Chapter 2

Image Representation

2.1 Introduction

Computer Graphics is principally concerned with the generation of images, with wide ranging applications from entertainment to scientific visualisation. In this chapter we begin our exploration of Computer Graphics by introducing the fundamental data structures used to represent images on modern computers. We describe the various formats for storing and working with image data, and for representing colour on modern machines.

2.2 The Digital Image

Virtually all computing devices in use today are digital; data is represented in a discrete form using patterns of binary digits (**bits**) that can encode numbers within finite ranges and with limited precision. By contrast, the images we perceive in our environment are analogue. They are formed by complex interactions between light and physical objects, resulting in continuous variations in light wavelength and intensity. Some of this light is reflected in to the retina of the eye, where cells convert light into nerve impulses that we interpret as a visual stimulus.

Suppose we wish to 'capture' an image and represent it on a computer e.g. with a scanner or camera (the machine equivalent of an eye). Since we do not have infinite storage (bits), it follows that we must convert that analogue signal into a more limited digital form. We call this conversion process **sampling**. Sampling theory is an important part of Computer Graphics, underpinning the theory behind both image capture and manipulation — we return to the topic briefly in Chapter 4 (and in detail next semester in CM20220).

For now we simply observe that a digital image can not encode arbitrarily fine levels of detail, nor arbitrarily wide (we say 'dynamic') colour ranges. Rather, we must compromise on accuracy, by choosing an appropriate method to sample and store (i.e. represent) our continuous image in a digital form.

2.2.1 Raster Image Representation

The Computer Graphics solution to the problem of **image representation** is to break the image (picture) up into a regular grid that we call a '**raster**'. Each grid cell is a 'picture cell', a term often contracted to **pixel**. The pixel is the atomic unit of the image; it is coloured uni-

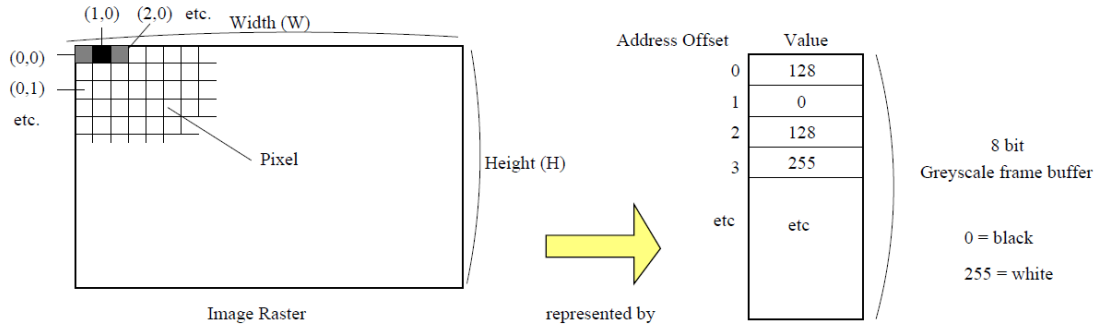

Figure 2.1: Rasters are used to represent digital images. Modern displays use a rectangular raster, comprised of $W \times H$ pixels. The raster illustrated here contains a greyscale image; its contents are represented in memory by a greyscale frame buffer. The values stored in the frame buffer record the intensities of the pixels on a discrete scale (0=black, 255=white).

formly — its single colour representing a discrete sample of light e.g. from a captured image. In most implementations, rasters take the form of a rectilinear grid often containing many thousands of pixels (Figure 2.1). The raster provides an orthogonal two-dimensional basis with which to specify pixel coordinates. By convention, pixels coordinates are zero-indexed and so the origin is located at the **top-left** of the image. Therefore pixel $(W-1, H-1)$ is located at the bottom-right corner of a raster of width W pixels and height H pixels. As a note, some Graphics applications make use of hexagonal pixels instead [1], however we will not consider these on the course.

The number of pixels in an image is referred to as the image's **resolution**. Modern desktop displays are capable of visualising images with resolutions around 1024×768 pixels (i.e. a million pixels or one **mega-pixel**). Even inexpensive modern cameras and scanners are now capable of capturing images at resolutions of several mega-pixels. In general, the greater the resolution, the greater the level of spatial detail an image can represent.

2.2.2 Hardware Frame Buffers

We represent an image by storing values for the colour of each pixel in a structured way. Since the earliest computer Visual Display Units (VDUs) of the 1960s, it has become common practice to reserve a large, **contiguous** block of memory specifically to manipulate the image currently shown on the computer's display. This piece of memory is referred to as a **frame buffer**. By reading or writing to this region of memory, we can read or write the colour values of pixels at particular positions on the display[2].

Note that the term 'frame buffer' as originally defined, strictly refers to the area of memory reserved for direct manipulation of the currently displayed image. In the early days of

[1] Hexagonal displays are interesting because all pixels are equidistant, whereas on a rectilinear raster neighbouring pixels on the diagonal are $\sqrt{(2)}$ times further apart than neighbours on the horizontal or vertical.

[2] Usually the frame buffer is not located on the same physical chip as the main system memory, but on separate graphics hardware. The buffer 'shadows' (overlaps) a portion of the logical address space of the machine, to enable fast and easy access to the display through the same mechanism that one might access any 'standard' memory location.

Graphics, special hardware was needed to store enough data to represent just that single image. However we may now manipulate hundreds of images in memory simultaneously, and the term 'frame buffer' has fallen into informal use to describe any piece of storage that represents an image.

There are a number of popular formats (i.e. ways of encoding pixels) within a frame buffer. This is partly because each format has its own advantages, and partly for reasons of backward compatibility with older systems (especially on the PC). Often video hardware can be switched between different **video modes**, each of which encodes the frame buffer in a different way. We will describe three common frame buffer formats in the subsequent sections; the greyscale, pseudo-colour, and true-colour formats. If you do Graphics, Vision or mainstream Windows GUI programming then you will likely encounter all three in your work at some stage.

2.2.3 Greyscale Frame Buffer

Arguably the simplest form of frame buffer is the greyscale frame buffer; often mistakenly called 'black and white' or 'monochrome' frame buffers. Greyscale buffers encodes pixels using various shades of grey. In common implementations, pixels are encoded as an unsigned integer using 8 bits (1 byte) and so can represent $2^8 = 256$ different shades of grey. Usually black is represented by value 0, and white by value 255. A mid-intensity grey pixel has value 128. Consequently an image of width W pixels and height H pixels requires $W \times H$ bytes of memory for its frame buffer.

The frame buffer is arranged so that the first byte of memory corresponds to the pixel at coordinates $(0, 0)$. Recall that this is the top-left corner of the image. Addressing then proceeds in a left-right, then top-down manner (see Figure 2.1). So, the value (grey level) of pixel $(1, 0)$ is stored in the second byte of memory, pixel $(0, 1)$ is stored in the $(W + 1)th$ byte, and so on. Pixel (x, y) would be stored at buffer offset A where:

$$A = x + Wy \tag{2.1}$$

i.e. A bytes from the start of the frame buffer. Sometimes we use the term **scan-line** to refer to a full row of pixels. A scan-line is therefore W pixels wide.

Old machines, such as the ZX Spectrum, required more CPU time to iterate through each location in the frame buffer than it took for the video hardware to refresh the screen. In an animation, this would cause undesirable flicker due to partially drawn frames. To compensate, byte range $[0, (W - 1)]$ in the buffer wrote to the first scan-line, as usual. However bytes $[2W, (3W - 1)]$ wrote to a scan-line one third of the way down the display, and $[3W, (4W - 1)]$ to a scan-line two thirds down. This interleaving did complicate Graphics programming, but prevented visual artifacts that would otherwise occur due to slow memory access speeds.

2.2.4 Pseudo-colour Frame Buffer

The **pseudo-colour frame buffer** allows representation of colour images. The storage scheme is identical to the greyscale frame buffer. However the pixel values do not represent shades of grey. Instead each possible value $(0 - 255)$ represents a particular colour; more

specifically, an index into a list of 256 different colours maintained by the video hardware.

The colours themselves are stored in a "**Colour Lookup Table**" (**CLUT**) which is essentially a map $< colourindex, colour >$ i.e. a table indexed with an integer key $(0-255)$ storing a value that represents colour. In alternative terminology the CLUT is sometimes called a **palette**. As we will discuss in greater detail shortly (Section 2.3), many common colours can be produced by adding together (mixing) varying quantities of Red, Green and Blue light. For example, Red and Green light mix to produce Yellow light. Therefore the value stored in the CLUT for each colour is a triple (R, G, B) denoting the quantity (**intensity**) of Red, Green and Blue light in the mix. Each element of the triple is 8 bit i.e. has range $(0 - 255)$ in common implementations.

The earliest colour displays employed pseudo-colour frame buffers. This is because memory was expensive and colour images could be represented at identical cost to greyscale images (plus a small storage overhead for the CLUT). The obvious disadvantage of a pseudo-colour frame buffer is that only a limited number of colours may be displayed at any one time (i.e. 256 colours). However the colour range (we say **gamut**) of the display is $2^8 \times 2^8 \times 2^8 = 2^24 = 16,777,216$ colours.

Pseudo-colour frame buffers can still be found in many common platforms e.g. both MS and X Windows (for convenience, backward compatibility etc.) and in resource constrained computing domains (e.g. low-spec games consoles, mobiles). Some low-budget (in terms of CPU cycles) animation effects can be produced using pseudo-colour frame buffers. Consider an image filled with an expanses of colour index 1 (we might set CLUT $< 1, Blue >$, to create a blue ocean). We could sprinkle consecutive runs of pixels with index '2,3,4,5' sporadically throughout the image. The CLUT could be set to increasing, lighter shades of Blue at those indices. This might give the appearance of waves. The colour values in the CLUT at indices 2,3,4,5 could be rotated successively, so changing the displayed colours and causing the waves to animate/ripple (but without the CPU overhead of writing to multiple locations in the frame buffer). Effects like this were regularly used in many '80s and early '90s computer games, where computational expense prohibited updating the frame buffer directly for incidental animations.

2.2.5 True-Colour Frame Buffer

The true-colour frame-buffer also represents colour images, but does not use a CLUT. The RGB colour value for each pixel is stored directly within the frame buffer. So, if we use 8 bits to represent each Red, Green and Blue component, we will require 24 bits (3 bytes) of storage per pixel.

As with the other types of frame buffer, pixels are stored in left-right, then top-bottom order. So in our 24 bit colour example, pixel $(0, 0)$ would be stored at buffer locations 0, 1 and 2. Pixel $(1, 0)$ at 3, 4, and 5; and so on. Pixel (x, y) would be stored at offset A where:

$$\begin{aligned} S &= 3W \\ A &= 3x + Sy \end{aligned} \tag{2.2}$$

where S is sometimes referred to as the **stride** of the display.

The advantages of the true-colour buffer complement the disadvantages of the pseudo-colour buffer We can represent all 16 million colours at once in an image (given a large enough image!), but our image takes 3 times as much storage as the pseudo-colour buffer. The image would also take longer to update (3 times as many memory writes) which should be taken under consideration on resource constrained platforms (e.g. if writing a video codec on a mobile phone).

Alternative forms of true-colour buffer

The true colour buffer, as described, uses 24 bits to represent RGB colour. The usual convention is to write the R, G, and B values in order for each pixel. Sometime image formats (e.g. Windows Bitmap) write colours in order B, G, R. This is primarily due to the little-endian hardware architecture of PCs, which run Windows. These formats are sometimes referred to as RGB888 or BGR888 respectively.

2.3 Representation of Colour

Our eyes work by focussing light through an elastic lens, onto a patch at the back of our eye called the **retina**. The retina contains light sensitive **rod** and **cone** cells that are sensitive to light, and send electrical impulses to our brain that we interpret as a visual stimulus (Figure 2.2).

Cone cells are responsible for colour vision. There are three types of cone; each type has evolved to be optimally sensitive to a particular wavelength of light. Visible light has wavelength 700-400nm (red to violet). Figure 2.3 (left) sketches the response of the three cone types to wavelengths in this band. The peaks are located at colours that we have come to call "Red", "Green" and "Blue"[3]. Note that in most cases the response of the cones decays monotonically with distance from the optimal response wavelength. But interestingly, the Red cone violates this observation, having a slightly raised secondary response to the Blue wavelengths.

Given this biological apparatus, we can simulate the presence of many colours by shining Red, Green and Blue light into the human eye with carefully chosen intensities. This is the basis on which all colour display technologies (CRTs, LCDs, TFTs, Plasma, Data projectors) operate. Inside our machine (TV, Computer, Projector) we represent pixel colours using values for Red, Green and Blue (RGB triples) and the video hardware uses these values to generate the appropriate amount of Red, Green and Blue light (subsection 2.2.2).

Red, Green and Blue are called the "**additive primaries**" because we obtain other, non-primary colours by blending (adding) together different quantities of Red, Green and Blue light. We can make the additive **secondary colours** by mixing pairs of primaries: Red and Green make Yellow; Green and Blue make Cyan (light blue); Blue and Red make Magenta (light purple). If we mix all three additive primaries we get **White**. If we don't mix any amount of the additive primaries we generate zero light, and so get **Black** (the absence of colour).

[3] Other animals have cones that peak at different colours; for example bees can see ultra-violet as they have three cones; one of which peaks at ultra-violet. The centres of flowers are often marked with ultra-violet patterns invisible to humans.

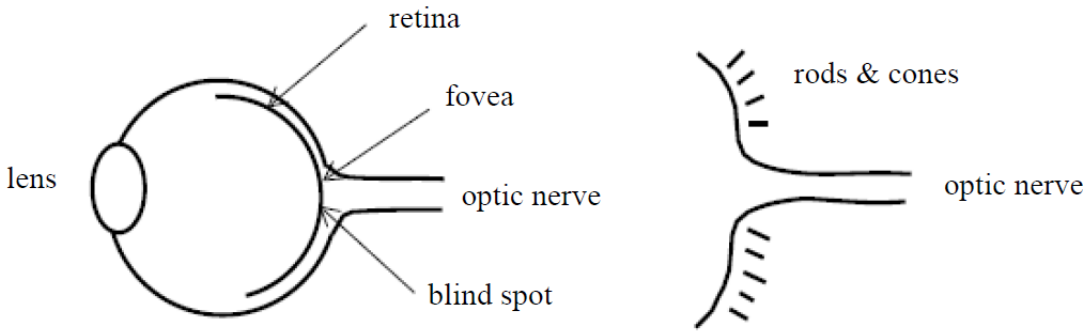

Figure 2.2: In the human eye, light is focused through an elastic lens onto a light sensitive patch (the retine). Special cells (rods and cones) in the retina convert light into electrical impulses that travel to the brain via the optic nerve. The site where the nerve exits the eye contains no such cells, and is termed the "blind spot". The cone cells in particular are responsible for our *colour* vision.

2.3.1 Additive vs. Subtractive Primaries

You may recall mixing paints as a child; being taught (and experimentally verifying) that Red, Yellow and Blue were the primary colours and could be mixed to obtain the other colours. So how do we resolve this discrepancy; are the primary colours Red, Green, and Blue; or are they Red, Yellow, and Blue?

When we view a Yellow object, we say that it is Yellow because light of a narrow band of wavelengths we have come to call "Yellow" enters our eye, stimulating the Red and Green cones. More specifically, the Yellow object reflects ambient light of the Yellow wavelength and absorbs all other light wavelengths.

We can think of Yellow paint as reflecting a band wavelengths spanning the Red-Green part of the spectrum, and absorbing everything else. Similarly, Cyan paint reflects the Green-Blue part of the spectrum, and absorbs everything else. Mixing Yellow and Cyan paint causes the paint particles to absorb all but the Green light. So we see that mixing Yellow and Cyan paint gives us Green. This allies with our experience mixing paints as a child; Yellow and Blue make Green. Figure 2.3 (right) illustrates this diagrammatically.

In this case adding a new paint (Cyan) to a Yellow mix, caused our resultant mix to become more restrictive in the wavelengths it reflected. We earlier referred to Cyan, Magenta and Yellow as the additive secondary colours. But these colours are more often called the "**subtractive primaries**". We see that each subtractive primary we contribute in to the paint mix "subtracts" i.e. absorbs a band of wavelengths. Ultimately if we mix all three primaries; Cyan, Yellow and Magenta together we get Black because all visible light is absorbed, and none reflected.

So to recap; RGB are the additive primaries and CMY (Cyan, Magenta, Yellow) the subtractive primaries. They are used respectively to mix light (e.g. on displays), and to mix ink/paint (e.g. when printing). You may be aware that colour printer cartridges are sold

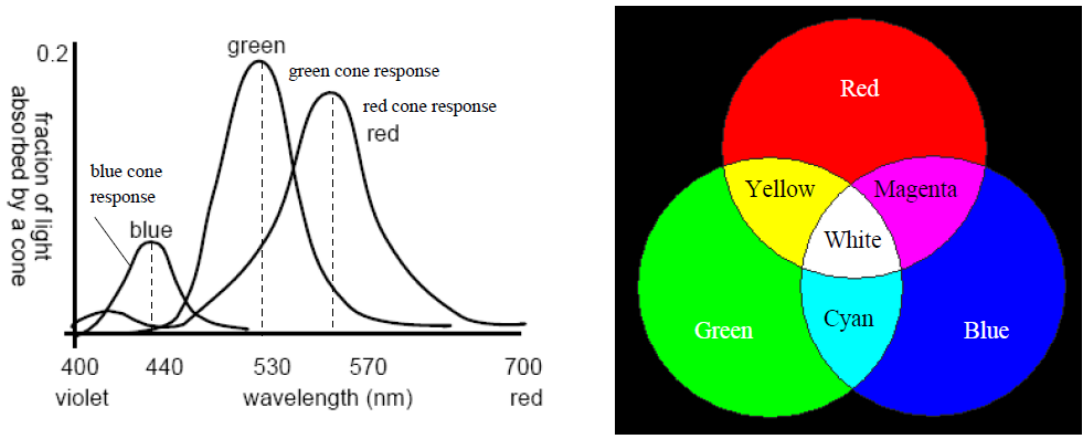

Figure 2.3: Left: Sketching the response of each of the three cones to differing wavelengths of light. Peaks are observed around the colours we refer to as Red, Green and Blue. Observe the secondary response of the Red cone to Blue light. Right: The RGB additive primary colour wheel, showing how light is mixed to produce different colours. All 3 primaries (Red, Green, Blue) combine to make white.

containing Cyan, Magenta and Yellow (CMY) ink; the subtractive primaries. Returning to our original observation, CMY are the colours approximated by the child when mixing Red, Yellow and Blue paint (blue is easier to teach than 'cyan'; similarly magenta).

2.3.2 RGB and CMYK colour spaces

We have seen that displays represent colour using a triple (R,G,B). We can interpret each colour as a point in a three dimensional space (with axes Red, Green, Blue). This is one example of a **colour space** — the RGB colour space. The RGB colour space is cube-shaped and is sometimes called the **RGB colour cube**. We can think of picking colours as picking points in the cube (Figure 2.4, left). Black is located at $(0, 0, 0)$ and White is located at $(255, 255, 255)$. Shades of grey are located at (n, n, n) i.e. on the diagonal between Black and White.

We have also seen that painting processes that deposit pigment (i.e. printing) are more appropriately described using colours in CMY space. This space is also cube shaped.

We observed earlier that to print Black requires mixing of all three subtractive primaries (CMY). Printer ink can be expensive, and Black is a common colour in printed documents. It is therefore inefficient to deposit three quantities of ink onto a page each time we need to print something in Black. Therefore printer cartridges often contain four colours: Cyan, Magenta, Yellow, and a pre-mixed Black. This is written CMYK, and is a modified form of the CMY colour space.

CMYK can help us print non-black colours more efficiently too. If we wish to print a colour (c, y, m) in CMY space, we can find the amount of Black in that colour (written k for 'key')

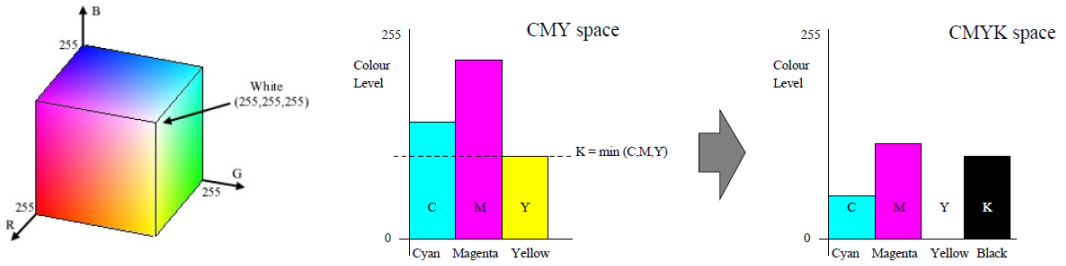

Figure 2.4: Left: The RGB colour cube; colours are represented as points in a 3D space each axis of which is an additive primary. Right: CMYK; the quantity of black (C, M, and Y ink mix) contained within a colour is substituted with pure black (K) ink. This saves ink and improves colour tone, since a perfectly dark black is difficult to mix using C, M, and Y inks.

by computing:

$$k = \min(c, y, m) \tag{2.3}$$

We then compute $c = c - k$, $y = y - k$, $m = m - k$ (one or more of which will be zero) and so represent our original CMY colour (c, y, m) as a CMYK colour (c, m, y, k). Figure 2.4 (right) illustrates this process. It is clear that a lower volume of ink will be required due to economies made in the Black portion of the colour, by substituting CMY ink for K ink.

You may be wondering if an "RGB plus White" space might exist for the additive primaries. Indeed some devices such as data projectors do feature a fourth colour channel for White (in addition to RGB) that works in a manner analogous to Black in CMYK space. The amount of white in a colour is linked to the definition of **colour saturation**, which we will return to in subsection 2.3.6.

2.3.3 Greyscale Conversion

Sometimes we wish to convert a colour image into a greyscale image; i.e. an image where colours are represented using shades of grey rather than different hues. It is often desirable to do this in Computer Vision applications, because many common operations (edge detection, etc) require a scalar value for each pixel, rather than a vector quantity such as an RGB triple. Sometimes there are aesthetic motivations behind such transformations also.

Referring back to Figure 2.3 we see that each type of cones responds with varying strength to each wavelength of light. If we combine (equation 2.4) the overall response of the three cones for each wavelength we would obtain a curve such as Figure 2.5 (left). The curve indicates the *perceived* **brightness** (the Graphics term is **luminosity**) of each light wavelength (colour), given a light source of *constant brightness* output.

By experimenting with the human visual system, researchers have derived the following equation to model this response, and so obtain the luminosity (l) for a given RGB colour (r, g, b) as:

$$l(r, g, b) = 0.30r + 0.59g + 0.11b \tag{2.4}$$

Figure 2.5: Greyscale conversion. Left: The human eye's combined response (weighted sum of cone responses) to light of increasing wavelength but constant intensity. The perceived luminosity of the light changes as a function of wavelength. Right: A painting (Impressionist Sunrise by Monet) in colour and greyscale; note the isoluminant colours used to paint the sun against the sky.

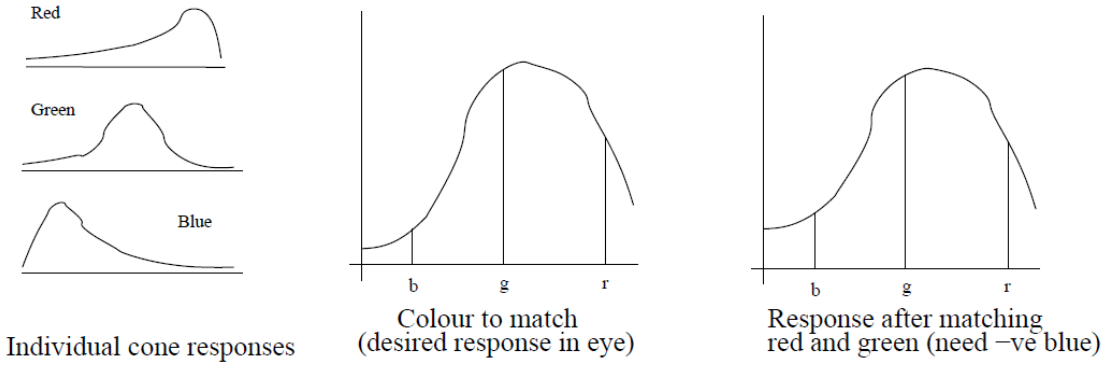

Figure 2.6: Tri-stimulus experiments attempt to match real-world colours with a mix of red, green and blue (narrow bandwidth) primary light sources. Certain colours cannot be matched because cones have a wide bandwidth response e.g. the blue cone responds a little to red and green light (see subsection 2.3.4).

As a note, usually we denote luminosity using Y — do not confuse this with the Y in CMYK.

You can write Matlab code to convert a colour JPEG image into a greyscale image as follows:

```
img=double(imread('image.jpg'))./255;
r=img(:,:,1);
g=img(:,:,2);
b=img(:,:,3);
y=0.3.*r + 0.69.*g + 0.11.*b;
imshow(y);
```

This code was applied to the source image in Figure 2.5. This Figure gives clear illustration that visually distinct colours (e.g. of the sun and sky in the image) can map to similar greyscale values. Such colours are called "isoluminant" (i.e. of same luminance).