observer adjusts the standard to get a match by eye. A photoelectric photometer responds to intensity like an eye and matches the sample directly to a calibrated standard.

Psychophysically, colourimetry [12–14] measures variations in the spectral power distribution (SPD) of wavelengths across the visible spectrum. It is based on the fact that any sample light can be matched by some combination of three fixed single-wavelength 'primary' lights. The matching is done by a standard observer (or an equivalent sensor), using elaborate equipment to adjust a mixture of three calibrated standard lights to match a test sample. A colour is then specified by three numbers, the relative intensities of the three primary sources. In the CIE 1931 xyY system, the

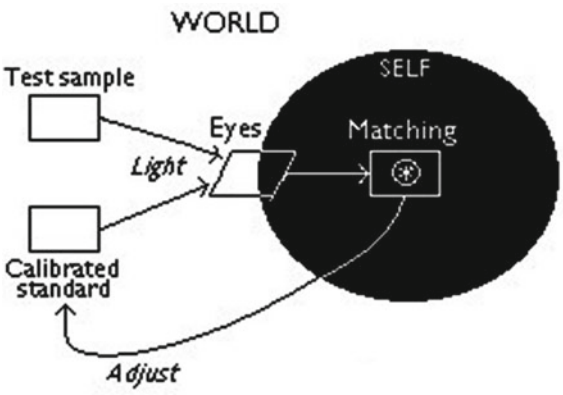


Fig. 1.5 The measuring environment. A self matches a sample to a reference standard, with or without instrumental help

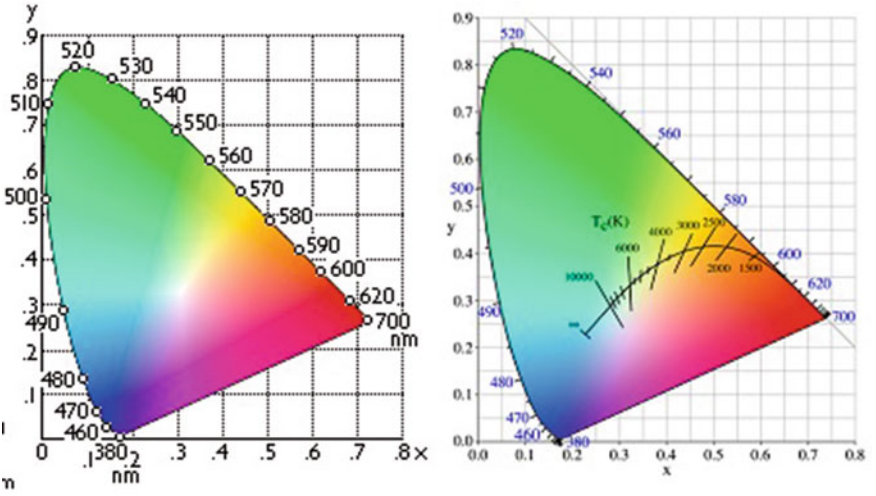


Fig. 1.6 CIE colourimetry. **a** The CIE 1931 xyY chromaticity chart. **b** Colour temperature of various light sources

Fig. 1.7 **a** Munsell tree **b** NCS swatch

whole gamut of visible colours can be plotted on a horseshoe chromaticity chart, as shown in Fig. 1.6a (with a further overall luminance value). In the 1976 LAB version, the gamut is plotted on a red/green versus blue/yellow chromaticity chart.

The colour of light reflected or transmitted by a sample is of course dependent on the colour of the illuminating source. Source colour is measured as the temperature in degrees Kelvin of a standard black body, as shown in Fig. 1.6b.

A somewhat different approach to measuring colour is by providing a reference atlas of standard surface colour patches, to which a sample is matched by direct inspection. The Munsell system [15] arranges the visible gamut in three purported psychological dimensions of hue (corresponding to CIE dominant wavelength), chroma or saturation (CIE excitation purity) and value (CIE luminance). The Munsell atlas has 1600 colours. It is presented in various forms, such as the tree shown

in Fig. 1.7a. The more recent Natural Colour System (NCS) atlas [16] has 1950 colours, based on the purported Hering opponent pairs of black/white, red/green and blue/yellow. It is also presented in various forms, such as the swatch shown in Fig. 1.7b. Colour specifications in CIE, Munsell and NCS are interconvertible.

In this environment, colour means the physical tri-stimulus values, or the atlas standard codes, which match the sample.

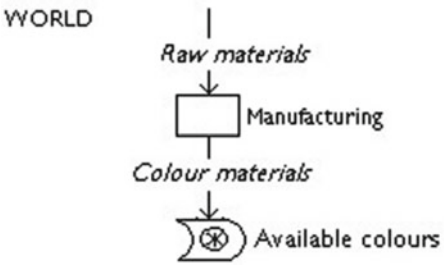


Fig. 1.8 The manufacturing environment. Raw materials are processed to colour materials, which are made available to other manufacturers and end users

Fig. 1.9 Typical manufacturer's swatch of paint colours available for interior decoration

1.5 Manufacturing Colour Materials

Figure 1.8 shows the main components of manufacturing colour materials.

Today, almost all colour materials are made by various industries and marketed to other industries, thence to trade and end users. Technologically, a great variety of raw materials, animal, vegetable and mineral are processed physically and chemically to become colour materials as paints [17, 18], dyes [19], inks [20], powders and solids.

In this environment, colour means the material manufactured and available for use. Available colours for various purposes are usually presented as colour swatches, as shown in Fig. 1.9.

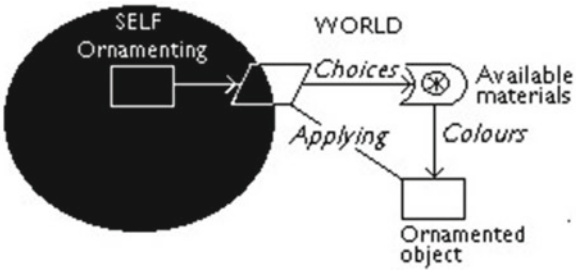


Fig. 1.10 The ornamenting environment. A self uses hands materials and tools to apply colours to objects

Fig. 1.11 An exemplar from Owen Jones' *Grammar of Ornament*

1.6 Ornamenting

Figure 1.10 shows the main components of ornamenting by colour.

Craftwise, we may often apply available colour materials to change the natural colour of objects. For example, paints to walls, dyes to clothes, cosmetics to face hair and nails, tattoos to bodies and so on. Each field of activity develops its own materials and fashions of ornament, varying widely with time and place, and often widely published and emulated (Fig. 1.11) [21]. Available materials are variously specified by swatches of reference patches, samples of dyed material and so on; to which manufacturers and marketers often attach fanciful names [22]. In this environment, colour means the available gamut of materials in the activity of interest.

1.7 Picturing

Figure 1.12 shows the main components of picturing a scene.

From childhood onward, we make pictures of scenes, real or imagined. Some develop high skills of picturing and may become professional in technical or artistic fields, making diagrams, maps, engineering drawings, architectural plans, paintings and so on. For example, Fig. 1.13 shows a painting (digitized, much-reduced, colour-changed and here published as a display in the e-book or as a print in the printed book).

A picture is essentially a projection from a three-dimensional scene to a two-dimensional surface, marked in colour of some sort. The projecting may be done

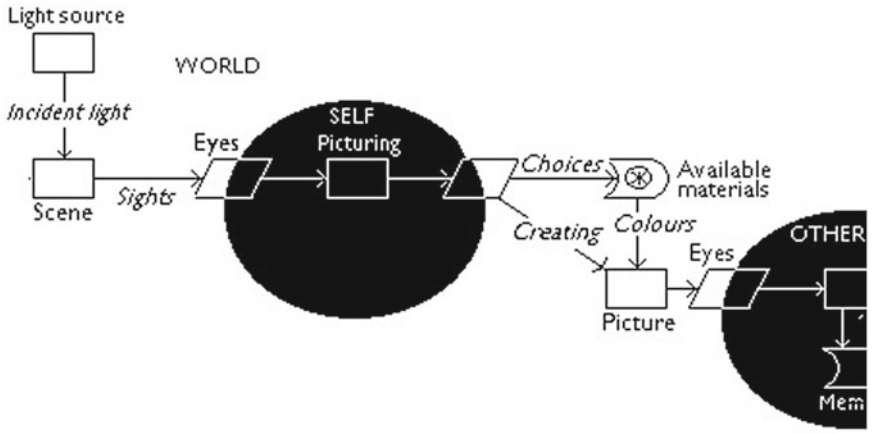


Fig. 1.12 The picturing environment. A self views a scene and makes a representation of it, live or from memory, using available paints and tools. The resulting picture can be seen by oneself and by others

Fig. 1.13 Oil painting *Brighton pierrots* by Walter Sickert,1915. Original 25 × 30 inches

mentally, or by progressive construction, possibly from measurements of the scene objects. Picture colours are chosen from available materials. There are usually practical reasons to narrow the choice of picture colours, perhaps to the extreme of black and white. In general, the gamut available for the picture cannot match the gamut of the scene, so various compromises are always needed to get an acceptable overall result [23].

The picture, rough or precise, is a convenient stand-in for the original scene and is habitually used to communicate with oneself as time passes and with others.

In this environment, colour means the gamut of materials available for a chosen technique. Casual amateurs may pay little attention to niceties of colour, while professionals may have exacting requirements [24].

1.8 Photographing

Figure 1.14 shows the main components of photographing a scene.

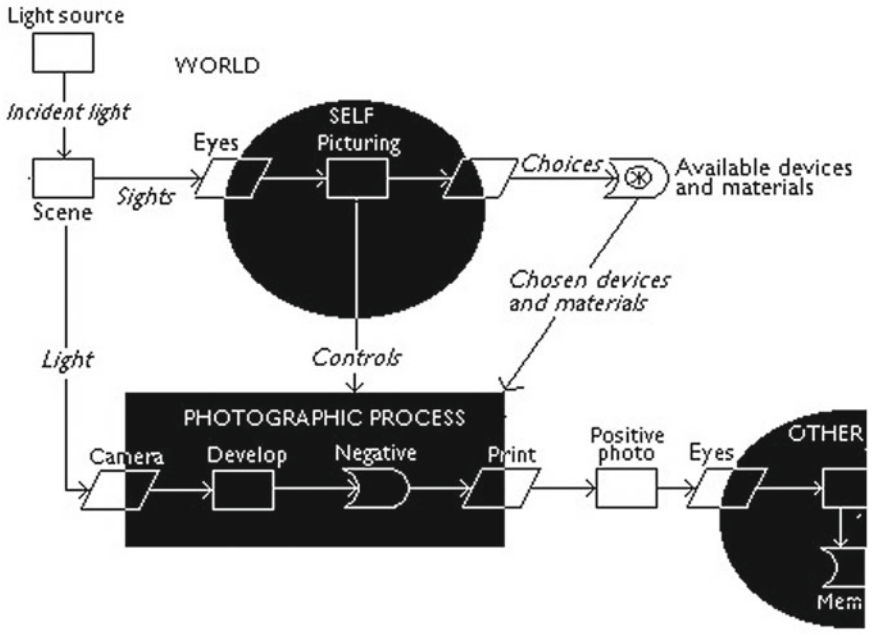


Fig. 1.14 The photographing environment. A self chooses a scene and uses a photographic process to make a representation of it. The self controls the process at various stages, making choices from available materials and tools. The resulting photograph can be seen by oneself and by others

Photographing is a special case of picturing, where a photographer uses a mechanical process to project a scene and colour the resulting representation. The photographer chooses the original scene and sets a camera to make an optical image on a light-sensitive surface. In a black-and-white photographic process [25], the materials are based on silver salts which blacken proportionately with light intensity to form a developable negative image. The negative can be stored and printed as a positive photograph any number of times. Various colour processes [26, 27] (Fig. 1.15) use three transparent dyes to filter the optical image and produce either a projectable transparency (Kodachrome) or a printable negative (Kodacolor). Historically [28], chemical photography had a wonderful run from the 1830s to around 2000 when Kodak and most other manufacturers stopped supplying the materials, overwhelmed by the success of digital photography.

In this environment, colour, broadly interpreted to include black and white, means the gamut available in the finished photograph from the chosen manufacturer's process.

Fig. 1.15 **a** Autochrome by an unknown photographer about 1910. **b** Kodachrome by Andreas Feininger 1942

1.9 Printing

Figure 1.16 shows the main components of the printing environment.

Historically [29], mass reproduction of an original image by printing has been done by several different processes: letterpress, engraving, photogravure, lithography and today's favourite offset lithography. The essential stages are to capture an original as a master printing plate (formerly by hand, nowadays by photography), and then use a press to print large numbers of reproductions for publication. Print reproduction has been revolutionized by the advent of digital processes [30–32].

Technologically, a printing press can deposit either ink or no ink at each point of an image: it cannot vary the intensity of its ink. Hence, reproducing a tonal image depends on some form of halftoning, whereby a range of greys is got by denser or sparser distributions of small dots or lines of solid black, in effect diluting the ink

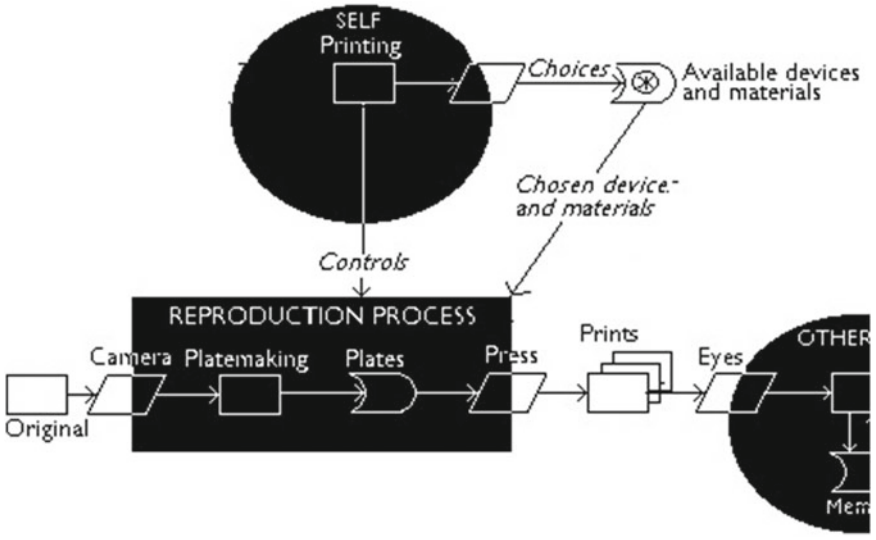


Fig. 1.16 The printing environment. Technicians use devices to photograph an original object, make plates and mass-print reproductions for publication

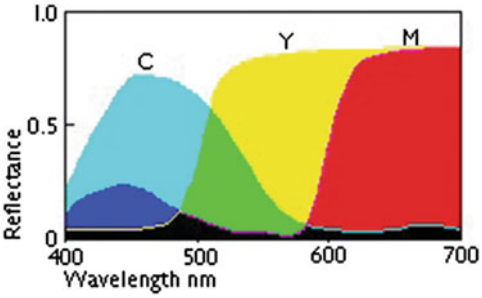


Fig. 1.17 The spectral power distributions of the four printing inks, cyan, magenta, yellow and black. Overprinting of the transparent inks filters incident light, which reflects from the white paper

with more or less of the white of the paper. In early printing, halftoning was done in the original by hand stippling or cross-hatching; subsequently, it was done by photographing a tonal original through a halftone screen; and today by digital means [33].

Optically, printed inks are transparent films, acting as filters on the incident light, unlike opaque paint coats. Thus, a red ink absorbs the red long wavelengths and transmits the green middle wavelengths and the blue short wavelengths. A reasonably good gamut can be got using inks which are the inverses of red green and blue, that is to say, cyan, magenta and yellow (CMY). Ideally, the overlaying of all three inks should show as black; but unfortunately available inks do not do this. So a black ink (K) is added.

Practically, reproducing a colour original is done by preparing a halftone plate for each of the four CMYK inks, imposed in register on the paper by the press. Figure 1.17 shows the SDPs of a set of CMYK inks and their overprints.

1.10 Digital Imaging

Figure 1.18 shows the main components of the digital imaging environment.

In digital imaging, a person uses a computer and various peripheral devices to create, store, analyse, transform, display, print and communicate an image as an array of coloured picture elements (pixels). The processes are open to user control throughout [34].

Common methods of creating a digital image are by hand, using a graphic user interface (GUI); by program, using a suitable programming language; by optical projection from a scene, using a digital camera; by contact capture from a given flat image, using a scanner; by capture from a remote source, using a downloaded file; or by calculating a projection from a numerically specified three-dimensional model. All these methods produce a machine-readable numerical representation of an image.

Storing is done in various file structures, using standard read/write routines.

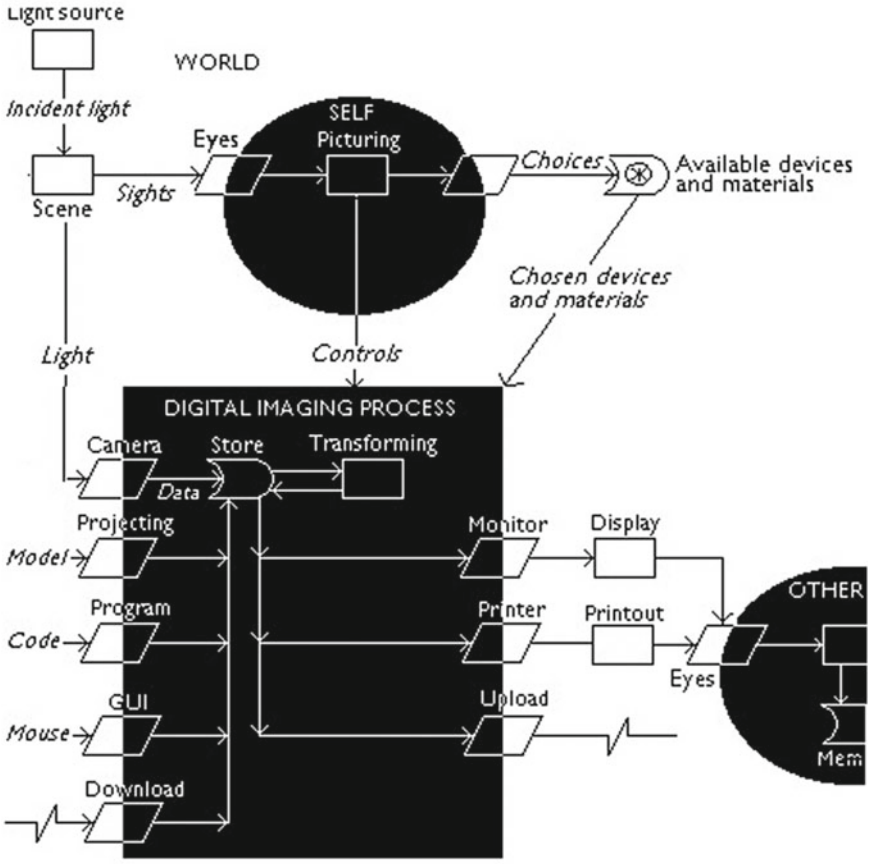


Fig. 1.18 The digital imaging environment. A self chooses a scene and uses a digital process to make a representation of it. The self controls the process at various stages, making choices from available materials and tools. The resulting image can be seen by oneself and by others, possibly widely disseminated

Analysing and transforming is done by programs, either in commercially available packages or own-written scripts.

Additive displaying uses a monitor screen, via a manufacturer's driver software.

Subtractive printing uses a printer, via a manufacturer's driver software.

Communicating uses a transducer to send a file to a remote address on the Internet.

In this environment, colour, broadly interpreted to include black and white, means the gamut which survives the successive stages of imaging from input to output.

It is colour in this environment which we now wish to investigate.

References

1. Scruton R (2004) Modern philosophy. Penguin, London
2. Wittgenstein L (2003) Remarks on colour. Blackwell, Oxford
3. Hardin CL (1988) Color for philosophers. Hackett Publishing Co, Cambridge
4. Berlin B, Kay P (1969) Basic color terms. Univ. Calif. Press, Berkeley
5. Hardin CL, Maffi L (2008) Color categories in thought and language. CUP, Cambridge
6. Judd DB, Wyszecki G (1975) Color in business science and industry, 3rd edn. Wiley, New York
7. Wyszecki G, Stiles WS (1988) Color science, 2nd edn. Wiley, New York
8. Bass M et al (2010) Handbook of optics vol III vision, 3rd edn. McGraw-Hill, New York
9. Dennett DC (1991) Consciousness explained. Allen Lane, London
10. Radiometry. https://en.wikipedia.org/wiki/Radiometry
11. Photometry. https://en.wikipedia.org/wiki/Photometry_(optics)
12. Colorimetry. https://en.wikipedia.org/wiki/Colorimetry
13. Wright WD (1944) The measurement of colour. Adam Hilger, London
14. Commission Internationale d'Eclairage. www.cie.co.at
15. Munsell Color System. https://en.wikipedia.org/wiki/Munsell_color_system
16. Natural Color System. https://en.wikipedia.org/wiki/Natural_Color_System
17. Lambourne R, Stevens TD (1999) Paint and surface coatings, 2nd edn. Woodhead, Cambridge
18. Talbert R (2007) Paint technology handbook. CRC Press, Boca Raton
19. Society of Dyers and Colourists (2017), Colour Index. https://colour-index.com
20. Leach R (2012) The printing ink manual, 4th edn. Springer, Heidelberg
21. Jones O (1856) The grammar of ornament. Day, London, p 1856
22. List of colors. https://en.wikipedia.org/wiki/List_of_colors_(compact)
23. Ruskin J (1843), Modern painters, vol I Pt II Sect II Chap I: Of truth of tone. https://www.gutenberg.org/files/29907/29907-h/29907-h.htm
24. Handprint. www.handprint.com
25. Mees CEK (1942) The theory of the photographic process. Macmillan, New York
26. Autochrome Lumiere. https://en.wikipedia.org/wiki/Autochrome_Lumiere
27. Kodachrome. https://en.wikipedia.org/wiki/Kodachrome
28. Gernsheim H, Gernsheim A (1960) The history of photography. Thames and Hudson, London
29. Twyman M (1970) Printing 1770–1970. Eyre and Spottiswoode, London
30. Yule JAC (2001) Principles of color reproduction, 2nd edn. Wiley, NJ
31. Kipphan H (2001) Handbook of Print Media. Springer, Heidelberg
32. Hunt RWG (2004) The reproduction of color, 6th edn. Wiley, New York
33. Ulichney R (1987) Digital halftoning. MIT Press, Cambridge
34. Parkin A (2016) Digital imaging primer. Springer, Heidelberg

Chapter 2
Digital Imaging Fundamentals

2.1 Digital Image

Visually, a *digital image* is a rectangular array of (nominally) square elements called *pixels*, each showing a colour. Figure 2.1 shows a simple example, displayed on a screen (if you are reading this as an e-book) or printed on paper (if you are reading it as a print book).

2.2 sRGB Colour Space

sRGB is a standard [1] defining a colour space and viewing conditions for digital images [2]. It is available in virtually all current personal computers, digital cameras, scanners, displays and printers.

In brief, sRGB has:

Three *variables*: red, green and blue.
In each variable, a range of integer *intensities* R, G, B, where $0 \leq R \leq 255$, $0 \leq G \leq 255$, $0 \leq B \leq 255$.
In the whole space, $256^3 = 16.7$ million *colours*, where a colour is an additive mixture of three intensities: (R, G, B).
A subset of 256 *neutrals*, colours where $R = G = B$.

The sRGB variables are defined as three primary *light sources* (the same as for HDTV [4]), which have the CIExyY chromaticity coordinates [3]:

R: x = 0.64, y = 0.33.
G: x = 0.30, y = 0.60.
B: x = 0.15, y = 0.06.
White point: x = 0.3127, y = 0.3290.

Figure 2.2a shows the three primary colours, which combine additively as white, and (b) shows the sRGB gamut within the full CIE gamut. Thus, sRGB colour space is

A. Parkin, *Computing Colour Image Processing*,
https://doi.org/10.1007/978-3-319-74076-8_2

Fig. 2.1 A simple digital image. The array has 8 columns and 8 rows, hence 64 pixels, which are coloured white, black, red, green and blue

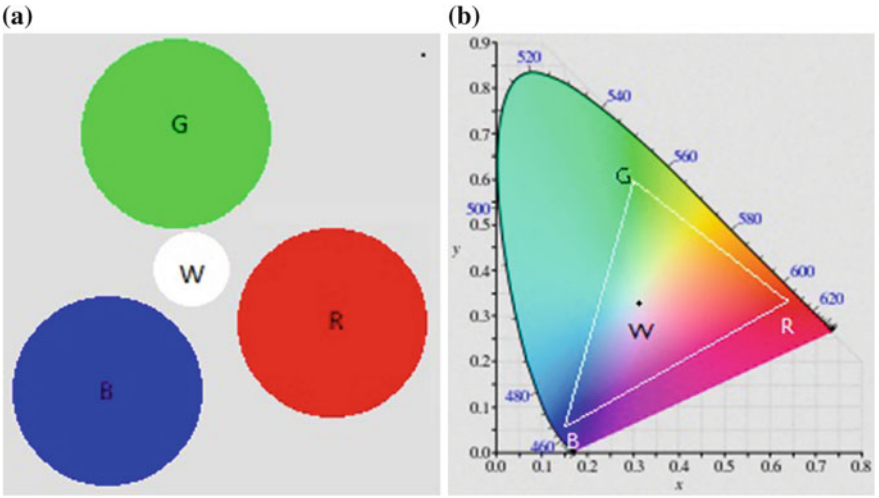


Fig. 2.2 **a** The three fundamental sRGB colours, which combine additively to show white. **b** The gamut of sRGB colours lies within the triangle of the three fundamental colours, which lies within the CIExyY chromaticity chart. (Notice that since this chromaticity chart is itself displayed here in sRGB, the colours are merely indicative, not CIE-accurate)

a subspace of CIE colour space. Any sRGB colour has a CIE equivalent, and some but not all CIE colours have an sRGB equivalent.

Visually, sRGB colour space is best modelled as a Cartesian cube, shown in Fig. 2.3 in front and back views. One vertex of the cube is the origin (0 0 0) black. The three edges from the origin are coordinate axes calibrated in integer steps of intensity from 0 to 255. The R-axis goes from (0 0 0) black to (255 0 0) red, the G-axis to (0 255 0) green and the B-axis to (0 0 255) blue. The other four vertices

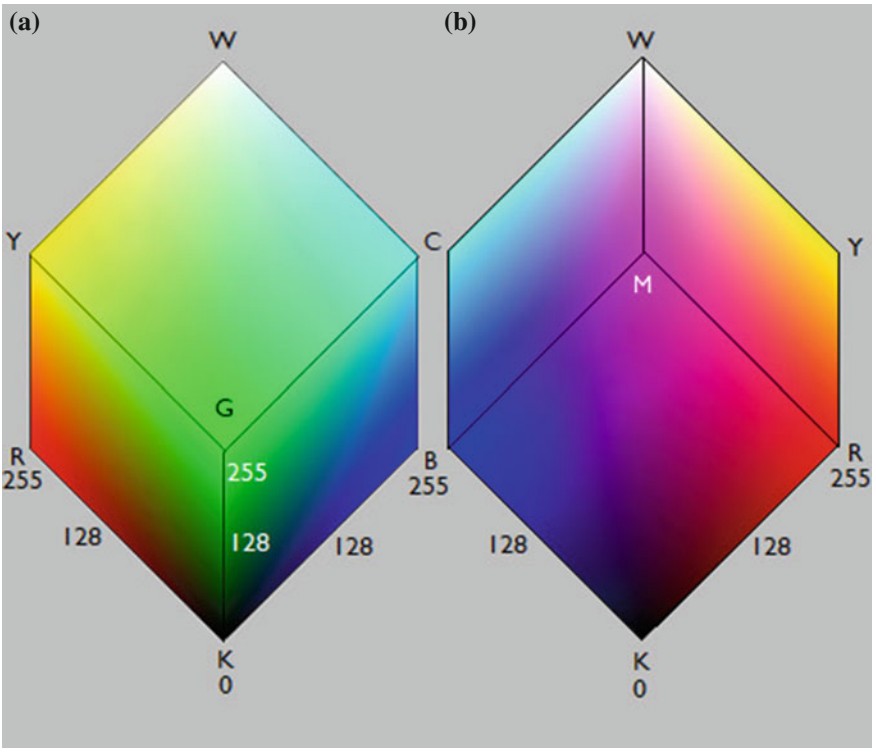


Fig. 2.3 SRGB cube model. **a** Front view. **b** Back view

are then (255 255 0) yellow, (255 255 255) white, (0 255 255) cyan and (255 0 255) magenta. The four body diagonals of the cube meet in the centre at (127 127 127) mid-grey. (Notice that in sRGB there is a systematic equivocation between intensity values 127 and 128. The middle value between 0 and 255 is 127.5, which can be arbitrarily rounded down or up without visual effect.)

2.3 Numerical Representation

Numerically, a digital image has:

Width W pixels.
Height H pixels.
Hence, *Extent* $E = W \times H$ pixels.

and a pixel has:

Location (X, Y), integers where $0 \leq X < W - 1$ and $0 \leq Y < H - 1$.
Colour (R, G, B), integers where $0 \leq R < 255$, $0 \leq G < 255$, $0 \leq B < 255$.

2.4 Scan Sequence

The conventional *scan sequence* of pixels in an image is shown in Fig. 2.4. Then, the complete numerical representation of the image is W, H followed by a list of (R, G, B) triples in scan sequence. For example, the simple image in Fig. 2.1 is represented numerically by the sequence:

```
8,8,(255,0,0),(255,255,255),(0,0,0),(255,255,255),(0,0,0),
(255,255,255),(0,0,0)),(255,255,255),(0,0,0),(255,255,255),
(0,0,0),(255,255,255),(0,0,0)),(255,255,255),(0,0,0),
(255,255,255),(255,0,0),(255,255,255),(0,0,0),(255,255,255),
(0,0,0),(255,255,255),(0,0,0)),(255,255,255),(0,0,0),
(255,255,255),(0,0,0),(0,255,0),(0,0,0)),(255,255,255),
(0,0,0),(255,255,255),(255,0,0),(255,255,255),(0,0,0),
(255,255,255),(0,0,0),(255,255,255),(0,0,0)),(255,255,255),
(0,0,0),(255,255,255),(0,0,0),(255,255,255),(0,0,0)),
(255,255,255),(0,0,0),(255,255,255),(255,0,0),(255,255,255),
(0,0,0),(255,255,255),(0,0,0),(255,255,255),(0,0,0)),
(255,255,255),(0,0,0),(255,255,255),(0,0,0),(255,255,255),
(0,0,0)),(255,255,255),(0,0,0),(0,0,255)
```

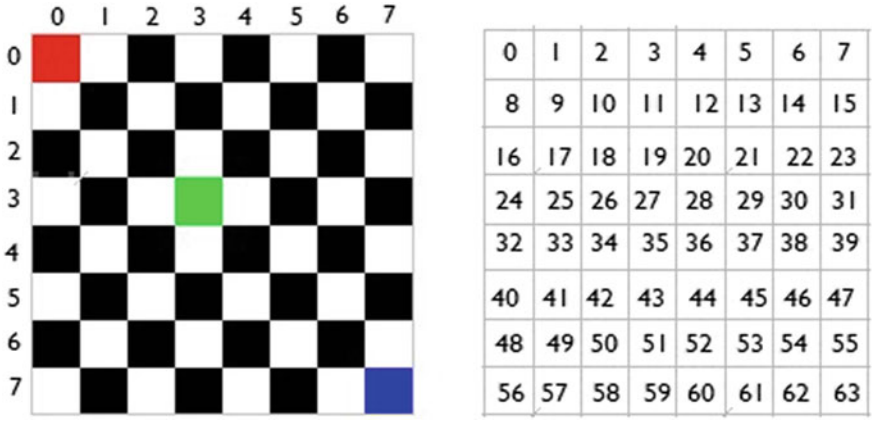


Fig. 2.4 Scan sequence

2.5 Computer Processing of Images

Computationally, the numerical representation of an image can be created, stored, transformed, displayed, printed and transmitted via a computer, using any suitable programming language. Python [5] is particularly suitable, as a scripting language with the associated Python Imaging Library (PIL) [6], Tkinter, numpy and scipy languages. For example, the following Python script will select, open and display any .bmp image in the current user directory:

```
# pyopsh: Python script to open and show a .bmp image.
# Written by Alan Parkin 2017.

from PIL import Image
import os, sys

# Enter any .bmp filemame in working directory.
imagefilename = raw_input(''Image filename?'')

#Open it, print filename, W, H, E, and show it.
im = Image.open(imagefilename)
width = im.size[0]
height = im.size[1]
extent = width * height
print ''filename'', imagefilename
print ''width W'', width
print ''height H'', height
print ''extent E'', extent
im.show()
```

The Python output is:

```
Python 2.7.11 |Anaconda 2.4.1 (64-bit)| (default,
 Jan 19 2016, 12:08:31) [MSC v.1500 64 bit (AMD64)]
 on win32
Type ''copyright'', ''credits'' or ''license()''
 for more information.
>>>
================ RESTART: C:\Users\Alan
 Parkinalan\pyopsh.py ================
Image filename? sim8.bmp
filename sim8.bmp
width W 8
height H 8
extent E 64
>>>
```

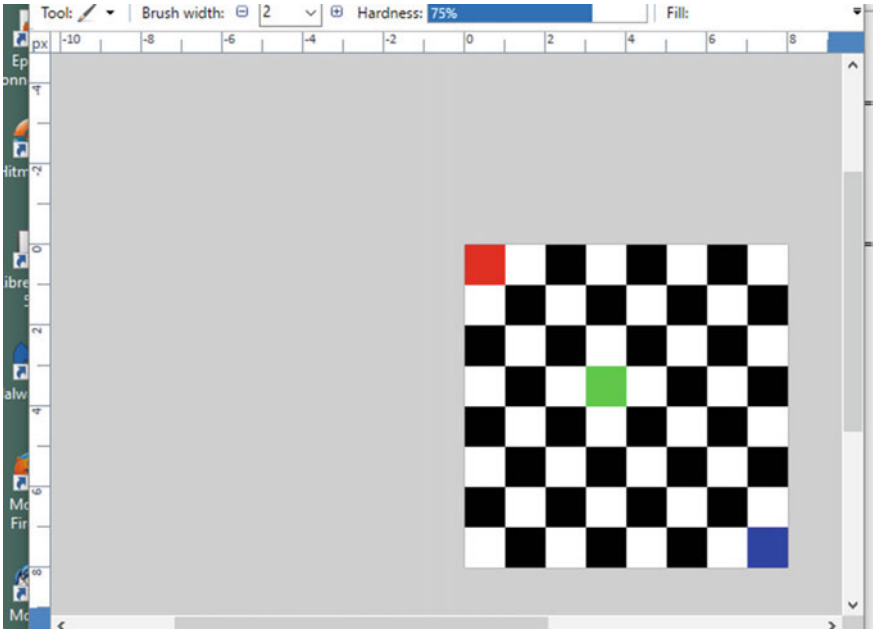


Fig. 2.5 Simple image sim8.bmp opened and displayed in Paint (where it is magnified by a factor of eight, as shown by the marginal rulers)

and the image displayed in the Microsoft Paint image editor, as shown in Fig. 2.5, whence it can be saved to storage or otherwise disposed of.

2.6 Location Resolution

Numerically, the *location resolution* of an image is the smallest detail which it can show. In a digital image, pixels are indivisible, so the smallest detail is one pixel distinguished (by colour) from its neighbours. Numerically, the *location resolution limit* $LOCRES = 1/E$, one pixel in E, where E is the extent, that is, width $\times$ height, of the image.

Notice that pixels, extent and location resolution are of indefinite size: they take on size only when displayed on a device which has a fixed *pixel pitch* of so many pixels per inch (ppi) or pixels per millimetre (ppm). For example, the simple image in Fig. 2.1 has width $W = 8$, height $H = 8$, extent $E = 64$ and location resolution limit $LOCRES = 1/64$. The smallest detail it can show is 1/64 of the extent, at whatever size it is displayed or printed.

A digital image with large extent E, such as a typical camera or scanner image, has extremely fine location resolution: perhaps 1 in 10 million or more. Such fine resolution is often unnecessary and inconvenient. It is common practice to reduce location resolution to suit the purpose in hand, by resizing down or by severe cropping. For methods, see Chap. 5.

2.7 Colour Resolution

Numerically, the *colour resolution* of an image is the least difference of colours which it can show. In a digital image, the least difference is one *step* in the colour space of the image. Numerically, the sRGB colour space has three axes, each with $S = 256$ steps of intensity; hence, the cube contains $S^3 = 256 \times 256 \times 256 = 16.7$ million colours, each different from its neighbours. Call this number the *diversity* D of the sRGB colour space. We define the *colour resolution limit COLRES* as $1/D$, one colour in D.

A digital image with large colour diversity D, such as a typical sRGB camera or scanner image, has an extremely fine colour resolution: 1 in 16.7 million. Such fine resolution is often unnecessary and inconvenient: as when, for example, we want to analyse colour distribution in an image, or discern essentials from inessentials, or make systematic changes of colour. We can simplify an image by reducing colour diversity to a subspace of sRGB, thus coarsening colour resolution.

We can define a series of subspaces, or restricted *palettes*, within sRGB by taking fewer than $S = 256$ steps of intensity per axis of the sRGB cube. For example, a minimal palette P2 has $S = 2$, hence diversity $D = S^3 = 8$ colours, just those at the vertices of the cube. Figure 2.6 shows palette P2.

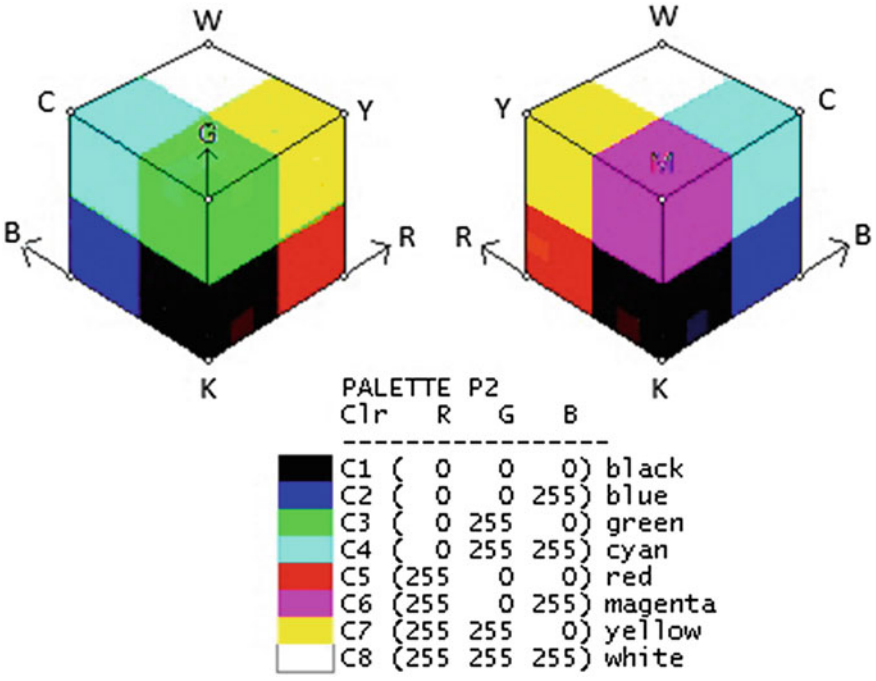


Fig. 2.6 Palette P2, a subset of sRGB, with $S = 2$ steps per cube axis

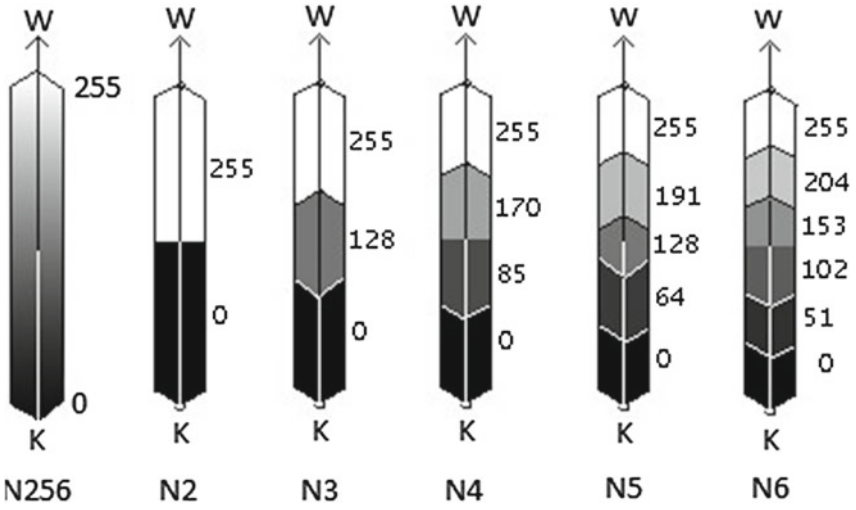


Fig. 2.7 Neutral palette N256, and subsets N2, N3, N4, N5, N6 with 2, 3, 4, 5 and 6 steps along the axis

An important subspace of sRGB is the *neutral palette* or greyscale N256, where ($R = G = B$). This has just one axis, the body diagonal of the cube from black (0,0,0) to white (255,255,255). It has $S = 256$ steps of intensity I, and hence contains just 256 neutrals, with diversity $D = 256$, and $COLRES = 1/256$, one grey in D. A series of neutral greyscales can be made by taking fewer than $S = 256$ steps of intensity on the diagonal axis. For example, a minimal neutral greyscale N2 has $S = 2$, hence diversity $D = 2$ colours, just black and white. Figure 2.7 shows neutral palettes N256, N2, N3, N4, N5 and N6.

For methods of reducing colour resolution, see Chap. 6.

References

1. sRGB Color Space. https://webstore.iec.ch/publication/6168
2. sRGB Color Space. https://en.wikipedia.org/wiki/SRGB
3. CIE 1931 color space. https://en.wikipedia.org/wiki/CIE_1931_color_space
4. ITU Rec.709. https://en.wikipedia.org/wiki/Rec._709
5. Python Software Foundation. https://www.python.org
6. Fredrik Lundh effbot. http://effbot.org

Chapter 3
Creating a Digital Image

3.1 Creating by Image Editor

To create an image by an image editor, such as Microsoft Paint [1], typical steps are as follows:

Open the editor.
Open New image, setting width W, height H, background colour, filename and storage format.
For comfort, adjust editor magnification for very small images, or diminution for very large images.
Select colour for drawing and filling shapes.
Draw features, possibly with undo/redo.
Cut/paste/move/erase/redraw features.
Save image.

For example, Fig. 3.1 shows some of the steps for creating a simple image, in Paint. Colours can be selected visually, and/or numerically by sRGB triple (R, G, B). The image is saved numerically to storage in the usual scan-sequence.

Figure 3.2 shows a 32×32 pixel icon created in an editor. An editor is particularly suitable for creating moderately complex images with multiple repetitions, as shown in Fig. 3.3.

3.2 Creating by Program

To create an image by program, such as a Python script [2], typical steps are:

Open the script editor.
Create Open New image, setting width W, height H, background colour, filename and storage format.
Write code for the required operations, with progressive output messages.

A. Parkin, *Computing Colour Image Processing*,
https://doi.org/10.1007/978-3-319-74076-8_3

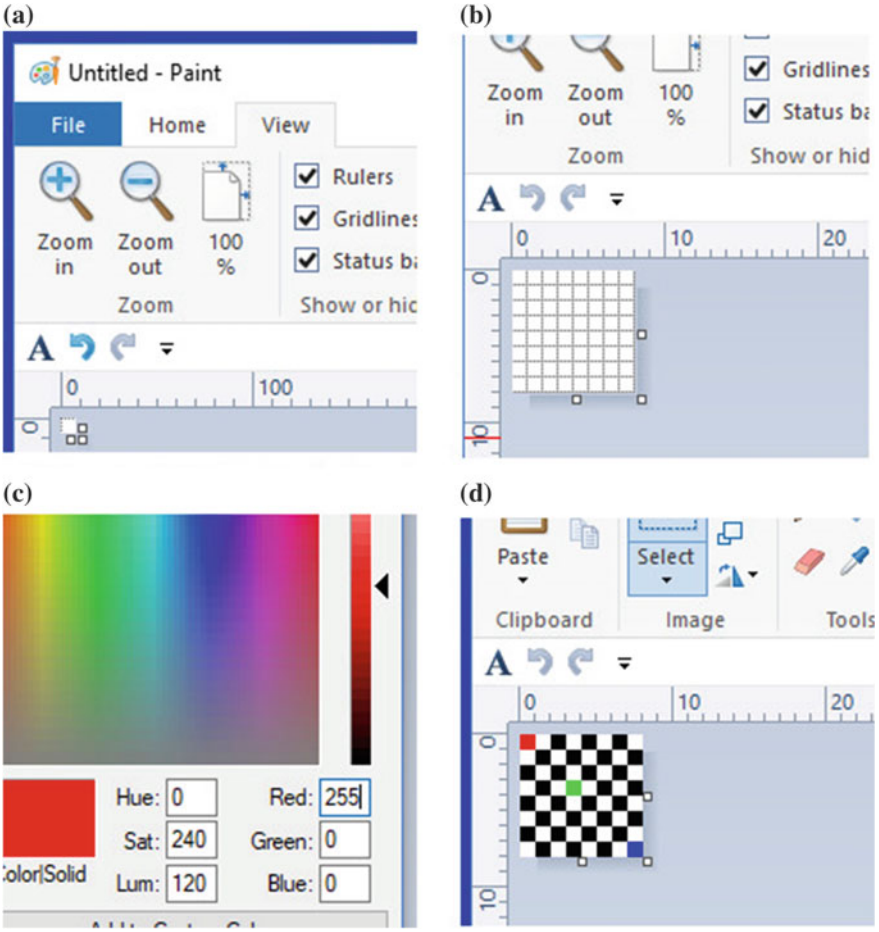


Fig. 3.1 **a** Part of Paint screen with new 8 × 8 pixel white blank, tiny at actual size. **b** Image and marginal rulers magnified to 800% for comfort. **c** Setting a colour. **d** Finished simple image, still magnified

Debug code.
Remove superfluous output messages.
Save script.

A program can do anything that an image editor can do. For example, the following Python [2] script creates the simple image in Sect. 1:

```
# pycrsim: Create simple image, using xy coordinates.
# Written by Alan Parkin 2017.

from PIL import Image
```

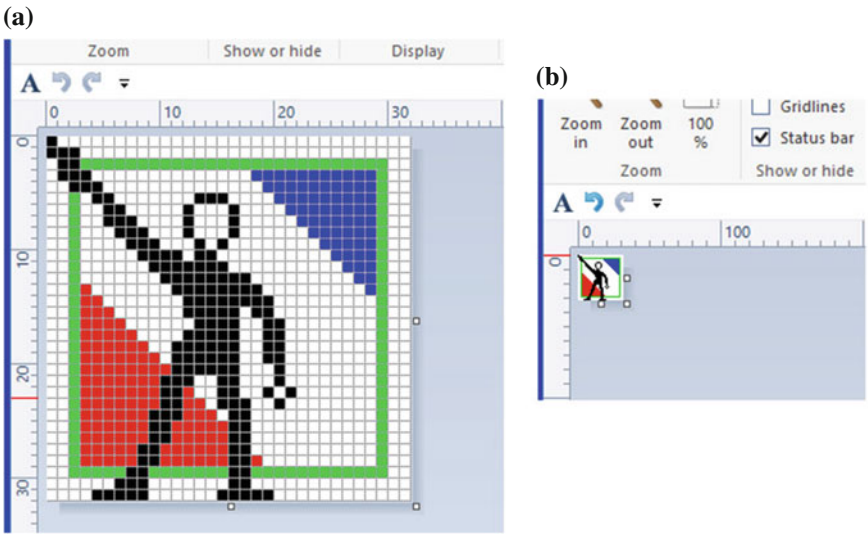


Fig. 3.2 **a** Part of Paint screen with 32 × 32 icon, magnified to 800% for comfort. **b** Icon at actual size

```
import os, sys

# Ask width and height, and create all-white array.
imwidth = int(raw_input("Width? "))
imheight = int(raw_input("Height? "))
imrgb = Image.new("RGB", (imwidth, imheight), 'white')
imrgb.show()

# Call PIL load to get pixy all-white xy array
# of rgb tuples.
pixy = imrgb.load()
print "All-white loaded"

# Set chequer in pixy.
for y in range(imheight):
    for x in range(imwidth):
        if y%2 == 0:
            if x%2 == 0:
                r = 0
                g = 0
                b = 0

            else:
```

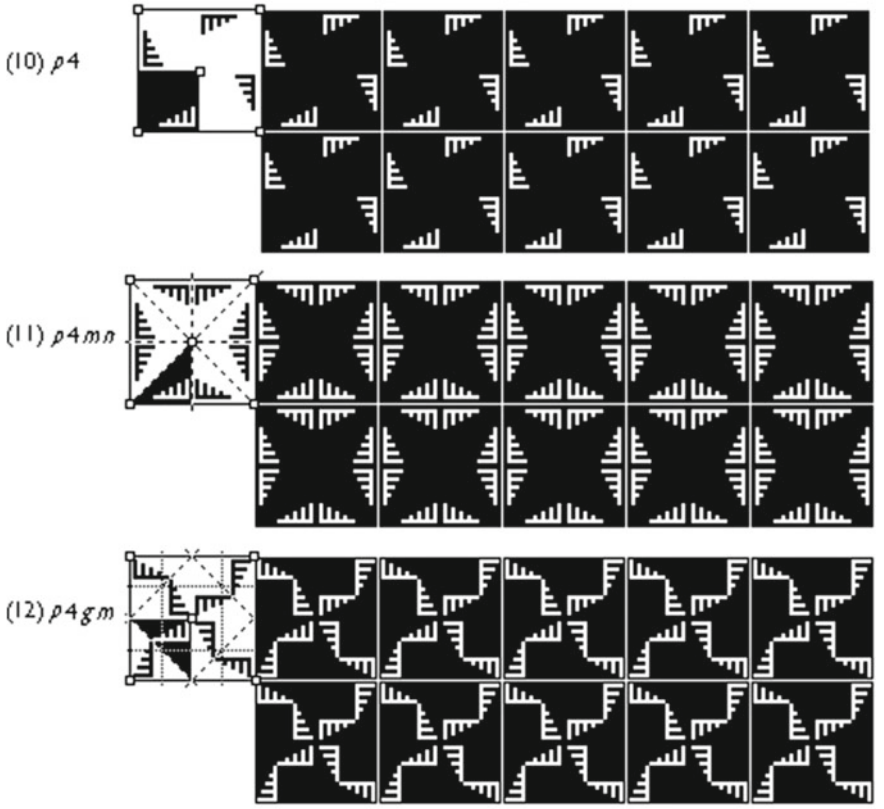


Fig. 3.3 An image editor is well suited to creating images of moderate complexity with repetitions, such as this book illustration

```
            r = 255
            g = 255
            b = 255

    else:
        if x%2 == 0:
            r = 255
            g = 255
            b = 255

        else:
            r = 0
            g = 0
            b = 0
```

```
            pixy[x,y] = (r,g,b)

    imrgb.show()
    print "Chequer set"

    # Set red green and blue pixels.
    pixy[0,0] = (255,0,0)
    pixy[3,3] = (0,255,0)
    pixy[7,7] = (0,0,255)

    imrgb.show()
    print "Red green blue set"
```

But a program can create images far beyond the reach of a GUI image editor. For example, the following Python script creates one face of a CMYK colour space cube model, 65 536 pixels created in a moment, as shown in Fig. 3.4.

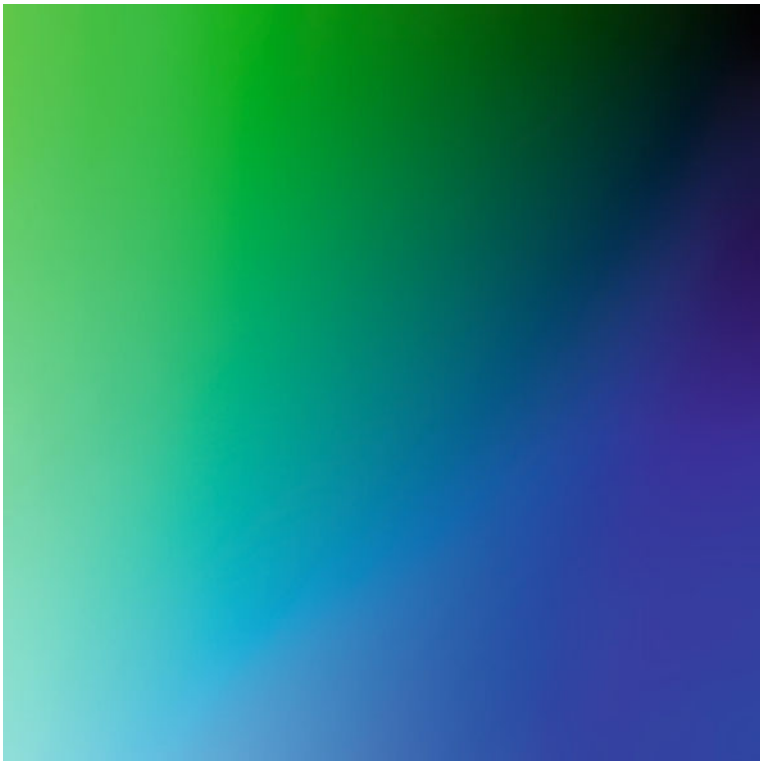


Fig. 3.4 A complex image 256 × 256 pixels, created by the program pycu6 listed here

```
# pycuf6: Create cmyk cube face 6 usng xy coordinates.
# Written by Alan Parkin 2017

from PIL import Image
import os, sys

# Create all-white array 256 x 256 pixels.
imrgb = Image.new("RGB", (256,256), 'white')

# Call PIL load to get pixy all-white xy array
# of rgb tuples.
pixy = imrgb.load()

# Generate pixel colours.
# Scanning origin k, x coords to green, y coords
# to blue.
for y in range(256):
    for x in range(256):
        r = 0
        g = 0 + x
        b = 0 + y
        pixy[x,y] = (r,g,b)

imreflr = imrgb.transpose(0)

imreflr.show()
print "Face 6 done"
```

3.3 Creating by Camera

To create an image by camera typical steps are:

IN THE CAMERA:
Power-up the camera.
Set camera parameters.
Choose view of scene.
Shoot photo.
IN THE COMPUTER:
Transfer photo to computer storage.
Display original photo in image editor.

An sRGB digital camera [3] focuses an optical image onto a sensor array with a very large count of pixels, typically $E = 10$ million or more. A mosaic of filters on the array restricts each sensor pixel to red or green or blue light. Each pixel

emits an analogue voltage directly proportional to the light which falls on it. An analogue-to-digital converter quantizes this to a digital intensity between 0 and 255. A de-mosaicking process constructs an (R, G, B) triple per pixel. These raw intensity values are encoded under the sRGB convention that subsequent display will be on a device with decoding gamma $\gamma = 2.2$ [4–6] (see Chap. 7).

sRGB incorporates the CIELAB scale of lightness L*, where perceptual *mid-grey* L* = 52% is taken as the physical light intensity reflected by an 18% photographic grey card [7]. sRGB applies exponent 1/2.2 = 0.4545 to counteract the assumed display decoding. Thus, mid-grey codes to $(0.18^{0.4545} * 255) = 127$. The encoding formula is

File code 0 to 255 = ((camera sensor value normalized between 0 and 1)$^{0.4545}$ * 255)

Figure 3.5 plots the encoding calculation. (For the corresponding decoding see Chap. 7)

The coded image file is delivered to storage, usually in compressed .jpg file format, or possibly in a proprietary uncompressed *raw* format (see Chap. 4).

After transferring to computer storage, a camera image can be displayed in an image editor, such as Paint. The working window of an editor screen has a fixed

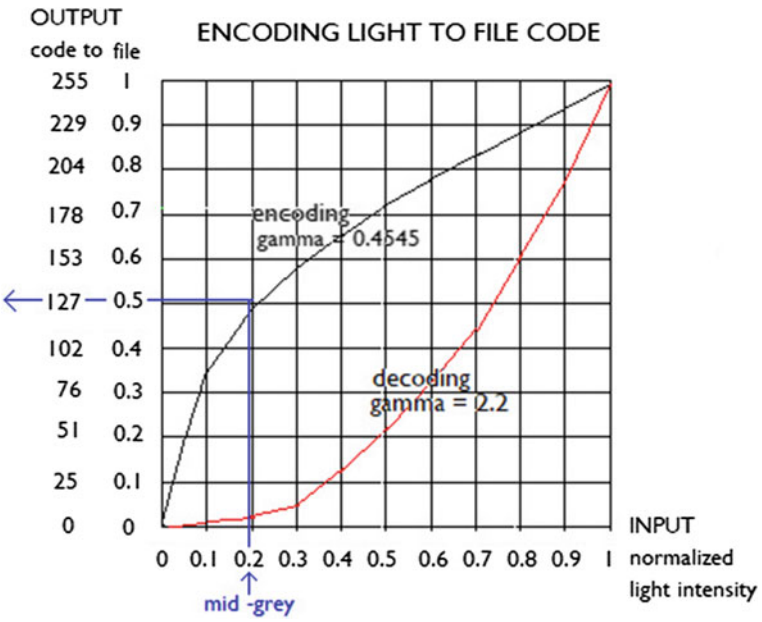


Fig. 3.5 Encoding a camera or scanner image. Encoder input is light intensity at each pixel, normalized to range 0–1. sRGB mid-grey is a photographic grey card reflecting 18% of incident light. Encoder output is code $((\text{input}^{0.4545}) * 255)$, so that mid-grey is coded 127. The red curve is the countervailing gamma = 2.2 used in decoding

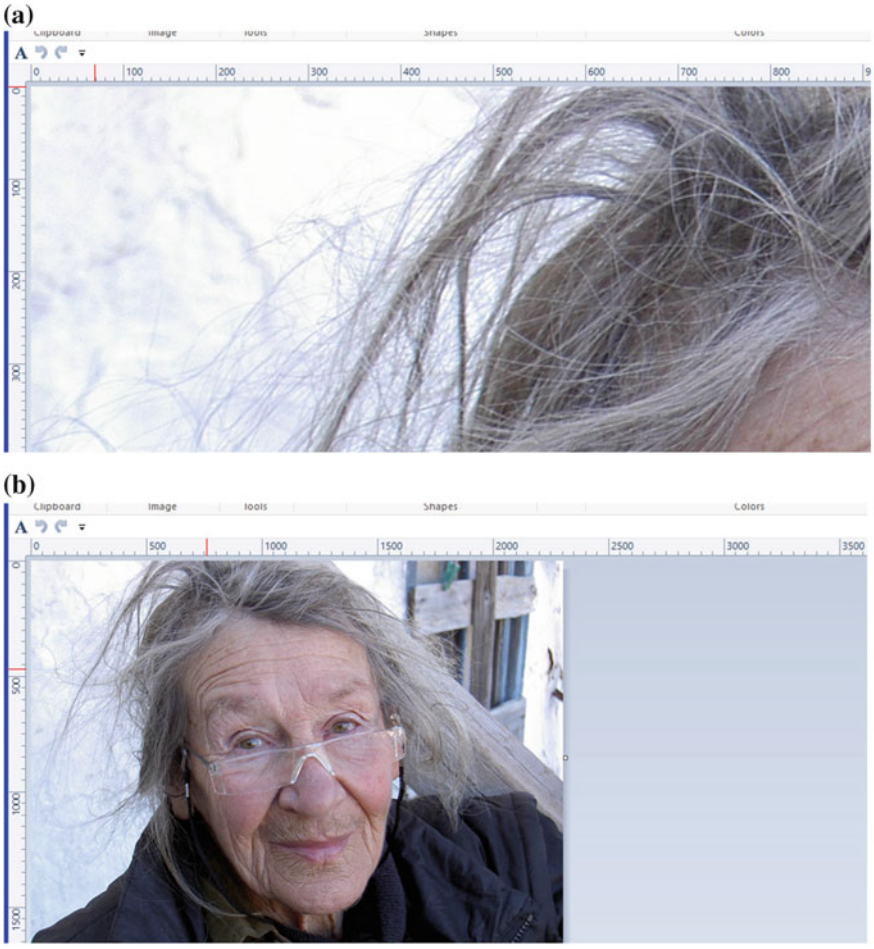


Fig. 3.6 Displaying a camera image. **a** Paint screen displaying at most 1000 × 500 px of a camera image which is 4000 × 3000 px. The display window can be panned and scrolled to show other parts of the image; or, as in (**b**), the image can be diminished (here to 12.5%)

array of pixels at a fixed pitch of around 100 pixels per inch (ppi); so the extent of the window is, say, 1000 × 500 px. Since the camera image has, say, 4000 × 3000 px, clearly it cannot be displayed whole in the editor window. The editor window can be panned and scrolled across the image, or the image can be diminished to fewer pixels, as shown in Fig. 3.6. Notice that this diminution is local to the editor, and does not affect the stored image. For permanent resizing, see Chap. 6.

3.4 Creating by Scanner

To create an image by scanner, typical steps are as follows:

Power-up the scanner.
Set scanning parameters.
Place original on platen.
Scan into storage.
Display scanner image in an image editor.

An sRGB scanner [8] has a platen, typically A4 size 11.75 × 8.25 in, on which a flat or nearly flat object is placed. An optical mechanism containing a fixed light source traverses the object at a settable pixel pitch, registering a very large count of sensor pixels. Scanners are manufactured with maximum pixel pitch of 300 pixels per inch (ppi), 600 ppi, 1200 ppi or more. The platen can be masked to less than its whole area. A full-platen scan at 300 pixels per inch delivers an image with extent $E = 8.7$ Mpx, comparable with a digital camera. Each pixel can register the very large sRGB count of colours, $D = 16.7$ million.

The sensor has pixels filtered to receive red or green or blue light, and their outputs are processed to sRGB triples (R, G, B) in much the same way as in a camera, intended for an sRGB display device.

A scanner delivers an image in .bmp, or other chosen format such as .jpg.

A scanner image in storage can be displayed in an image editor, such as Paint. As for a camera image, the editor window can be diminished, or, for close examination, magnified. Figure 3.7 shows a scanner image diminished and magnified. Diminution or magnification in an editor does not affect the stored image. For permanent re-sizing, see Chap. 6.

3.5 Creating by Modelling

To create an image by modelling typical steps are as follows:

Get coordinates of salient vertices of the object, by measuring an actual object or by assigning to an imagined object, in a suitable reference frame.
Enter the coordinate data to the computer.
Compute a chosen projection of the object vertices to a digital image.
Complete the image with edges between the vertices for a wire-frame representation, possibly with hidden line removal, possibly with lighting rendering
Store the image.
Display the modelled image in an image editor.

For example, Fig. 3.8 shows some steps of a simple modelling [9]. Much advanced work goes into creating digital images from models of various kinds. In some fields massive three-dimensional coordinate data sets are available in digital form, as in

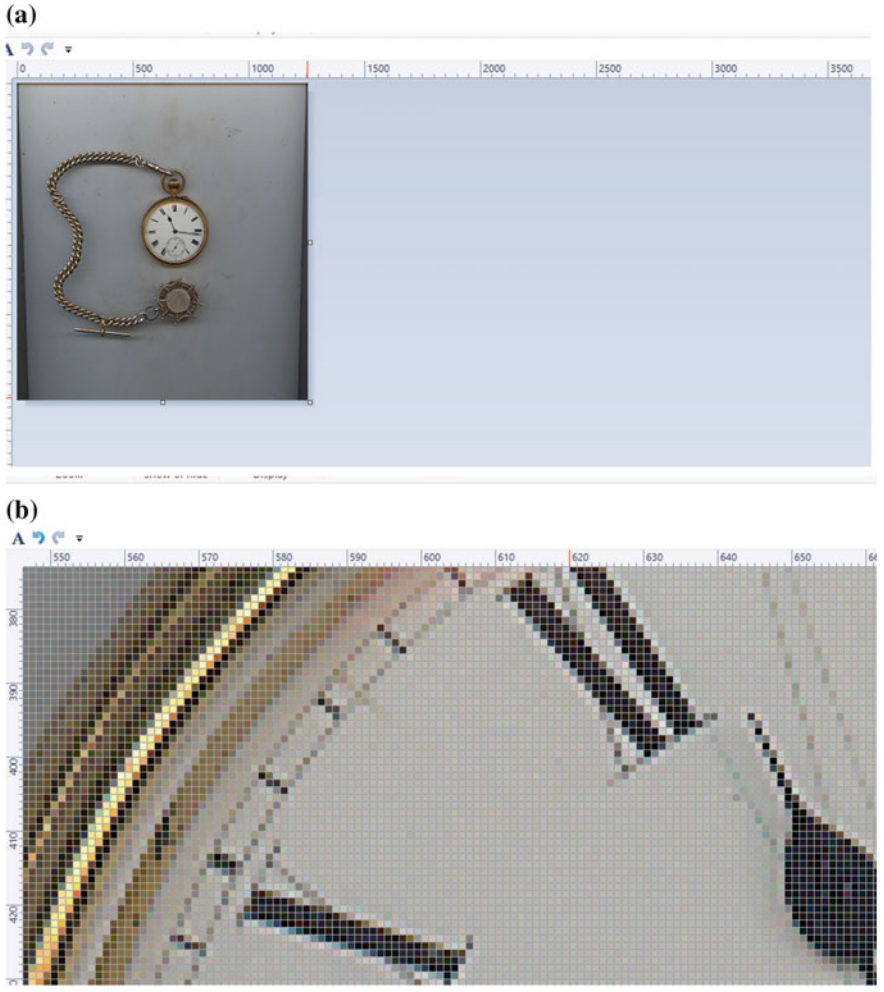


Fig. 3.7 Scanner image displayed. Object scanned at 150 pixels per inch, extent $E = 1256 \times 1374$ px $= 1.7$ Mpx. **a** Displayed in Paint window, diminished to 25% of native pitch. **b** Magnified to 800% of native pitch, with pixel grid marked

medical MRI scans, land surveys, etc. There are ongoing developments in three-dimensional scanners, for small objects. The design specification of engineering and architectural components and assemblies is increasingly available in digital form.

Computed projection from three to two dimensions is a well-developed field [9], with many elaborations of renderings [10].

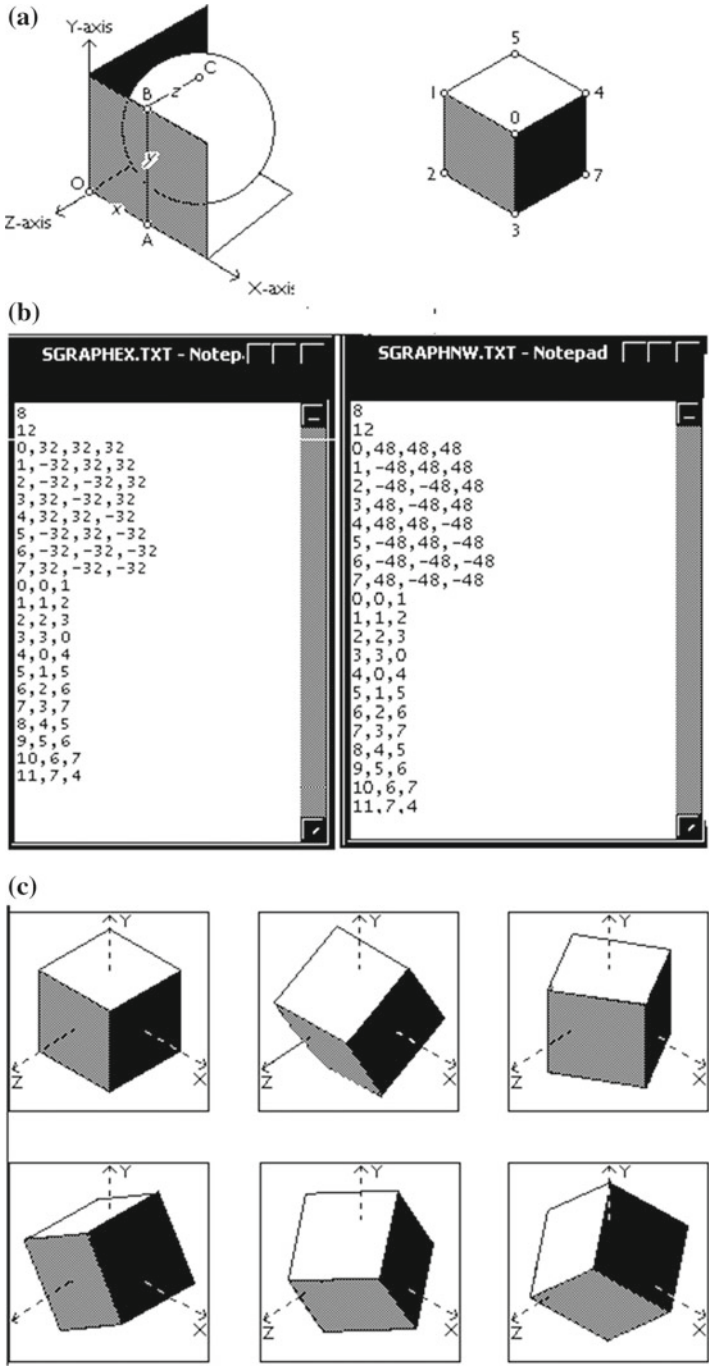


Fig. 3.8 Creating by modelling. **a** Reference frame for model. **b** Coordinate data for cube model. **c** Computed projections of cube model

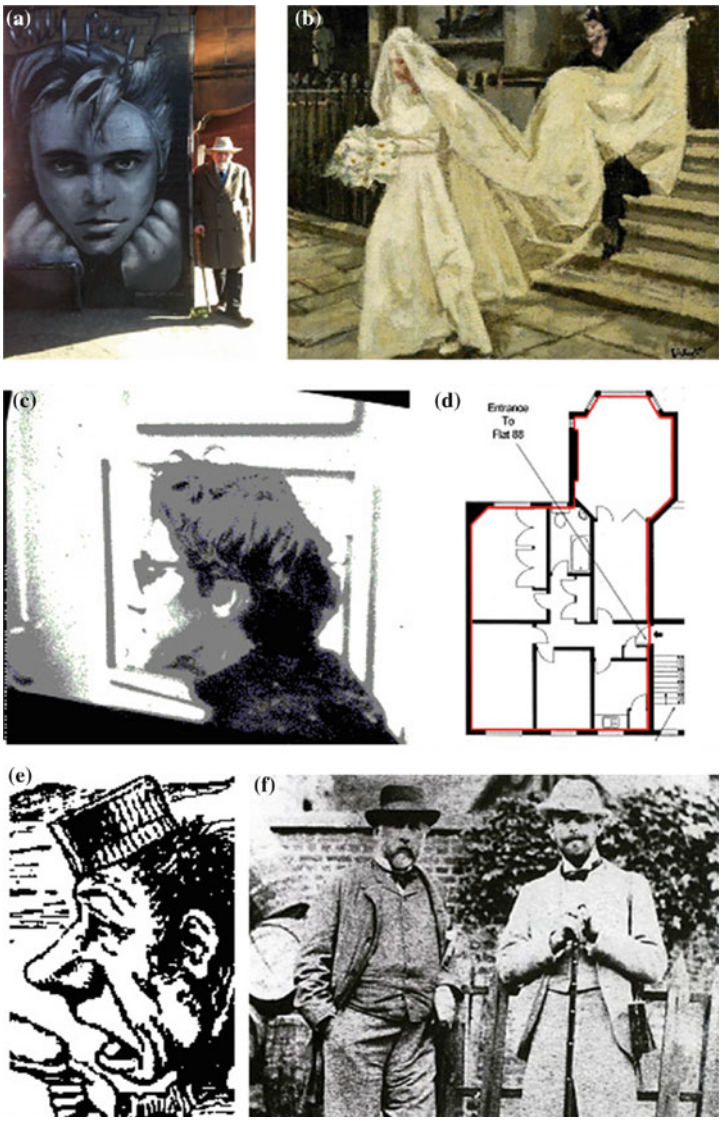


Fig. 3.9 Images created elsewhere. **a** Capture by camera from a wall-painting: Billy Fury, by an unknown artist, West Hampstead. **b** Capture by camera or scanner from a book: The Bride, painting from a newspaper photograph, by Walter Sickert. **c** Capture by scanner from a 35 mm film negative: Michael Horovitz by Alan Parkin. **d** Capture by scanner from an A4 original: apartment plan. **e** Capture by scanner from a book: Tenniel's Carpenter from *Through the Looking-glass*. **f** Capture by download from a website: Photograph of Degas and Sickert 1885

3.6 Hijacking an Image Created Elsewhere

Any image created elsewhere and accessible here can be captured in digital form by camera or scanner. And any image on the Web can be downloaded (subject to copyright or other restrictions). Figure 3.9 shows some typical cases.

References

1. Microsoft paint. https://en.wikipedia.org/wiki/Microsoft_Paint
2. Fredrik Lundh (effbot). http://effbot.org
3. Digital camera. https://en.wikipedia.org/wiki/Digital_camera
4. Gamma-correction. https://en.wikipedia.org/wiki/Gamma_correction
5. Poynton C, Frequently asked questions about gamma. www.poynton.ca/notes/color/GammaFQA.html
6. Hoffmann G, The gamma question. www.docs-hoffmann.de/gamquest18102001.pdf
7. Middle-gray. https://en.wikipedia.org/wiki/Grayscale
8. Image Scanner. https://en.wikipedia.org/wiki/Image_scanner
9. Parkin A (2016) Digital imaging primer. Springer, Heidelberg
10. 3D Computer Graphics. https://en.wikipedia.org/wiki/3D_computer_graphics

Chapter 4
Storing a Digital Image

4.1 Storing an Image as a File

Each of the methods of creating a digital image described in Chap. 3, (and of transforming an image in Chaps. 5 and 7), concludes with a display of the image, usually in an editor such as Paint. To store an image, the option is as follows:

File
—Save as
——Filename
———Save as type.

The stored image is then accessible as a file with the chosen filename and extension.

4.2 Image File

Computer storage is organized as nested *directories* or folders, containing *files*, each with a *filename* and *extension*. The extension indicates the type of file, each of which has a specific *file format* [1]. We can access stored files by means of a file manager utility, such as Microsoft File Explorer, which opens, closes, copies, pastes or deletes files.

A file holds a sequence of *fields*; a field holds one or more *bytes*; and a byte holds eight *binary digits* 0 or 1. The file format defines the position of each field, and what each field represents. For example, Fig. 4.1 shows the .BMP image file format. For user convenience, a file is usually displayed in hexadecimal digits 0 to F [2], so a byte is written as two hex digits 00 to FF. For example, Fig. 4.2 shows a simple image and its .BMP file.

4.3 File Format .BMP

.BMP (for bitmap) is a venerable image file format developed by Microsoft for Windows operating systems, in successive versions [3]. It is an uncompressed format, and has large file size. It is a simple format, very widely available for ordinary use,

A. Parkin, *Computing Colour Image Processing*,
https://doi.org/10.1007/978-3-319-74076-8_4

Group	*Field*	*Type*	*dec.*	*hex.*	*Remarks*
FileHeader			1-14	01-0E	File type, properties
	bfType	integer	1-2	01-02	Always 'BM'
	bfSize	long	3-6	03-06	File length, bytes
	bfReserved1	integer	7-8	07-08	Always 0
	bfReserved2	integer	9-10	09-0A	Always 0
	bfOffBits	long	11-14	0B-0E	Header length, bytes
InfoHeader			15-54	0F-36	Image size, properties
	biSize	long	15-18	0F-12	Infoheader len., bytes
	biWidth	long	19-22	13-16	Image width, pixels
	biHeight	long	23-26	17-1A	Image height, pixels
	biPlanes	integer	27-28	1B-1C	Always 1
	biBitCount	integer	29-30	1D-1E	Bits per pixel = 24
	biCompression	long	31-34	1F-22	Not compr. = 0
	biSizeImage	long	35-38	23-26	Image data len., bytes
	biXPelsPerMeter	long	39-42	27-2A	Printer p.p.m., horiz.
	biYPelsPerMeter	long	43-46	2B-2E	Printer p.p.m., vert.
	biClrUsed	long	47-50	2F-32	Always 0
	biClrImportant	long	51-54	33-36	Clrs considered imp.
Image data			55-	37-	In scan sequence
	aBitmapBits[0]	byte	55	37	Pixel 0, blue value
	aBitmapBits[1]	byte	56	38	Pixel 0, green value
	aBitmapBits[2]	byte	57	39	Pixel 0, red value
	aBitmapBits[3]	byte	58	3A	Pixel 1, blue value
	aBitmapBits[4]	byte	59	3B	Pixel 1, green value
	aBitmapBits[5]	byte	60	3C	Pixel 1, red value
	etc. ...				

Fig. 4.1 The .BMP 24-bit format, with two headers and a body as long as it takes

though other formats with compression are often preferred. Figure 4.1 shows the structure. Figure 4.3 shows a typical camera image stored in .BMP format displayed in a Paint editor, at 100% size and magnified to 400% to show the pixel detail.

Offset (h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00000000	42	4D	F6	00	00	00	00	00	00	00	36	00	00	00	28	00
00000010	00	00	08	00	00	00	08	00	00	00	01	00	18	00	00	00
00000020	00	00	C0	00	00	00	C3	0E	00	00	C3	0E	00	00	00	00
00000030	00	00	00	00	00	00	FF	FF	FF	00	00	00	FF	FF	FF	00
00000040	00	00	FF	FF	FF	00	00	00	FF	FF	FF	FF	00	00	00	00
00000050	00	FF	FF	FF	00	00	00	FF	FF	FF	00	00	00	FF	FF	FF
00000060	00	00	00	FF	FF	FF	FF	FF	FF	00	00	00	FF	FF	FF	00
00000070	00	00	FF	FF	FF	00	00	00	FF	FF	FF	00	00	00	00	00
00000080	00	FF	FF	FF	1D	E6	B5	1D	E6	B5	1D	E6	B5	FF	FF	FF
00000090	00	00	00	FF	FF	FF	FF	FF	FF	00	00	00	1D	E6	B5	00
000000A0	FF	00	1D	E6	B5	00	00	00	FF	FF	FF	00	00	00	00	00
000000B0	00	FF	FF	FF	1D	E6	B5	1D	E6	B5	1D	E6	B5	FF	FF	FF
000000C0	00	00	00	FF	FF	FF	FF	FF	FF	00	00	00	FF	FF	FF	00
000000D0	00	00	FF	FF	FF	00	00	00	FF	FF	FF	00	00	00	00	00
000000E0	FF	FF	FF	FF	00	00	00	FF	FF	FF	00	00	00	FF	FF	FF
000000F0	00	00	00	FF	FF	FF										

Fig. 4.2 A simple image and its .BMP file in hexadecimal dump, two hex digits per byte. In multi-byte fields, most-significant bytes are to the right, least-significant left. Header fields, above the thick line, are: two-byte "magic cookie" 42 4Dh = ASCII BM; four-byte file size F6h = 246d; two two-byte fields ignored; four-byte header length 36h = 54d; four-byte info header 28h = 40d; four-byte image width 08h = 8d; four-byte image height 08h = 8d; two-byte planes field 01h = 1d; two-byte bits per pixel 18h = 24d; four-byte compression 00h = 0d; image data length C0h = 192d; two four-byte pixels per meter C3 0Eh = 49934d; four-byte colors used 00h = 0d; and four-byte colors important 00h = 0d. After the thick line: sixty-four triples of one-byte fields showing the (B, G, R) values of the image pixels in scan-sequence starting at bottom left

4.4 File Format .GIF

.GIF (for graphic interchange format) was developed in 1987 for 8-bit displays limited to 256 colours [4]. It has been largely replaced by .PNG, but is still widely available for website use. It has lossless compression, hence small file size. Figure 4.4 shows a typical camera image stored in .GIF format displayed in a Paint editor, at 100%

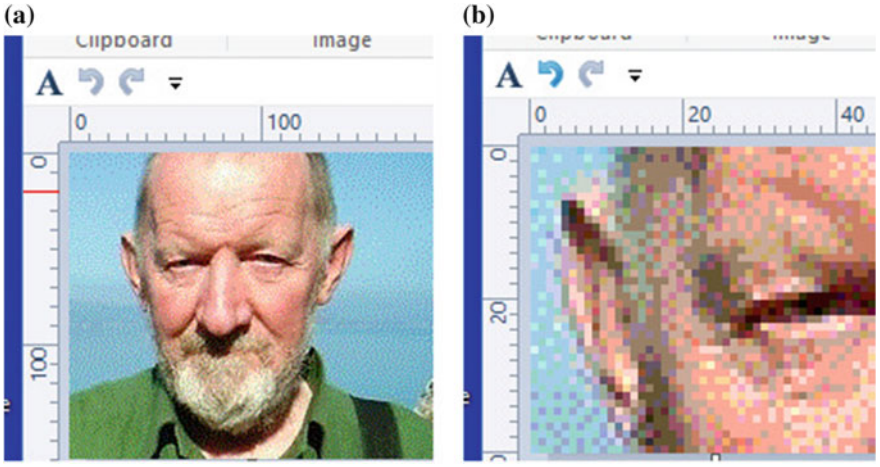


Fig. 4.3 BMP format. **a** Image 190 × 160 px stored in .BMP format; file size 89 KB. **b** Magnified to 400%

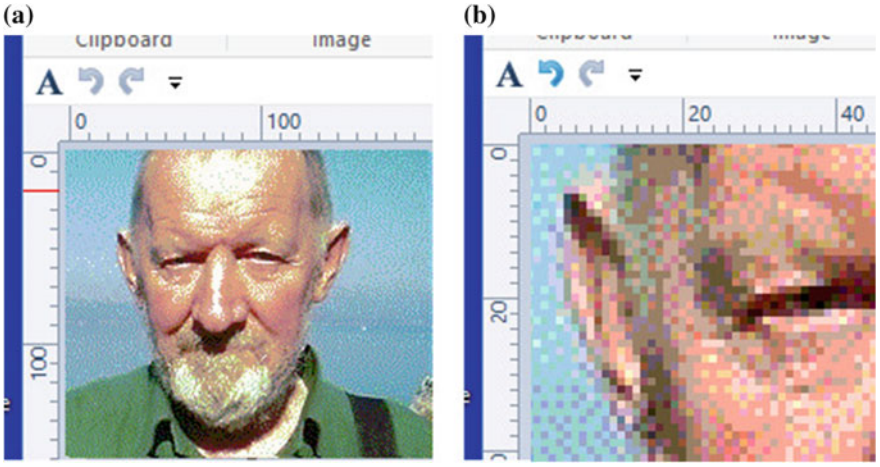


Fig. 4.4 GIF format. **a** Image 190 × 160 px stored in .GIF format; file size 20 KB. **b** Magnified to 400%

size and magnified to 400% to show the pixel detail. The original sRGB colours are severely altered.

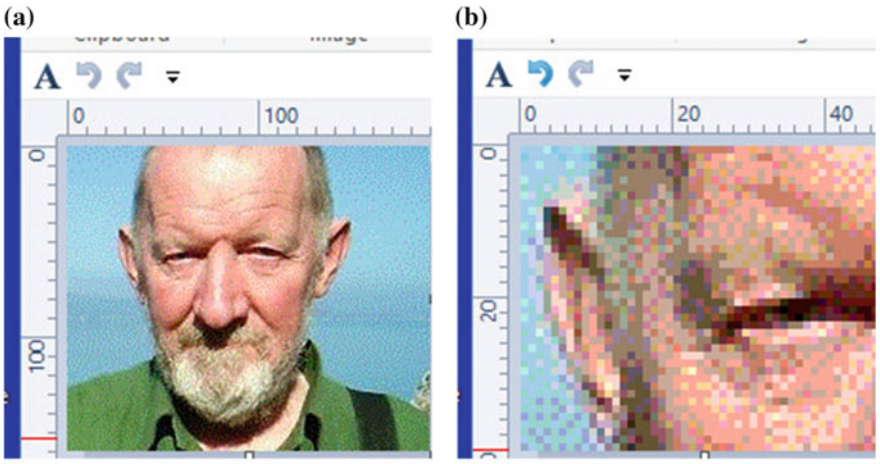


Fig. 4.5 PNG format. **a** Image 190 × 160 px stored in .PNG format; file size 23 KB. **b** Magnified to 400%

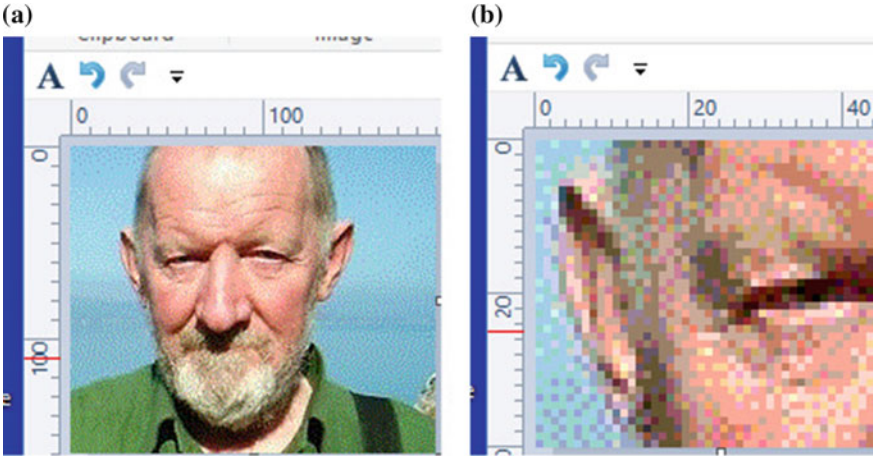


Fig. 4.6 TIF format. **a** Image 190 × 160 px stored in .TIF format; file size 26 KB. **b** Magnified to 400%

4.5 File Format .PNG

.png (for portable network graphics) was developed in 1996 as a replacement for .BMP and .GIF [5]. It has lossless compression, hence small file size. Figure 4.5 shows a typical camera image stored in .png format displayed in a Paint editor, at 100% size and magnified to 400% to show the pixel detail.

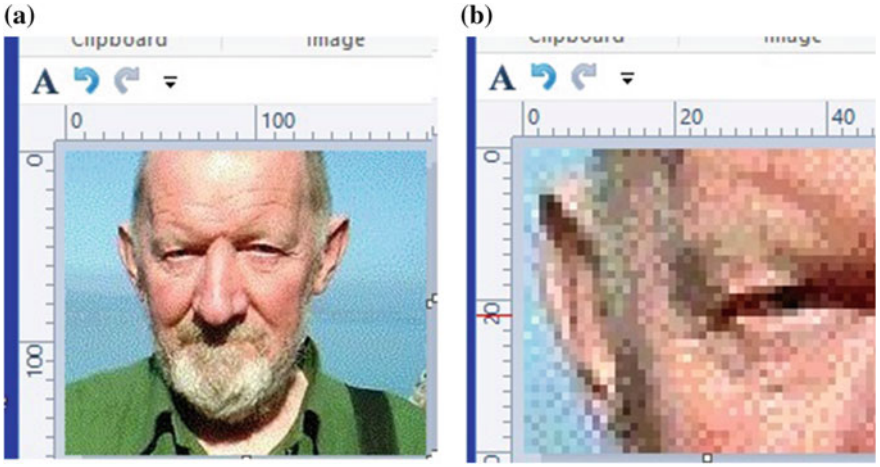


Fig. 4.7 JPG format, high quality. **a** Image 190 × 160 px stored in .JPG format with 94% quality; file size 20 KB. **b** Magnified to 400%

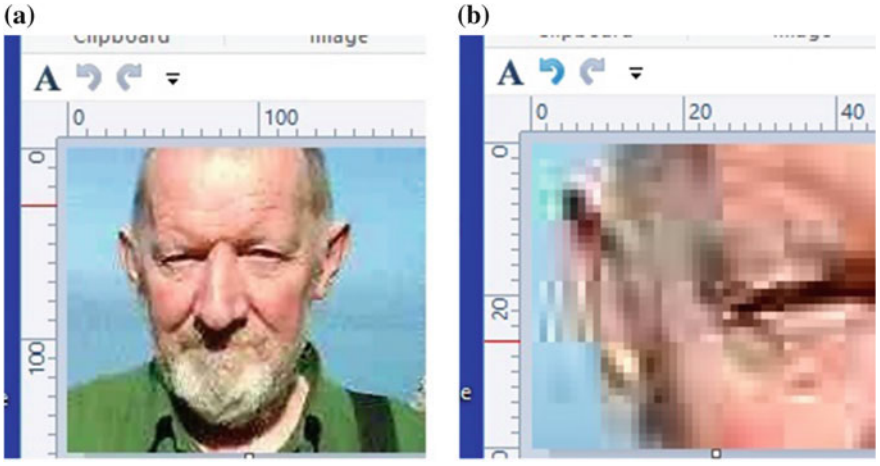


Fig. 4.8 JPG format, low quality **a** Image 190 × 160 px stored in .JPGG format with 20% quality; file size 2 KB. **b** Magnified to 400%

4.6 File Format .TIF

.TIF (for tagged image file) was developed in 1986 for desktop and commercial printing, where it remains the preferred format [6]. It has lossless compression, hence small file size, and can contain additional image information. Figure 4.6 shows a typical camera image stored in .TIF format displayed in a Paint editor, at 100% size and magnified to 400% to show the pixel detail.

4.7 File Format .JPG

.JPG (for joint photographic experts group) was developed in 1992 for making adjustable-quality compressions of camera and scanner images by removing the least visually significant image data [7]. It is very widely used to store and transfer such images. Figure 4.7 shows a typical camera image stored in .JPG format with 94% quality, and Fig. 4.8 with 20% quality.

References

1. Image file formats. https://en.wikipedia.org/wiki/Image_file_formats
2. Hexadecimal number system. https://en.wikipedia.org/wiki/Hexadecimal
3. BMP file format. https://en.wikipedia.org/wiki/BMP_file_format
4. GIF file format. https://en.wikipedia.org/wiki/GIF
5. PNG file format. https://en.wikipedia.org/wiki/Portable_Network_Graphics
6. TIF file format. https://en.wikipedia.org/wiki/TIFF
7. JPEG file format. https://en.wikipedia.org/wiki/JPEG

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.

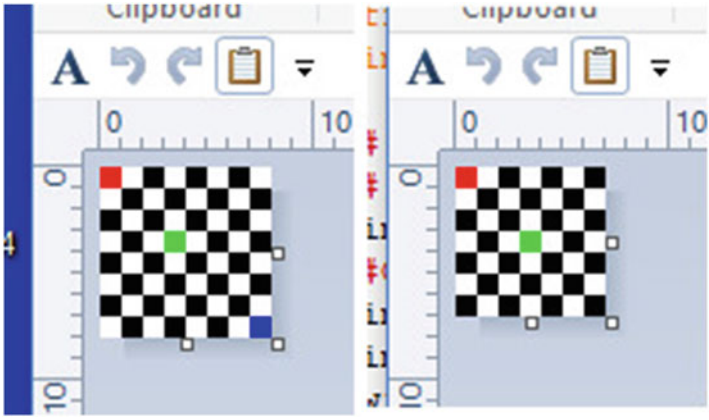


Fig. 5.1 Cropping an image. **a** Simple image 8 × 8 px, magnified in a Paint window. **b** Image cropped to 7 × 7 px, without interpolation

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 -

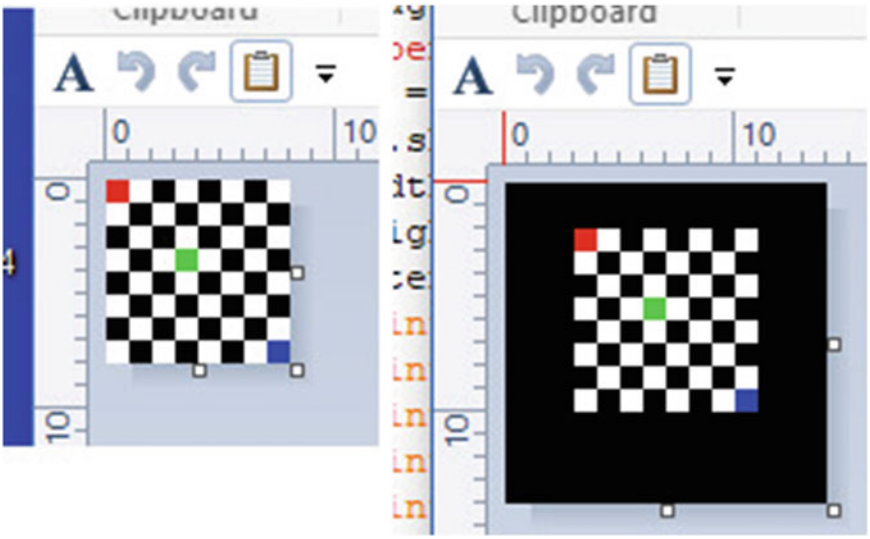


Fig. 5.3 Framing an image. **a** Simple image 8 × 8 px, magnified in a Paint window. **b** The imaged framed, to 14 × 14 px

(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 × 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 × 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 × 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 × 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:

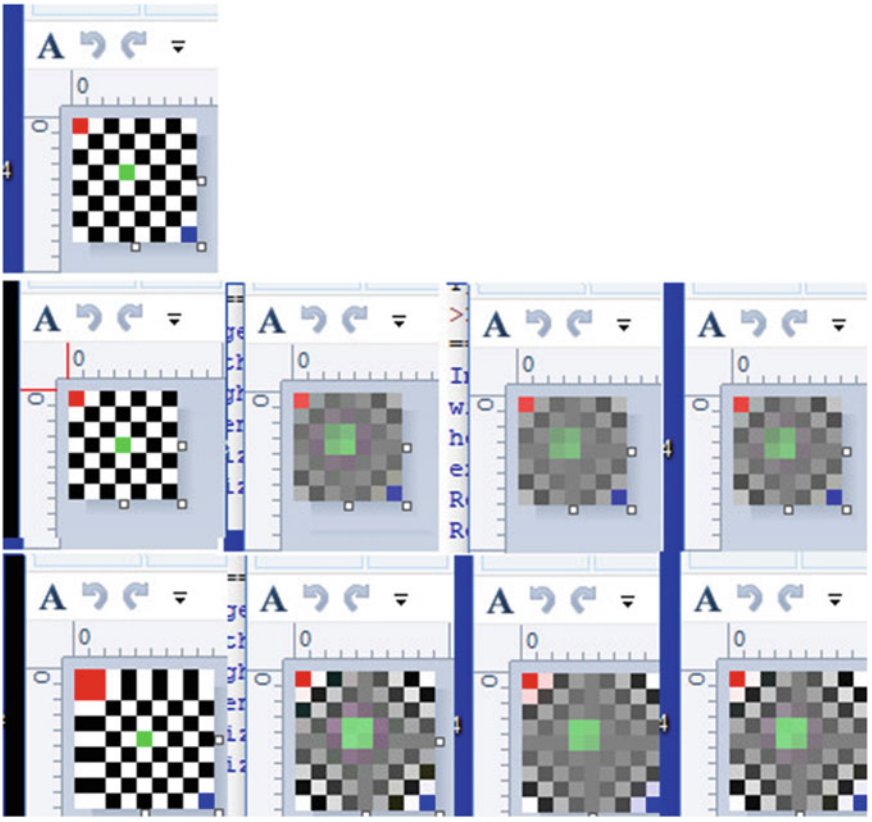


Fig. 5.4 Dilating an image. Top row: **a** Source image 8 × 8 px, magnified in Paint window. Middle row: **b** Dilated down to 7 × 7 px without interpolation. **c** With nearest neighbour interpolation. **d** With bilinear 2 × 2 interpolation. **e** With bicubic 4 × 4 interpolation. Bottom row: **f**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

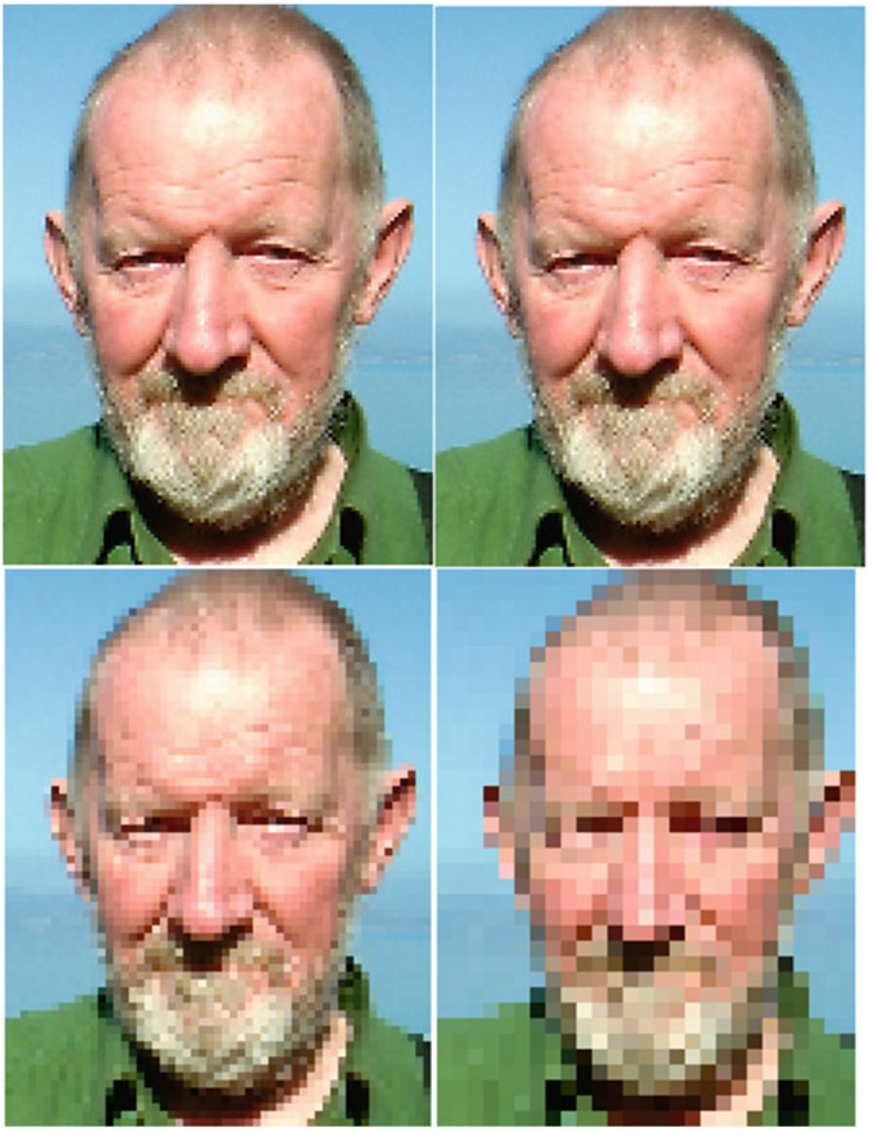


Fig. 5.5 Dilating down then up. **a** Source image 222 × 290 px (100 percent). **b** After dilating (**a**) down by factor 2–50 percent, then dilating up again by factor 2–100 percent. **c** After dilating (**a**) down by factor 4–25 percent, then dilating up again to 100 percent. **d** After dilating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

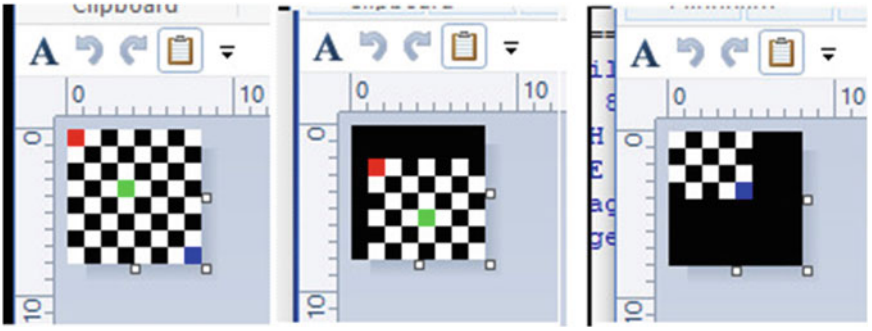


Fig. 5.6 Translating an image. **a** Simple image in original position. **b** Translated 1 px rightwards and 2 px downwards. **c** Translated 3 px leftwards and 4 px upwards

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).

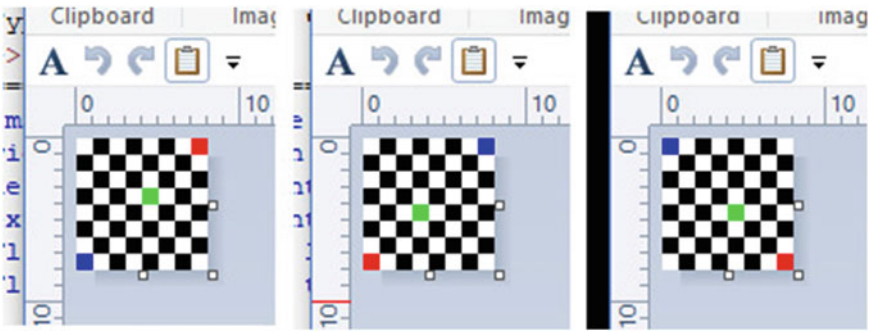


Fig. 5.7 Reflecting an image. **a** The simple image from Fig. 5.6 reflected left/right. **b** Reflected top/bottom. **c** Reflected left/right and top/bottom

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).

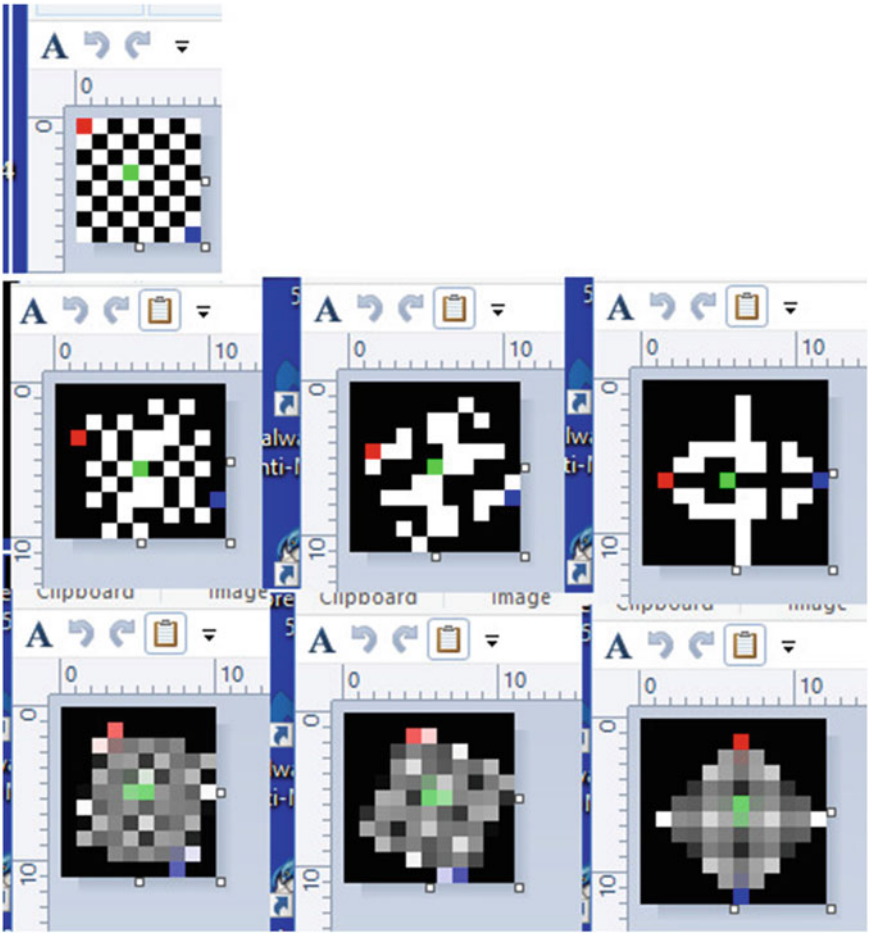


Fig. 5.8 Rotating a simple image. Top row: **a** Simple image 8 × 8 px. Middle row, without interpolation. **b** Rotated 15 degrees counterclockwise; frame expanded to 11 × 10 px. **c** Rotated by 30 degrees counterclockwise; frame expanded to 11 × 11 px. **d** Rotated by 45 degrees counterclockwise;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

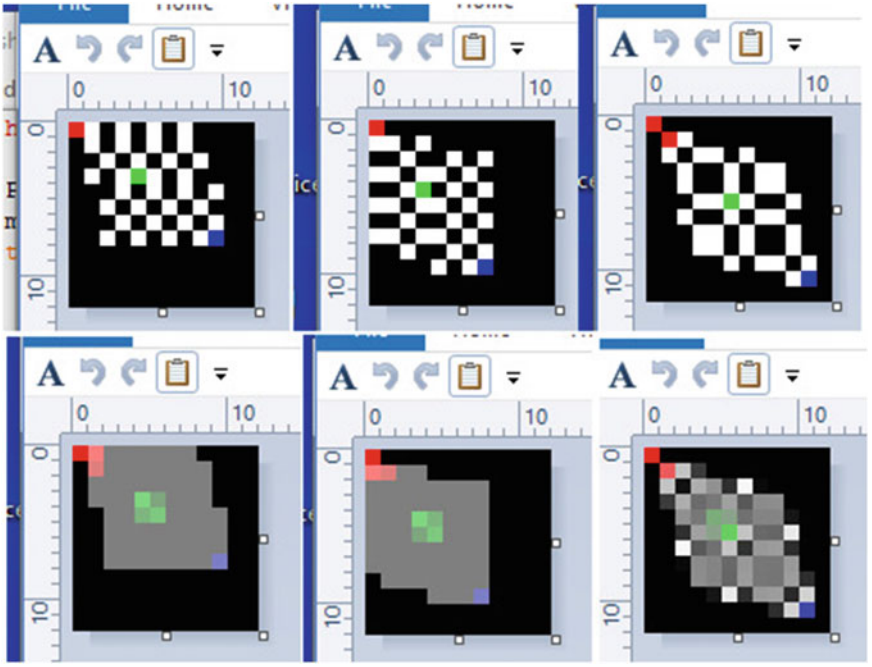


Fig. 5.10 Shearing a simple image. Top row, without interpolation: **a** Simple image 8 × 8 px sheared away from the Y-axis by 15 degrees. **b** Sheared away from the X-axis by 15 degrees. **c** Sheared away from both axes by 15 degrees. Bottom row: **d** As **a** with interpolation. **e** As **b** with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

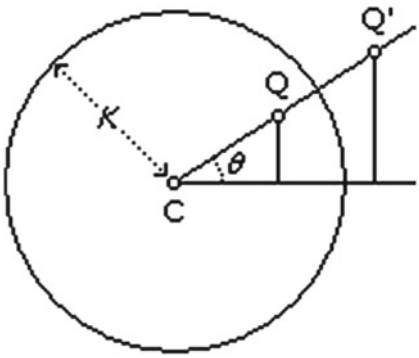


Fig. 5.12 The geometry of inverting. CQ * CQ' = K^2

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
```

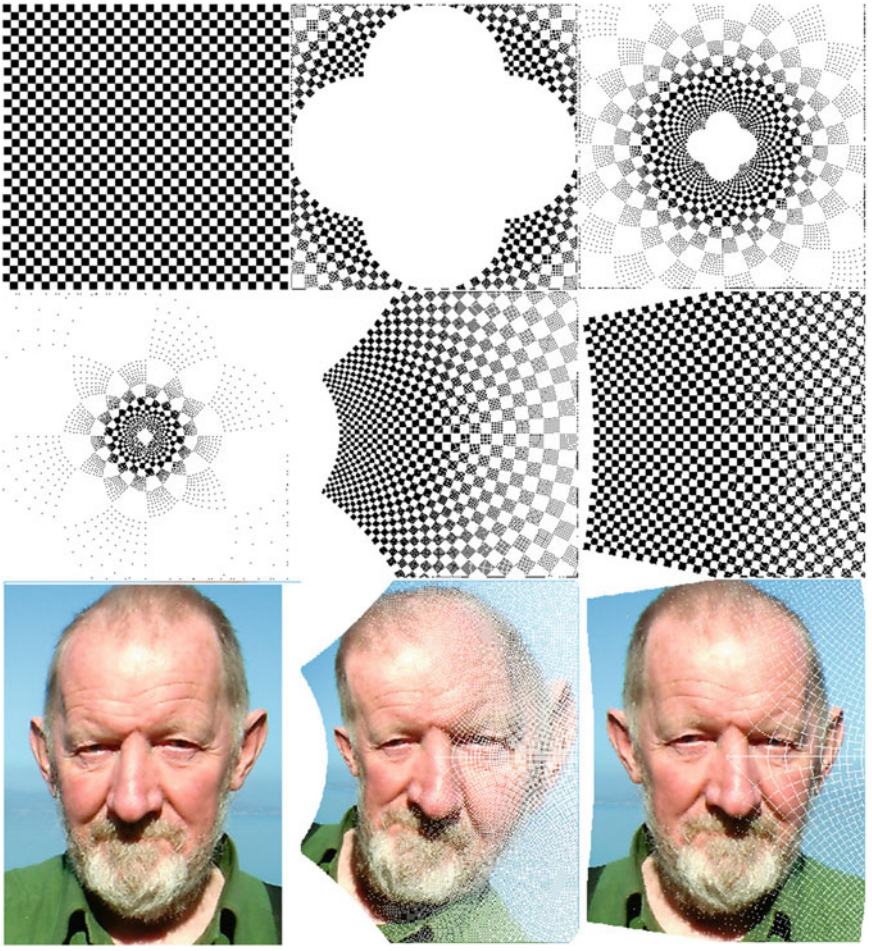


Fig. 5.13 Inverting an image. Top row: **a** Source image 288 × 288 px. **b** C = (144, 144), K = 144. **c** C = (144, 144), K = 72. Middle row: **d** C = (144, 144), K = 36. **e** C = (−144, 144), K = 288. **f** C = (−288, 144), K = 576. Bottom row: **g** Camera source image 375 × 480 px. **h** C =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

Chapter 6
Transforming Image Colours

6.1 Colour Palettes

sRGB colour space has $S = 256$ steps of intensity on all three axes, hence diversity $D = S^3 = 16.7$ million distinct colours, and colour resolution $COLRES = 1/16.7$ million, one colour in 16.7 million. But for many purposes, such fine colour resolution is unnecessary and inconvenient. We can define a series of subspaces of sRGB, or *palettes*, by setting fewer steps of intensity on the three axes, thus reducing the diversity of colours and coarsening the colour resolution. We can then convert an sRGB image to a smaller palette to give a simpler and more manageable result.

A subspace or palette of sRGB has $S < 256$ steps of intensity on each of the R G B axes, and is here labelled P(S). For example, Fig. 6.1a shows the full sRGB cube model; (b) the minimal palette P2 as a sparse cube, covering the same range as sRGB but with only $S = 2$ steps of intensity on each axis. It has diversity $D = 2^3 = 8$ distinct colours. Figure 6.1c shows the P3 palette with $S = 3$ steps of intensity on each axis, hence diversity $D = 3^3 = 27$ distinct colours. Clearly, the series can be continued as P4, P5, P6, ... up to P256, which is the full sRGB palette. In practice, anything beyond P6 is rarely needed.

To convert an image to a restricted palette the steps are as follows:

Select the source image file from storage.
Open it.
Select the palette wanted.
Do the conversion.
Save the converted image to storage under a new filename.

Various image editors offer various conversions of images to various restricted palettes. For example, Microsoft Paint offers Save-as .BMP Monochrome, .BMP 16-colour, .BMP 256-colour and .GIF 256-colour. We can write a script which will perform conversions to the P-series of palettes, as listed below. For example, Fig. 6.2 shows original sRGB images (full gamut and camera image) and their conversions to palettes P2–P6 by this script.

A. Parkin, *Computing Colour Image Processing*,
https://doi.org/10.1007/978-3-319-74076-8_6

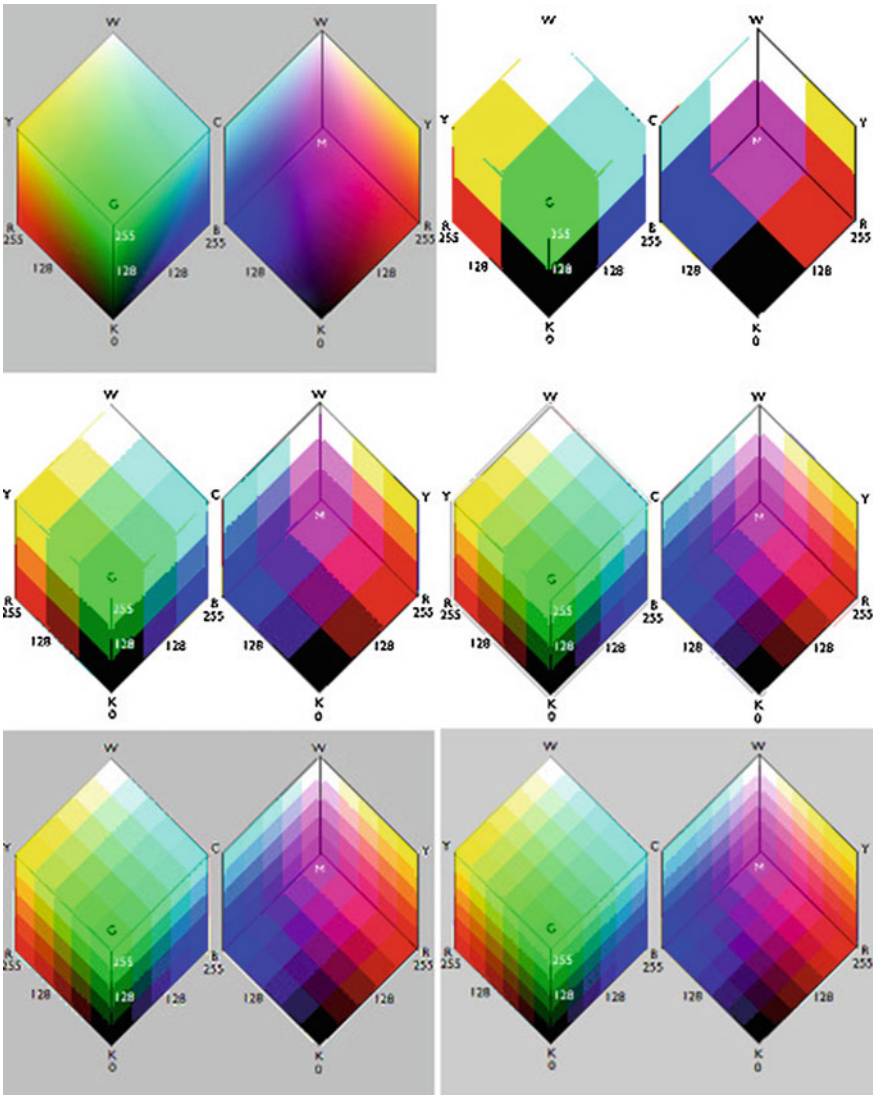


Fig. 6.1 Palette subsets of sRGB. **a** Full sRGB cube model, front and back. **b** Palette P2 sparse cube model: it has $S = 2$ steps on each of the three axes, hence diversity $D = 2^3 = 8$ colours, the vertices of the cube. **c** Palette P3, with $S = 3$ steps per axis, hence $D = 3^3 = 27$ colours. **d** Palette P4, $D = 4^3 = 64$ colours. **e** Palette P5, $D = 5^3 = 125$ colours. **f** Palette P6, $D = 6^3 = 216$ colours

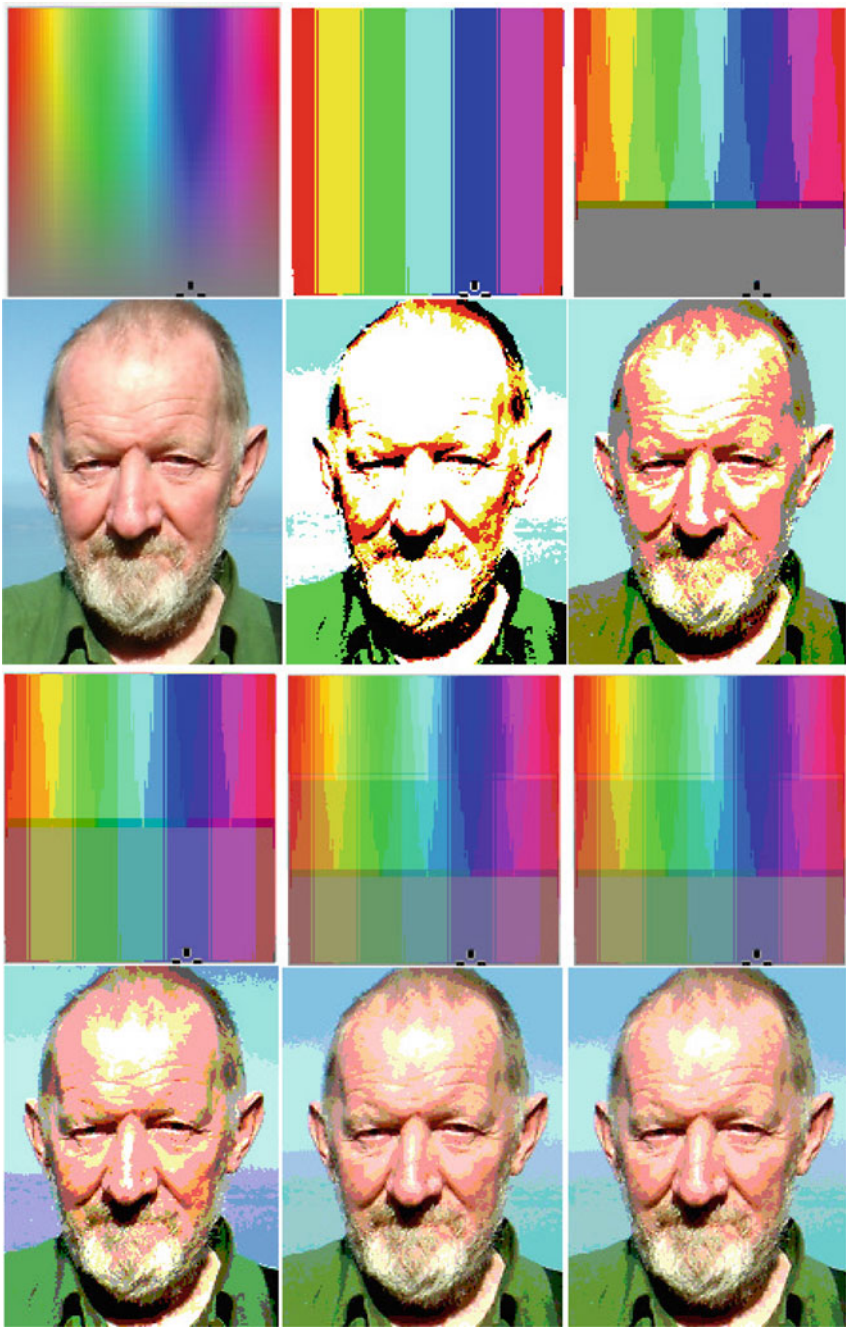


Fig. 6.2 Examples of images converted to restricted palettes. **a** Original images: above is the sRGB gamut as displayed in the Paint colour selector; below is a typical camera image. **b a** converted to palette P2. **c a** converted to palette P3. **d** converted to palette P4. **e** Converted to palette P5. **f** Converted to palette P6

```
# pypale: Change colour resolution from full sRGB
# to palette P2 P3 P4 P5 or P7 P6. Use PIL to ask
# .bmp image filename in working directory, open it,
# show it, and get pix = xy array of rgb colour
# tuples. Ask which palette wanted, and do the one
# wanted. Apply reduced values to image, and show
# in Paint, to be saved where wanted.
# Written by Alan Parkin 2017.

from PIL import Image
# import numpy as np
import os, sys

#Get filename of .bmp image which is in working directory
imagefilename = raw_input("Image filename? ")
#Open it and show it
imrgb = Image.open(imagefilename)
imrgb.show()
print "imrgb sizet34b.bmp", imrgb.size
width = imrgb.size[0]
height = imrgb.size[1]
extent = width * height
print "width", width
print "height", height
print "extent", extent

# Call PIL load for this image, to get xy array of rgb tuples.
pix = imrgb.load()

# Convert xy array of tuples to xy array of lists.
pixlist = []
for y in range(height):
    for x in range(width):
        for j in range(3):
            thisval = pix[x,y][j]
            pixlist.append(thisval)

# Ask which colres to do.
palette = raw_input("Palette P2 P3 P4 P5 P6 ? ")
print "palette ", palette

# Reduce pixlist values to palette.
if palette == 'P2':
    for i in range (extent * 3):
        if pixlist[i] <= 127:
```

```
            pixlist[i] = 0
        if pixlist[i] >= 128:
            pixlist[i] = 255

if palette == 'P3':
    for i in range (extent * 3):
        if pixlist[i] <= 85:
            pixlist[i] = 0
        if pixlist[i] >= 86 and pixlist[i] <= 170:
            pixlist[i] = 127
        if pixlist[i] >= 171:
            pixlist[i] = 255

if palette == 'P4':
    for i in range (extent * 3):
        if pixlist[i] <= 64:
            pixlist[i] = 0
        if pixlist[i] >= 65 and pixlist[i] <= 127:
            pixlist[i] = 85
        if pixlist[i] >= 128 and pixlist[i] <= 191:
            pixlist[i] = 170
        if pixlist[i] >= 192:
            pixlist[i] = 255

if palette == 'P5':
    for i in range (extent * 3):
        if pixlist[i] <= 51:
            pixlist[i] = 0
        if pixlist[i] >= 52 and pixlist[i] <= 102:
            pixlist[i] = 64
        if pixlist[i] >= 103 and pixlist[i] <= 153:
            pixlist[i] = 127
        if pixlist[i] >= 154 and pixlist[i] <= 204:
            pixlist[i] = 191
        if pixlist[i] >= 205:
            pixlist[i] = 255


if palette == 'P6':
    for i in range (extent * 3):
        if pixlist[i] <= 42:
            pixlist[i] = 0
        if pixlist[i] >= 43 and pixlist[i] <= 85:
            pixlist[i] = 51
        if pixlist[i] >= 86 and pixlist[i] <= 127:
```

```
            pixlist[i] = 102
        if pixlist[i] >= 128 and pixlist[i] <= 170:
            pixlist[i] = 153
        if pixlist[i] >= 171 and pixlist[i] <= 212:
            pixlist[i] = 204
        if pixlist[i] >= 213:
            pixlist[i] = 255

# Apply reduced values to image.
m = width * 3
for y in range(height):
    for x in range(width):
        pix[x,y] = (pixlist[(3*x)+(m*y)], pixlist[(3*x)+(m*y)+1],
        pixlist[(3*x)+(m*y)+2])

# Show reduced image.
imrgb.show()

print "Done"
```

6.2 Neutral Palettes

A *neutral palette* or greyscale is one where ($R = G = B$). To convert an image to a neutral palette, several somewhat different formulas have been used [1]. The Python Imaging Library uses the ITU-R 6021-2 formula:

$$\text{Grey (R' = G' = B')} = ((\text{R} * 299/1000) + (\text{G} * 587/1000) + (\text{B} * 114/1000))$$

Figure 6.3 shows the sRGB cube model converted to neutral palette N256. It has $S = 256$ steps of intensity I, diversity $D = S = 256$ and $COLRES = 1/256$, one grey in 256. Indeed a neutral palette reduces the sRGB cube model to just one axis, the body diagonal of the cube from (0,0,0) black to (255,255,255) white. A series of neutral greyscales can be made by taking fewer than $S = 256$ steps of intensity on the diagonal axis. For example, a minimal neutral greyscale N2 has $S = 2$, hence diversity $D = 2$ colours, just black and white. Figure 6.4 shows neutral greyscales N256, N2, N3, N4, N5 and N6. Clearly, the series can be continued up to N256 which is the full greyscale.

We can write a script to convert any sRGB .bmp image to a neutral palette, as listed below. Figure 6.5 shows original gamut and camera images converted to neutral palettes N256, N2, N3, N4, N5 and N6

```
# pyneut: Change colour resolution from full sRGB to
# neutral palette N2 N3 N4 N5 N6 or N256(= greyscale),
# show it in Paint (to be saved if wanted).
# Written by Alan Parkin 017.
```

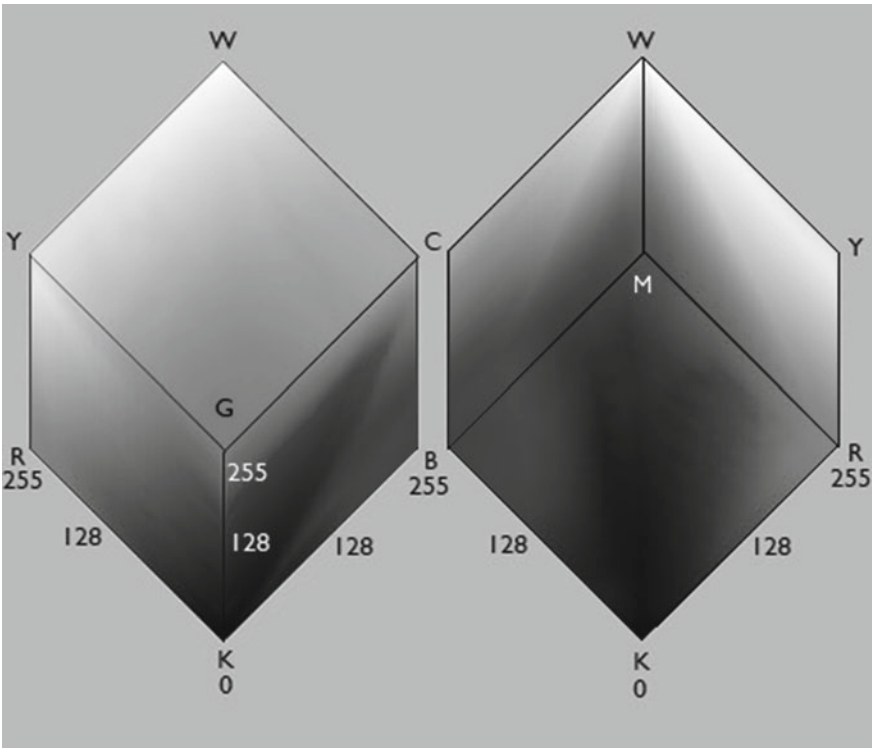


Fig. 6.3 sRGB cube model converted to neutral palette N256. **a** Front view **b** Back view

```
from PIL import Image
# import numpy as np
import os, sys

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

# Call PIL load for this image, to get xy array of rgb tuples.
pix = imrgb.load()

# Convert xy array of tuples to xy array of lists.
pixlist = []
```

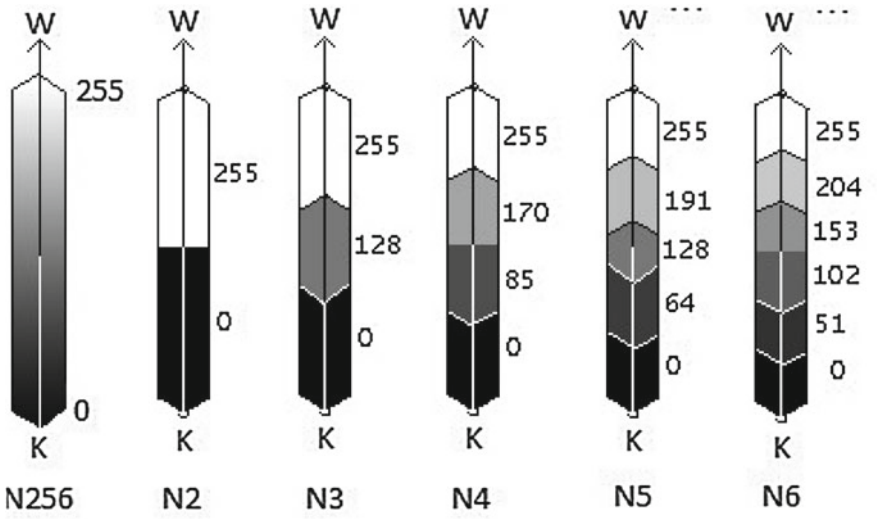


Fig. 6.4 Neutral palette subsets of sRGB. **a** Neutral palette N256, the greyscale of sRGB, with S = 256 steps along the body diagonal of the sRGB cube model, from (0, 0, 0) black to (255, 255, 255) white. Diversity $D = S = 256$. **b** Neutral palette N2, with $D = S = 2$ **(c)–(f)** Neutral palettes N2, N3, N4, N5 and N6

```
for y in range(height):
    for x in range(width):
        thisval = pix[x,y]
        pixlist.append(thisval)

# Convert each rgb triple in pixlist to neutral, as
# factors = (0.30r + 0.59g + 0.1b) then
# neut = (factors, factors, factors)
pixnlist = []
for j in range(extent):
    factors = int((pixlist[j][0]*0.30) +
        (pixlist[j][1]*0.59) + (pixlist[j][2]*0.1))
    pixnlist.append(factors)
    pixnlist.append(factors)
    pixnlist.append(factors)

# Ask which neutral palette wanted.
palette = raw_input("Neutral palette N2 N3 N4 N5 N6 N256(greyscale) ? ")
print "neutral palette ", palette

# Reduce pixlist values to palette.
if palette == 'N2':
    for i in range (extent * 3):
        if pixnlist[i] <= 127:
            pixnlist[i] = 27
        if pixnlist[i] >= 128:
            pixnlist[i] = 255
```

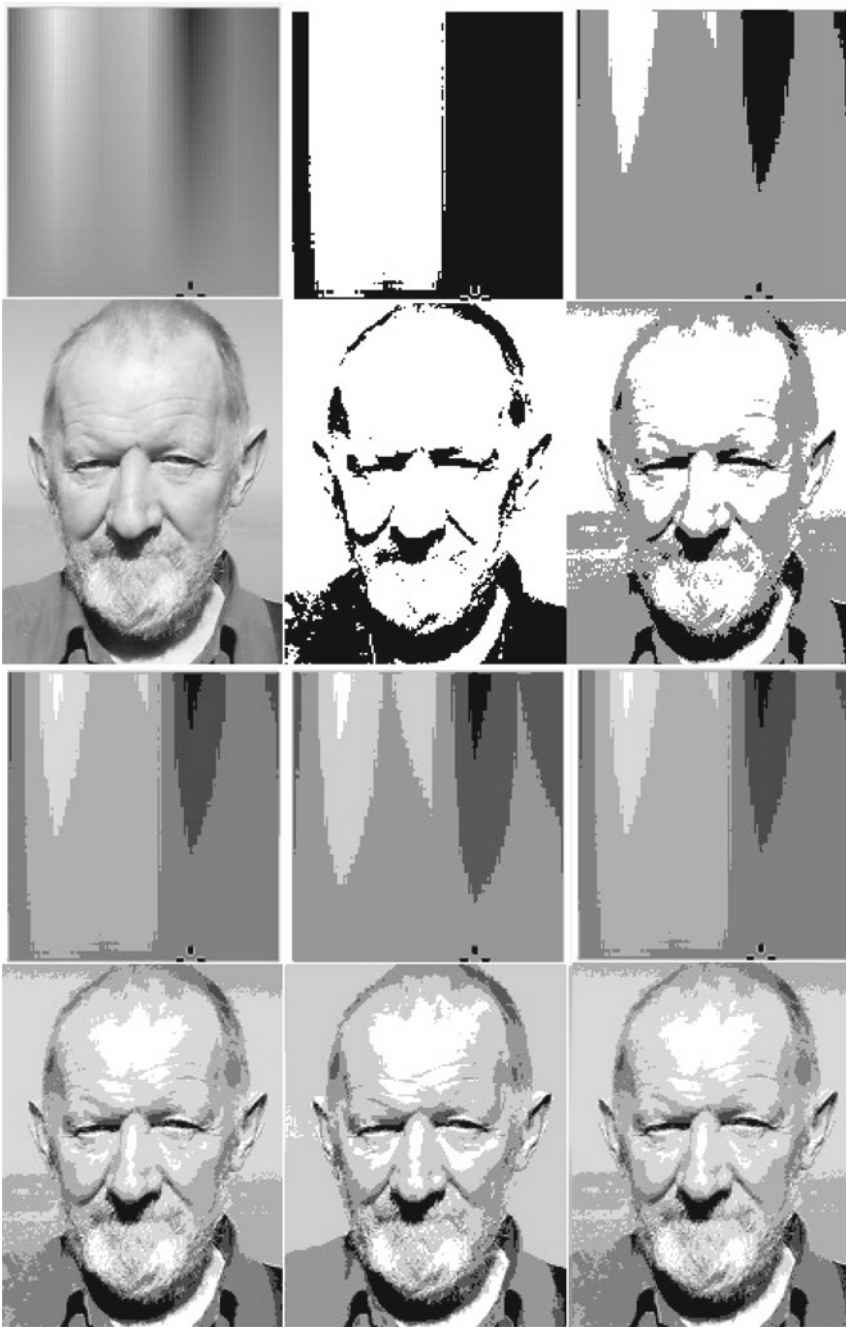


Fig. 6.5 Examples of images converted to neutral palettes. **a** Images converted to N256 full greyscale. **b** Converted to N2. **c** Converted to N3. **d** Converted to N4. **e** Converted to N5. **f** Converted to N6

```
if palette == 'N3':
    for i in range (extent * 3):
        if pixnlist[i] <= 64:
            pixnlist[i] = 27
        if pixnlist[i] >= 65 and pixnlist[i] <= 191:
            pixnlist[i] = 127
        if pixnlist[i] >= 1921:
            pixnlist[i] = 255

if palette == 'N4':
    for i in range (extent * 3):
        if pixnlist[i] <= 43:
            pixnlist[i] = 27
        if pixnlist[i] >= 44 and pixnlist[i] <= 127:
            pixnlist[i] = 85
        if pixnlist[i] >= 128 and pixnlist[i] <= 212:
            pixnlist[i] = 171
        if pixnlist[i] >= 212:
            pixnlist[i] = 255

if palette == 'N5':
    for i in range (extent * 3):
        if pixnlist[i] <= 32:
            pixnlist[i] = 27
        if pixnlist[i] >= 33 and pixnlist[i] <= 96:
            pixnlist[i] = 64
        if pixnlist[i] >= 97 and pixnlist[i] <= 160:
            pixnlist[i] = 127
        if pixnlist[i] >= 161 and pixnlist[i] <= 223:
            pixnlist[i] = 191
        if pixnlist[i] >= 224:
            pixnlist[i] = 255



if palette == 'N6':
    for i in range (extent * 3):
        if pixnlist[i] <= 92:
            pixnlist[i] = 27
        if pixnlist[i] >= 93 and pixnlist[i] <= 149:
            pixnlist[i] = 124
        if pixnlist[i] >= 150 and pixnlist[i] <= 187:
            pixnlist[i] = 169
        if pixnlist[i] >= 188 and pixnlist[i] <= 217:
            pixnlist[i] = 203
        if pixnlist[i] >= 218 and pixnlist[i] <= 243:
            pixnlist[i] = 231
        if pixnlist[i] >= 244:
            pixnlist[i] = 255

# Apply reduced values to image.
m = width * 3
for y in range(height):
    for x in range(width):
        pix[x,y] = (pixnlist[(3*x)+(m*y)], pixnlist[(3*x)+(m*y)+1],
        pixnlist[(3*x)+(m*y)+2])
```

```
# Convert image to N256 (full greyscale).
if palette == 'N256':
    imL = imrgb.convert("L")
    imL.show()

# Show reduced image.
if palette != 'N256':
    imrgb.show()
```

6.3 Halftone Palettes

Halftoning is an ancient trick to overcome the limitations of subtractive printing, which deposits ink or no-ink at each location of its paper or other sheet without graduation of intensity (see Chap. 8). Historically, halftoning has been done by hatching, intaglio engraving, mezzotint engraving, lithographic dotting, screened photoengraving and screened litho-offset, in black-and-white and in colour [2].

Digital halftoning can be done by grouping image pixels into cells to create small dots of varying size or spacing, which appear as graduation of intensity [3, 4]. Whereas a neutral palette converts an sRGB coloured image to a scale of greys, a *halftone palette* further converts a neutral image to black-only pixels.

Any sRGB image can be converted to a black-only halftone palette by the following script, which uses the ingenious Floyd–Steinberg error-distributed formula [5] in the Python Imaging Library:.

```
# pyhalft: Change colour resolution from full
# sRGB to halftone palette H256. Convert to
# mode "1" with "dither=1"  and show in PAINT
# (save here if wanted).
# Written by Alan Parkin 2017.

from PIL import Image
# import numpy as np
import os, sys

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

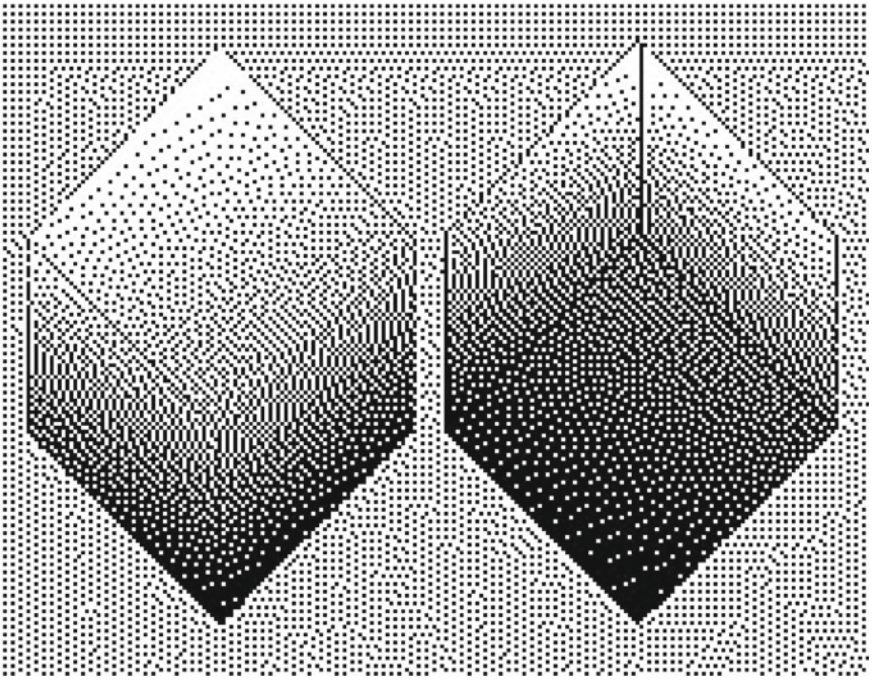


Fig. 6.6 sRGB cube model converted to halftone palette H256. Figure 6.3 after halftoning from varied intensity greys to black-only

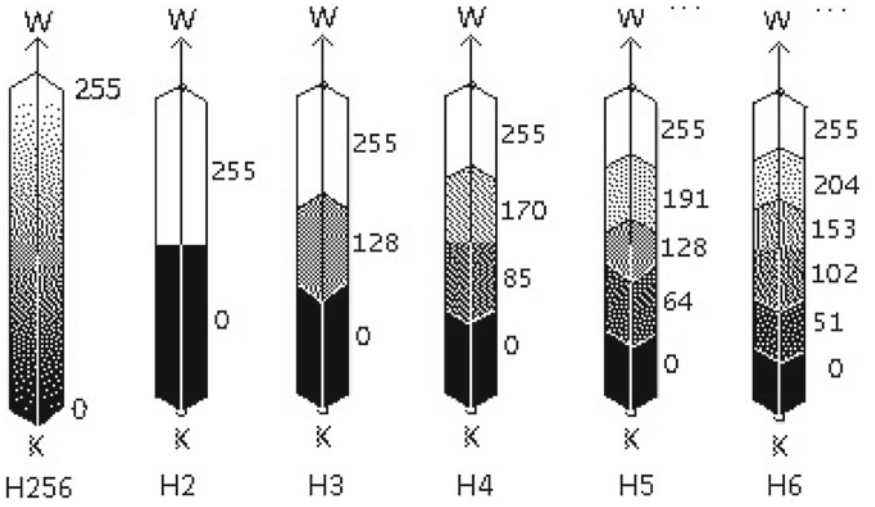


Fig. 6.7 Halftone neutral palettes. **a** Halftone palette H256 is neutral palette N256 after halftoning, with $S = 256$ steps along the body diagonal of the sRGB cube model, from (0, 0, 0) black K to (255, 255, 255) white W, and diversity $D = S = 256$. **b** Halftone palette H2, with $D = S = 2$ **(c)–(f)** Halftone palettes H3, H4, H5 and H6

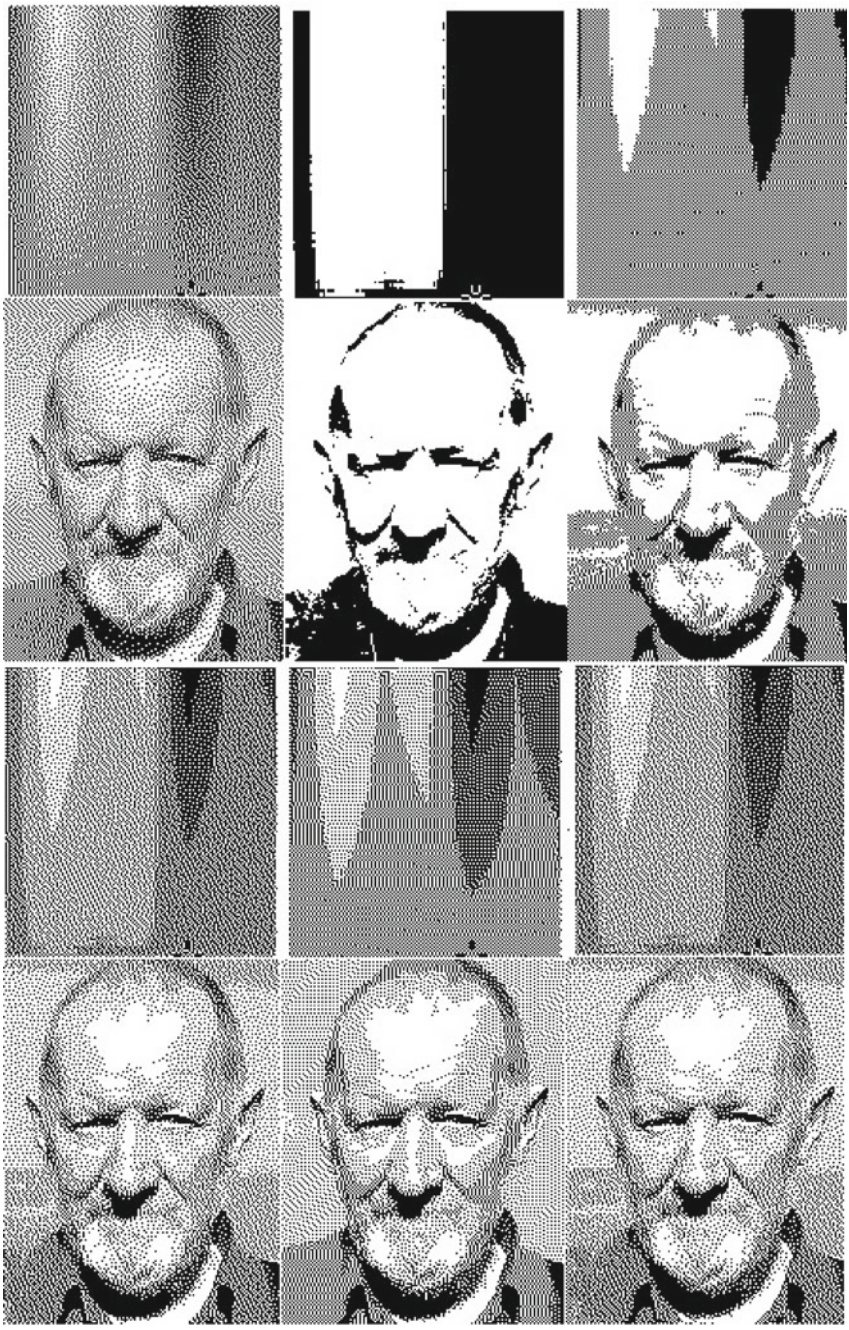


Fig. 6.8 Examples of images converted to halftone palettes: Figure 6.5 halftoned. The palettes are then: **a** H256. **b** H2. **c** H3. **d** H4. **e** H5. **f** H6

```
print "extent", extent

# Convert image to halftone.
imht = imrgb.convert("1", dither = 1)

# Show reduced image.
imht.show()
```

Figure 6.6 shows the sRGB cube in a halftone palette Fig. 6.7 shows the neutral axis in a halftone palette; and Fig. 6.8 shows some example images in halftone palette. Notice that a halftone image is very sensitive to the pixel pitch of a display device, and the dot pitch of a print device.

6.4 General Colour Shifts

In a digital image, each pixel has a location (X, Y) and a colour (R, G, B). A *general colour transformation* is a systematic change of all the pixel colours of an image, without change of location. We can make such a transformation by increasing or

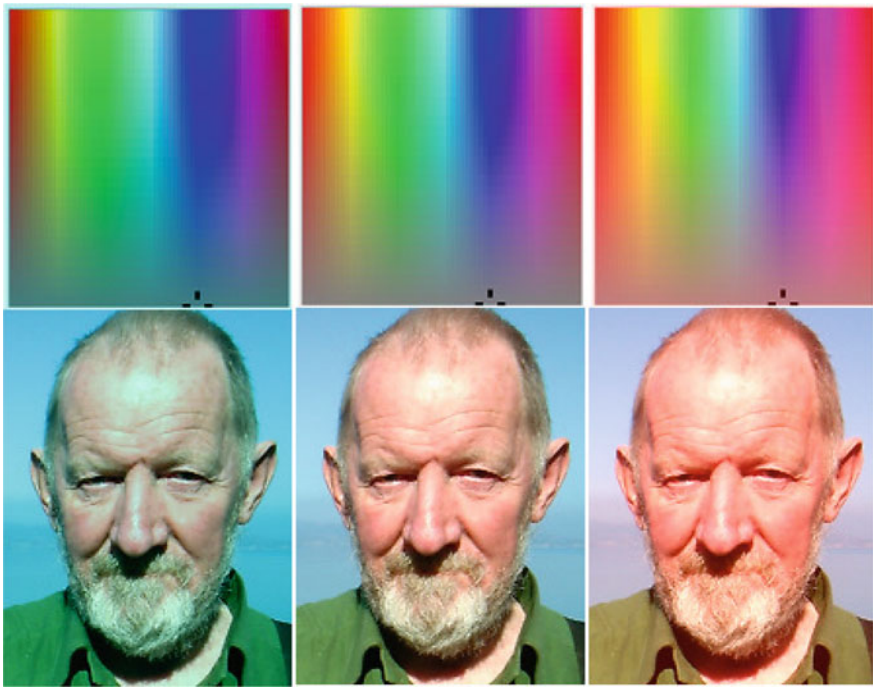


Fig. 6.9 Red-shift. The source images are in the middle. On the left, all red intensities are decreased by 64. On the right, all red intensities are increased by 64

Fig. 6.10 Green-shift. The source images are in the middle. On the left, all green intensities are decreased by 64. On the right, all green intensities are increased by 64

decreasing intensity values in one, two or all three of the fundamental red green and blue variables.

Elementary general colour transformations are as follows:

Red-shift: change intensity of the R variable
Green-shift: change intensity of the G variable
Blue-shift: change intensity of the B variable
Yellow-shift: change intensity of the R and G variables
Magenta-shift: change intensity of the R and B variables
Cyan-shift: change intensity of the G and B variables
All-shift: change intensity of the R, G and B variables

Figures 6.9, 6.10, 6.11, 6.12, 6.13, 6.14 and 6.15 show examples of each of the general colour transformations. The source images for all the examples are an sRGB gamut and a camera image.

To do a general colour transformation the steps are as follows:

Select the source image file in storage
Open it
Select the transformation wanted

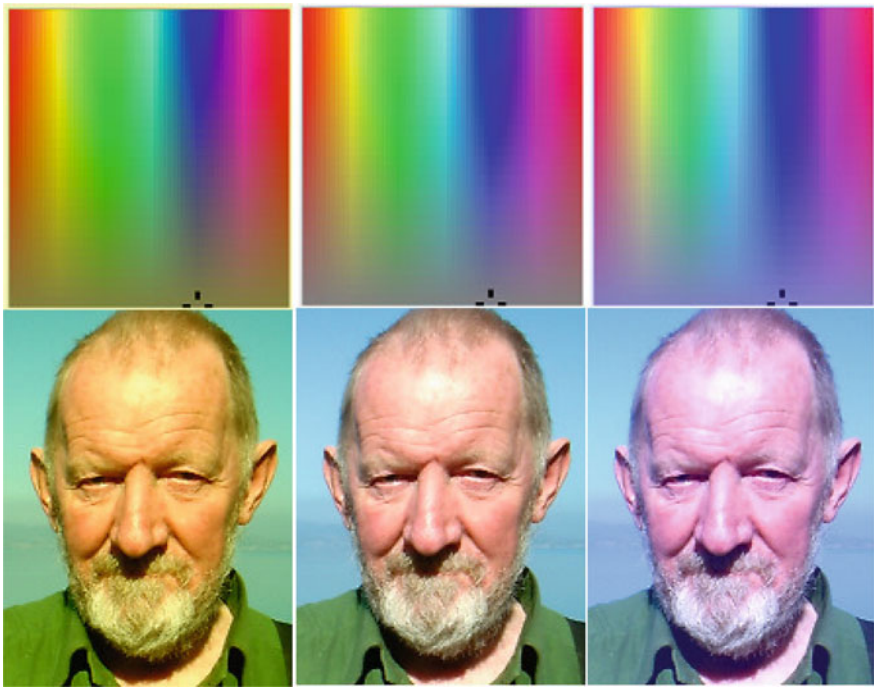


Fig. 6.11 Blue-shift. The source images are in the middle. On the left, all blue intensities are decreased by 64. On the right, all blue intensities are increased by 64

Set the transformation parameters
Do the transformation
Save the transformed image to storage under a new filename

We can write a script which will perform any of the R G B Y M C A shifts, listed below. Figures 6.9, 6.10, 6.11, 6.12, 6.13, 6.14 and 6.15 show examples of each of the general colour transformations. The source images for all the examples are an sRGB gamut and a camera image.

```
# pytrcog: transform colours (general)
# in .bmp image in current directory.
# Written by Alan Parkin 2017.

from PIL import Image
import os, sys
import StringIO

# Get source image and show it: enter any .bmp
# filemame in working directory.
print "SOURCE IMAGE"
imagefilename = raw_input("Image filename? ")
```

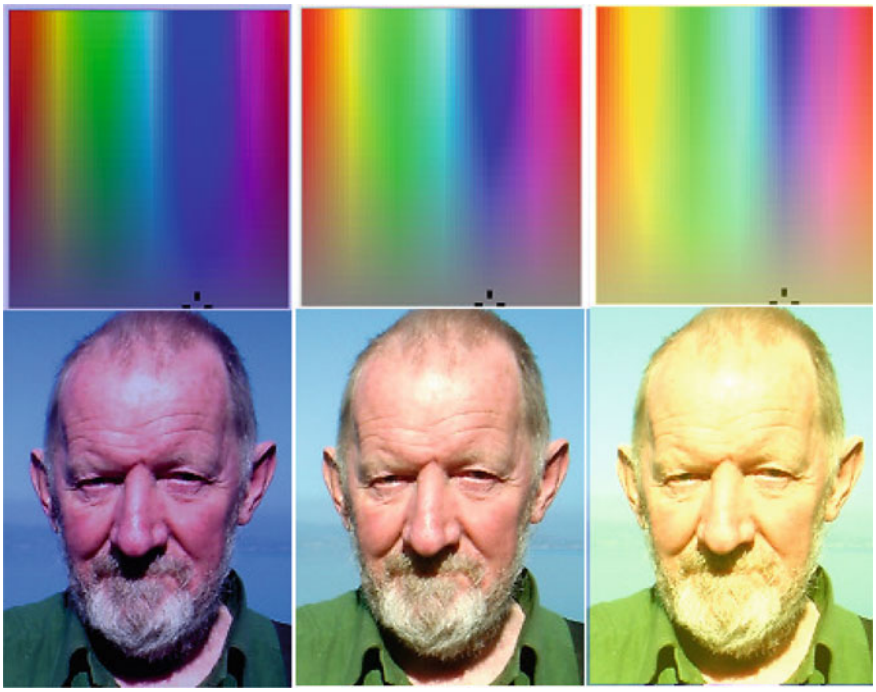


Fig. 6.12 Yellow-shift. The source images are in the middle. On the left, all red and green intensities are decreased by 64. On the right, all red and green intensities are increased by 64

```
#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

# Create input and output xy arrays.
inxy = im.load()
outim = Image.new("RGB", (width,height), "white")
outxy = outim.load()

# Enter menu choices.
print "--------"
print "MENU"
band = raw_input("Enter band(s) to transform R G B Y M C or A for all? ")
addsub = raw_input("Enter intensity to add/subtract as -255 to 255? ")
addsubi = int(addsub)
print "band", band
print "addsubi", addsubi
```

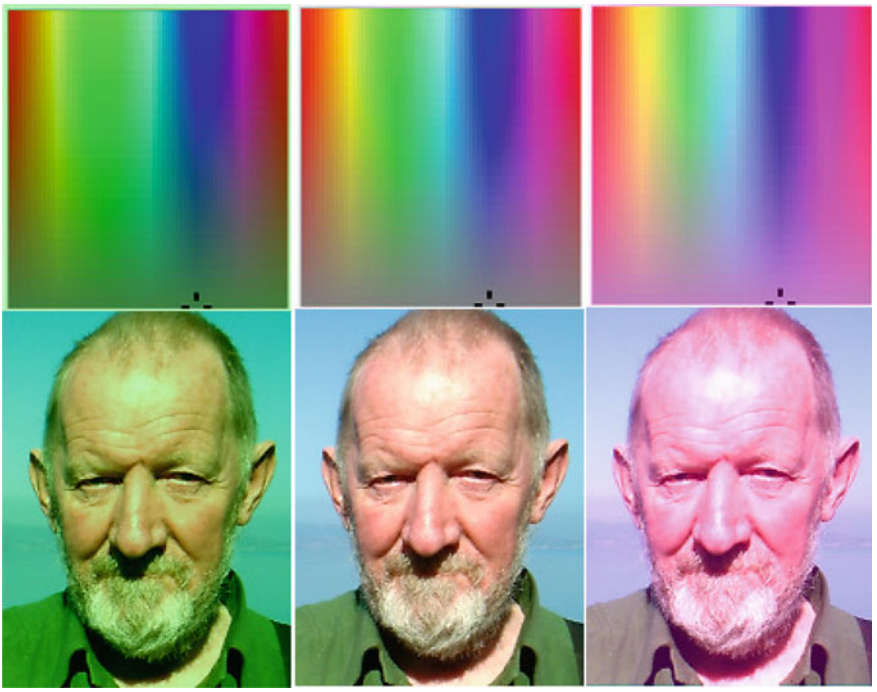


Fig. 6.13 Magenta-shift. The source images are in the middle. On the left, all red and blue intensities are decreased by 64. On the right, all red and blue intensities are increased by 64

```
# For each source pixel in inxy calculate new colour into outxy.
# Scan each row.
for y in range(height):
    # Scan each column.
    for x in range(width):
        rr = inxy[x, y][0]
        gg = inxy[x, y][1]
        bb = inxy[x, y][2]
        if band == "R" or band == "Y" or band == "M" or band == "A":
            rr = inxy[x, y][0] + addsubi
            if rr > 255:
                rr = 255

            if rr < 0:
                rr = 0

        if band == "G" or band == "Y" or band == "C" or band == "A":
            gg = inxy[x, y][1] + addsubi
            if gg > 255:
                gg = 255

            if gg < 0:
                gg = 0
```

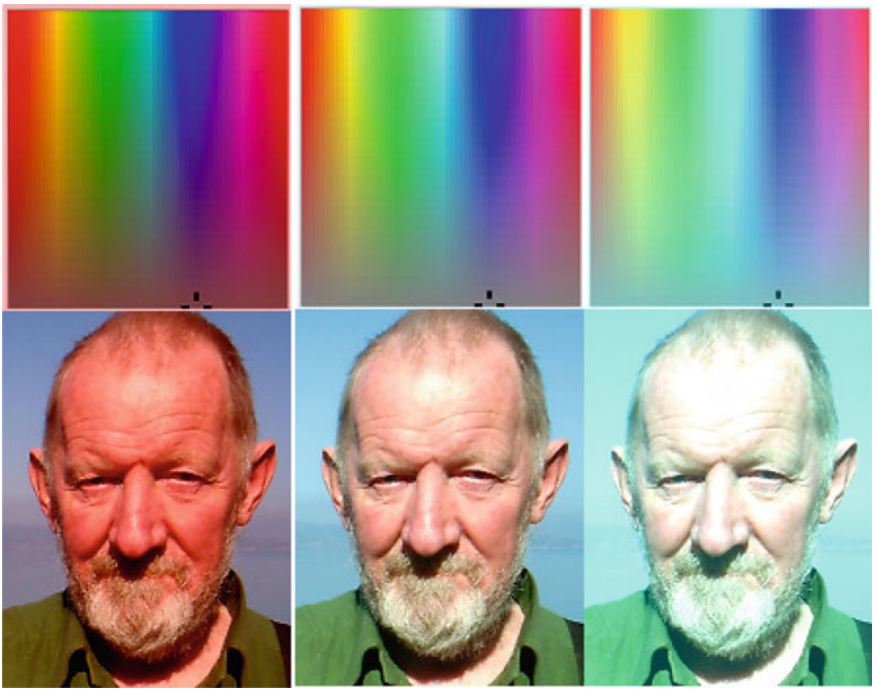


Fig. 6.14 Cyan-shift. The source images are in the middle. On the left, all green and blue intensities are decreased by 64. On the right, all green and blue intensities are increased by 64

```
            if band == "B" or band == "C" or band == "M" or band == "A":
                bb = inxy[x, y][2] + addsubi
                if bb > 255:
                    bb = 255

                if bb < 0:
                    bb = 0

            outxy[x, y] = (rr, gg, bb)

    # Show transformed image in Paint: save under new filename
    outim.show()
```

6.5 Muting Colours

A *muting* or neutral-shift changes all colours of an image towards their neutral equivalent.

To do a muting towards neutral, a script is listed below. Figure 6.16 shows examples of muting the source images.

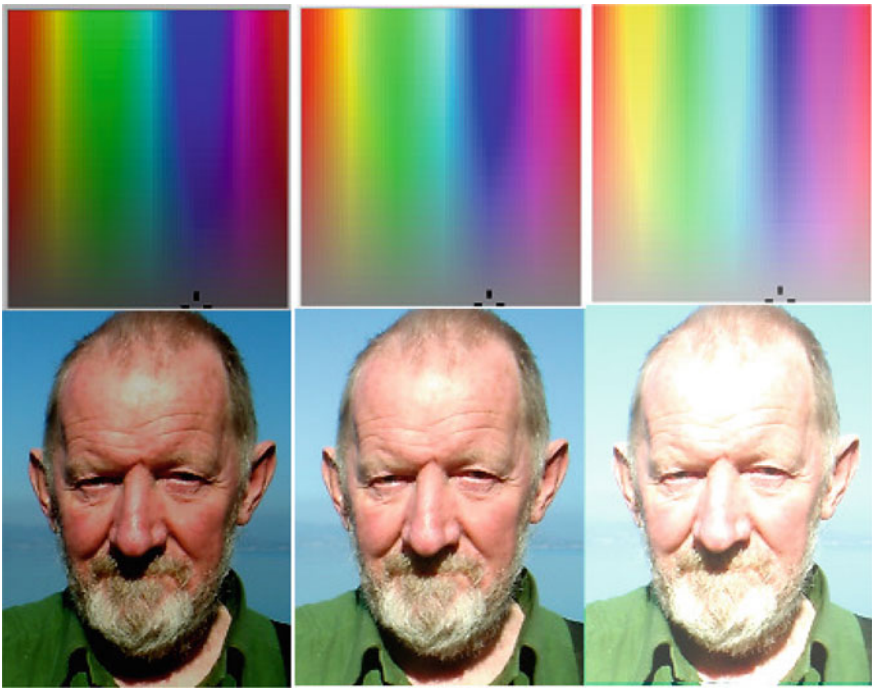


Fig. 6.15 All-shift. The source images are in the middle. On the left, all red green and blue intensities are decreased by 64. On the right, all red green and blue intensities are increased by 64

```
# pymute: using PIL to ask .bmg image filename, and
# ask muting fraction 0 to 1. Load rgb image as
# pix = xy array of rgb tuples. Calculate muted
# R G B values and impose them on the image.

from PIL import Image
import numpy as np
import os, sys

#Get filename of .bmp image which is in working directory
imagefilename = raw_input("Image filename? ")
#Open it and show it
imrgb = Image.open(imagefilename)
imrgb.show()
print "imrgb sizet34b.bmp", imrgb.size
width = imrgb.size[0]
height = imrgb.size[1]
pixtot = width * height
print "width", width
```

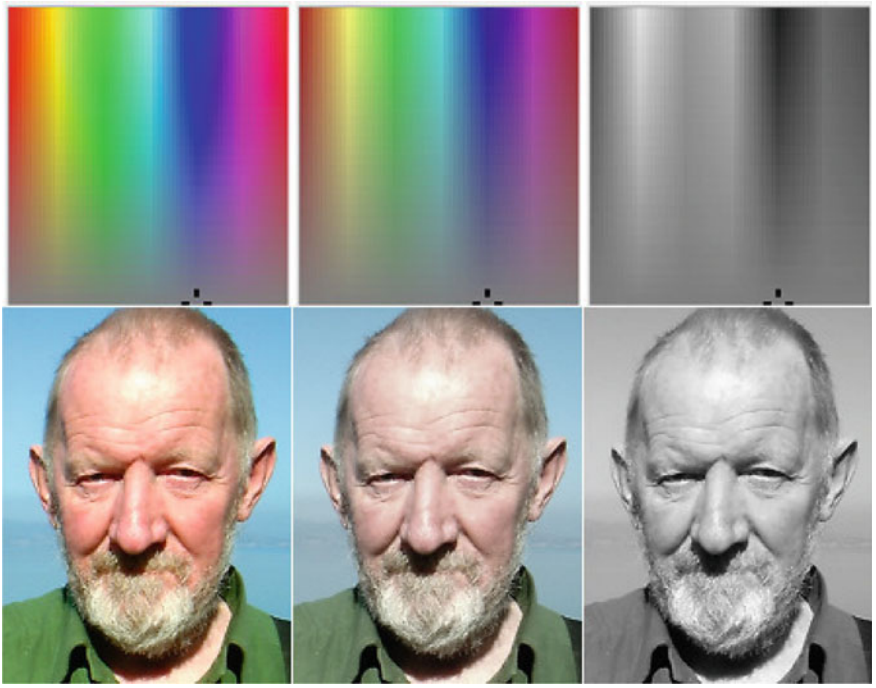


Fig. 6.16 Muting towards neutral. The source images are on the left (muting fraction = 0). In the middle, all colours are halfway muted to neutrals (muting fraction = 0.5). On the right, all colours are fully muted to neutral (muting fraction = 1.0)

```
print "height", height
print "pixtot", pixtot

# Get muting fraction.
mi = raw_input("Muting fraction 0.0 to 1.0? ")
mf = float(mi)
print "muting fraction ", mf

# Call PIL load for this image, to get xy array of rgb tuples.
pix = imrgb.load()

# Process values of pix.
for y in range(height):
    for x in range(width):
        r = pix[x,y][0]
        g = pix[x,y][1]
        b = pix[x,y][2]
        gsi = r * 299/1000 + g * 587/1000 + b * 114/1000
```

```
            rr = r * (1 - mf) + gsi * mf
            gg = g * (1 - mf) + gsi * mf
            bb = b * (1 - mf) + gsi * mf
            pix[x,y] = (int(rr), int(gg), int(bb))

    # Show muted image.
    imrgb.show()
```

6.6 Specific Colour Substitution

A *specific colour transformation* is a change of all occurrences of a specified colour to another specified colour, without change of location, throughout an image.

To do a specific colour transformation the steps are as follows:

Select the source image file in storage.
Open it.
Specify the old colour in the image, and the new colour wanted.
Do the transformation.
Save the transformed image to storage under a new filename.

An image editor such as Microsoft Paint has an option to replace one colour by another; but only within one connected region at a time. If we want to change a colour which has many occurrences in an image, we can write a script as listed below. Figure 6.17 shows examples of specific colour transformations done by the this script. The original pattern (a) is changed step by step in (b) (c) and (d).

```
# pytrsp: transforms one specific colour in image
# to another specific colour. Ask any .bmp image
# filename in current directory, open and show it.
# Ask colour (RGB) from and colour (RGB) to.
# Load image as pix = xy array of rgb tuples.
# Scan pix to change specified pixels from-to.
# Show changed image in Paint (save to new filename).
# Written by Alan Parkin 2017.

from PIL import Image
# import numpy as np
import os, sys

#Get filename of .bmp image which is in working directory
imagefilename = raw_input("Image filename? ")
#Open it and show it
imrgb = Image.open(imagefilename)
imrgb.show()
print "imrgb sizet34b.bmp", imrgb.size
```

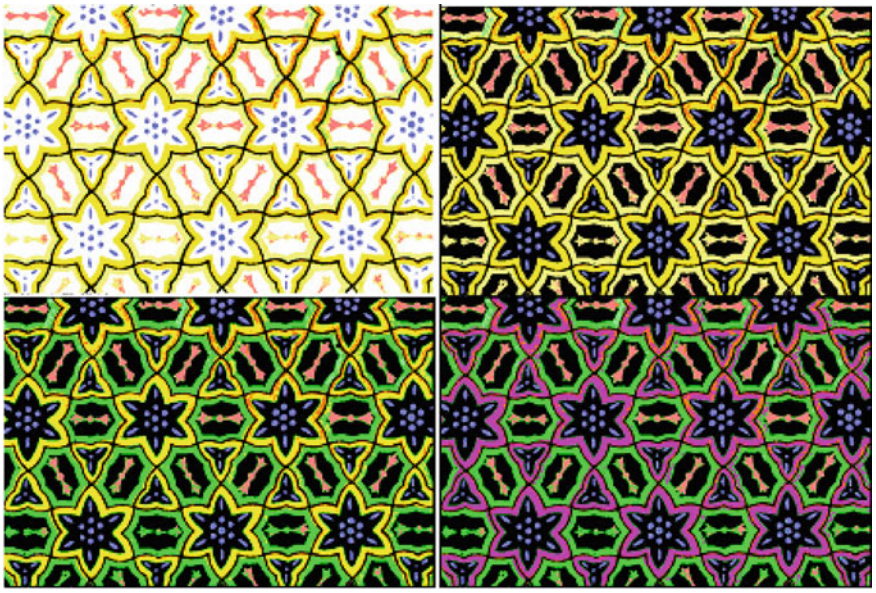


Fig. 6.17 Specific colour transformation. **a** Image is Plate 45 from Owen Jones' Grammar of Ornament, downloaded from the Internet in full sRGB, then converted to palette P3 to even out the colours. **b** Colour (255, 255, 255) white changed to (0, 0, 0) black. **c** Colour (255, 255, 127) dull yellow (border of bow ties) changed to (0, 255, 0) green. **d** Colour (255, 255, 0) bright yellow (border of stars) changed to (255, 0, 255) magenta

```
width = imrgb.size[0]
height = imrgb.size[1]
pixtot = width * height
print "width", width
print "height", height
print "pixtot", pixtot
print "--------"

# Ask colour from and colour to.
fromr = raw_input("Colour to change from, R-intensity 0-255 ? ")
fromg = raw_input("Colour to change from, G-intensity 0-255 ? ")
fromb = raw_input("Colour to change from, B-intensity 0-255 ? ")
ifromr = int(fromr)
ifromg = int(fromg)
ifromb = int(fromb)
tor = raw_input("Colour to change to, R-intensity 0-255 ? ")
tog = raw_input("Colour to change to, G-intensity 0-255 ? ")
tob = raw_input("Colour to change to, B-intensity 0-255 ? ")
itor = int(tor)
itog = int(tog)
itob = int(tob)
print "--------"
```

```
print "Change from (RGB) ", ifromr, ifromg, ifromb
print "Change to (RGB) ", itor, itog, itob
print "--------"

# Call PIL load for this image, to get xy array of rgb tuples.
pix = imrgb.load()

# Scan pix, changing each from pixel to a to pixel.
for y in range(height):
    for x in range(width):
        if pix[x,y] == (ifromr, ifromg, ifromb):
            pix[x,y] = (itor, itog, itob)

# Show changed image.
imrgb.show()
```

References

1. Grayscale. https://en.wikipedia.org/wiki/Grayscale
2. Twyman M (1970) Printing 1790–1970. Eyre and Spottiswoode, London
3. Ulichney R (1987) Digital halftoning. MIT Press, Cambridge
4. Parkin A (2016) Digital imaging primer. Springer, Heidelberg
5. Floyd-Steinberg dithering. https://en.wikipedia.org/wiki/Floyd-Steinberg_dithering

Chapter 7
Displaying an Image

7.1 Display Screen

A digital image is created, stored, processed and possibly transmitted, in numerical representation. It can be presented visually at any stage by an additive display device [1]. The standard such device is an sRGB computer screen [2].

Numerically, a computer screen has:

Fixed width $W*$ pixels.
Fixed height $H*$ pixels.
Fixed pixel pitch $Q*$ pixels per inch (ppi) (or pixels per millimetre (ppmm)).
In each pixel, standard red green and blue light sources independently settable at intensities 0–255.

The fixed width and height of a display screen vary widely. A typical laptop PC might have something like $W* = 11$ in, $H* = 8$ in, hence extent $E* = 88$ in ^{2}, about the same as an A4 sheet. Typical screens have pixel pitch around 100 ppi (though high-end computer and smartphone screens may go up to 600 ppi or even more [3]). Thus, a typical screen has fixed maximum extent around $E* = 1100 \times 800$=880 000 px (Fig. 7.1).

A digital image has indeterminate size: a particular display of an image has a size determined by the extent and pixel pitch of the display device. An image with small extent, such as 8×8 px, has a tiny display $8/100 \times 8/100$ in (which can be magnified), while a camera image with large extent, such as 4000×3000 px, has a huge display 40×30 in (which can be panned and scrolled, or diminished). Display magnification repeats every pixel in width and height 2, 3, ..., 8 times, in effect halving, thirding, ..., eighth-ing the pixel pitch Q* ppi. Display diminution omits half, three-quarters, seven-eighths, ... of the pixels in the image width and height. Figure 7.2 shows an example.

A. Parkin, *Computing Colour Image Processing*,
https://doi.org/10.1007/978-3-319-74076-8_7

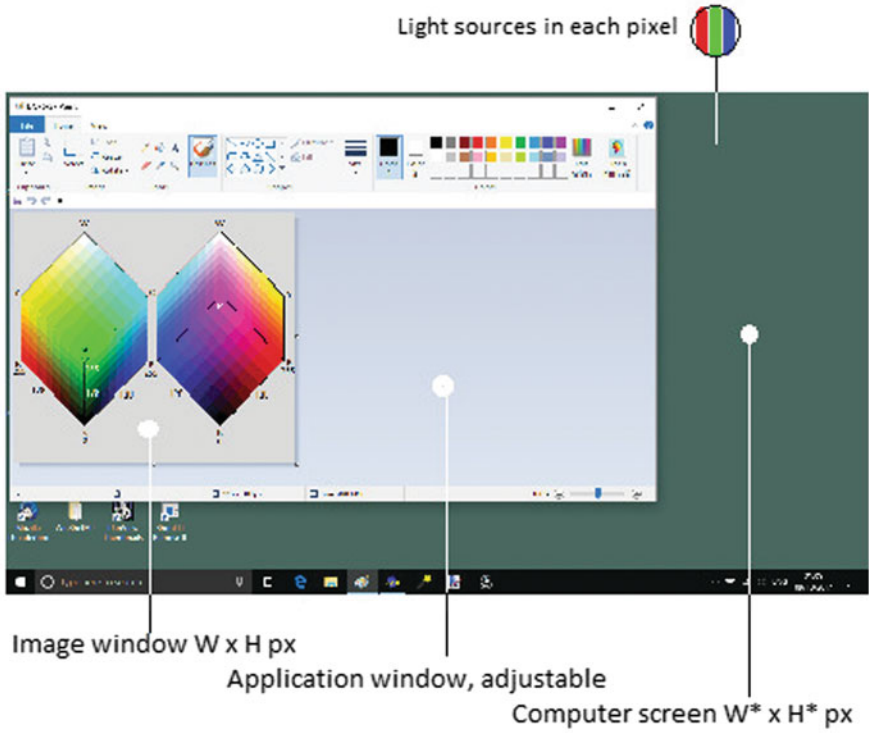


Fig. 7.1 Display screen showing an image editor application containing an image window

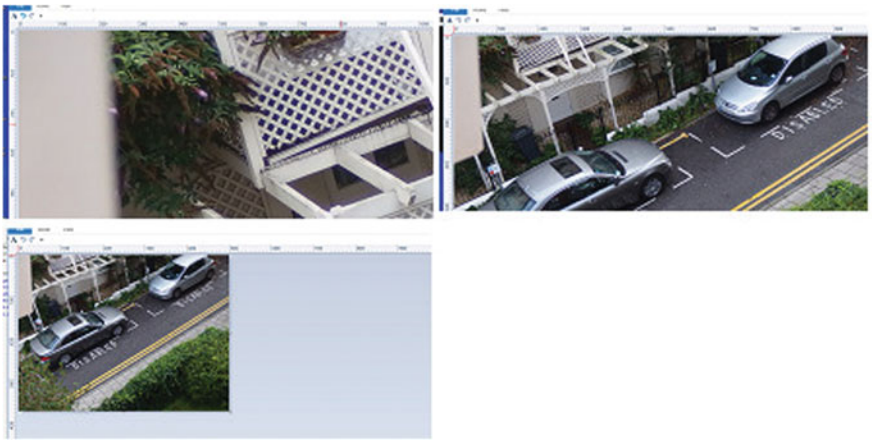


Fig. 7.2 Display diminution. **a** Part of a camera image 3970 × 2980 px, extent E approximately 12 Mpx, displayed in a Paint window 1000 × 450 px, extent $E* = 0.45$ Mpx. **b** Image diminished to 25 %, 1000 × 750 px, $E = 0.75$ Mpx, displayed in same window. **c** Image diminished to 12.5 %, 500 × 375 px, $E = 0.1875$ Mpx, displayed in same window

7.2 Display Location Resolution

The *location resolving power* of a display device is the smallest detail which it can display, measured as $1/E*$. Thus, a typical display screen has a location resolving power of (say) 1/880 000.

7.3 Display Colour Resolution

The *colour resolving power* of a display device is the smallest difference of colours which it can show. An sRGB device has diversity $D = 256^3 = 16.7$ million colours, each different from its neighbours, so the colour resolving power is 1/16.7 million. When displaying an image with a restricted colour space, the device uses only a subset of the sRGB colours, so its effective colour resolving power is accordingly restricted to 1/subset D.

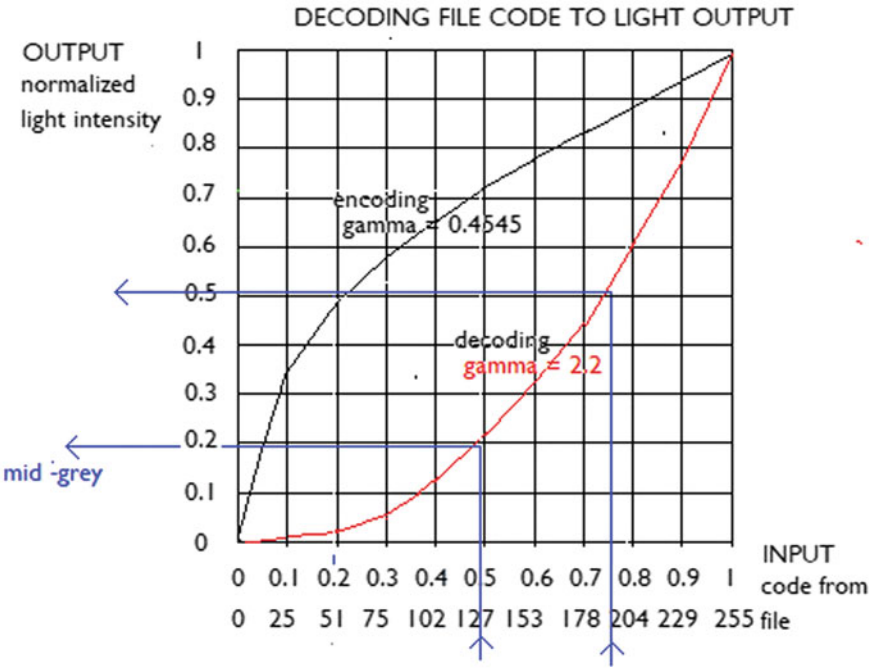


Fig. 7.3 Decoding a stored file image. Decoder input per pixel is code 0–255, normalized to 0–1. Decoder output is light intensity = ((input code) / 255)$^{2.2}$. Thus, code 127 is displayed as mid-grey = 0.18 luminance, corresponding to a photographic 18 % test card. Code 187 is displayed as 0.5 luminance. (The black curve shows encoding gamma = 0.4545, symmetrical with 2.2 about the plot diagonal (0,0)–(1,1), see Chap. 3.)

A display device receives input as a per-pixel triple of numeric codes (R, G, B), each in the range 0–255. A digital-to-analogue converter (DAC) passes this as a triple of voltages to the three subpixel light sources. In an earlier generation, virtually all displays were based on a cathode ray tube (CRT). An inherent physical characteristic of a CRT is that

$$\text{Output light} = (\text{input voltage})^{2.2}$$

where 2.2 is the so-called *decoding gamma* γ of the device [7]. To counteract this non-linear decoding gamma, the inverse *encoding gamma* = 1/2.2 = 0.4545 was routinely applied to the driving signal. Today virtually all displays use liquid crystal technology (LCD), which has no inherent gamma. But to maintain continuity, LCDs are given a CRT-like decoding gamma of 2.2. It is a fundamental convention of sRGB that cameras and other input devices have encoding gamma 0.4545, and display devices have decoding gamma 2.2. Figure 7.3 plots sRGB display decoding.

7.4 Perceptually Equal-Step Scales

sRGB also incorporates the CIELAB scale of lightness L* [8]. L* sets *perceptual mid-grey* as the physical light intensity reflected by an 18 % photographic grey card [9], and hence encodes mid-grey as (127, 127, 127) (as shown in the encoding plot in Chap. 3). It is claimed that this setting 'recognizes the logarithmic nature of visual perception', stemming from the Weber–Fechner law that just-noticeable differences are small for low light intensities and increasingly larger for higher intensities. It is further claimed that this gives a good perceptual result for the average photographed scene.

But Hoffmann's cogent objection is: ' All this cannot be applied to images, because the Weber law is a result of variable adaptation (sitting in a dark room and observing two large patches). For imaging, the adaptation is more or less fixed, the Weber law is not valid. ... The resolution for dark greys is not generally better than for light greys' [10].

It turns out that a perceptually equal-step greyscale has a Stevens' power-law [12] exponent of 1, not the 0.33 used by L* [11, 13]. That is to say, we perceive a scale of intensities pretty much as they physically are. To get such a scale, the sRGB scale anchored at mid-grey = 18 % reflection = code (127, 127, 127) should be re-anchored at mid-grey = 50 % reflection = code (187, 187, 187) [9]. The re-anchoring is done by applying power 0.45 to each intensity in the image file. The script below does just this, for any stored sRGB image. Figure 7.4 compares the standard sRGB greyscale with the perceptually equal-step greyscale, and Fig. 7.5 compares the R G B scales. 'Who you gonna believe, me or your own eyes?' [14].

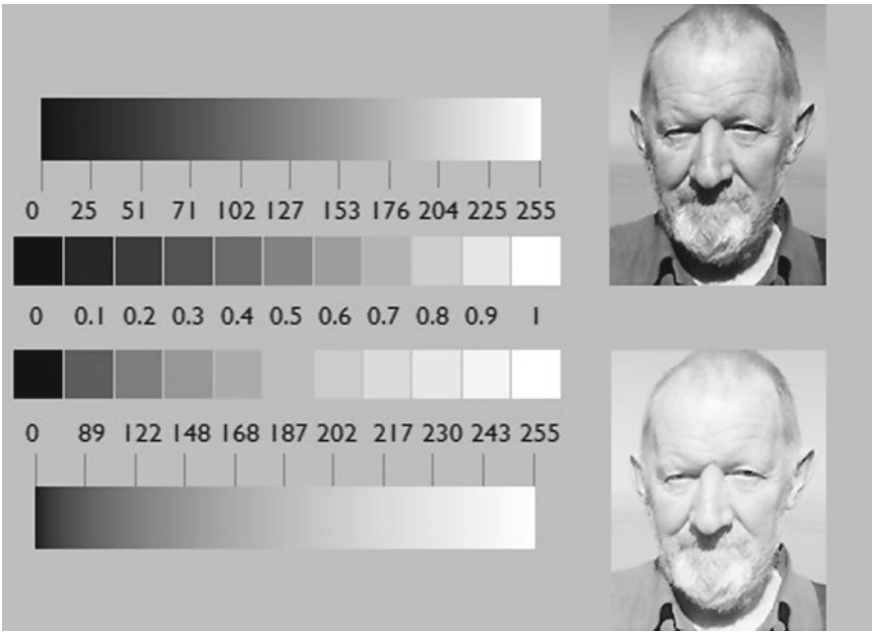


Fig. 7.4 Perceptually equal-step greyscale. The top scale is a display of 256 greys from black (0, 0, 0) to white (255, 255, 255), under the sRGB L* standard anchored at mid-grey = (127, 127, 127). The second scale separates the gradation into 11 steps which should be perceptually equal but are clearly not. The third scale is a better set of 11 steps, separated from the bottom gradation. The bottom scale is a display of 256 greys anchored at mid-grey = (187, 187, 187). The top image is displayed using the first coding, and the bottom image using the second coding

```
# pychagam: Change gamma in stored image file.
# User picks .bmp file to change; computer shows it.
# User enters new gamma to be applied. Computer
# calculates new image and shows it (to be saved
# where wanted).
# 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
imrgb = Image.open(imagefilename, 'r')
imrgb.show()
width = imrgb.size[0]
height = imrgb.size[1]
```

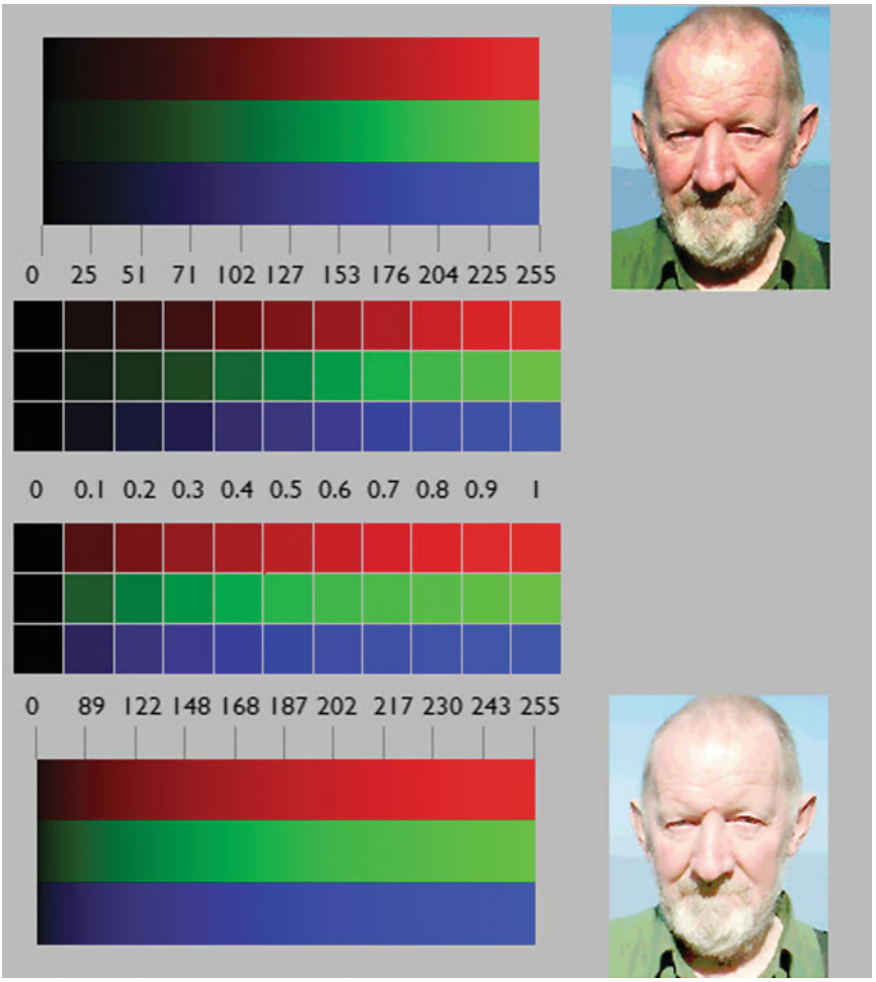


Fig. 7.5 Perceptually equal-step RGB scales. The top triple is a display of 256 intensities from black (0, 0, 0) to red (255, 0, 0), 256 from black to green (0, 255, 0) and 256 from black to blue (0, 0, 255), under the sRGB L* standard anchored at mid-grey = (127, 127, 127). The second triple separates the gradation into 11 steps which should be perceptually equal but are clearly not. The third triple is a better set of 11 steps, separated from the bottom gradation. The bottom triple is a display of the 256 intensities anchored at mid-grey = (187, 187, 187). The top image is displayed using the first coding, and the bottom image using the second coding

```
extent = width * height
print imagefilename + " open"

# Enter new gamma value wanted.
newgam = raw_input("Enter new gamma as float 1.0 etc) ? ")
```

```
fnewgam = float(newgam)
print "new gamma ", fnewgam

# Call PIL getdata for this image, and length.
gd = imrgb.getdata()
gdlist = list(gd)
lengdlist = len(gdlist)

# Create input and output xy arrays.
inxy = imrgb.load()
outim = Image.new("RGB", (width,height), "white")
outxy = outim.load()

# For each source pixel in inxy calculate new colour in outxy.
for y in range(height):
    for x in range(width):
        oldr = inxy[x, y][0]
        oldg = inxy[x, y][1]
        oldb = inxy[x, y][2]
        newr = int(((oldr/255.) ** fnewgam) * 255.)
        newg = int(((oldg/255.) ** fnewgam) * 255.)
        newb = int(((oldb/255.) ** fnewgam) * 255.)
        outxy[x, y] = (newr, newg, newb)

outim.show()

print "done"
```

7.5 Display Viewing Environment

The sRGB recommended viewing environment is shown in Fig. 7.6.

An LCD screen is highly sensitive to the precise viewing angle. When this is 90 degrees at the centre of the screen, the display shows a greyscale anchored at mid-grey = code 127, giving poor resolution of darker colours, as shown in Fig. 7.5. But when viewed from rather higher, it is as if the mid-grey anchor goes up towards code 187; and from lower the anchor goes down towards code 0. We may on occasion prefer to set an image file for a perceptually equal-step display anchored at mid-grey = code 187 (or even another), using program pychgam listed above, to get always the display shown in Fig. 7.5.

We view an image displayed on a screen at a suitable distance. Figure 7.7 shows the trigonometry of the situation. It is known that a normal Snellen 20/20 eye resolves detail which subtends α = one minute of arc [15]. For a display screen with pixel

Parameter	Value
Screen luminance level	80 cd/m^2
Illuminant white point	x = 0.3127, y = 0.3290 (D65)
Image surround reflectance	20% (~medium gray)
Encoding ambient illuminance level	64 lux
Encoding ambient white point	x = 0.3457, y = 0.3585 (D50)
Encoding viewing flare	1.0%
Typical ambient illuminance level	200 lux
Typical ambient white point	x = 0.3457, y = 0.3585 (D50)
Typical viewing flare	5.0%

Fig. 7.6 sRGB viewing environment (taken from [5])

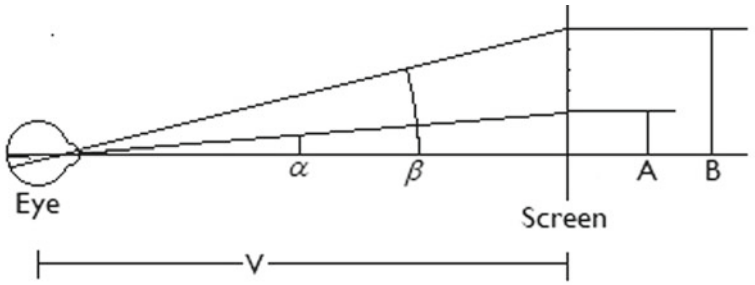


Fig. 7.7 Viewing a display screen. α is 1 arc-min, and A is the smallest detail resolved by a normal Snellen 20/20 eye. β is 3 arc-min, the angle subtended at the eye by one pixel at near viewing distance V = 10 in (equivalent to an eye with Snellen 20/65 acuity)

pitch 1/100 ppi, and near viewing distance V = 10 in = 1000 px, $A = V * \tan(\alpha)$ = 1000 * 0.0002908 = 0.3 px. So to distinguish B = one pixel, we must have angle β = about 3 arc-min, which is equivalent to an eye with Snellen 20/65 acuity.

References

1. Display device. https://en.wikipedia.org/wiki/Display_device
2. Computer monitor. https://en.wikipedia.org/wiki/Computer_monitor
3. Retina display. https://en.wikipedia.org/wiki/Retina_Display
4. Snellen chart. https://en.wikipedia.org/wiki/Snellen_chart
5. sRGB colour space. https://en.wikipedia.org/wiki/SRGB
6. Cathode ray tube. https://en.wikipedia.org/wiki/Cathode_ray_tube
7. Gamma-correction. https://en.wikipedia.org/wiki/Gamma_correction
8. Lab color space: CIELAB. https://en.wikipedia.org/wiki/Lab_color_space
9. Middle-gray. https://en.wikipedia.org/wiki/Middle_gray
10. Hoffmann G. The gamma question. http://docs-hoffmann.de/gamquest18102001.pdf
11. Parkin A (2016) Digital imaging primer, Chapter 17. Springer, Heidelberg
12. Stevens's power law. https://en.wikipedia.org/wiki/Stevens's_power_law
13. Hoffmann G. Optimized grayscale. http://docs-hoffmann.de/optigray06102001.pdf
14. Chico Marx playing Chicolini disguised as Rufus T Firefly in Duck Soup, Paramount pictures (1933)
15. Visual acuity. http://webvision.med.utah.edu/book/part-viii-gabac-receptors/visual-acuity/

Chapter 8
Printing an Image

8.1 Subtractive Printing

To make permanent prints of digital images on paper, the technology is drawn from the printing trades [1]. Optically, printing inks are transparent films, which act as filters to subtract (absorb) bands of wavelengths from the incident light. Thus, a cyan 'not-red' ink absorbs the red long wavelengths and transmits the green middle and blue short wavelengths; a magenta 'not-green' ink absorbs green and transmits red and blue; and a yellow 'not-blue' ink absorbs blue and transmits red and green (see Chap. 1, Section Printing). Available cyan magenta and yellow (CMY) inks are deficient in various ways, particularly in not making a good black when all three are overprinted. A black ink K, which absorbs all wavelengths and transmits none, is used to replace equal densities of CMY overprints ('undercolour-removal'). The incident light which survives the filtering of the ink layers is reflected by the white paper. Figure 8.1 shows the subtractive combinations of CMYK inks when printed solid.

But printing ink on paper is an essentially binary process. Each location of the master plate can either deposit ink or leave the paper showing the intensity of the ink deposit cannot be varied. Halftoning of one kind or another is the way around this limitation that has been used since the beginning of printing (see Chap. 6, Section Halftone Palettes). Hatched lines or small dots of solid ink are spaced out to appear as a range of intensities, in effect diluting the ink with the white paper. Digital halftoning [2, 3] is an important aspect of printing digital images, by either four-colour or black-only processes. A black-only halftone image can also be displayed on an additive sRGB screen.

Thus, printing uses a *CMYK colour space* [4], which is inverse to the sRGB space: instead of red we have cyan, instead of green magenta, instead of blue yellow and instead of equal intensities of red green and blue we have black. Figure 8.2a shows the four ink colours, and Fig. 8.2b shows the gamut of print colours, which is noticeably different from the sRGB gamut. Figure 8.3 shows a cube model of the CMYK space, front and back.

A. Parkin, *Computing Colour Image Processing*,
https://doi.org/10.1007/978-3-319-74076-8_8

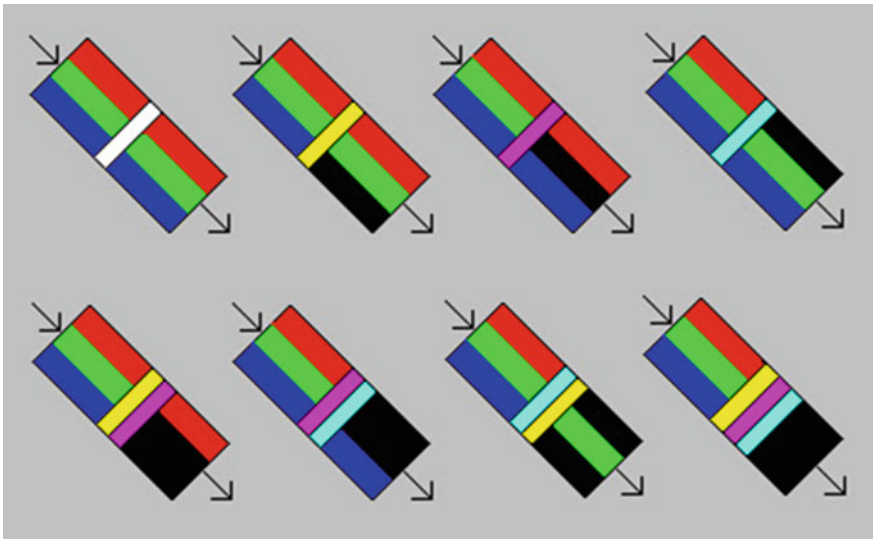


Fig. 8.1 Subtractive-colour filtering. Top row: **a** No ink passes all three wavelength bands of incident light and stops none. **b** Yellow ink passes red and green, stops blue. **c** Magenta ink passes red and blue, stops green. **d** Cyan ink passes green and blue, stops red. Bottom row: **e** Yellow and magenta inks together pass red. **f** Magenta and cyan inks together pass blue. **g** Cyan and yellow inks together pass green. **h** Ideal yellow magenta and cyan inks together would stop all light and pass none, to show black. Practical inks need a black-ink booster to show a good black

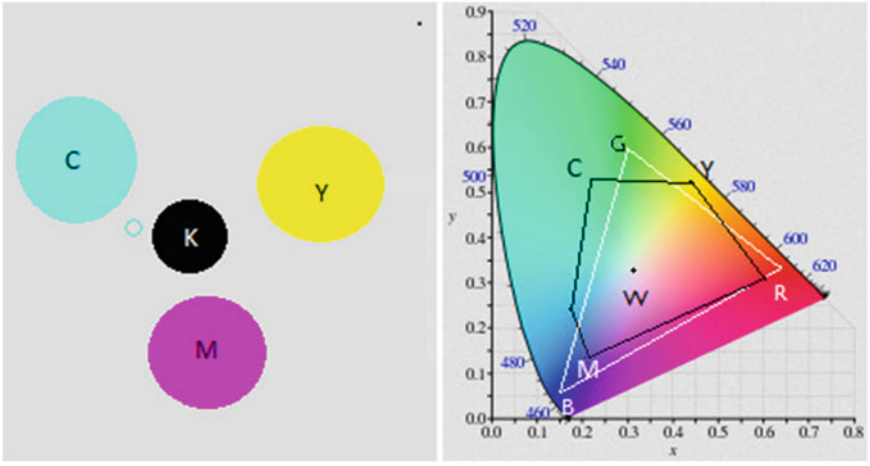


Fig. 8.2 The CMYK inks and their gamut of CIE hues. **a** Cyan magenta yellow and black inks. **b** The subtractive gamut of typical inks, compared to the sRGB additive gamut

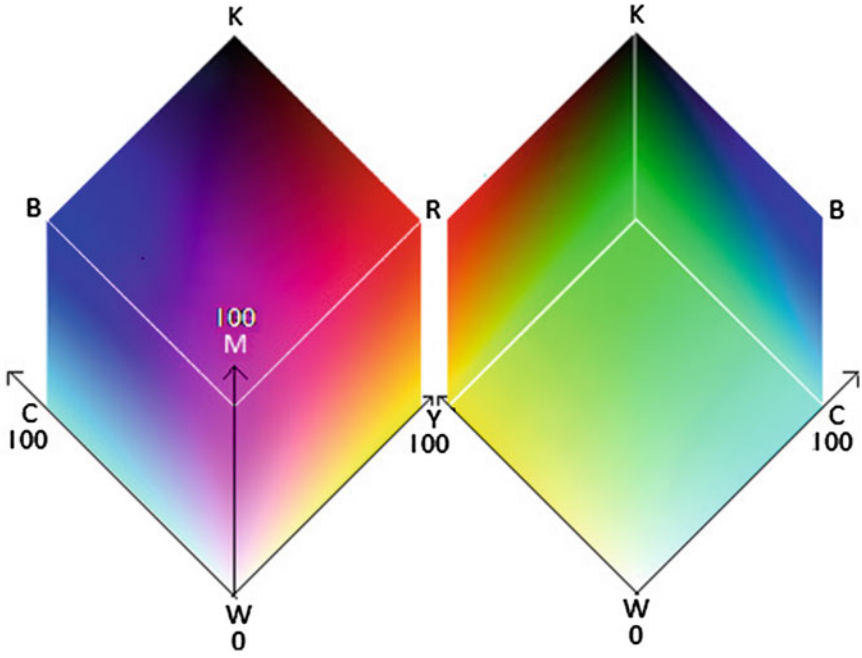


Fig. 8.3 CMYK colour space. Cube model, front and back, inverse to sRGB space. The origin is W white, with axes Y yellow, M magenta and C cyan, each axis calibrated 0–100%. K black ink replaces neutral combinations (C = M = Y)

8.2 Location Resolution

Numerically, an inkjet printer has:

Fixed line length $L*$ in.
Fixed page length $M*$ in.
Fixed dot pitch $N*$ dots per inch (dpi) (or dots per millimetre (dpmm)).
In each pixel, cyan magenta yellow and black dot densities 0–100%.

A typical A4 printer has a fixed line length of 8 in and a fixed page length of 11 in. The fixed dot pitch is usually 600 dpi for an inkjet printer, or 600–1200 or higher for a laser printer. So, an A4 600 dpi printer has a print extent of fixed width $W* = 8 \times 600 = 4800$ dots and a fixed height of $H* = 11 \times 600 = 6600$ dots. The printer software allocates a square of (say) 6×6 dots per pixel, so the extent of the printed page is $E* = 4800/6 \times 6600/6 = 880000$ px, about the same as an A4 display screen.

The computations for converting an sRGB image to CMYK colour space, and for halftoning the colours, are supplied in the printer manufacturer's driver software. The basic calculations are described in [3].

8.3 Colour Resolution

Compared to a screen display at 100 pixels per inch, a printer can thus allocate 6 × 6 dots per pixel. Each dot is an unvaried intensity of yellow magenta cyan or black ink. The dots are variably spaced by halftoning percentage from 0 to 100. At normal viewing distance, the dots combine subtractively and additively to give an approximate match to the additive display colours.

A CMYK printed image thus has four ink colours, each in 100% halftones, colour diversity $D = 100^4 = 100$ million apparent colours. Clearly, halftoning to say four levels would provide $D = 4^4 = 256$ colours, $COLRES = 1/256$, sufficient for most purposes.

8.4 Viewing Environment

We view an image printed on paper at a suitable distance and under suitable ambient lighting. Figure 8.4 shows the trigonometry of the situation. Best viewing distance is usually reckoned as the diagonal of the image. As shown in Chap. 7, a pixel can easily be seen by a normal Snellen 20/20 eye. But in a printout, a pixel is formed of (say) 6 × 6 dots, which are well below the threshold of normal visibility. A printout thus appears as a continuous-tone image with no visible pixel structure.

Since a printout depends on filtering out some parts of the incident light, it will always need bright incident light to match the 80–100 cd/m^2 recommended for an additive screen [3]. But the eye is remarkably adaptive to lighting conditions, and involuntarily compensates for differences of incident and reflected light [5].

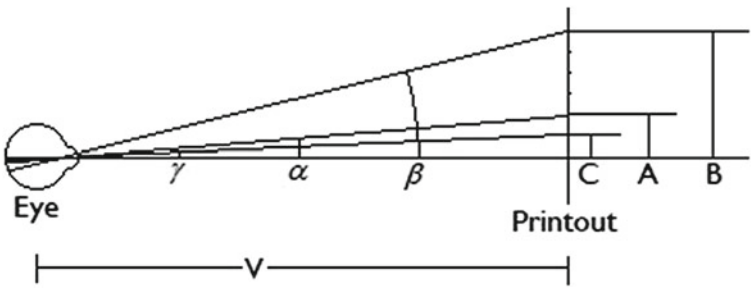


Fig. 8.4 Viewing a printout. α is 1 arc-minute, and A is the smallest detail resolved by a normal Snellen 20/20 eye. β is 3 arc-minutes, the angle subtended at the eye by one pixel at near viewing distance $V = 10$ in. γ is 0.5 arc-minute, the angle subtended at the eye by a typical printer dot C: too small to resolve

References

1. Idealalliance SWOP. https://en.wikipedia.org/wiki/Specifications_for_Web_Offset_Publications
2. Ulichney R (1987) Digital halftoning. MIT Press, Cambridge
3. Parkin A (2016) Digital imaging primer. Springer, Heidelberg
4. CMYK color model. https://en.wikipedia.org/wiki/CMYK_color_model
5. Bass M et al (2010) Handbook of optics, vol 3, 3rd edn. Vision. McGraw-Hill, New York

Chapter 9
Analysing Image Colour

9.1 Image Colour Distribution

The distribution of colour in a digital image is complex: we are faced with an extent of perhaps ten million pixels and a diversity of 16.7 million colours in each pixel. A statistical approach is appropriate, to reduce this complexity to a manageable summary [1]. Since a digital image is available in numerical representation, we can use computational methods to analyse the distribution of colour in an image.

Statistically speaking, an sRGB image presents as a finite population of E pixels, with three fundamental variables, the red green and blue light sources. A colour is a triple of intensity values (R, G, B) of the fundamental variables. The colour space of an image is the totality of possible intensity values, with a known diversity D. The colour distribution of an image is the proportional occupation of the E pixels by the D colours.

We wish to summarize this distribution and find its mean value. A procedure is sketched below, using the very simple image shown in Fig. 9.1 as an example.

9.2 Constructing a Colour Scheme Table

First, scan the entire population and count the frequency of occurrence of each distinct colour (R, G, B). Write a frequency table of N rows, one for each colour in this image. Each row shows the colour identifier, the frequency count in pixels Epx of that colour and the three intensity values of the fundamental variables (R, G, B):

```
Colour Epx (R   G   B)
------------------------
[[  0   1 255   0   0 ]  (red)
 [  1  32 255 255 255 ]  (white)
 [  2  29   0   0   0 ]  (black)
```


A. Parkin, *Computing Colour Image Processing*,
https://doi.org/10.1007/978-3-319-74076-8_9

```
 [  3   1   0 255   0 ]  (green)
 [  4   1   0   0 255 ]] (blue)
------------------------
    5  64
```

Next, sort the rows by frequency Epx:

```
Colour Epx (R   G   B)
------------------------
[[  0  32 255 255 255 ]
 [  1  29   0   0   0 ]
 [  2   1   0   0 255 ]
 [  3   1 255   0   0 ]
 [  4   1   0 255   0 ]]
------------------------
    5  64
```

Next, calculate the percentage of E occupied by the colour of each row: $Epc = (Epx * 100)/E$:

```
Colour Epx (R   G   B) Epc
---------------------------
[[  0  32 255 255 255  50]
 [  1  29   0   0   0  45]
 [  2   1   0   0 255   1]
 [  3   1 255   0   0   1]
 [  4   1   0 255   0   1]]
---------------------------
    5  64
```

For the final table, the Epx column is not needed, so replace it by the Epc column:

```
Colour Epc (R   G   B)
----------------------
[[  0  50 255 255 255]
 [  1  45   0   0   0]
 [  2   1   0   0 255]
 [  3   1 255   0   0]
 [  4   1   0 255   0]]
----------------------
 [  5 100(130 130 130)]
```

Calculate the values at the foot of the table as shown below:

```
 Clr  Epc  (R   G    B)
```

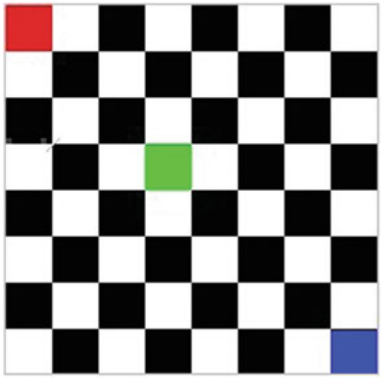


Fig. 9.1 A simple digital image. The image array has width = 8, height = 8, hence $E = 64$ pixels. The image shows $N = 5$ colours: white, black, red, green and blue

```
-----------------------
[[C0  Epc0 R0  G0   B0]
 [C1  Epc1 R1  G1   B1]
 [... ... ...  ... ...]
 [Ck  Epck Rk  Gk   Bk]]
---------------------------
[ N   100 Rav Gav  Bav ]
```

where in each row i:

Ci is the colour-row identifier,
$Epci =$ ((Epxi * 100)/E) is the extent of this colour as a percentage of the whole image extent E,
Ri, Gi, Bi are the intensities of the three variables in this colour,
$N = Ck + 1$ is the count of colours in this image,
$100 = \Sigma(Epc)$ the checksum of the percentages column (rarely exact if percentages are rounded to integer),
$Rav =$ ((R0 * Epx0) + (R1 * Epx1) + (R2 * Epx2) $+ \cdots +$ (Rk * Epxk))/100, the weighted-average intensity of red in the whole image that is the contribution of red from each colour weighted by the extent of that colour,
Gav ditto for green and
Bav ditto for blue.

9.3 Constructing a Colour Scheme Bar Graph

Having made a colour scheme table, we can show the distribution graphically by a stack of bars, one for each colour, occupying their proportional percentages of the extent. And we can show the mean value (Rav, Gav, Bav) of the whole image in an adjoining bar. Figure 9.2 shows the graph for the simple example.

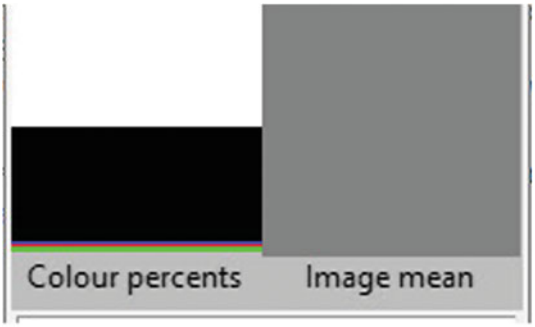


Fig. 9.2 Colour scheme bar graph for a simple digital image. On the left, the stack is 100 pixels high, containing 50% white, 45% black and 1% each of blue, red and green. On the right, the bar shows the image mean colour (130, 130, 130) a neutral just lighter than mid-grey (127, 127, 127)

9.4 Conditioning the Colour Scheme

sRGB colour space has huge diversity $D = 16.7$ million possible colours. An image created by program, camera or scanner in sRGB colour space will typically contain a very large count N of these colours and produce an impractically large colour scheme table, where most of the colours will be far down the table, with very small percentages Epc. Standard statistical practice in such circumstances is to group variables and ignore outliers.

We can group the colour intensity values by converting the original image to a restricted palette, as in Chap. 6, thus radically reducing diversity and bringing out the main structure of the colour distribution. Further, we can omit colours which occupy very small percentage extents in the image.

9.5 Scripts for a Colour Scheme

A pair of scripts which deliver a colour scheme table and bar graph for any .bmp image paletted to $D \leq 216$ colours are listed below. They are run in tandem, pyanalyA first to do the analysis and write two data files, then pyanalyB to read the data files and produce the bar graph and table as a Tkinter image.

```
# pyanalyA: Analyze .bmp image in current directory.
# User sets threshold for colour counts to pass.
# Computer counts colours and extents, and writes
# file cstabl and file csbars.dat.
# (Then run pyanalyB to show bars and table.)
# Written by Alan Parkin 2017.

from PIL import Image
import numpy as np
import os, sys
```

```
#Get filename of .bmp image which is in working directory
imagefilename = raw_input("Image filename? ")
#Open it and show it
imrgb = Image.open(imagefilename)
imrgb.show()
width = imrgb.size[0]
height = imrgb.size[1]
extent = width * height
print imagefilename + " open"

# Call PIL getcolors for this image, and length.
gc = imrgb.getcolors()
gclist = list(gc)
lengclist = len(gclist)

# Ask count pass value 0 or 1,2,3,4,5 percent.
passval = raw_input("Pass counts above 0 or 1 or 2 or 3 or 4 or
5 percent ? ")
if passval == "0":
    threshold = 0

if passval != "0":
    intpassval = int(passval)
    threshold = extent / (100. / intpassval)
    print "Omitting counts below percent ", intpassval

# Convert gclist to inlist, omitting items where
# count of px is less than asked..
inlist = []
leninlist = 0
for i in range(lengclist):
    count = gclist[i][0]
    if count > threshold:
        inlist.append(gclist[i][0])
        leninlist = leninlist + 1
        for j in range(3):
            thistuple = gclist[i][1]
            thisitem = thistuple[j]
            inlist.append(thisitem)

#Create numpy array of one dimension.
narra = np.array(list(inlist))
# Reshape as leninlist x 4.
narrb = narra.reshape((leninlist, 4))
```

```
# Sort rows by column 0, ascending.
narrc = narrb[narrb[:,0].argsort()]
# flip updown for descending..
narrd = np.flipud(narrc)

# Create index list, then index column.
indlis = []
for i in range(leninlist):
    indlis.append(i)

indarr = np.array(list(indlis))
indcol = indarr.reshape(leninlist, 1)

# Create pc list, then pc column.
pclis =[]
for i in range(leninlist):
    pc = (narrd[i][0] * 100) / extent
    pclis.append(pc)

pcarr = np.array(list(pclis))
pccol = pcarr.reshape(leninlist, 1)

# Delete col 0 from narrd.
narre = np.delete(narrd,0,1)

# Concatenate indcol and pccol to narre.
x = indcol
y = pccol
z = narre
narrf = np.concatenate((x,y,z), axis=1)

# Create weighted-average row.
rw = 0
gw = 0
bw = 0
for i in range(leninlist):
    rw = rw + (narrf[i,1] * narrf[i,2])
    gw = gw + (narrf[i,1] * narrf[i,3])
    bw = bw + (narrf[i,1] * narrf[i,4])

rwa = rw / 100
gwa = gw / 100
bwa = bw / 100
wavrowlist = (leninlist, 100, rwa, gwa, bwa)
wavrow = np.array(wavrowlist)
```

```
# Write to table data file cstabl.
f = open('cstabl', 'w')
f.write('COLOUR SCHEME FOR \n')
f.write('IMAGE ' + imagefilename)
f.write('\n')
f.write('Width px ' + str(width))
f.write('\n')
f.write('Height px ' + str(height))
f.write('\n')
f.write('Extent px ' + str(extent))
f.write('\n')
f.write('\n')
f.write('  Clr Epc  (R   G   B)\n')
f.write('---------------------- \n')
value = (narrf)
s = str(value)
f.write(s)
f.write('\n')
f.write('---------------------- \n')
s = str(wavrow)
f.write(s)
f.close()

# Write to bars data file csbars.dat.
bf = open('csbars.dat', 'w')
bf.write(imagefilename)
bf.write('\n')
bf.write(str(leninlist))
bf.write('\n')
for i in range(leninlist):
    bf.write(str(narrf[i,0]))
    bf.write('\n')
    bf.write(str(narrf[i,1]))
    bf.write('\n')
    bf.write(str(narrf[i,2]))
    bf.write('\n')
    bf.write(str(narrf[i,3]))
    bf.write('\n')
    bf.write(str(narrf[i,4]))
    bf.write('\n')

bf.write(str(rwa))
bf.write('\n')
bf.write(str(gwa))
```

```
bf.write('\n')
bf.write(str(bwa))
bf.write('\n')
bf.close()

print imagefilename + " done"

----------------
\index{Scripts!pyanalyB}
# pyanalyB: (Follows pyanalyA). In Tkinter, read
# file csbars.dat and create bars of colour scheme.
# Then read file cstabl and create text table.
# Written by Alan Parkin 2017.

from Tkinter import *

bf = open('C:/Users/Alan Parkinalan/csbars.dat', 'r')
filename = bf.readline()
print "Filename = ", filename
leninlist = int(bf.readline())

root = Tk()

w = Canvas(root, width=200, height=120, background="gray")
w.pack()

epc = 0
sumepc = 0
for i in range (2, leninlist + 2):
    xepc = epc
    clrindex = int(bf.readline())
    epc = int(bf.readline())
    rin = int(bf.readline())
    gin = int(bf.readline())
    bin = int(bf.readline())
    sumepc = sumepc + epc
    tkrgb = "#%02x%02x%02x" % (rin, gin, bin)
    w.create_rectangle(0, (sumepc - epc), 100, sumepc, outline=
    tkrgb, fill=tkrgb)

# Read image mean colours and create bar.
rwa = int(bf.readline())
gwa = int(bf.readline())
bwa = int(bf.readline())
tkrgb = "#%02x%02x%02x" % (rwa, gwa, bwa)
```

```
w.create_rectangle(100, 0, 200, 100, outline=tkrgb, fill=tkrgb)

# Create caption below bars.
w.create_text(50,108, text="Colour percents")
w.create_text(150,108, text="Image mean")

# Create text box below captions.
t = Text(root, width=24,height=33, background="white")
t.pack()

# Open file cstabl and insert lines in text box
tf = open('C:/Users/Alan Parkinalan/cstabl', 'r')
for j in range (leninlist + 10):
    line = tf.readline()
    t.insert(END, line, )

tf.close()

root.mainloop()

print filename + "done "
```

9.6 Colour Scheme Examples

Figures 9.3, 9.4, 9.5, 9.6 and 9.7 show examples of various types of images with colour schemes analysed by the scripts above.

Figure 9.3a has 11 colours. Colour 0 Pink (255, 153, 204) dominates, with two much smaller green extents 1 and 2. Colours 3 to 10 are tiny details. (b) has 11 colours in equal extents, so no significant ordering.

Figure 9.4a has six colours with extents 1% or greater: black, four greys and white. The red green and blue in the image occupy less than 1%. (b) has two colours, white and black.

Figure 9.5a is a camera image with many colours, all in small extents. This analysis is set to show just the top 12 colours with extents from 3% upwards. (b) is another camera image with many colours, dominated by black and the rest in small extents. This analysis is set to show just the colours occupying 1% or more.

Figure 9.6a is a scanner image with many colours, dominated by the two greys, with the rest in small extents. This analysis is set to show just the top 12 colours with extents from 1 percent upwards. (b) is another scanner image, severely diminished in extent, with many colours, where the analysis is set to show just the colours occupying 1% or more.

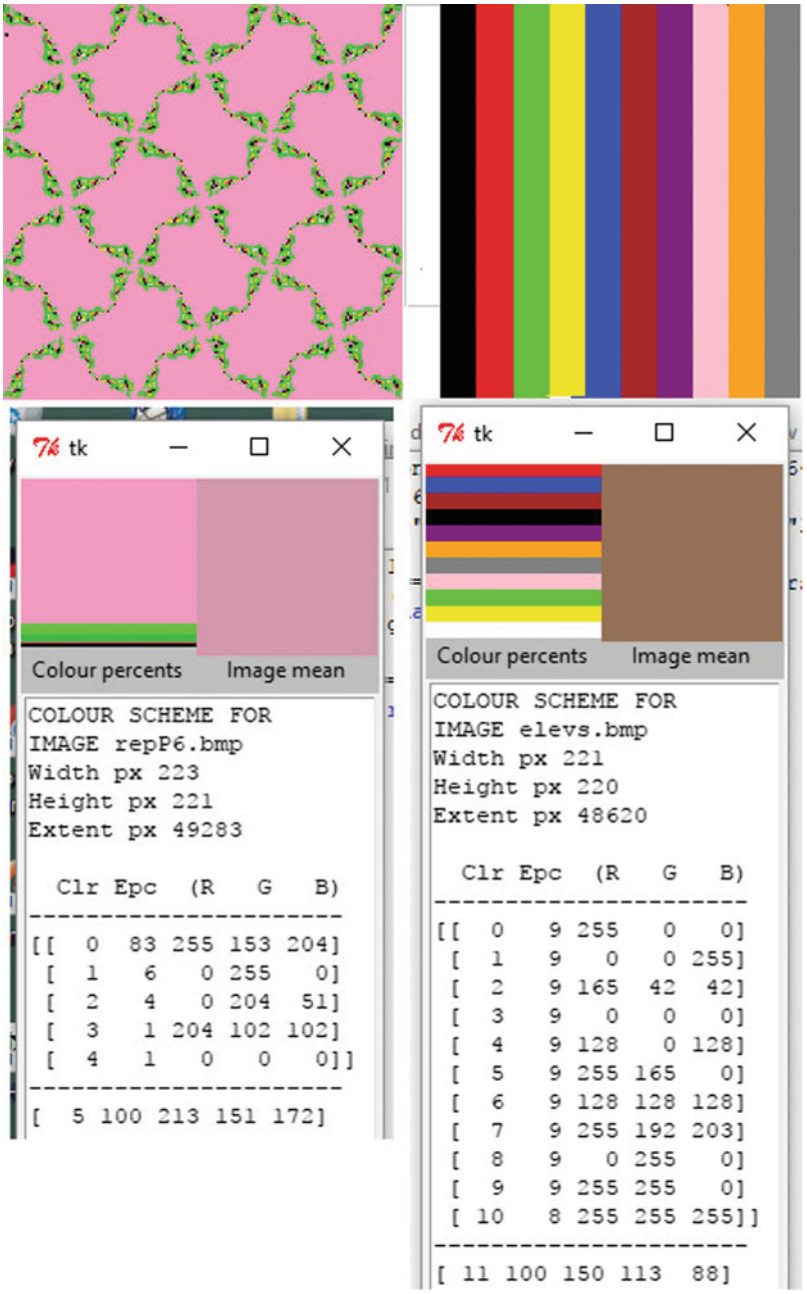


Fig. 9.3 Colour scheme examples. **a** Repeating pattern created by GUI image editor, 223 × 221 px, converted to palette P6. **b** Stripes of the 11 colours with common English names (see Chap. 1), created by GUI image editor, 223 × 220 px, converted to palette P6

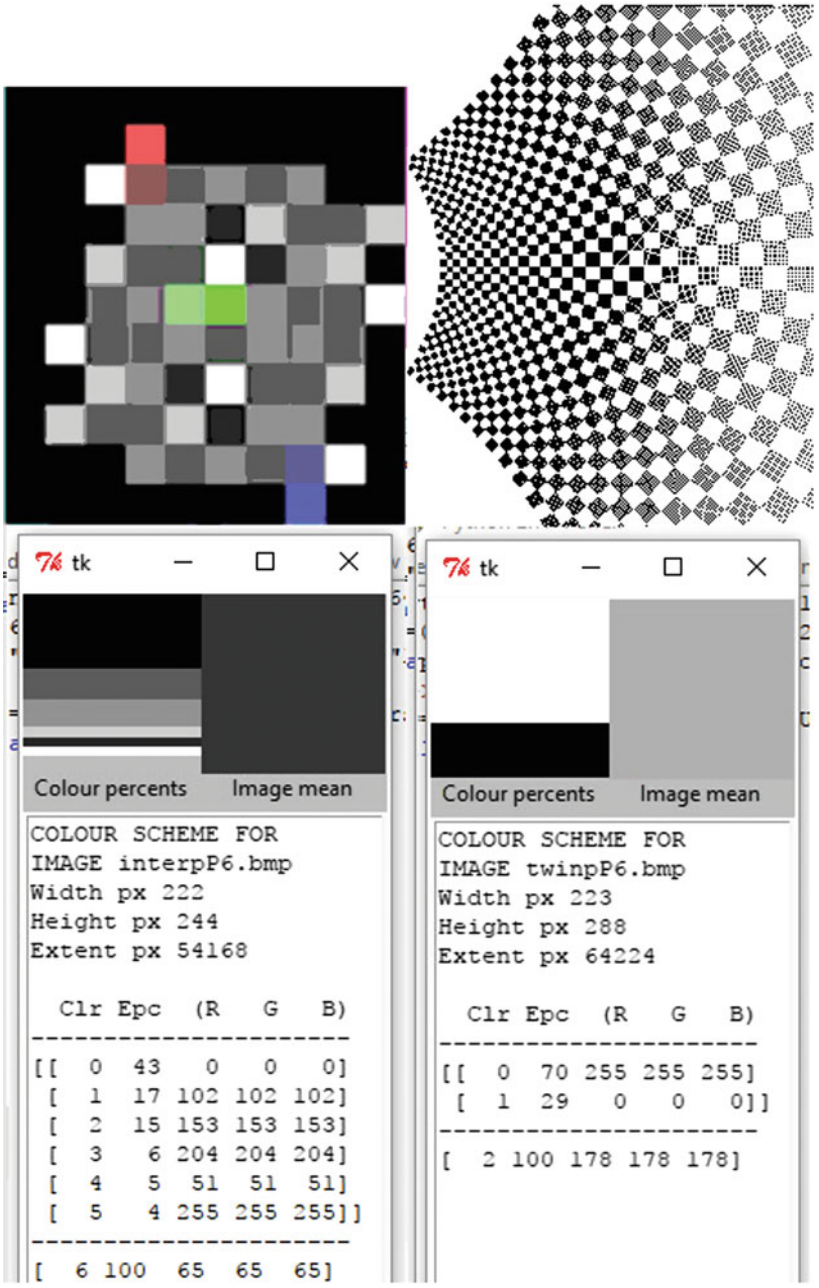


Fig. 9.4 Colour analysis examples. **a** Interpolated rotation created by program, 222 × 244 px, converted to palette P6. **b** Inversion of chequer created by program, 223 × 288 px, converted to palette P6

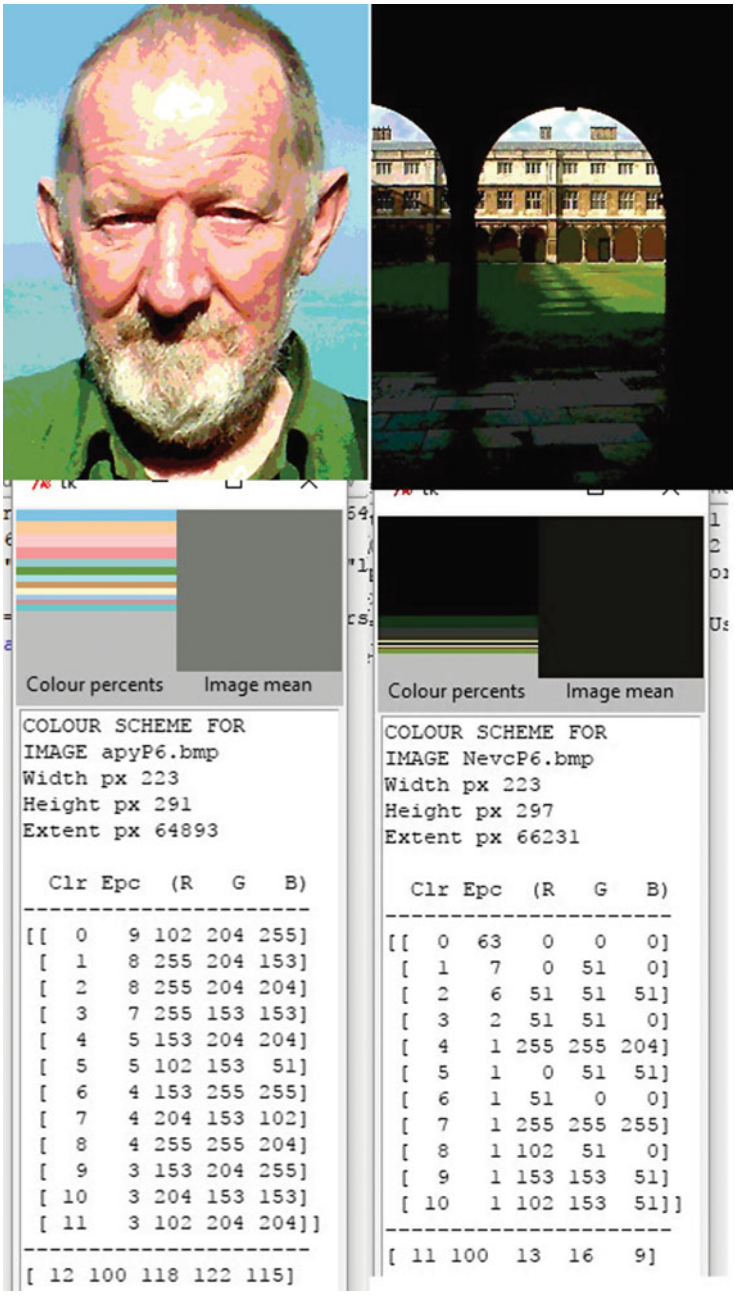


Fig. 9.5 Colour analysis examples. **a** Near camera image 223 × 291 px, converted to palette P6. **b** Far camera image 223 × 297 px, converted to palette P6

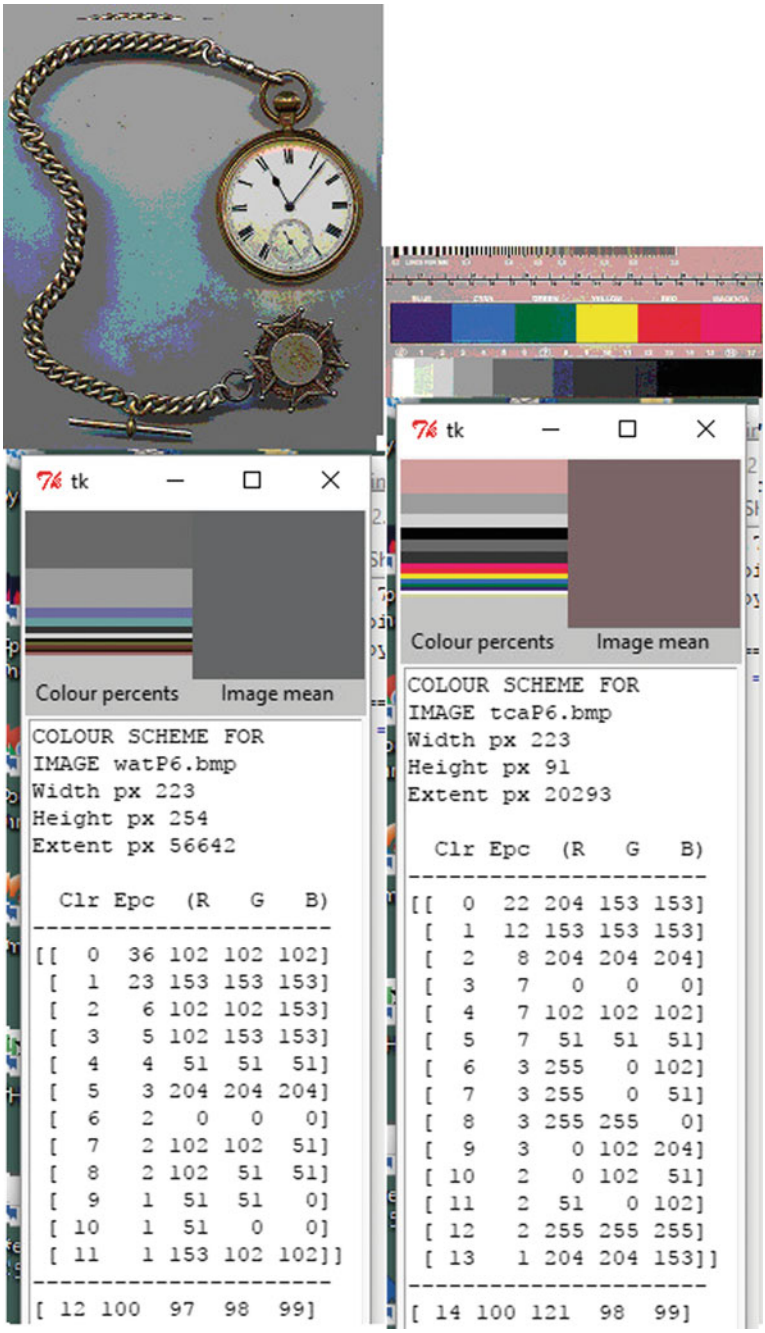


Fig. 9.6 Colour analysis examples. **a** Scanner image of watch 223 × 254 px, converted to palette P6. **b** Scanner image of photographic test card 223 × 91 px, converted to palette P6

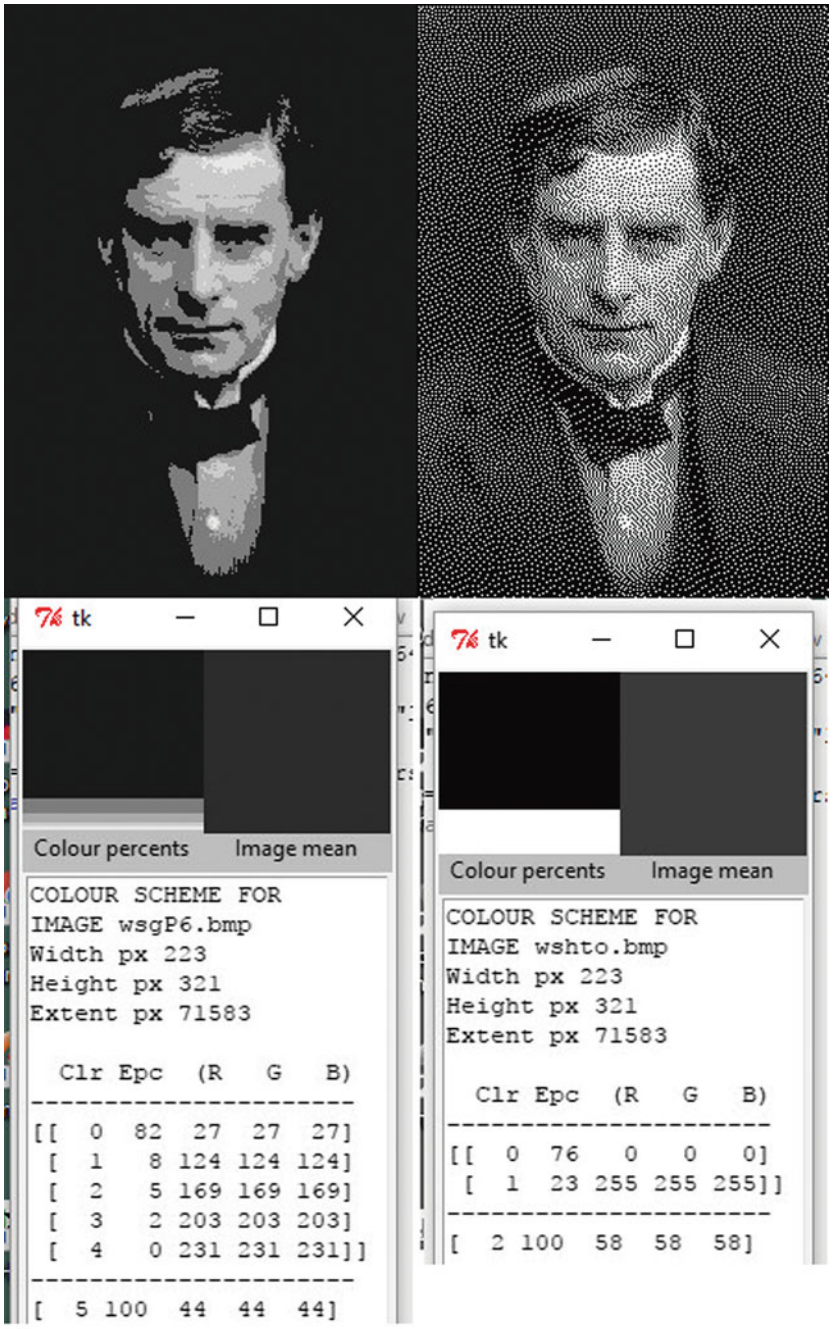


Fig. 9.7 Colour analysis examples. **a** Camera image of Walter Sickert by G.C.Beresford 1911, downloaded from the Internet, diminished to 223 × 321 px, and converted to palette N6. **b** Image (a) converted to halftone

Figure 9.7a is a camera image downloaded from the Internet, diminished and converted to neutral palette N6. It has five greys, with near-black (27, 27, 27) heavily dominant, and very pale (231, 231, 231) insignificant. (b) is (a) converted from greyscale to black-and-white halftone.

Reference

1. Statistics. https://en.wikipedia.org/wiki/Statistics

Chapter 10
Balancing Image Colour

10.1 Neutral Colour Balance

Doctrines of colour harmony, that is to say combinations of colours which are pleasing in nature and art, are many and varied [1]. An old and persistent theme is that of *neutral colour balance*, that is to say a set of colours which are in some sense equilibrated around a middle grey. From a distance, the effect is repose, where nothing should be added nor anything taken away. From closer, the components emerge as a bloom of varied intensities and extents, belonging to identifiable objects. The idea was developed systematically with remarkable insight and taste by George Field [2] and applied with vigour to the ornamental arts by Owen Jones [3]. Unfortunately, they were working with pigments, and an unsatisfactory RYB subtractive primary model (and even more unsatisfactory secondary and tertiary mixtures). However, with an sRGB additive colour space and numerical methods, the principle of neutral balance can be realized in a straightforward and precise way.

A *balanced colour scheme* is one where the mean colour of the whole image is mid-grey (127 127 127). That is to say, the average value of the intensities in the R channel, weighted by their respective extents, is 127; and likewise in the G channel and in the B channel. The image has equal amounts of red, green and blue, and is thus overall neutral in colour.

Given an unbalanced image, we can balance it in either of the following two ways:

Change some or all of the colours, to get a neutral balance.
Adjoin a coloured frame, to get a neutral ensemble image-plus-frame.

10.2 Balancing by Changing Colours

To balance the colours of a given sRGB image, the steps are as follows:

Analyse the colour scheme of given image IMIN.BMP.

A. Parkin, *Computing Colour Image Processing*,
https://doi.org/10.1007/978-3-319-74076-8_10

Calculate changes needed for a balanced colour scheme.
Create balanced image IMOUT.BMP.

Computationally, a colour scheme table has the general form:

```
 Clr  Epc  (R   G   B)
----------------------
[[C0  Epc0 R0  G0  B0]
 [C1  Epc1 R1  G1  B1]
 [... ... ... ... ...]
 [Ck  Epck Rk  Gk  Bk]]
----------------------
[ N   100 Rwa Gwa Bwa]
```

as described in Chap. 9. In this table, we have three equations relating the values:

$$(R0 * Epc0/100) + (R1 * Epc1/100) + \ldots + (Rk * Epck/100) = Rav \quad (10.1)$$

$$(G0 * Epc0/100) + (G1 * Epc1/100) + \ldots + (Gk * Epck/100) = Gav \quad (10.2)$$

$$(B0 * Epc0/100) + (B1 * Epc1/100) + \ldots + (Bk * Epck/100) = Bav \quad (10.3)$$

We can use these equations to achieve a balanced colour scheme in the following way:

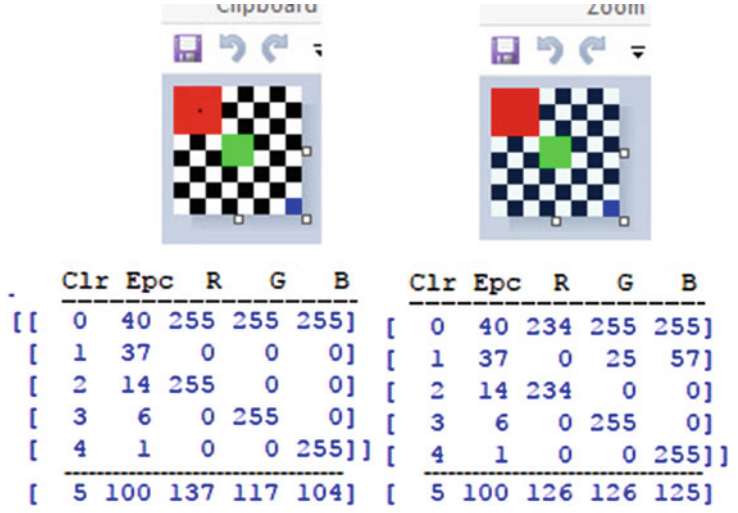


Fig. 10.1 Balancing image colour. On the left, original image is 8 × 8 px. Colour 0 is white (40%), 1 is black (37%), 2 is red (14%), 3 is green (6%) and 4 is blue (1%). At the foot of the table, the image mean (137, 117, 104) is high in red, low in green and lower still in blue. On the right, balanced image has colours in the same extents as the original, but the white is a little cyanish, the black is also a bluer cyanish, the red is a little duller and the green and blue are unchanged. At the foot of the table, the image mean (126, 126, 125) is balanced as middle grey (within the integer rounding of the calculations)

In the R-equation, if the weighted average Rav < 127, there is too little red by an additive term of (127 - Rav); so to get a balance, we can increase the major red intensity R0 by (term/Epc0), keeping all percentage extents unchanged. If this would result in an intensity greater than 255, we can apply the increase to the next-largest red R1.

But if Rav > 127, there is too much red by a factor of (127/Rav); so to get a balance, we can reduce all red intensities Ri by the same factor, keeping the percentage extents unchanged. Since factor 127/Rav is less than 1, this cannot overshoot intensity 255 nor undershoot intensity 0.

And similarly in the G-equation and in the B-equation.

Figure 10.1 shows a simple example of balancing a colour scheme: the imbalance and adjustment is small.

10.3 Script for Balancing by Changing Image Colours

The following script balances a given colour scheme:

```
# pybala: Converts a .bmp image with unbalanced
# colour to balanced mean (127 127 127). First
# analyzes the given image, exactly as
# pyanalyz.py; then calculates balanced image
# and shows it in Paint, to be saved where
# wanted.
# Written by Alan Parkin 2017.

from PIL import Image
import numpy as np
import os, sys

# Get filename of .bmp image which is in working directory
imagefilename = raw_input("Image filename? ")
#Open it and show it
imrgb = Image.open(imagefilename, 'r')
imrgb.show()
width = imrgb.size[0]
height = imrgb.size[1]
extent = width * height
print " "
print "Image file ",imagefilename
print "Width px ", width
print "Height px ", height
print "Extent px ", extent


# Call PIL getcolors for this image, and length.
gc = imrgb.getcolors()
gclist = list(gc)
lengclist = len(gclist)
# Convert gclist to inlist, omitting items where
# count of px is less than 1 percent of extent..
inlist = []
leninlist = 0
for i in range(lengclist):
    count = gclist[i][0]
    if count > (extent /100.):
        inlist.append(gclist[i][0])
```

```
        leninlist = leninlist + 1
        for j in range(3):
            thistuple = gclist[i][1]
            thisitem = thistuple[j]
            inlist.append(thisitem)

#Create numpy array of one dimension.
narra = np.array(list(inlist))

# Reshape as leninlist x 4.
narrb = narra.reshape((leninlist, 4))

# Sort rows by column 0, ascending.
narrc = narrb[narrb[:,0].argsort()]
# flip updown for descending..
narrd = np.flipud(narrc)

# Create index list, then index column.
indlis = []
for i in range(leninlist):
    indlis.append(i)


indarr = np.array(list(indlis))
indcol = indarr.reshape(leninlist, 1)

# Create pc list, then pc column.
pclis =[]
for i in range(leninlist):
    pc = (narrd[i][0] * 100) / extent
    pclis.append(pc)

pcarr = np.array(list(pclis))
pccol = pcarr.reshape(leninlist, 1)

# Delete col 0 from narrd.
narre = np.delete(narrd,0,1)

#flatten narre.
narreflat = list(narre.flatten())


# Concatenate indcol and pccol to narre.
x = indcol
y = pccol
z = narre
narrf = np.concatenate((x,y,z), axis=1)
print "Unbalanced colour scheme "
print "  Clr Epc   R   G   B"
print "  ------------------- "
print narrf

# Create weighted-average row.
rw = 0
gw = 0
bw = 0
for i in range(leninlist):
    rw = rw + (narrf[i,1] * narrf[i,2])
    gw = gw + (narrf[i,1] * narrf[i,3])
    bw = bw + (narrf[i,1] * narrf[i,4])

rwa = rw / 100
gwa = gw / 100
bwa = bw / 100
wavrowlist = (leninlist, 100, rwa, gwa, bwa)
wavrow = np.array(wavrowlist)
print "  -------------------"
```

```
print" ", wavrow
print " "


# Create nnarre.
nnarre = narre

# Calculate new R' values in nnarre.
if rwa < 127:
    addterm = (127 - rwa) * 100 / narrf[0,1]
    if (nnarre[0,0] + addterm) <= 255:
        nnarre[0,0] = nnarre[0,0] + addterm
    else:
        if (narre[1,0] + addterm) <= 255:
            narre[i,0] = narre[i,0] + addterm

if rwa > 127:
    mulfac = (12700 / rwa)
    for i in range(leninlist):
        nnarre[i,0] = (nnarre[i,0] * mulfac) / 100

# Calculate new G' values in nnarre.
if gwa < 127:
     addterm = (127 - gwa) * 100 / narrf[0,1]
     if (nnarre[0,1] + addterm) <= 255:
         nnarre[0,1] = nnarre[0,1] + addterm
     else:
         nnarre[1,1] = nnarre[1,1] + addterm

if gwa > 127:
    mulfac = (12700 / gwa)
    for i in range(leninlist):
        nnarre[i,1] = (nnarre[i,1] * mulfac) / 100

# Calculate new B' values in narrf.
if bwa < 127:
     addterm = (127 - bwa) * 100 / narrf[0,1]
     if (nnarre[0,2] + addterm) <= 255:
         nnarre[0,2] = nnarre[0,2] + addterm
     else:
         nnarre[1,2] = nnarre[1,2] + addterm

if bwa > 127:
    mulfac = (12700 / bwa)
    for i in range(leninlist):
        nnarre[i,2] = (nnarre[i,2] * mulfac) / 100

#flatten nnarre.
nnarreflat = list(nnarre.flatten())

# Concatenate indcol and pccol to nnarre.
x = indcol
y = pccol
z = nnarre
nnarrf = np.concatenate((x,y,z), axis=1)
print "Balanced colour scheme "
print "  Clr Epc   R   G   B"
print "  ------------------- "
print nnarrf


# Create new weighted-average row.
nrw = 0
ngw = 0
nbw = 0
for i in range(leninlist):
    nrw = nrw + (nnarrf[i,1] * nnarrf[i,2])
```

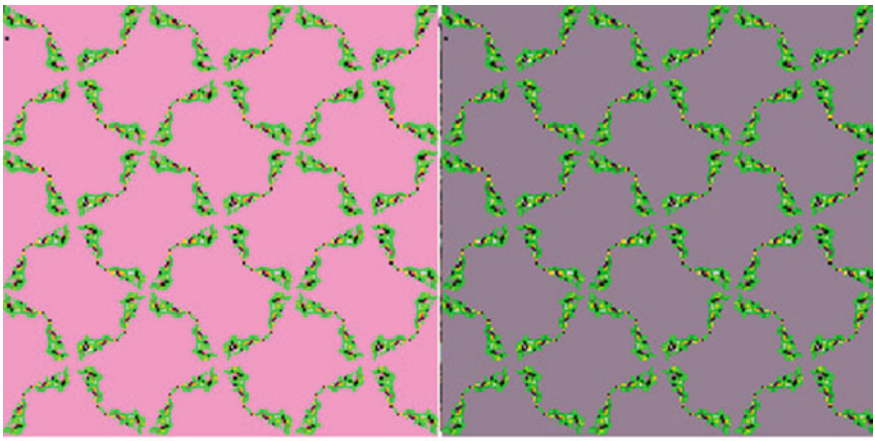
```
        ngw = ngw + (nnarrf[i,1] * nnarrf[i,3])
        nbw = nbw + (nnarrf[i,1] * nnarrf[i,4])

    nrwa = nrw / 100
    ngwa = ngw / 100
    nbwa = nbw / 100
    nwavrowlist = (leninlist, 100, nrwa, ngwa, nbwa)
    nwavrow = np.array(nwavrowlist)
    print "  --------------------"
    print" ",nwavrow
    print " "

    # Call PIL load for this image, to get xy array of rgb tuples.
    pix = imrgb.load()

    # Convert xy array of tuples to xy array of lists.
    pixlist = []
    for y in range(height):
        for x in range(width):
            for j in range(3):
                thisval = pix[x,y][j]
                pixlist.append(thisval)

    # Compare colour triples.
    for i in range(extent):
        for j in range(leninlist):
           # print "i j narreflat nnarreflat "
           # print i, j, narreflat[(j*3):((j*3)+3)], nnarreflat[(j*3):((j*3)+3)]
            if pixlist[(i*3):((i*3)+3)] == narreflat[(j*3):((j*3)+3)]:
                pixlist[(i*3):((i*3)+3)] = nnarreflat[(j*3):((j*3)+3)]



    # Apply reduced values to image.
    m = width * 3
    for y in range(height):
        for x in range(width):
            pix[x,y] = (pixlist[(3*x)+(m*y)], pixlist[(3*x)+(m*y)+1], pixlist[(3*x)+(m*y)+2])

    # Show reduced image in Paint, to be saved
    # where you will.
    imrgb.show()

    print "Balancing done "
```

10.4 Examples of Balancing by Changing Colours

Figures 10.2, 10.3, 10.4, 10.5 and 10.6 show examples of various types of image before and after balancing colour using the script listed above.

In Fig. 10.2a, the weighted mean (213, 151, 172) of the image is too high in red green and blue. In (b), the intensity values in the R column have been reduced proportionately to their extents, so that their weighted mean is 125; and similarly the intensity values in the G column and the B column. The resulting image is balanced around mid-grey (127, 127, 127) (allowing for rounding of integer values).

In Fig. 10.3a, the weighted mean (178, 178, 178) of the image is too high in red green and blue. In (b), the intensity values of colour 0 have been reduced propor-



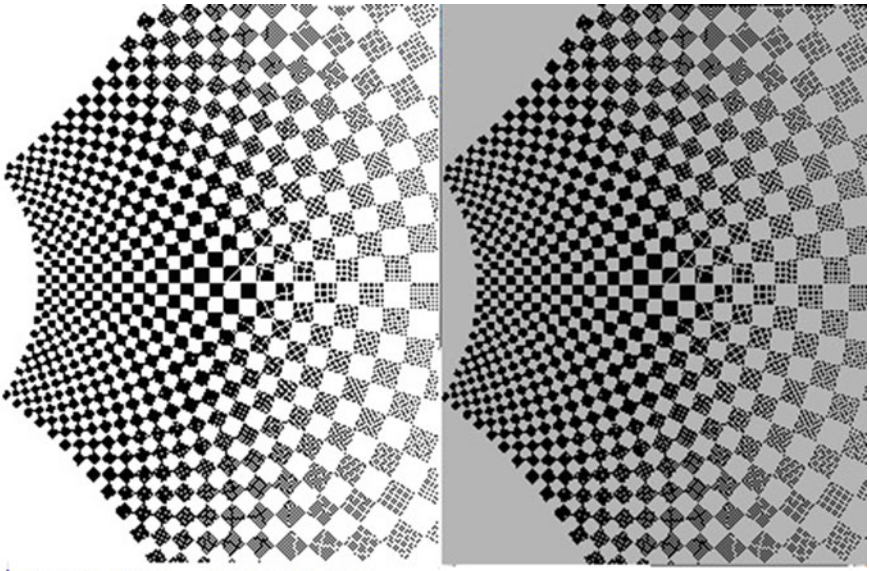
```
Image file  repP6.bmp
Width px  223
Height px  221
Extent px  49283
Unbalanced colour scheme
  Clr Epc   R   G   B
  ------------------
[[  0  83 255 153 204]
 [  1   6   0 255   0]
 [  2   4   0 204  51]
 [  3   1 204 102 102]
 [  4   1   0   0   0]]
  ------------------
  [  5 100 213 151 172]
```

```
Balanced colour scheme
  Clr Epc   R   G   B
  ------------------
[[  0  83 150 128 148]
 [  1   6   0 214   0]
 [  2   4   0 171  37]
 [  3   1 120  85  74]
 [  4   1   0   0   0]]
  ------------------
  [  5 100 125 126 125]
```

Fig. 10.2 Balancing a colour scheme. **a** Repeating pattern created by GUI image editor, 223 × 221 px, converted to palette P6. **b** After balancing

tionately to their extents, so that their weighted mean is 126. The resulting image is balanced around mid-grey (127, 127, 127) (allowing for rounding to integer values).

In Fig. 10.4a, the camera image has 26 colours (some omitted here to fit in the page), which are close in diversity and close in extent. The weighted mean (142, 143, 131) is almost balanced, but at a somewhat high intensity. In (b), the intensity values of all the colours have been reduced proportionately to their extents, so that their weighted mean is (126, 125, 125). The resulting image is of slightly lower intensity throughout.



```
Image file  twinpP6.bmp
Width px  223
Height px  288
Extent px  64224
Unbalanced colour scheme
  Clr Epc   R   G   B
  ------------------
[[  0  70 255 255 255]
 [  1  29   0   0   0]]
  ------------------
  [  2 100 178 178 178]
```

```
Balanced colour scheme
  Clr Epc   R   G   B
  ------------------
[[  0  70 181 181 181]
 [  1  29   0   0   0]]
  ------------------
  [  2 100 126 126 126]
```

Fig. 10.3 Balancing a colour scheme. **a** Inversion of chequer created by program, 223 × 288 px, converted to palette P6. **b** After balancing

In Fig. 10.5a, the scanner image has 12 colours, with the top two greys dominant. The weighted mean (97, 98, 99) is balanced, but at a rather low intensity. In (b), the intensity values of the top grey have been increased proportionately to its their extent, so that the weighted mean is now (127, 127, 127).

In Fig. 10.6a, the downloaded image has four greys, with the top near-black dominant. The weighted mean (44, 44, 44) is balanced, but at a low intensity. In (b), the intensity values of the near-black have been increased proportionately to its extent, so that the weighted mean is now (127, 127, 127).

```
Image file  apyP6.bmp
Width px  223
Height px  291
Extent px  64893
Unbalanced colour scheme
  Clr Epc   R   G   B
  ------------------
[[  0   9 102 204 255]
 [  1   8 255 204 153]
 [  2   8 255 204 204]
 [  3   7 255 153 153]
 [  4   5 153 204 204]
 [  5   5 102 153  51]
 [  6   4 153 255 255]
 [  7   4 204 153 102]
 [  8   4 255 255 204]
 [  9   3 153 204 255]
 [ 10   3 204 153 153]
 [ 11   3 102 204 204]
 [ 12   2 153 102 102]
 [ 13   2 153 153 102]
 [ 14   2 102 102  51]
 [ 15   2   0   0   0]
 [ 16   1 204 204 153]
 [ 17   1 102 153 102]

 [ 24   1 153 153 153]
 [ 25   1 102 102 102]]
  ------------------
  [ 26 100 142 143 131]
```

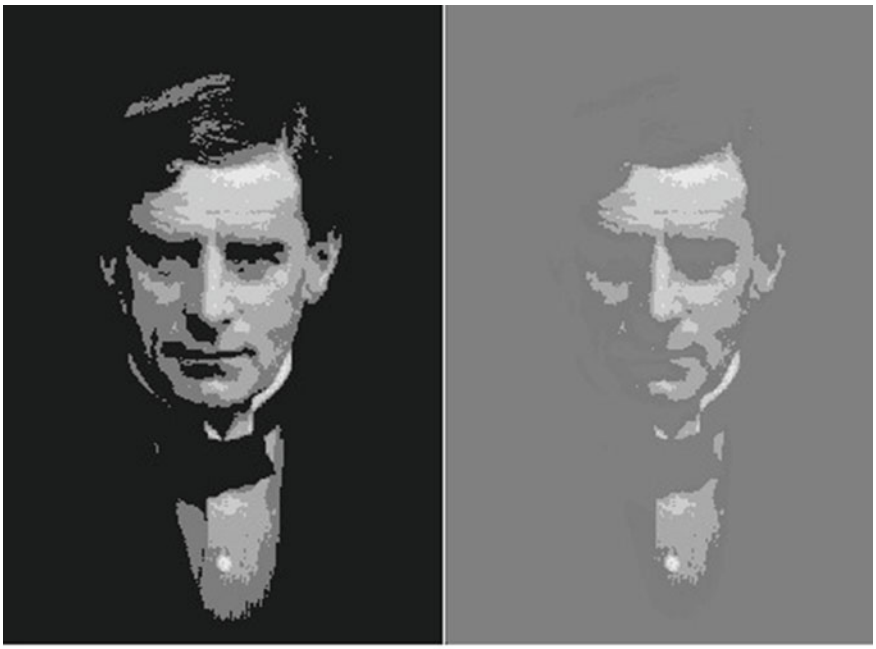
```
Balanced colour scheme
  Clr Epc   R   G   B
  ------------------
[[  0   9  90 179 244]
 [  1   8 226 179 146]
 [  2   8 226 179 195]
 [  3   7 226 134 146]
 [  4   5 136 179 195]
 [  5   5  90 134  48]
 [  6   4 136 224 244]
 [  7   4 181 134  97]
 [  8   4 226 224 195]
 [  9   3 136 179 244]
 [ 10   3 181 134 146]
 [ 11   3  90 179 195]
 [ 12   2 136  89  97]
 [ 13   2 136 134  97]
 [ 14   2  90  89  48]
 [ 15   2   0   0   0]
 [ 16   1 181 179 146]
 [ 17   1  90 134  97]

 [ 24   1 136 134 146]
 [ 25   1  90  89  97]]
  ------------------
  [ 26 100 126 125 125]
```

Fig. 10.4 Balancing a colour scheme. **a** Image created by camera, 223 × 288 px, converted to palette P6. **b** After balancing

```
Image file  watP6.bmp
Width px  223
Height px  254
Extent px  56642
Unbalanced colour schem
  Clr Epc   R   G   B
  -------------------
[[  0  36 102 102 102]
 [  1  23 153 153 153]
 [  2   6 102 102 153]
 [  3   5 102 153 153]
 [  4   4  51  51  51]
 [  5   3 204 204 204]
 [  6   2   0   0   0]
 [  7   2 102 102  51]
 [  8   2 102  51  51]
 [  9   1  51  51   0]
 [ 10   1  51   0   0]
 [ 11   1 153 102 102]]
  -------------------
  [ 12 100  97  98  99]
```

```
Balanced colour scheme
  Clr Epc   R   G   B
  -------------------
[[  0  36 185 182 179]
 [  1  23 153 153 153]
 [  2   6 102 102 153]
 [  3   5 102 153 153]
 [  4   4  51  51  51]
 [  5   3 204 204 204]
 [  6   2   0   0   0]
 [  7   2 102 102  51]
 [  8   2 102  51  51]
 [  9   1  51  51   0]
 [ 10   1  51   0   0]
 [ 11   1 153 102 102]]
  -------------------
  [ 12 100 127 127 127]
```

Fig. 10.5 Balancing a colour scheme. **a** Image created by scanner, 223 × 254 px, converted to palette P6. **b** After balancing



```
Image file  wsgP6.bmp
Width px  223
Height px  321
Extent px  71583
Unbalanced colour scheme
  Clr Epc   R   G   B
  ------------------
[[  0  82  27  27  27]
 [  1   8 124 124 124]
 [  2   5 169 169 169]
 [  3   2 203 203 203]]
  ------------------
  [  4 100  44  44  44]
```

```
Balanced colour scheme
  Clr Epc   R   G   B
  ------------------
[[  0  82 128 128 128]
 [  1   8 124 124 124]
 [  2   5 169 169 169]
 [  3   2 203 203 203]]
  ------------------
  [  4 100 127 127 127]
```

Fig. 10.6 Balancing a colour scheme. **a** Image downloaded from the Internet, 223 × 321 px, converted to palette P6. **b** After balancing (but perhaps not what was wanted!)

10.5 Balancing by Adjoining a Frame

There are many circumstances in which we might want to keep an image in its original colour scheme, yet achieve some kind of balanced norm. (For example, the image may be an accurate representation of a scene, or an already carefully considered arrangement by a painter.) In such cases, we can always adjoin a frame of a colour which achieves an image-plus-frame mean of mid-grey (127, 127, 127).

Traditionally, frames of various materials and designs have been adjoined to images for several reasons. Paintings are framed for convenient attachment to walls, and for visual separation from their surroundings. Water colours and prints are framed with a mat and glass for protection. Images in books are naturally framed by the white page, which separates them from type matter and from their surroundings. There is a long history of changing practice amongst artists, framers, printers and collectors as to frames and margins [4].

A digital image is like an image in a book, in that it is naturally framed by a window which it shares with text matter, on a screen which it shares with other windows. An additive display is temporary, soon giving way to other images and other windows. A digital image for more permanent presentation can be made by subtractive printout. It is then very like an image in a book, and indeed often becomes an image in a book. It may sometimes be treated like a painting, and framed for attachment, protection and visual separation from surroundings.

To balance the colours of a given sRGB image by adjoining a frame, the steps are as follows:

Analyse the colour scheme of given image IMIN.BMP.
Calculate extent and colour of frame needed for a balanced colour scheme.
Create balanced image IMOUT.BMP.

To balance any image mean whatsoever, from (0, 0, 0) to (255, 255, 255), the extent of the frame needs to be (at least) equal to the extent of the image, each 50% of the ensemble. Setting:

$$xW * xH = 2(W * H)$$
$$x^2 * (W * H) = 2 * (W * H)$$
$$x^2 = 2$$
$$x = \sqrt{2}$$

we see that the frame size should be as shown in Fig. 10.7. Furthermore, we can set the margins of the frame around the image however we please; as, for example, in the same figure consistent with classical canons [5].

Computationally, the script below analyses any .bmp image, calculates the mean value of its colour scheme, calculates the balancing colour and adjoins a frame of that colour. User can set margins as a centred arrangement, or as book-style recto/verso:

```
# pyanbafr: Analyze .bmp image, calculate balancing
# colour required, and adjoin frame of that colour.
# Written by Alan Parkin 2017.

from PIL import Image
import numpy as np
import os, sys

# Get filename of .bmp image which is in working directory
imagefilename = raw_input("Image filename? ")
#Open it and show it
imrgb = Image.open(imagefilename, 'r')
imrgb.show()
width = imrgb.size[0]
height = imrgb.size[1]
extent = width * height
```

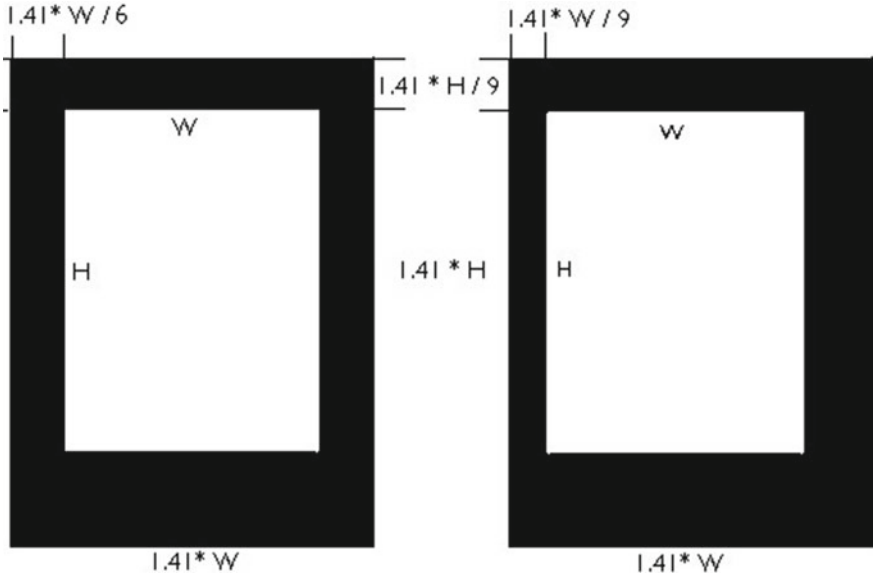


Fig. 10.7 Frame geometry. **a** For a central frame, frame width is 1.41 × image width W, frame height is 1.41 × image height H, top margin is 1/9 of frame height, and side margin is 1/6 of frame width. **b** For a verso–recto page frame, gutter nmargin is 1/9 of frame width

```
# Enter C for centred, R for recto, or V for verso page.
cenrecver = raw_input("Enter C for centred, R for recto, V for verso page? ")
print " "

# Call PIL getcolors for this image, and length.
gc = imrgb.getcolors()
gclist = list(gc)
lengclist = len(gclist)

# Convert gclist to inlist, omitting items where
# count of px is less than 1 percent of extent..
inlist = []
leninlist = 0
for i in range(lengclist):
    count = gclist[i][0]
    if count > (extent / 100.):
        inlist.append(gclist[i][0])
        leninlist = leninlist + 1
        for j in range(3):
            thistuple = gclist[i][1]
            thisitem = thistuple[j]
            inlist.append(thisitem)

#Create numpy array of one dimension.
narra = np.array(list(inlist))
# Reshape as leninlist x 4.
narrb = narra.reshape((leninlist, 4))

# Sort rows by column 0, ascending.
narrc = narrb[narrb[:,0].argsort()]
```

```
# flip updown for descending..
narrd = np.flipud(narrc)


# Create index list, then index column.
indlis = []
for i in range(leninlist):
    indlis.append(i)

indarr = np.array(list(indlis))
indcol = indarr.reshape(leninlist, 1)

# Create pc list, then pc column.
pclis =[]
for i in range(leninlist):
    pc = (narrd[i][0] * 100) / extent
    pclis.append(pc)

pcarr = np.array(list(pclis))
pccol = pcarr.reshape(leninlist, 1)

# Delete col 0 from narrd.
narre = np.delete(narrd,0,1)

# Concatenate indcol and pccol to narre.
x = indcol
y = pccol
z = narre
narrf = np.concatenate((x,y,z), axis=1)


# Create weighted-average row.
rw = 0
gw = 0
bw = 0
for i in range(leninlist):
    rw = rw + (narrf[i,1] * narrf[i,2])
    gw = gw + (narrf[i,1] * narrf[i,3])
    bw = bw + (narrf[i,1] * narrf[i,4])

rwa = rw / 100
gwa = gw / 100
bwa = bw / 100
wavrowlist = (leninlist, 100, rwa, gwa, bwa)
wavrow = np.array(wavrowlist)

# Calculate balancing colour for frame.
if rwa >= 128:
    rexcess = rwa - 128
    rbal = 128 - rexcess

if rwa <= 127:
    rdeficit = 127 - rwa
    rbal = 127 + rdeficit

if gwa >= 127:
    gexcess = gwa - 127
    gbal = 127 - gexcess

if gwa < 127:
    gdeficit = 127 - gwa
    gbal = 127 + gdeficit

if bwa >= 127:
    bexcess = bwa - 127
    bbal = 127 - bexcess
```

```
if bwa < 127:
    bdeficit = 127 - bwa
    bbal = 127 + bdeficit

# Calculate frame for centred or recto or verso.
framewidth = int(width * 1.414)
frameheight = int(height * 1.414)
imoffsety = frameheight // 9
if cenrecver == "C":
    imoffsetx = (framewidth - width) // 2

if cenrecver == "R":
    imoffsetx = framewidth // 9

if cenrecver == "V":
    imoffsetx = ((framewidth - width) - framewidth // 9)


print "Image width px", width
print "Image height px", height
print "Image extent px", width * height
print "Frame width px", framewidth
print "Frame height px", frameheight
print "Frame extent px", framewidth * frameheight
print "Centred/recto/verso ", cenrecver
print "Image offset x, y ", imoffsetx, imoffsety
print "Image colour ",rwa,gwa,bwa
print "Frame colour ",rbal,gbal,bbal

# Do frame, insert image, and show.
frame = Image.new("RGB", (framewidth,frameheight), (rbal, gbal, bbal))
frame. paste(imrgb, (imoffsetx, imoffsety), 0)
frame.show()

print " "
print "Framing done"
```

10.6 Examples of Balancing by Adjoining a Frame

Figures 10.8, 10.9, 10.10, 10.11 and 10.12 show examples of various types of image after balancing colour by a frame, using the script listed above.

In Fig. 10.8a, the incoming image is the repeating pattern shown in Fig. 10.2a, dilated down to $1/\sqrt{2} = 71\%$ to allow for the subsequent frame. (Notice that the extent percentages are unaffected by dilations.) The image has mean (211, 152, 170): too high in red green and blue. Script pyanbafr calculates a balancing frame extent equal to the image extent, a balancing frame colour (45, 102, 84), and imposes the image centred left/right in the frame. Figure 10.8b is the colour scheme of the balanced frame-plus-image ensemble, done by scripts pyanalyzA and pyanalyzB. This shows the frame as the new top colour 0 occupying 50% of the ensemble.

In Fig. 10.9a, the incoming image is the inversion shown in Fig. 10.3a, dilated down to $1/\sqrt{2} = 71\%$ to allow for the subsequent frame. The image has mean (178, 178, 178): too high in red green and blue. Script pyanbafr calculates a balancing frame extent equal to the image extent, a balancing frame colour (78, 76, 76), and

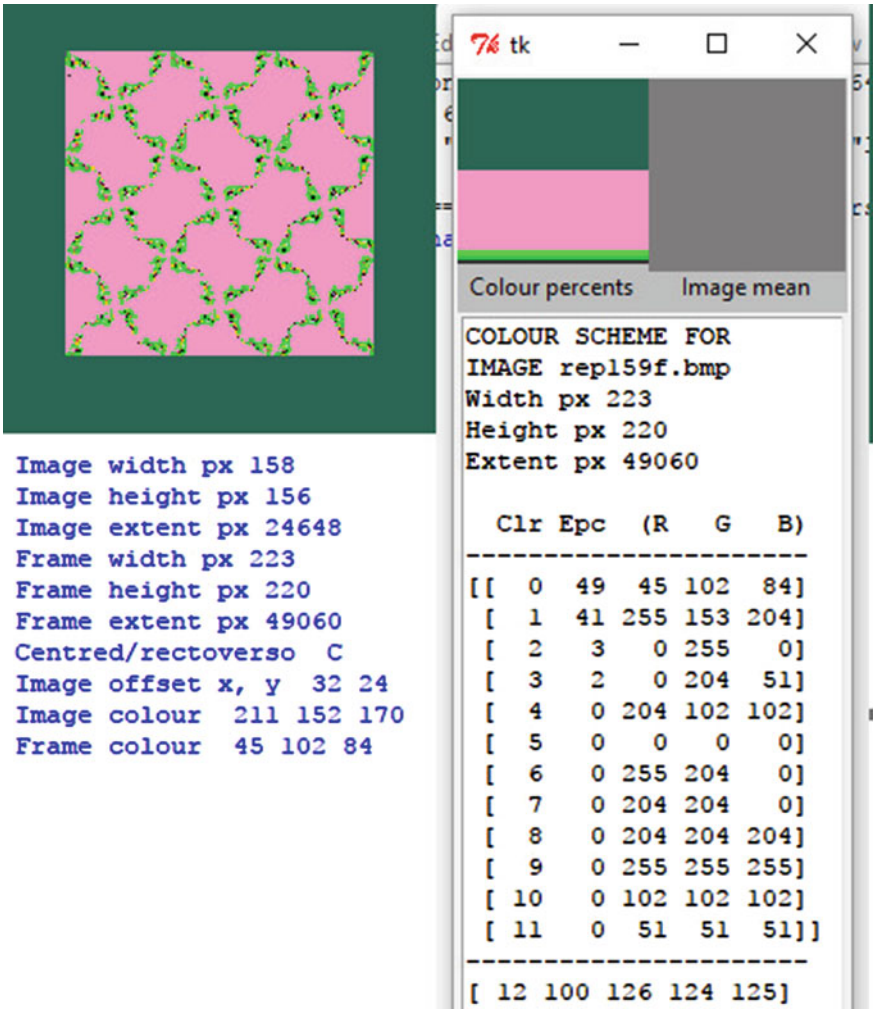


Fig. 10.8 Balancing by framing. **a** Repeating pattern created by image editor 159 × 156 px and converted to P6, then balanced by frame. **b** Colour scheme of frame-plus-image

imposes the image centred in the frame. Figure 10.9b is the colour scheme of the balanced frame-plus-image ensemble, done by scripts pyanalyzA and pyanalyzB. This shows the frame as the new top colour 0 occupying 50% of the ensemble.

In Fig. 10.10a, the incoming image is the camera image shown in Fig. 10.4a, dilated down to $1/\sqrt{2} = 71\%$ to allow for the subsequent frame. The image has mean (143,142,131): too high in red and green and less so in blue. Script pyanbafr calculates a balancing frame extent equal to the image extent, a balancing frame colour (113, 111, 123), and imposes the image centred in the frame. Figure 10.10b is the colour scheme of the balanced frame-plus-image ensemble, done by scripts

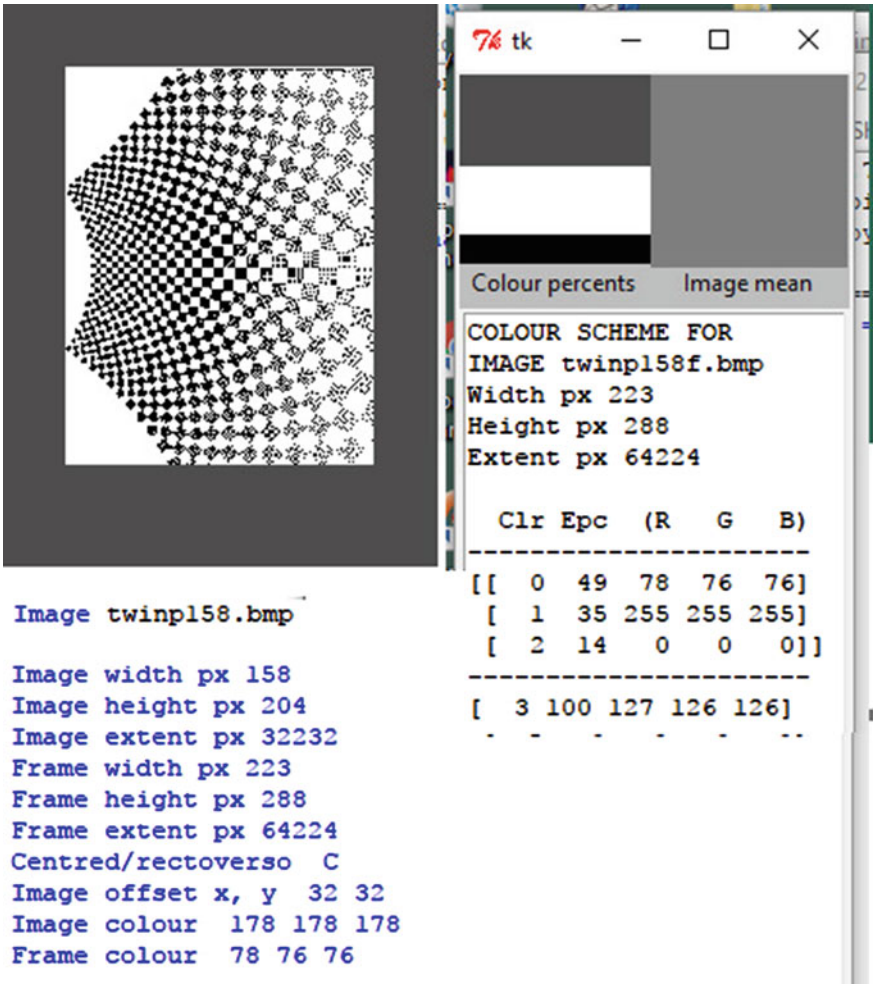


Fig. 10.9 Balancing by framing. **a** Inversion of chequer created by program, 158 × 204 px and converted to P6, then balanced by frame. **b** Colour scheme of frame-plus-image

pyanalyzA and pyanalyzB. This shows the frame as the new top colour 0 occupying 50% of the ensemble.

In Fig. 10.11a, the incoming image is the scanner image shown in Fig. 10.5a, dilated down to $1/\sqrt{2} = 71\%$ to allow for the subsequent frame. The image has mean (97, 98, 99): too low in red green and blue. Script pyanbafr calculates a balancing frame extent equal to the image extent, a balancing frame colour (157, 156, 155), and imposes the image centred in the frame. Figure 10.10b is the colour scheme of the balanced frame-plus-image ensemble, done by scripts pyanalyzA and pyanalyzB. This shows the frame as the new top colour 0 occupying 50% of the ensemble.

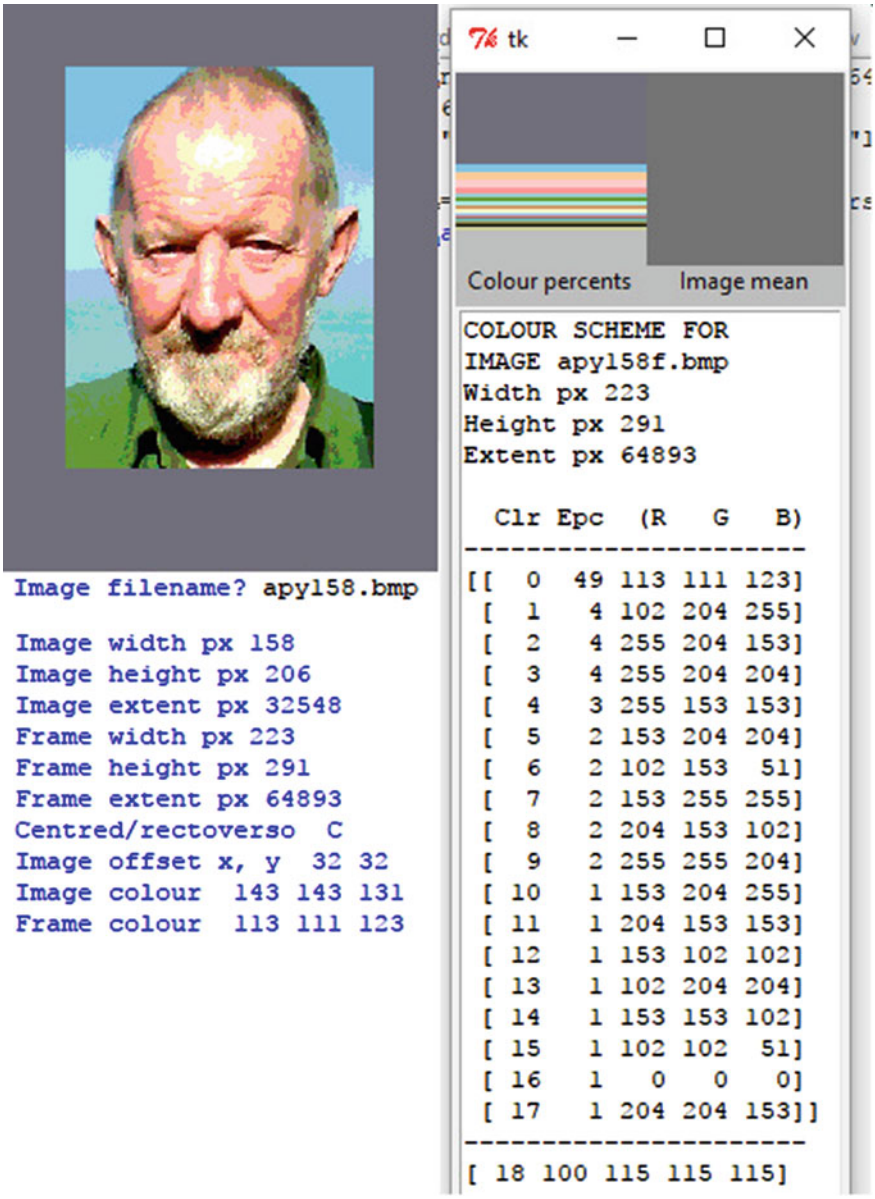


Fig. 10.10 Balancing by framing. **a** Camera image 158 × 206 px and converted to P6, then balanced by frame. **b** Colour scheme of frame-plus-image

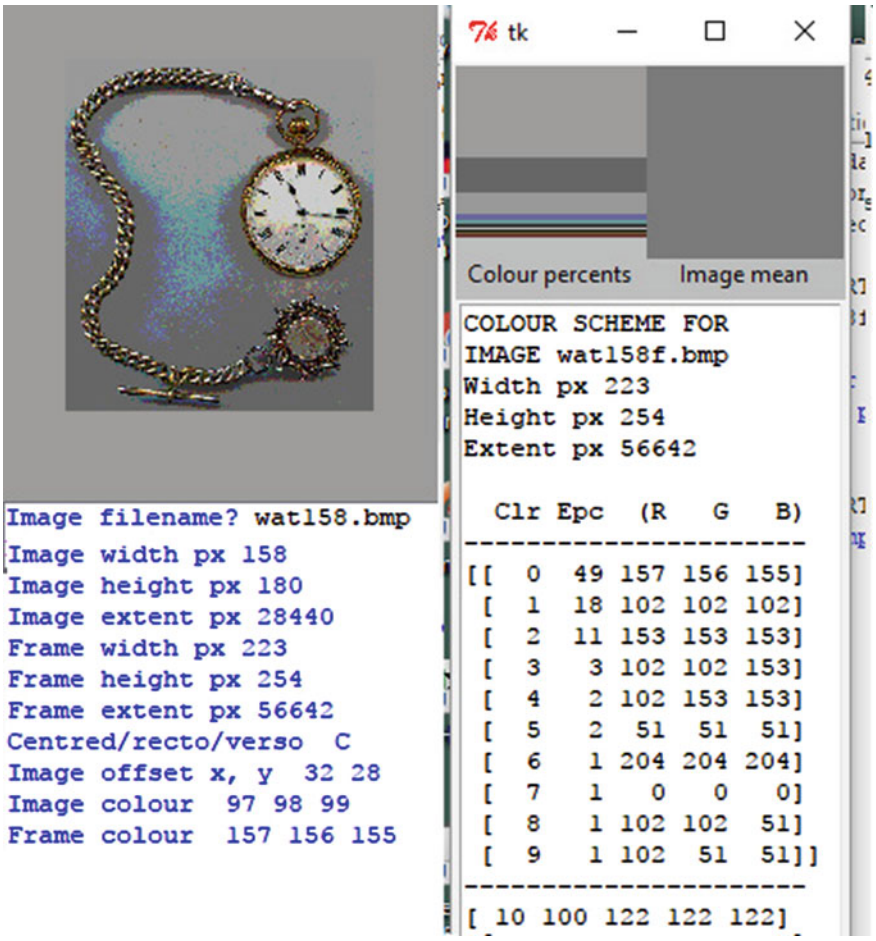


Fig. 10.11 Balancing by framing. **a** Scanner image 158 × 180 px and converted to P6, then balanced by frame. **b** Colour scheme of frame-plus-image

In Fig. 10.12a the incoming image is the greyscale image downloaded from the Internet shown in Fig. 10.6a, dilated down to $1/\sqrt{2} = 71\%$ to allow for the subsequent frame. The image has mean (44, 44, 44): very low in red green and blue. Script pyanbafr calculates a balancing frame extent equal to the image extent, a balancing frame colour (210, 210, 210), and imposes the image centred in the frame. Figure 10.12b is the colour scheme of the balanced frame-plus-image ensemble, done by scripts pyanalyzA and pyanalyzB. This shows the frame as the new top colour 0 occupying 50% of the ensemble.

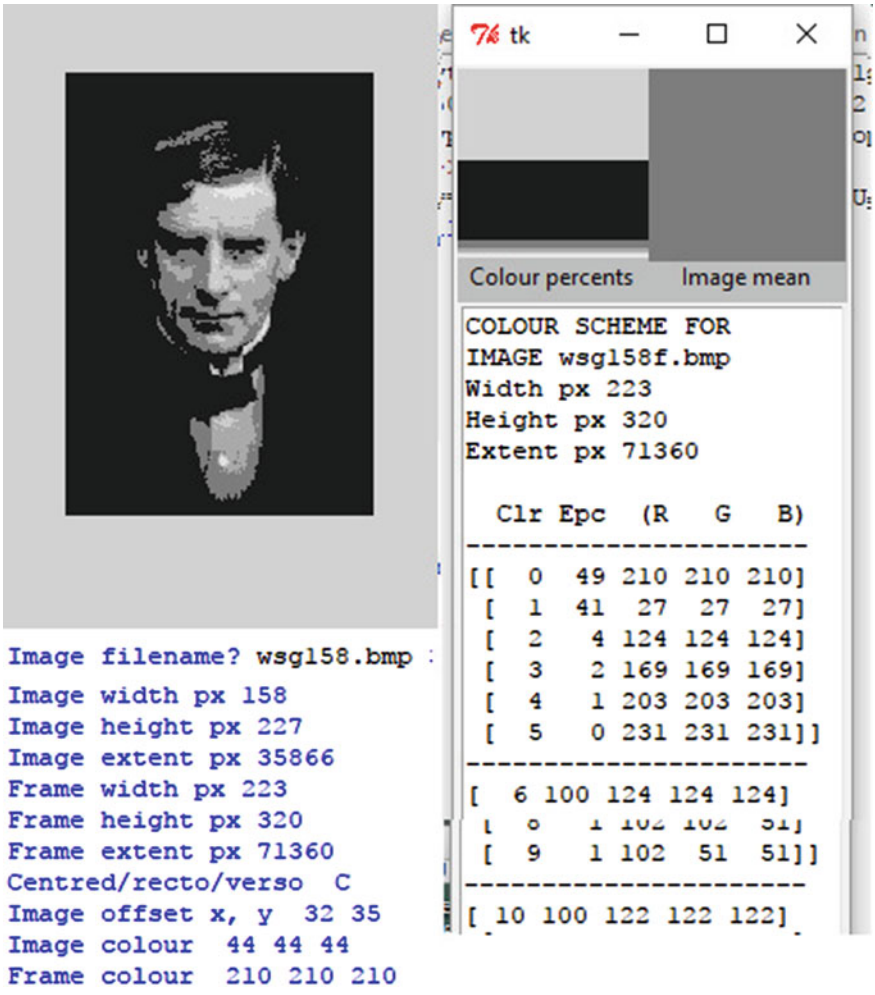


Fig. 10.12 Balancing by framing. **a** Greyscale image downloaded from the internet 158 × 227 px and converted to P6, then balanced by frame. **b** Colour scheme of frame-plus-image

10.7 Why Balance?

A short answer is: because, in sRGB imaging, we computationally can. Balancing gives a canonical normal form to any image which is worth it.

More generally, balancing is a powerful tool for training one's understanding of and sensitivity to colour harmony. The facts are often surprising and sometimes useful, amongst the tangle of personal preferences and prejudices surrounding colour.

References

1. Gage J (1993) Colour and culture. Thames and Hudson, London
2. Field G (1835) Chromatography. Charles Tilt, London
3. Jones O (1856) The grammar of ornament. Day, London
4. Tschichold J (1991) The form of the book: essays on the moraity of good design. Hartley and Marks, Point Roberts, WA
5. Canons of page construction. https://en.wikipedia.org/wiki/Canons_of_page_construction

Index

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