Image Acquisition

2

Before any video or image processing can commence an image must be captured by a camera and converted into a manageable entity. This is the process known as *image acquisition*. The image acquisition process consists of three steps; *energy* reflected from the object of interest, an *optical system* which focuses the energy and finally a *sensor* which measures the amount of energy. In Fig. 2.1 the three steps are shown for the case of an ordinary camera with the sun as the energy source. In this chapter each of these three steps are described in more detail.

2.1 Energy

In order to capture an image a camera requires some sort of measurable energy. The energy of interest in this context is light or more generally *electromagnetic waves*. An electromagnetic (EM) wave can be described as massless entity, a *photon*, whose electric and magnetic fields vary sinusoidally, hence the name wave. The photon belongs to the group of fundamental particles and can be described in three different ways:

- A photon can be described by its energy E, which is measured in electronvolts [eV]
- A photon can be described by its frequency f, which is measured in Hertz [Hz]. A frequency is the number of cycles or wave-tops in one second
- A photon can be described by its wavelength λ, which is measured in meters [m]. A wavelength is the distance between two wave-tops

The three different notations are connected through the speed of light c and Planck's constant h:

$$\lambda = \frac{c}{f}, \qquad E = h \cdot f \quad \Rightarrow \quad E = \frac{h \cdot c}{\lambda} \tag{2.1}$$

An EM wave can have different wavelengths (or different energy levels or different frequencies). When we talk about all possible wavelengths we denote this as the *EM spectrum*, see Fig. 2.2.

T.B. Moeslund, *Introduction to Video and Image Processing*,
Undergraduate Topics in Computer Science,
DOI 10.1007/978-1-4471-2503-7_2, © Springer-Verlag London Limited 2012

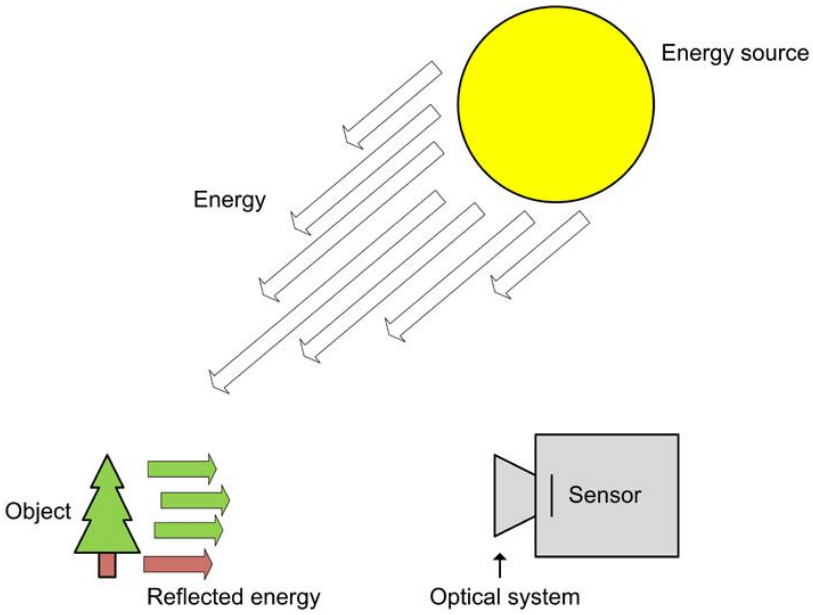


Fig. 2.1 Overview of the typical image acquisition process, with the sun as light source, a tree as object and a digital camera to capture the image. An analog camera would use a film where the digital camera uses a sensor

In order to make the definitions and equations above more understandable, the EM spectrum is often described using the names of the applications where they are used in practice. For example, when you listen to FM-radio the music is transmitted through the air using EM waves around $100 \cdot 10^6$ Hz, hence this part of the EM spectrum is often denoted "radio". Other well-known applications are also included in the figure.

The range from approximately 400–700 nm (nm = nanometer = 10^{-9}) is denoted the visual spectrum. The EM waves within this range are those your eye (and most cameras) can detect. This means that the light from the sun (or a lamp) in principle is the same as the signal used for transmitting TV, radio or for mobile phones etc. The only difference, in this context, is the fact that the human eye can sense EM waves in this range and not the waves used for e.g., radio. Or in other words, if our eyes were sensitive to EM waves with a frequency around $2 \cdot 10^9$ Hz, then your mobile phone would work as a flash light, and big antennas would be perceived as "small suns". Evolution has (of course) not made the human eye sensitive to such frequencies but rather to the frequencies of the waves coming from the sun, hence visible light.

2.1.1 Illumination

To capture an image we need some kind of energy source to illuminate the scene. In Fig. 2.1 the sun acts as the energy source. Most often we apply visual light, but other frequencies can also be applied, see Sect. 2.5.

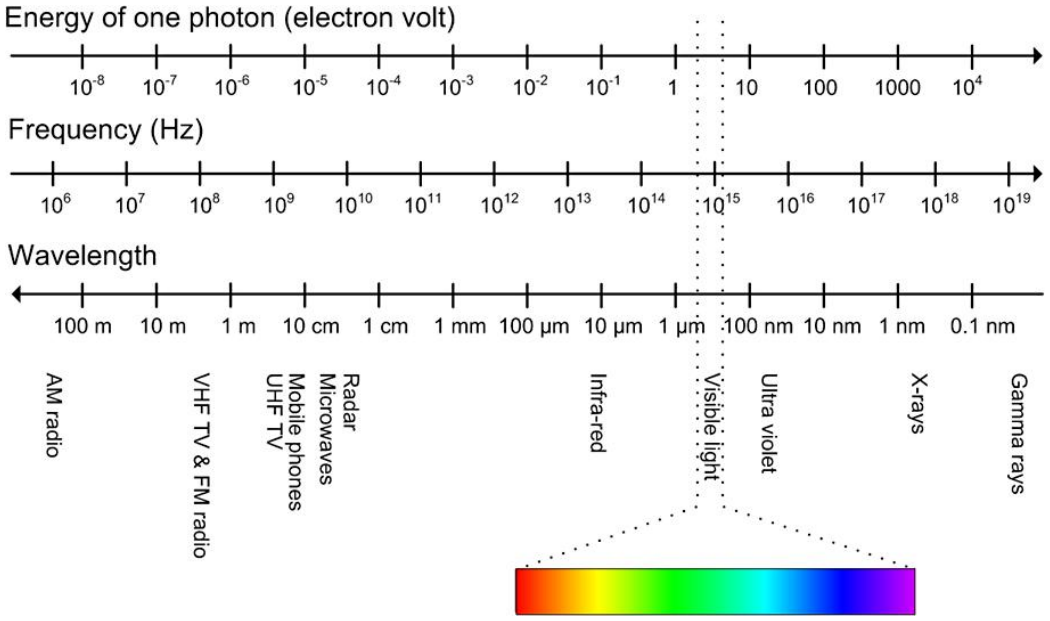


Fig. 2.2 A large part of the electromagnetic spectrum showing the energy of one photon, the frequency, wavelength and typical applications of the different areas of the spectrum

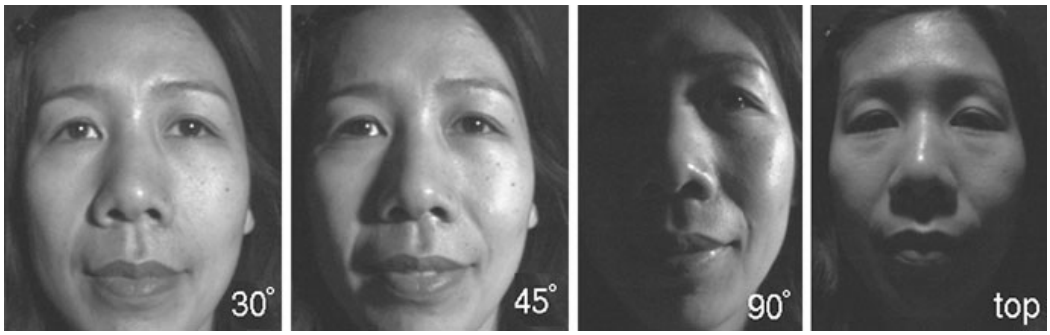


Fig. 2.3 The effect of illuminating a face from four different directions

If you are processing images captured by others there is nothing much to do about the illumination (although a few methods will be presented in later chapters) which was probably the sun and/or some artificial lighting. When you, however, are in charge of the capturing process yourselves, it is of great importance to carefully think about how the scene should be lit. In fact, for the field of Machine Vision it is a rule-of-thumb that illumination is 2/3 of the entire system design and software only 1/3. To stress this point have a look at Fig. 2.3. The figure shows four images of the same person facing the camera. The only difference between the four images is the direction of the light source (a lamp) when the images were captured!

Another issue regarding the direction of the illumination is that care must be taken when pointing the illumination directly toward the camera. The reason being that this might result in too bright an image or a nonuniform illumination, e.g., a bright circle in the image. If, however, the outline of the object is the only infor-

Fig. 2.4 Backlighting. The light source is behind the object of interest, which makes the object stand out as a black silhouette. Note that the details inside the object are lost

mation of interest, then this way of illumination—denoted *backlighting*—can be an optimal solution, see Fig. 2.4. Even when the illumination is not directed toward the camera overly bright spots in the image might still occur. These are known as *highlights* and are often a result of a shiny object surface, which reflects most of the illumination (similar to the effect of a mirror). A solution to such problems is often to use some kind of diffuse illumination either in the form of a high number of less-powerful light sources or by illuminating a rough surface which then reflects the light (randomly) toward the object.

Even though this text is about visual light as the energy form, it should be mentioned that infrared illumination is sometimes useful. For example, when tracking the movements of human body parts, e.g. for use in animations in motion pictures, infrared illumination is often applied. The idea is to add infrared reflecting markers to the human body parts, e.g., in the form of small balls. When the scene is illuminated by infrared light, these markers will stand out and can therefore easily be detected by image processing. A practical example of using infrared illumination is given in Chap. 12.

2.2 The Optical System

After having illuminated the object of interest, the light reflected from the object now has to be captured by the camera. If a material sensitive to the reflected light is placed close to the object, an image of the object will be captured. However, as illustrated in Fig. 2.5, light from different points on the object will mix—resulting in a useless image. To make matters worse, light from the surroundings will also be captured resulting in even worse results. The solution is, as illustrated in the figure, to place some kind of barrier between the object of interest and the sensing material. Note that the consequence is that the image is upside-down. The hardware and software used to capture the image normally rearranges the image so that you never notice this.

The concept of a barrier is a sound idea, but results in too little light entering the sensor. To handle this situation the hole is replaced by an *optical system*. This section describes the basics behind such an optical system. To put it into perspective, the famous space-telescope—the Hubble telescope—basically operates like a camera, i.e., an optical system directs the incoming energy toward a sensor. Imagine how many man-hours were used to design and implement the Hubble telescope. And still, NASA had to send astronauts into space in order to fix the optical system due

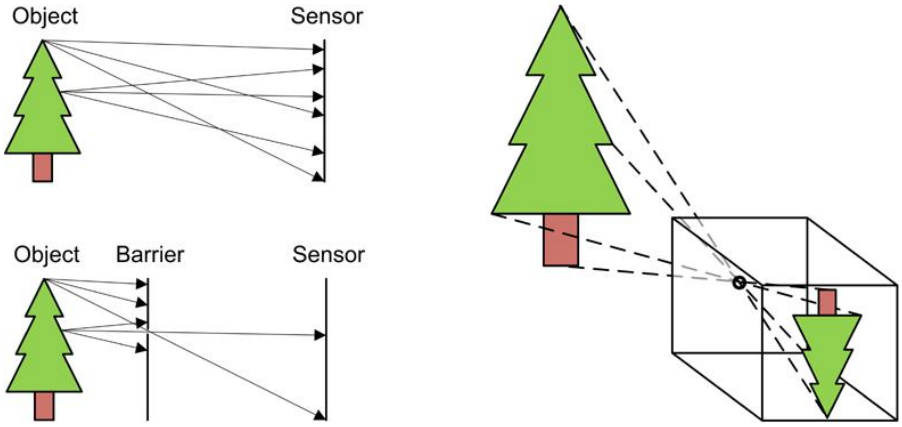


Fig. 2.5 Before introducing a barrier, the rays of light from different points on the tree hit multiple points on the sensor and in some cases even the same points. Introducing a barrier with a small hole significantly reduces these problems

to an incorrect design. Building optical systems is indeed a complex science! We shall not dwell on all the fine details and the following is therefore not accurate to the last micro-meter, but the description will suffice and be correct for most usages.

2.2.1 The Lens

One of the main ingredients in the optical system is the lens. A lens is basically a piece of glass which focuses the incoming light onto the sensor, as illustrated in Fig. 2.6. A high number of light rays with slightly different incident angles collide with each point on the object's surface and some of these are reflected toward the optics. In the figure, three light rays are illustrated for two different points. All three rays for a particular point intersect in a point to the right of the lens. Focusing such rays is exactly the purpose of the lens. This means that an image of the object is formed to the right of the lens and it is this image the camera captures by placing a sensor at exactly this position. Note that parallel rays intersect in a point, F, denoted the *Focal Point*. The distance from the center of the lens, the *optical center* O, to the plane where all parallel rays intersect is denoted the *Focal Length* f. The line on which O and F lie is *the optical axis*.

Let us define the distance from the object to the lens as, g, and the distance from the lens to where the rays intersect as, b. It can then be shown via similar triangles, see Appendix B, that

$$\frac{1}{g} + \frac{1}{b} = \frac{1}{f} \tag{2.2}$$

f and b are typically in the range [1 mm, 100 mm]. This means that when the object is a few meters away from the camera (lens), then $\frac{1}{g}$ has virtually no effect on the equation, i.e., $b = f$. What this tells us is that the image inside the camera is formed

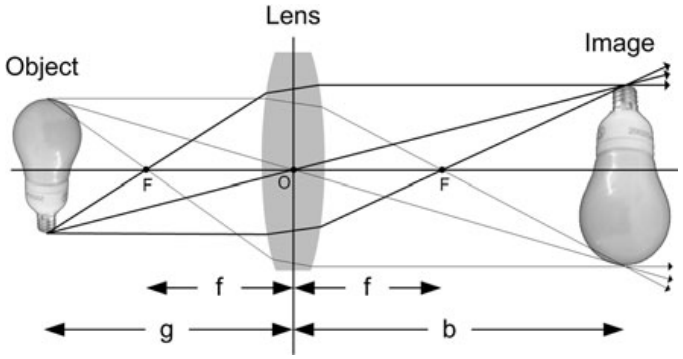


Fig. 2.6 The figure shows how the rays from an object, here a light bulb, are focused via the lens. The real light bulb is to the *left* and the image formed by the lens is to the *right*

at a distance very close to the focal point. Equation 2.2 is also called the *thin lens equation*.

Another interesting aspect of the lens is that the size of the object in the image, B, increases as f increased. This is known as *optical zoom*. In practice f is changed by rearranging the optics, e.g., the distance between one or more lenses inside the optical system.[1] In Fig. 2.7 we show how optical zoom is achieved by changing the focal length. When looking at Fig. 2.7 it can be shown via similar triangles that

$$\frac{b}{B} = \frac{g}{G} \tag{2.3}$$

where G is the real height of the object. This can for example be used to compute how much a physical object will fill on the imaging censor chip, when the camera is placed at a given distance away from the object.

Let us assume that we do not have a zoom-lens, i.e., f is constant. When we change the distance from the object to the camera (lens), g, Eq. 2.2 shows us that b should also be increased, meaning that the sensor has to be moved slightly further away from the lens since the image will be formed there. In Fig. 2.8 the effect of not changing b is shown. Such an image is said to be *out of focus*. So when you adjust focus on your camera you are in fact changing b until the sensor is located at the position where the image is formed.

The reason for an *unfocused* image is illustrated in Fig. 2.9. The sensor consists of pixels, as will be described in the next section, and each pixel has a certain size. As long as the rays from one point stay inside one particular pixel, this pixel will be focused. If rays from other points also intersect the pixel in question, then the pixel will receive light from more points and the resulting pixel value will be a mixture of light from different points, i.e., it is unfocused.

Referring to Fig. 2.9 an object can be moved a distance of g_l further away from the lens or a distance of g_r closer to the lens and remain in focus. The sum of g_l and g_r defines the total range an object can be moved while remaining in focus. This range is denoted as the *depth-of-field*.

[1] Optical zoom should not be confused with digital zoom, which is done through software.

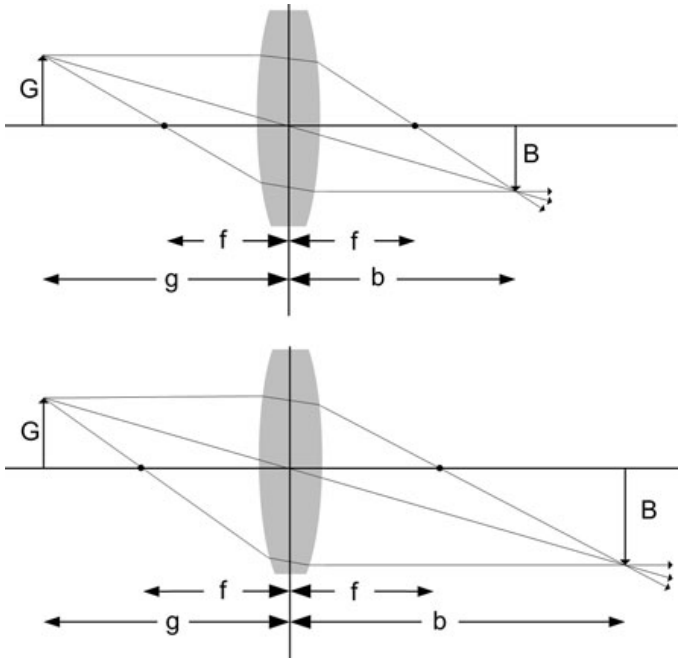


Fig. 2.7 Different focal lengths results in optical zoom

Fig. 2.8 A focused image (*left*) and an unfocused image (*right*). The difference between the two images is different values of b

A smaller depth-of-field can be achieved by increasing the focal length. However, this has the consequence that the area of the world observable to the camera is reduced. The observable area is expressed by the angle V in Fig. 2.10 and denoted the *field-of-view* of the camera. The field-of-view depends, besides the focal length, also on the physical size of the image sensor. Often the sensor is rectangular rather than square and from this follows that a camera has a field-of-view in both the horizontal and vertical direction denoted FOV_x and FOV_y, respectively. Based on right-angled triangles, see Appendix B, these are calculated as

$$\begin{aligned} \text{FOV}_x &= 2 \cdot \tan^{-1}\left(\frac{\text{width of sensor}/2}{f}\right) \\ \text{FOV}_y &= 2 \cdot \tan^{-1}\left(\frac{\text{height of sensor}/2}{f}\right) \end{aligned} \tag{2.4}$$

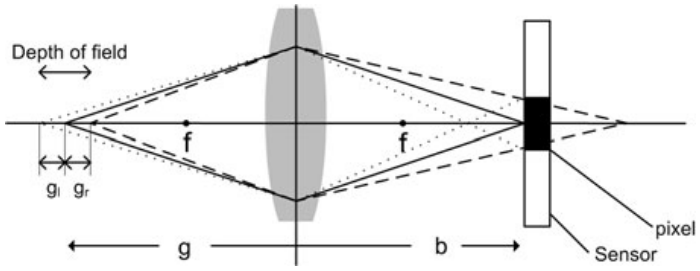


Fig. 2.9 Depth-of-field. The *solid lines* illustrate two light rays from an object (a point) on the optical axis and their paths through the lens and to the sensor where they intersect within the same pixel (illustrated as a *black rectangle*). The *dashed* and *dotted lines* illustrate light rays from two other objects (points) on the optical axis. These objects are characterized by being the most extreme locations where the light rays still enter the same pixel

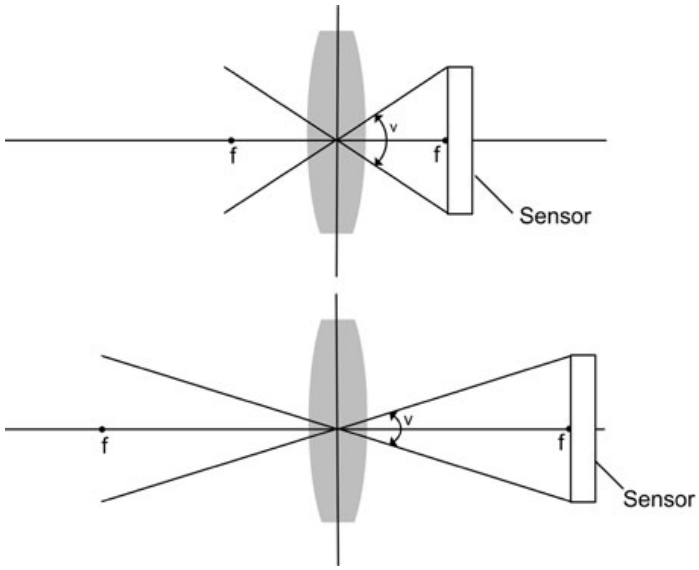


Fig. 2.10 The field-of-view of two cameras with different focal lengths. The field-of-view is an angle, V, which represents the part of the world observable to the camera. As the focal length increases so does the distance from the lens to the sensor. This in turn results in a smaller field-of-view. Note that both a horizontal field-of-view and a vertical field-of-view exist. If the sensor has equal height and width these two fields-of-view are the same, otherwise they are different

where the focal length, f, and width and height are measured in mm. So, if we have a physical sensor with width $= 14$ mm, height $= 10$ mm and a focal length $= 5$ mm, then the fields-of-view will be

$$\text{FOV}_x = 2 \cdot \tan^{-1}\left(\frac{7}{5}\right) = 108.9^\circ, \qquad \text{FOV}_y = 2 \cdot \tan^{-1}(1) = 90^\circ \tag{2.5}$$

Another parameter influencing the depth-of-field is the *aperture*. The aperture corresponds to the human iris, which controls the amount of light entering the human eye. Similarly, the aperture is a flat circular object with a hole in the center with adjustable radius. The aperture is located in front of the lens and used to control the amount of incoming light. In the extreme case, the aperture only allows rays through the optical center, resulting in an infinite depth-of-field. The downside is that the more light blocked by the aperture, the lower *shutter* speed (explained below) is required in order to ensure enough light to create an image. From this it follows that objects in motion can result in blurry images.

Fig. 2.11 Three different camera settings resulting in three different depth-of-fields

To sum up, the following interconnected issues must be considered: distance to object, motion of object, zoom, focus, depth-of-field, focal length, shutter, aperture, and sensor. In Figs. 2.11 and 2.12 some of these issues are illustrated. With this knowledge you might be able to appreciate why a professional photographer can capture better images than you can!

2.3 The Image Sensor

The light reflected from the object of interest is focused by some optics and now needs to be recorded by the camera. For this purpose an image sensor is used. An image sensor consists of a 2D array of cells as seen in Fig. 2.13. Each of these cells is denoted a *pixel* and is capable of measuring the amount of incident light and convert that into a voltage, which in turn is converted into a digital number.

The more incident light the higher the voltage and the higher the digital number. Before a camera can capture an image, all cells are emptied, meaning that no charge is present. When the camera is to capture an image, light is allowed to enter and charges start accumulating in each cell. After a certain amount of time, known as the *exposure time*, and controlled by the *shutter*, the incident light is shut out again. If the exposure time is too low or too high the result is an underexposed or overexposed image, respectively, see Fig. 2.14.

Many cameras have a built-in intelligent system that tries to ensure the image is not over- or underexposed. This is done by measuring the amount of incoming light and if too low/high correct the image accordingly, either by changing the exposure time or more often by an *automatic gain control*. While the former improves the image by changing the camera settings, the latter is rather a post-processing step. Both can provide more pleasing video for the human eye to watch, but for automatic video analysis you are very often better off disabling such features. This might sound counter intuitive, but since automatic video/image processing is all about manipulating the incoming light, we need to understand and be able to foresee incoming light in different situations and this can be hard if the camera interferes beyond our control and understanding. This might be easier understood after reading the next chapter. The point is that when choosing a camera you need to remember to check if the automatic gain control is mandatory or if it can be disabled. Go for a camera where it can be disabled. It should of course be added that if you capture video

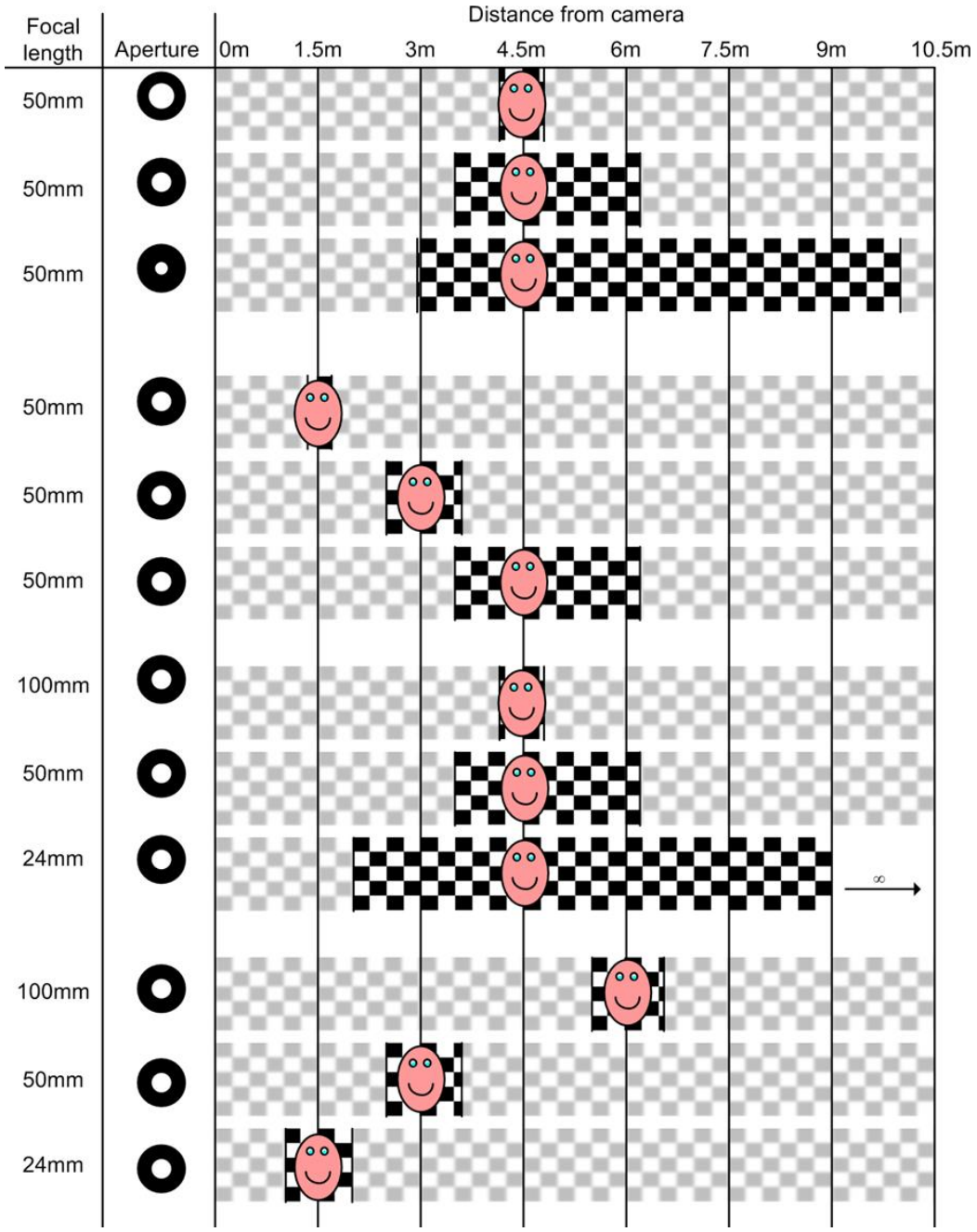


Fig. 2.12 Examples of how different settings for focal length, aperture and distance to object result in different depth-of-fields. For a given combination of the three settings the optics are focused so that the object (person) is in focus. The focused checkers then represent the depth-of-field for that particular setting, i.e., the range in which the object will be in focus. The figure is based on a Canon 400D

in situations where the amount of light can change significantly, then you *have* to enable the camera's automatic settings in order to obtain a useable image.

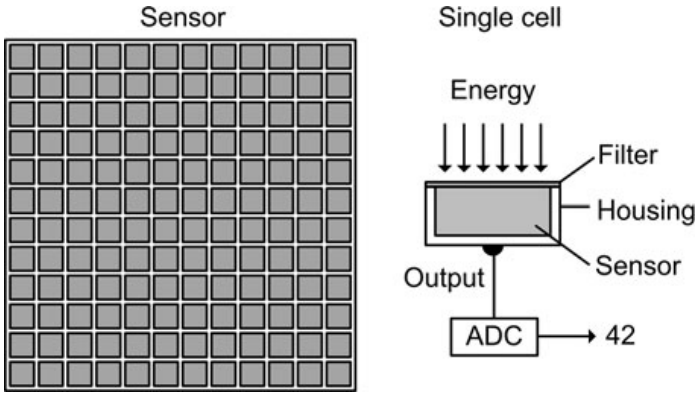


Fig. 2.13 The sensor consists of an array of interconnected cells. Each cell consists of a housing which holds a filter, a sensor and an output. The filter controls which type of energy is allowed to enter the sensor. The sensor measures the amount of energy as a voltage, which is converted into a digital number through an analog-to-digital converter (ADC)

Fig. 2.14 The input image was taken with the correct amount of exposure. The over- and underexposed images are too bright and too dark, respectively, which makes it hard to see details in them. If the object or camera is moved during the exposure time, it produces motion blur as demonstrated in the last image

Another aspect related to the exposure time is when the object of interest is in motion. Here the exposure time in general needs to be low in order to avoid *motion blur*, where light from a certain point on the object will be spread out over more cells, see Fig. 2.14.

The accumulated charges are converted into digital form using an *analog-to-digital converter*. This process takes the continuous world outside the camera and converts it into a digital representation, which is required when stored in the computer. Or in other words, this is where the image becomes digital. To fully comprehend the difference, have a look at Fig. 2.15.

To the left we see where the incident light hits the different cells and how many times (the more times the brighter the value). This results in the shape of the object and its intensity. Let us first consider the shape of the object. A cell is sensitive to

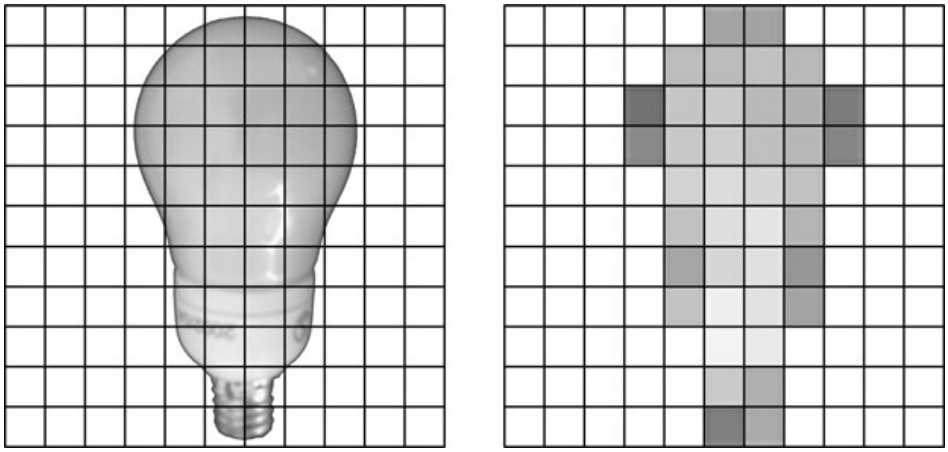


Fig. 2.15 To the *left* the amount of light which hits each cell is shown. To the *right* the resulting image of the measured light is shown

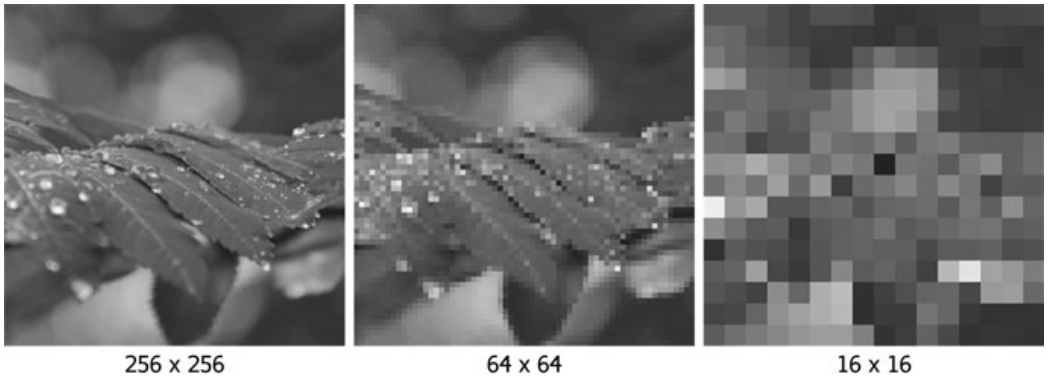


Fig. 2.16 The effect of spatial resolution. The spatial resolution is from *left* to *right*: 256×256, 64×64, and 16×16

incident light hitting the cell, but not sensitive to where exactly the light hits the cell. So if the shape should be preserved, the size of the cells should be infinitely small. From this it follows that the image will be infinitively large in both the x- and y-direction. This is not tractable and therefore a cell, of course, has a finite size. This leads to loss of data/precision and this process is termed *spatial quantization*. The effect is the blocky shape of the object in the figure to the right. The number of pixels used to represent an image is also called the *spatial resolution* of the image. A high resolution means that a large number of pixels are used, resulting in fine details in the image. A low resolution means that a relatively low number of pixels is used. Sometimes the words fine and coarse resolution are used. The visual effect of the spatial resolution can be seen in Fig. 2.16. Overall we have a trade-off between memory and shape/detail preservation. It is possible to change the resolution of an image by a process called *image-resampling*. This can be used to create a low resolution image from a high resolution image. However, it is normally not possible to create a high resolution image from a low resolution image.

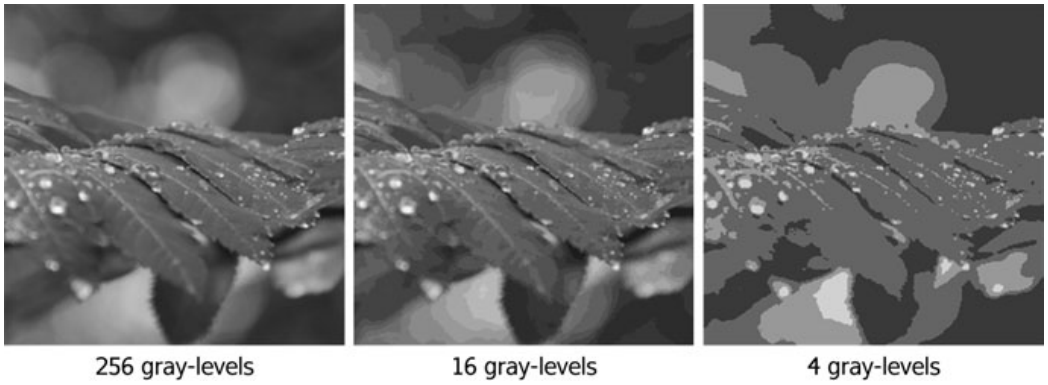


Fig. 2.17 The effect of gray-level resolution. The gray-level resolution is from *left* to *right*: 256, 16, and 4 gray levels

A similar situation is present for the representation of the amount of incident light within a cell. The number of photons hitting a cell can be tremendously high requiring an equally high digital number to represent this information. However, since the human eye is not even close to being able to distinguish the exact number of photons, we can quantify the number of photons hitting a cell. Often this quantization results in a representation of one byte (8 bits), since one byte corresponds to the way memory is organized inside a computer (see Appendix A for an introduction to bits and bytes). In the case of 8-bit quantization, a charge of 0 volt will be quantized to 0 and a high charge quantized to 255. Other gray-level quantizations are sometimes used. The effect of changing the gray-level quantization (also called the *gray-level resolution*) can be seen in Fig. 2.17. Down to 16 gray levels the image will frequently still look realistic, but with a clearly visible quantization effect. The gray-level resolution is usually specified in number of bits. While, typical gray-level resolutions are 8-, 10-, and 12-bit corresponding to 256, 1024, and 4096 gray levels, 8-bit images are the most common and are the topic of this text.

In the case of an overexposed image, a number of cells might have charges above the maximum measurable charge. These cells are all quantized to 255. There is no way of knowing just how much incident light entered such a cell and we therefore say that the cell is *saturated*. This situation should be avoided by setting the shutter (and/or aperture), and saturated cells should be handled carefully in any video and image processing system. When a cell is saturated it can affect the neighbor pixels by increasing their charges. This is known as *blooming* and is yet another argument for avoiding saturation.

2.4 The Digital Image

To transform the information from the sensor into an image, each cell content is now converted into a pixel value in the range: [0, 255]. Such a value is interpreted as the amount of light hitting a cell during the exposure time. This is denoted the *intensity* of a pixel. It is visualized as a shade of gray denoted a *gray-scale value* or *gray-level value* ranging from black (0) to white (255), see Fig. 2.18.

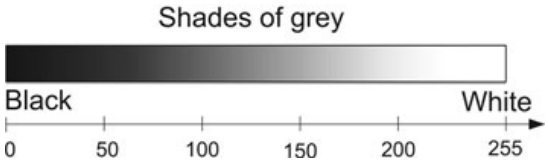


Fig. 2.18 The relationship between the intensity values and the different shades of gray

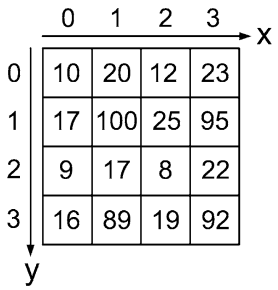


Fig. 2.19 Definition of the image coordinate system

A gray-scale image (as opposed to a color image, which is the topic of Chap. 3) is a 2D array of pixels (corresponding to the 2D array of cells in Fig. 2.13) each having a number between 0 and 255. In this text the coordinate system of the image is defined as illustrated in Fig. 2.19 and the image is represented as $f(x, y)$, where x is the horizontal position of the pixel and y the vertical position. For the small image in Fig. 2.19, $f(0, 0) = 10$, $f(3, 1) = 95$ and $f(2, 3) = 19$.

So whenever you see a gray-scale image you must remember that what you are actually seeing is a 2D array of numbers as illustrated in Fig. 2.20.

2.4.1 The Region of Interest (ROI)

As digital cameras are sold in larger and larger numbers the development within sensor technology has resulted in many new products including larger and larger numbers of pixels within one sensor. This is normally defined as the size of the image that can be captured by a sensor, i.e., the number of pixels in the vertical direction multiplied by the number of pixels in the horizontal direction. Having a large number of pixels can result in high quality images and has made, for example, digital zoom a reality.

When it comes to image processing, a larger image size is not always a benefit. Unless you are interested in tiny details or require very accurate measurements in the image, you are better off using a smaller sized image. The reason being that when we start to process images we have to process each pixel, i.e., perform some math on each pixel. And, due to the large number of pixels, that quickly adds up to quite a large number of mathematical operations, which in turn means a high computational load on your computer.

Say you have an image which is 500×500 pixels. That means that you have $500 \cdot 500 = 250{,}000$ pixels. Now say that you are processing video with 50 images per second. That means that you have to process $50 \cdot 250{,}000 = 12{,}500{,}000$ pixels per second. Say that your algorithm requires 10 mathematical operations per pixel, then in total your computer has to do $10 \cdot 12{,}500{,}000 = 125{,}000{,}000$ operations

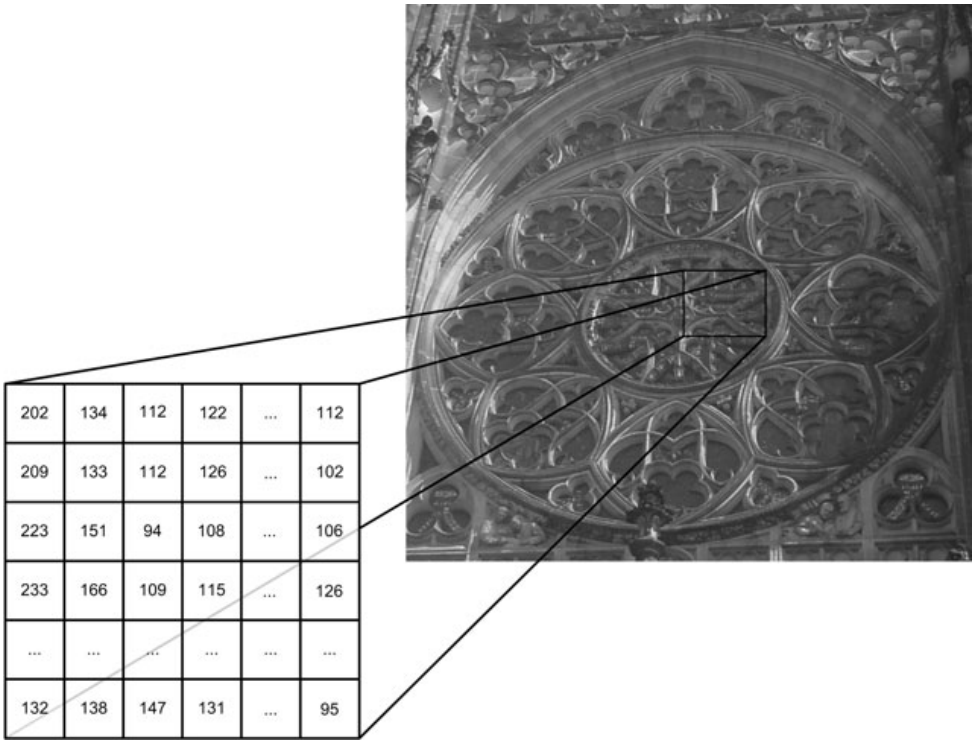


Fig. 2.20 A gray-scale image and part of the image described as a 2D array, where the cells represent pixels and the value in a cell represents the intensity of that pixel

per second. That is quite a number even for today's powerful computers. So when you choose your camera do not make the mistake of thinking that bigger is always better!

Besides picking a camera with a reasonable size you should also consider introducing a *region-of-interest* (ROI). An ROI is simply a region (normally a rectangle) within the image which defines the pixels of interest. Those pixels not included in the region are ignored altogether and less processing is therefore required. An ROI is illustrated in Fig. 2.21.

The ROI can sometimes be defined for a camera, meaning that the camera only captures those pixels within the region, but usually it is something you as a designer define in software. Say that you have put up a camera in your home in order to detect if someone comes through one of the windows while you are on holiday. You could then define an ROI for each window seen in the image and *only* process these pixels. When you start playing around with video and image processing you will soon realize the need for an ROI.

2.5 Further Information

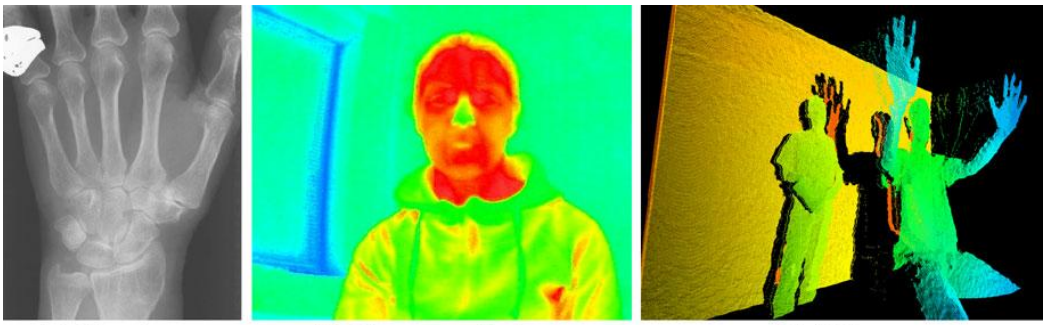
As hinted at in this chapter the camera and especially the optics are complicated and much more information is required to comprehend those in-depth. While a full understanding of the capturing process is mainly based on electrical engineering,

Fig. 2.21 The *white rectangle* defines a region-of-interest (ROI), i.e., this part of the image is the only one being processed

understanding optics requires a study on physics and how light interacts with the physical world. A more easy way into these fields can be via the FCam [1], which is a software platform for understanding and teaching different aspects of a camera.

Another way into these fields is to pick up a book on Machine Vision. Here you will often find a practical approach to understanding the camera and guidelines on picking the right camera and optics. Such books also contain practical information on how to make your image/video analysis easier by introducing special lightning etc.

While this chapter (and the rest of the book) focused solely on images formed by visual light it should be mentioned that other wavelengths from the electromagnetic spectrum can also be converted into digital images and processed by the methods in the following chapters. Two examples are X-ray images and thermographic images, see Fig. 2.22. An X-ray image is formed by placing an object between an X-ray emitter and an X-ray receiver. The receiver measures the energy level of the X-rays at different positions. The energy level is proportional to the physical properties of the object, i.e., bones stop the X-rays while blood does not. Thermographic images capture middle- or far-infrared rays. Heat is emitted from all objects via such wavelengths meaning that the intensity in each pixel in a thermographic image corresponds directly to the temperature of the observed object, see Fig. 2.22. Other types of image not directly based on the electromagnetic spectrum can also be captured and processed and in general all 2D signals that can be measured can be represented as an image. Examples are MR and CT images known from hospitals, and 3D (or depth) images obtained by a laser scanner, a time-of-flight camera or the Kinect sensor developed for gaming, see Fig. 2.22.



(a) X-ray image (b) Thermographic image (c) 3D image

Fig. 2.22 Three different types of image. (**a**) X-ray image. Note the ring on the finger. (**b**) Thermographic image. The more reddish the higher the temperature. (**c**) 3D image. The more blueish the closer to the camera

2.6 Exercises

Exercise 1: Explain the following concepts: electromagnetic spectrum, focal length, exposure time, backlighting, saturation, focus, depth-of-fields, motion blur, spatial quantization, ROI.

Exercise 2: Explain the pros and cons of backlighting.

Exercise 3: Describe the image acquisition process. That is, from light to a digital image in a computer.

Exercise 4: What is the purpose of the lens?

Exercise 5: What is the focal length and how does it relate to zoom?

Exercise 6: How many different 512×512 gray-scale (8-bit) images can be constructed?

Exercise 7: Which pixel value is represented by the following bit sequence: 00101010?

Exercise 8: What is the bit sequence of the pixel value: 150?

Exercise 9: In a 100×100 gray-scale image each pixel is represented by 256 gray levels. How much memory (bytes) is required to store this image?

Exercise 10: In a 100×100 gray-scale image each pixel is represented by 4 gray levels. How much memory (bytes) is required to store this image?

Exercise 11: You want to photograph an object, which is 1 m tall and 10 m away from the camera. The height of the object in the image should be 1 mm. It is assumed that the object is in focus at the focal point. What should the focal length be?

Exercise 12a: Mick is 2 m tall and standing 5 m away from a camera. The focal length of the camera is 5 mm. A focused image of Mick is formed on the sensor. At which distance from the lens is the sensor located?

Exercise 12b: How tall (in mm) will Mick be on the sensor?

Exercise 12c: The camera sensor contains 640×480 pixels and its physical size is 6.4 mm $\times$ 4.8 mm. How tall (in pixels) will Mick be on the sensor?

Exercise 12d: What are the horizontal field-of-view and the vertical field-of-view of the camera?

Exercise 13: Show that $\frac{1}{g} + \frac{1}{b} = \frac{1}{f}$.

Additional exercise 1: How does the human eye capture light and how does that relate to the operations in a digital camera?

Additional exercise 2: How is auto-focus obtained in a digital camera?

Additional exercise 3: How is night vision obtained in for example binoculars and riflescopes?

Color Images

3

So far we have restricted ourselves to gray-scale images, but, as you might have noticed, the real world consists of colors. Going back some years, many cameras (and displays, e.g., TV-monitors) only handled gray-scale images. As the technology matured, it became possible to capture (and visualize) color images and today most cameras capture color images.

In this chapter we turn to the topic of color images. We describe the nature of color images and how they are captured and represented.

3.1 What Is a Color?

In Chap. 2 it was explained that an image is formed by measuring the amount of energy entering the image sensor. It was also stated that only energy within a certain frequency/wavelength range is measured. This wavelength range is denoted the *visual spectrum*, see Fig. 2.2. In the human eye this is done by the so-called *rods*, which are specialized nerve-cells that act as *photoreceptors*. Besides the rods, the human eye also contains *cones*. These operate like the rods, but are not sensitive to all wavelengths in the visual spectrum. Instead, the eye contains three types of cones, each sensitive to a different wavelength range. The human brain interprets the output from these different cones as different colors as seen in Table 3.1 [4].

So, a color is defined by a certain wavelength in the electromagnetic spectrum as illustrated in Fig. 3.1.

Since the three different types of cones exist we have the notion of the *primary colors* being red, green and blue. Psycho-visual experiments have shown that the different cones have different sensitivity. This means that when you see two different colors with the same intensity, you will judge their brightness differently. On average, a human perceives red as being 2.6 times as bright as blue and green as being 5.6 times as bright as blue. Hence the eye is more sensitive to green and least sensitive to blue.

When all wavelengths (all colors) are present at the same time, the eye perceives this as a shade of gray, hence no color is seen! If the energy level increases the shade becomes brighter and ultimately becomes white. Conversely, when the energy

T.B. Moeslund, *Introduction to Video and Image Processing*,
Undergraduate Topics in Computer Science,
DOI 10.1007/978-1-4471-2503-7_3, © Springer-Verlag London Limited 2012

Table 3.1 The different types of photoreceptor in the human eye. The cones are each specialized to a certain wavelength range and peak response within the visual spectrum. The output from each of the three types of cone is interpreted as a particular color by the human brain: red, green, and blue, respectively. The rods measure the amount of energy in the visual spectrum, hence the shade of gray. The type indicators L, M, S, are short for long, medium and short, respectively, and refer to the wavelength

Photoreceptor cell	Wavelength in nanometers (nm)	Peak response in nanometer (nm)	Interpretation by the human brain
Cones (type L)	[400–680]	564	Red
Cones (type M)	[400–650]	534	Green
Cones (type S)	[370–530]	420	Blue
Rods	[400–600]	498	Shade of gray

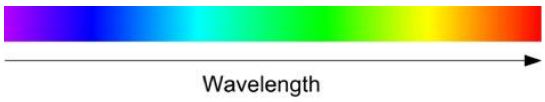


Fig. 3.1 The relationship between colors and wavelengths

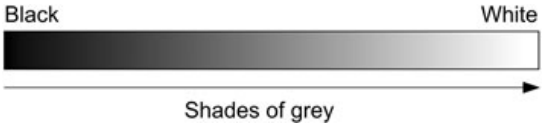


Fig. 3.2 Achromatic colors

level is decreased, the shade becomes darker and ultimately becomes black. This continuum of different gray-levels (or shades of gray) is denoted the *achromatic colors* and illustrated in Fig. 3.2. Note that this is the same as Fig. 2.18.

An image is created by sampling the incoming light. The colors of the incoming light depend on the color of the light source illuminating the scene and the material the object is made of, see Fig. 3.3. Some of the light that hits the object will bounce right off and some will penetrate into the object. An amount of this light will be absorbed by the object and an amount leaves again possibly with a different color. So when you see a green car this means that the wavelengths of the main light reflected from the car are in the range of the type M cones, see Table 3.1. If we assume the car was illuminated by the sun, which emits all wavelengths, then we can reason that all wavelengths *except* the green ones are absorbed by the material the car is made of. Or in other words, if you are wearing a black shirt all wavelengths (energy) are absorbed by the shirt and this is why it becomes hotter than a white shirt.

When the resulting color is created by illuminating an object by white light and then absorbing some of the wavelengths (colors) we use the notion of *subtractive colors*. Exactly as when you mix paint to create a color. Say you start with a white piece of paper, where no light is absorbed. The resulting color will be white. If you then want the paper to become green you add green paint, which absorbs everything but the green wavelengths. If you add yet another color of paint, then more wavelengths will be absorbed, and hence the resulting light will have a new color. Keep doing this and you will in theory end up with a mixture where all wavelengths are absorbed, that is, black. In practice, however, it will probably not be black, but rather dark gray/brown.

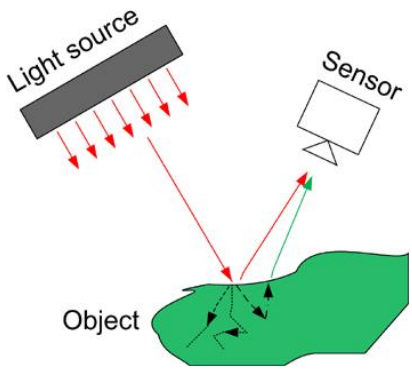


Fig. 3.3 The different components influencing the color of the received light

The opposite of subtractive colors is *additive colors*. This notion applies when you create the wavelengths as opposed to manipulating white light. A good example is a color monitor like a computer screen or a TV screen. Here each pixel is a combination of emitted red, green and blue light. Meaning that a black pixel is generated by not emitting anything at all. White (or rather a shade of gray) is generated by emitting the same amount of red, green, and blue. Red will be created by only emitting red light etc. All other colors are created by a combination of red, green and blue. For example yellow is created by emitting the same amount of red and green, and no blue.

3.2 Representation of an RGB Color Image

A color camera is based on the same principle as the human eye. That is, it measures the amount of incoming red light, green light and blue light, respectively. This is done in one of two ways depending on the number of sensors in the camera. In the case of three sensors, each sensor measures one of the three colors, respectively. This is done by splitting the incoming light into the three wavelength ranges using some optical filters and mirrors. So red light is only send to the "red-sensor" etc. The result is three images each describing the amount of red, green and blue light per pixel, respectively. In a color image, each pixel therefore consists of three values: red, green and blue. The actual representation might be three images—one for each color, as illustrated in Fig. 3.4, but it can also be a 3-dimensional vector for each pixel, hence an image of vectors. Such a vector looks like this:

$$\text{Color pixel} = [\text{Red, Green, Blue}] = [\text{R, G, B}] \tag{3.1}$$

In terms of programming a color pixel is usually represented as a *struct*. Say we want to set the RGB values of the pixel at position $(2, 4)$ to: Red $= 100$, Green $= 42$, and Blue $= 10$, respectively. In C-code this can for example be written as

```
f[2][4].R = 100;
f[2][4].G = 42;
f[2][4].B = 10;
```

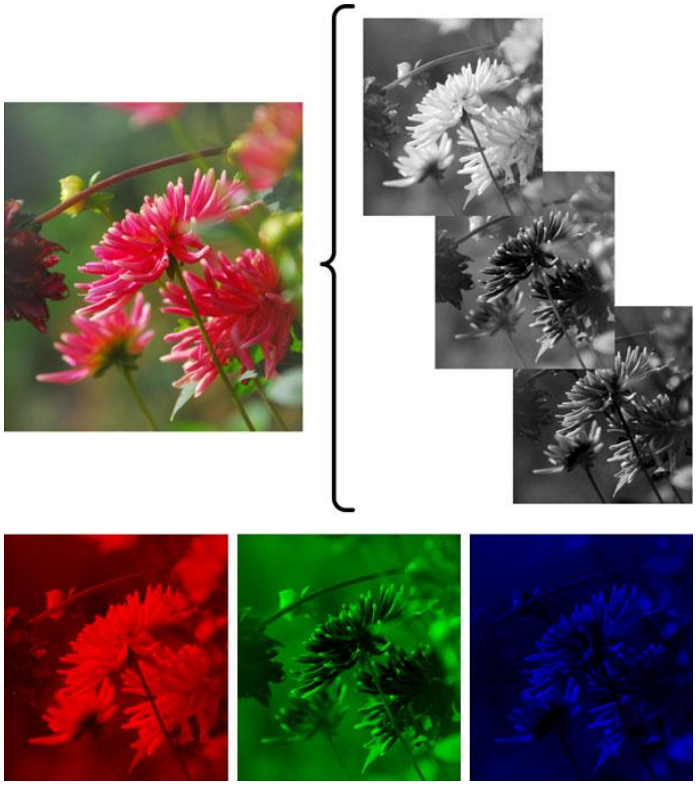


Fig. 3.4 A color image consisting of three images; red, green and blue

respectively; or alternatively:

```
SetPixel(image, 2, 4, R, 100);
SetPixel(image, 2, 4, G, 42)};
SetPixel(image, 2, 4, B, 10)};
```

Typically each color value is represented by an 8-bit (one byte) value meaning that 256 different shades of each color can be measured. Combining different values of the three colors, each pixel can represent $256^3 = 16,777,216$ different colors.

A cheaper alternative to having three sensors including mirrors and optical filters is to only have one sensor. In this case, each cell in the sensor is made sensitive to one of the three colors (ranges of wavelength). This can be done in a number of different ways. One is using a *Bayer pattern*. Here 50% of the cells are sensitive to green, while the remaining cells are divided equally between red and blue. The reason being, as mentioned above, that the human eye is more sensitive to green. The layout of the different cells is illustrated in Fig. 3.5.

The figure shows the upper-left corner of the sensor, where the letters illustrate which color a particular pixel is sensitive to. This means that each pixel only captures one color and that the two other colors of a particular pixel must be inferred from the neighbors. Algorithms for finding the remaining colors of a pixel are known as *demosaicing* and, generally speaking, the algorithms are characterized by the required processing time (often directly proportional to the number of neighbors included) and the quality of the output. The higher the processing time the better

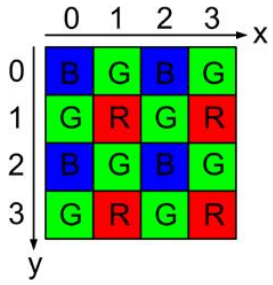


Fig. 3.5 The Bayer pattern used for capturing a color image on a single image sensor. R = red, G = green, and B = blue

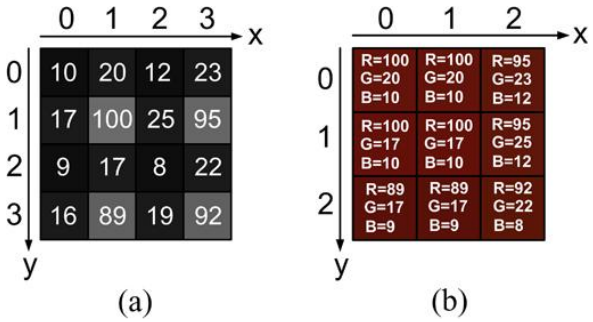


Fig. 3.6 (**a**) Numbers measured by the sensor. (**b**) Estimated RGB image using Eq. 3.2

the result. How to balance these two issues is up to the camera manufactures, and in general, the higher the quality of the camera, the higher the cost. Even very advanced algorithms are not as good as a three sensor color camera and note that when using, for example, a cheap web-camera, the quality of the colors might not be too good and care should be taken before using the colors for any processing. Regardless of the choice of demosaicing algorithm, the output is the same as when using three sensors, namely Eq. 3.1. That is, even though only one color is measured per pixel, the output for each pixel will (after demosaicing) consist of three values: R, G, and B.

An example of a simple demosaicing algorithm is to infer the missing colors from the nearest pixels, for example using the following set of equations:

$$g(x,y) \quad \begin{cases} [R,G,B]_B = [f(x+1,y+1), f(x+1,y), f(x,y)] \\ [R,G,B]_{GB} = [f(x,y+1), f(x,y), f(x-1,y)] \\ [R,G,B]_{GR} = [f(x+1,y), f(x,y), f(x,y-1)] \\ [R,G,B]_R = [f(x,y), f(x-1,y), f(x-1,y-1)] \end{cases} \tag{3.2}$$

where $f(x,y)$ is the input image (Bayer pattern) and $g(x,y)$ is the output RGB image. The RGB values in the output image are found differently depending on which color a particular pixel is sensitive to: $[R,G,B]_B$ should be used for the pixels sensitive to blue, $[R,G,B]_R$ should be used for the pixels sensitive to red, and $[R,G,B]_{GB}$ and $[R,G,B]_{GR}$ should be used for the pixels sensitive to green followed by a blue or red pixel, respectively.

In Fig. 3.6 a concrete example of this algorithm is illustrated. In the left figure the values sampled from the sensor are shown. In the right figure the resulting RGB output image is shown using Eq. 3.2.

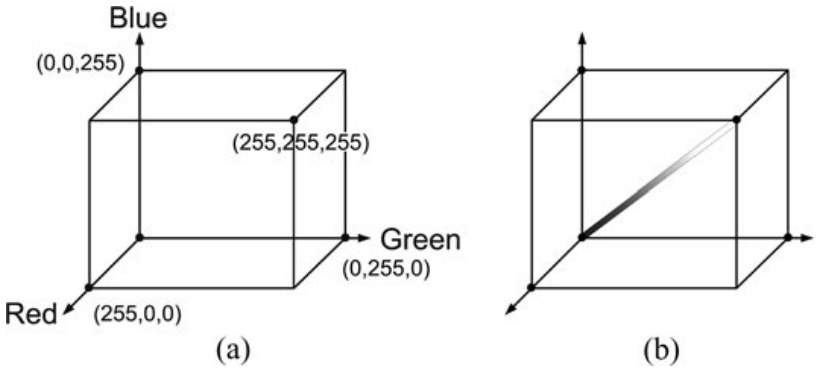


Fig. 3.7 (**a**) The RGB color cube. (**b**) The gray-vector in the RGB color cube

Table 3.2 The colors of the different corners in the RGB color cube

Corner	Color
(0, 0, 0)	Black
(255, 0, 0)	Red
(0, 255, 0)	Green
(0, 0, 255)	Blue
(255, 255, 0)	Yellow
(255, 0, 255)	Magenta
(0, 255, 255)	Cyan
(255, 255, 255)	White

3.2.1 The RGB Color Space

According to Eq. 3.1 a color pixel has three values and can therefore be represented as one point in a 3D space spanned by the three colors. If we say that each color is represented by 8-bits, then we can construct the so-called RGB color cube, see Fig. 3.7.

In the color cube a color pixel is one point or rather a vector from $(0, 0, 0)$ to the pixel value. The different corners in the color cube represent some of the *pure colors* and are listed in Table 3.2. The vector from $(0, 0, 0)$ to $(255, 255, 255)$ passes through all the gray-scale values and is denoted the *gray-vector*. Note that the gray-vector is identical to Fig. 3.2.

3.2.2 Converting from RGB to Gray-Scale

Even though you use a color camera it might be sufficient for your algorithm to apply the intensity information in the image and you therefore need to convert the color image into a gray-scale image. Converting from RGB to gray-scale is performed as

$$I = W_R \cdot \mathrm{R} + W_G \cdot \mathrm{G} + W_B \cdot \mathrm{B} \tag{3.3}$$

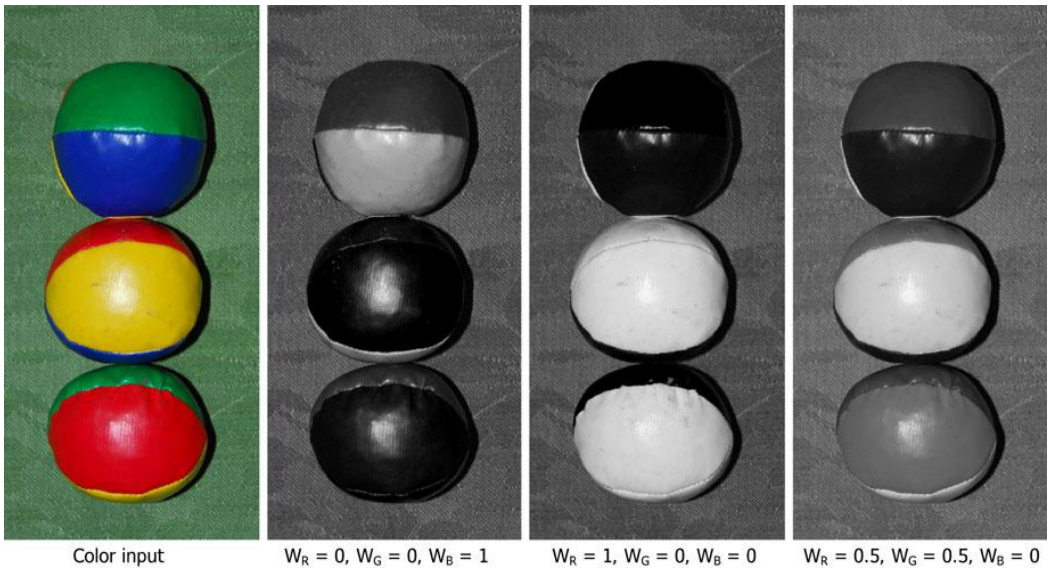


Fig. 3.8 A color image and how it can be mapped to different gray-scale images depending on the weights

where I is the intensity and W_R, W_G, and W_B are weight factors for R, G, and B, respectively. To ensure the value of Eq. 3.3 is within one byte, i.e. in the range $[0, 255]$, the weight factors must sum to one. That is $W_R + W_G + W_B = 1$. As default the three colors are equally important, hence $W_R = W_G = W_B = \frac{1}{3}$, but depending on the application one or two colors might be more important and the weight factors should be set accordingly. For example when processing images of vegetation the green color typically contains the most information or when processing images of metal objects the most information is typically located in the blue pixels. Yet another example could be when looking for human skin (face and hands) which has a reddish color. In general, the weights should be set according to your application and a good way of assessing this is by looking at the histograms of each color.[1] An example of a color image transformed into a gray-scale image can be seen in Fig. 3.8. Generally, it is not possible to convert a gray-scale image back into the original color image, since the color information is lost during the color to gray-scale transformation.

When the goal of a conversion from color to gray-scale is not to prepare the image for processing but rather for visualization purposes, then an understanding of the human visual perception can help decide the weight factors. The optimal weights vary from individual to individual, but the weights listed below are a good compromise, agreed upon by major international standardization organizations within TV and image/video coding. When the weights are optimized for the human visual system, the resulting gray-scale value is denoted *luminance* and usually represented as Y.

$$W_R = 0.299, \qquad W_G = 0.587, \qquad W_B = 0.114 \tag{3.4}$$

[1] An image histogram is defined in the next chapter.

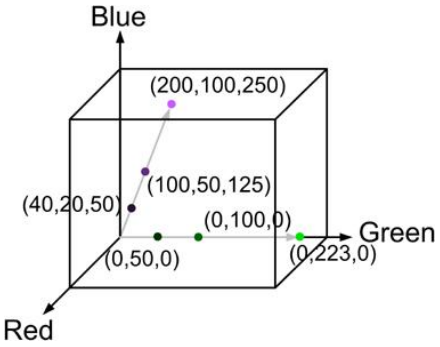


Fig. 3.9 The RGB color cube. Each dot corresponds to a particular pixel value. Multiple dots on the same line all have the same color, but different levels of illumination

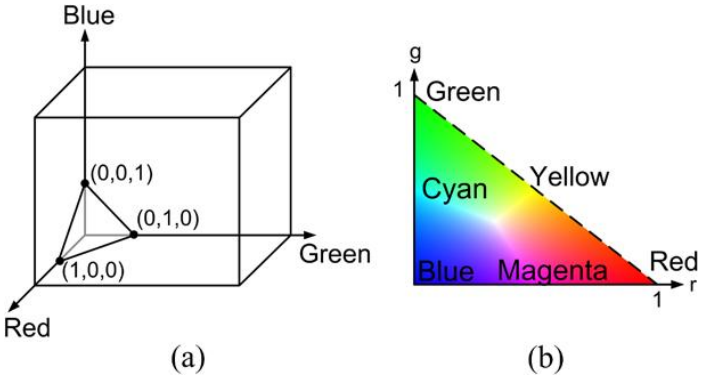


Fig. 3.10 (**a**) The triangle where all color vectors pass through. The value of a point on the triangle is defined using normalized RGB coordinates. (**b**) The chromaticity plane

3.2.3 The Normalized RGB Color Representation

If we have the following three RGB pixel values $(0, 50, 0)$, $(0, 100, 0)$, and $(0, 223, 0)$ in the RGB color cube, we can see that they all lie on the same vector, namely the one spanned by $(0, 0, 0)$ and $(0, 255, 0)$. We say that all values are a shade of green and go even further and say that they all have the same color (green), but different levels of illumination. This also applies to the rest of the color cube. For example, the points $(40, 20, 50)$, $(100, 50, 125)$ and $(200, 100, 250)$ all lie on the same vector and therefore have the same color, but just different illumination levels. This is illustrated in Fig. 3.9.

If we generalize this idea of different points on the same line having the same color, then we can see that all possible lines pass through the triangle defined by the points $(1, 0, 0)$, $(0, 1, 0)$ and $(0, 0, 1)$, see Fig. 3.10(a). The actual point (r, g, b) where a line intersects the triangle is found as[2]:

$$(r, g, b) = \left(\frac{R}{R+G+B}, \frac{G}{R+G+B}, \frac{B}{R+G+B} \right) \tag{3.5}$$

These values are named *normalized RGB* and denoted (r, g, b). In Table 3.3 the rgb values of some RGB values are shown. Note that each value is in the interval $[0, 1]$ and that $r + g + b = 1$. This means that if we know two of the normalized

[2]Note that the formula is undefined for $(R, G, B) = (0, 0, 0)$. We therefore make the following definition: $(r, g, b) \equiv (0, 0, 0)$ when $(R, G, B) = (0, 0, 0)$.

RGB values, then we can easily find the remaining value, or in other words, we can represent a normalized RGB color using just two of the values. Say we choose r and g, then this corresponds to representing the triangle in Fig. 3.10(a) by the triangle to the right, see Fig. 3.10(b). This triangle is denoted the *chromaticity plane* and the colors along the edges of the triangle are the so-called pure colors. The further away from the edges the less pure the color and ultimately the center of the triangle has no color at all and is a shade of gray. It can be stated that the closer to the center a color is, the more "polluted" a pure color is by white light.

Summing up we can now re-represent an RGB value by its "*true*" color, r and g, and the amount of light (intensity or energy or illumination) in the pixel. That is,

$$(R, G, B) \quad \Leftrightarrow \quad (r, g, I) \tag{3.6}$$

where $I = \frac{R+G+B}{3}$. In Table 3.3 the rgI values of some RGB values are shown.[3] Separating the color and the intensity like this can be a powerful notion in many applications. In Sect. 4.4.1 one will be presented.

In terms of programming the conversion from (R, G, B) to (r, g, I) can be implemented in C-Code as illustrated below:

```
for (y = 0; y < M; y = y+1)
{
  for (x = 0; x < N; x = x+1)
  {
     temp = GetPixel(input, x, y, R) +
            GetPixel(input, x, y, G) +
            GetPixel(input, x, y, B);
     value = GetPixel(input, x, y, R) / temp;
     SetPixel(output, x, y, r, value);
     value = GetPixel(input, x, y, G) / temp;
     SetPixel(output, x, y, g, value);
     value = temp / 3;
     SetPixel(output, x, y, I, value);
  }
}
```

where M is the height of the image, N is the width of the image, *input* is the RGB image, and *output* is the rgI image. The programming example primarily consists of two *FOR-loops* which go through the image, pixel-by-pixel, and convert from an input image (RGB) to an output image (rgI). The opposite conversion from (r, g, I) to (R, G, B) can be implemented as

[3] If r and g need to be represented using one byte for each color we can simply multiply each with 255 and the new values will be in the interval $[0, 255]$.

```
for (y = 0; y < M; y = y+1)
{
  for (x = 0; x < N; x = x+1)
  {
     temp = 3 * GetPixel(input,x,y,I);
     value = GetPixel(input, x, y, r) * temp;
     SetPixel(output, x, y, R, value);
     value = GetPixel(input, x, y, g) * temp;
     SetPixel(output, x, y, G, value);
     value = (1 - GetPixel(input, x, y, r) -
              GetPixel(input, x, y, g)) * temp;
     SetPixel(output, x, y, B, value);
  }
}
```

where M is the height of the image, N is the width of the image, *input* is the rgI image, and *output* is the RGB image.

3.3 Other Color Representations

From a human perception point of view the triangular representation in 3.10(b) is not intuitive. Instead humans rather use the notion of *hue* and *saturation*, when perceiving colors. The hue is the dominant wavelength in the perceived light and represents the pure color, i.e., the colors located on the edges of the triangle in Fig. 3.10(b). The saturation is the purity of the color and represents the amount of white light mixed with the pure color. To understand these entities better, let us look at Fig. 3.11(a). First of all we see that the point C corresponds to the neutral point, meaning the colorless center of the triangle where $(r, g) = (1/3, 1/3)$. Let us define a random point in the triangle as P. The hue of this point is now defined as an angle, θ, between the vectors $\overrightarrow{C_{r=1}}$ and $\overrightarrow{CP}$. So hue $= 0°$ means red and hue = 120° means green.

If the point P is located on the edge of the triangle then we say the saturation is 1, hence a pure color. As the point approaches C the saturation goes toward 0, and ultimately becomes 0 when $P = C$. Since the distance from C to the three edges of the triangle is not uniform, the saturation is defined as a relative distance. That is, saturation is defined as the ratio between the distance from C to P, and the distance from C to the point on the edge of the triangle in the direction of $\overrightarrow{CP}$. Mathematically we have

$$\text{Saturation} = \frac{\|\overrightarrow{CP}\|}{\|\overrightarrow{CP'}\|}, \qquad \text{Hue} = \theta \tag{3.7}$$

where $\|\overrightarrow{CP}\|$ is the length of the vector $\overrightarrow{CP}$. The representation of colors based on hue and saturation results in a circle as opposed to the triangle in Fig. 3.10(b). In Fig. 3.11(b) the hue–saturation representation is illustrated together with some of

Table 3.3 Some different colors and their representation in the different color spaces. ND = Not Defined

Color	(R, G, B)	(r, g, b)	(r, g, I)	(H, S, I)	(H, S, V)	(Y, U, V)	(Y, C_b, C_r)
Red	(255, 0, 0)	(1, 0, 0)	(1, 0, 85)	(0, 1, 85)	(0, 1, 255)	(76, −37, 157)	(76, 85, 255)
Yellow	(255, 255, 0)	(1/2, 1/2, 0)	(1/2, 1/2, 170)	(60, 1, 170)	(60, 1, 255)	(226, −111, 26)	(226, 255, 149)
Green	(0, 255, 0)	(0, 1, 0)	(0, 1, 85)	(120, 1, 85)	(120, 1, 255)	(150, −74, −131)	(150, 100, 115)
Cyan	(0, 255, 255)	(0, 1/2, 1/2)	(0, 1/2, 170)	(180, 1, 170)	(180, 1, 255)	(179, 38, −157)	(179, 171, 0)
Blue	(0, 0, 255)	(0, 0, 1)	(0, 0, 85)	(240, 1, 85)	(240, 1, 255)	(29, 111, −26)	(29, 255, 107)
Magenta	(255, 0, 255)	(1/2, 0, 1/2)	(1/2, 0, 170)	(300, 1, 170)	(300, 1, 255)	(105, 74, 131)	(105, 212, 235)
Black	(0, 0, 0)	(0, 0, 0)	(0, 0, 0)	(ND, 0, 0)	(ND, 0, 0)	(0, 0, 0)	(0, 128, 128)
White	(255, 255, 255)	(1/3, 1/3, 1/3)	(1/3, 1/3, 255)	(ND, 0, 255)	(ND, 0, 255)	(255, 0, 0)	(255, 128, 128)
25% white	(64, 64, 64)	(1/3, 1/3, 1/3)	(1/3, 1/3, 64)	(ND, 0, 64)	(ND, 0, 64)	(64, 0, 0)	(64, 128, 128)
50% white	(128, 128, 128)	(1/3, 1/3, 1/3)	(1/3, 1/3, 128)	(ND, 0, 128)	(ND, 0, 128)	(128, 0, 0)	(128, 128, 128)
25% Blue	(0, 0, 64)	(0, 0, 1)	(0, 0, 21)	(240, 1, 21)	(240, 1, 64)	(7, 28, −6)	(7, 160, 123)
50% Blue	(0, 0, 128)	(0, 0, 1)	(0, 0, 43)	(240, 1, 43)	(240, 1, 128)	(15, 58, −13)	(15, 192, 118)
75% Blue	(0, 0, 192)	(0, 0, 1)	(0, 0, 64)	(240, 1, 64)	(240, 1, 192)	(22, 84, −19)	(22, 224, 112)
Orange	(255, 165, 0)	(0.6, 0.4, 0)	(0.6, 0.4, 140)	(40, 1, 140)	(39, 1, 255)	(173, 10, 72)	(173, 30, 186)
Pink	(255, 192, 203)	(0.4, 0.3, 0.3)	(0.4, 0.3, 217)	(351, 0.1, 217)	(350, 0.2, 255)	(212, −4, 57)	(212, 123, 157)
Brown	(165, 42, 42)	(0.6, 0.2, 0.2)	(0.6, 0.2, 83)	(0, 0.5, 83)	(0, 0.7, 165)	(79, −18, 76)	(79, 107, 190)

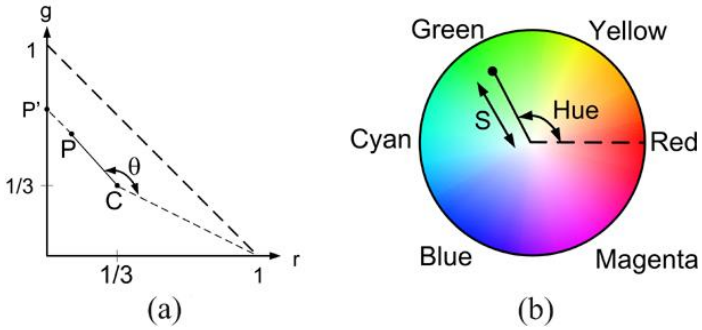


Fig. 3.11 (**a**) The definition of hue and saturation. (**b**) The hue–saturation representation. The color of a pixel (indicated by a dot) is represented by a hue value and a saturation value (denoted S in the figure). The figure also indicates the location of some of the pure colors

the pure colors. It is important to realize how this figure relates to Fig. 3.7, or in other words, how the hue–saturation representation relates to the RGB representation. The center of the hue–saturation circle in Fig. 3.11(b) is a shade of gray and corresponds to the gray-vector in Fig. 3.7. The circle is located so that it is perpendicular to the gray-vector. For a particular RGB value, the hue–saturation circle is therefore centered at a position on the gray-vector, so that the RGB value is included in the circle.

A number of different color representations exist, which are based on the notion of hue and saturation. Below two of these are presented.[4]

3.3.1 The HSI Color Representation

The HSI color representation is short for hue, saturation and intensity. The representation follows the exact definition mentioned above. That is, the intensity is defined as $I = \frac{R+G+B}{3}$ and hue and saturation is defined as illustrated in Fig. 3.11. When calculating the conversion from RGB to HSI we seek a way of avoiding fist converting from RGB to rg, i.e., we want to represent the conversion in terms of RGB values. In Appendix D it is shown how this is possible and the resulting conversion from RGB to HSI is defined as

$$H = \begin{cases} \cos^{-1}\left(1/2 \cdot \frac{(R-G)+(R-B)}{\sqrt{(R-G)(R-G)+(R-B)(G-B)}}\right), & \text{if } G \geq B; \\ 360^\circ - \cos^{-1}\left(1/2 \cdot \frac{(R-G)+(R-B)}{\sqrt{(R-G)(R-G)+(R-B)(G-B)}}\right), & \text{Otherwise} \end{cases} \tag{3.8}$$

$$H \in [0, 360[$$

$$S = 1 - 3 \cdot \frac{\min\{R, G, B\}}{R+G+B} \quad S \in [0, 1] \tag{3.9}$$

$$I = \frac{R+G+B}{3} \quad I \in [0, 255] \tag{3.10}$$

[4]It should be noted that the naming of the different color representations based on hue and saturation is not consistent throughout the body of literature covering this subject. Please have this in mind when studying other information sources.

where $\min\{R, G, B\}$ means the smallest of the R, G, and B values, see Appendix B. Saturation is defined to be zero when $(R, G, B) = (0, 0, 0)$ and hue is undefined for gray-values, i.e., when $R = G = B$. The conversion from HSI to RGB is given as

$$H_n = \begin{cases} 0, & \text{if } 0° \leq H \leq 120°; \\ H - 120°, & \text{if } 120° < H \leq 240°; \\ H - 240°, & \text{if } 240° < H < 360° \end{cases} \tag{3.11}$$

$$R = \begin{cases} I \cdot \left(1 + \frac{S \cdot \cos(H_n)}{\cos(60° - H_n)}\right), & \text{if } 0° \leq H \leq 120°; \\ I - I \cdot S, & \text{if } 120° < H \leq 240°; \\ 3I - G - B, & \text{if } 240° < H < 360° \end{cases} \tag{3.12}$$

$$G = \begin{cases} 3I - R - B, & \text{if } 0° \leq H \leq 120°; \\ I \cdot \left(1 + \frac{S \cdot \cos(H_n)}{\cos(60° - H_n)}\right), & \text{if } 120° < H \leq 240°; \\ I - I \cdot S, & \text{if } 240° < H < 360° \end{cases} \tag{3.13}$$

$$B = \begin{cases} I - I \cdot S, & \text{if } 0° \leq H \leq 120°; \\ 3I - R - G, & \text{if } 120° < H \leq 240°; \\ I \cdot \left(1 + \frac{S \cdot \cos(H_n)}{\cos(60° - H_n)}\right), & \text{if } 240° < H < 360° \end{cases} \tag{3.14}$$

In Table 3.3 the HSI values of some RGB pixels are shown.[5]

3.3.2 The HSV Color Representation

The HSV color representation is short for *hue*, *saturation* and *value*. One can think of HSV as an approximation of HSI, but much simpler to calculate. This is true, but it is important to notice that HSV is not defined to be an approximation of HSI. It is rather defined from an artist's point of view. Consider the situation when an artist mixes paint. She would choose a pure color and lighten it by adding white or darkening it by adding black. In the HSV representation the actions of the artist are modeled in the following way. The pure color obviously corresponds to hue. Increasing the whiteness (by adding white) corresponds to lowing the saturation. Finally, increasing the amount of black corresponds to lowering the intensity of R, G, and B. Concretely, this is modeled by the intensity of the maximum color and denoted *value*, i.e., value = $\max\{R, G, B\}$.

Following these definitions, a very elegant geometric argument can be made leading to a computationally simpler representation of hue, saturation, and value, than HSI. The conversion from RGB to HSV is given as (see Appendix E for details):

[5] Note that sometimes all parameters are normalized to the interval [0, 1]. For example for H this is done as $H_{\text{normalized}} = \frac{H}{360}$.

$$H = \begin{cases} \frac{G-B}{V-\min\{R,G,B\}} \cdot 60°, & \text{if } V = R \text{ and } G \geq B; \\ \left(\frac{B-R}{V-\min\{R,G,B\}} + 2\right) \cdot 60°, & \text{if } G = V; \\ \left(\frac{R-G}{V-\min\{R,G,B\}} + 4\right) \cdot 60°, & \text{if } B = V; \\ \left(\frac{R-B}{V-\min\{R,G,B\}} + 5\right) \cdot 60°, & \text{if } V = R \text{ and } G < B \end{cases} \qquad H \in [0°, 360°[\tag{3.15}$$

$$S = \frac{V - \min\{R, G, B\}}{V} \quad S \in [0, 1] \tag{3.16}$$

$$V = \max\{R, G, B\} \quad V \in [0, 255] \tag{3.17}$$

where $\min\{R, G, B\}$ and $\max\{R, G, B\}$ are the smallest and biggest of the R, G, and B values, respectively, see Appendix B. As for HSI saturation is defined to be zero when $(R, G, B) = (0, 0, 0)$ and hue is undefined for gray-values, i.e., when $R = G = B$. The conversion from HSV to RGB is given as

$$K = \left\lfloor \frac{H}{60°} \right\rfloor \tag{3.18}$$

$$T = \frac{H}{60°} - K \tag{3.19}$$

$$X = V \cdot (1 - S) \tag{3.20}$$

$$Y = V \cdot (1 - S \cdot T) \tag{3.21}$$

$$Z = V \cdot \big(1 - S \cdot (1 - T)\big) \tag{3.22}$$

$$(R, G, B) = \begin{cases} (V, Z, X), & \text{if } K = 0; \\ (Y, V, X), & \text{if } K = 1; \\ (X, V, Z), & \text{if } K = 2; \\ (X, Y, V), & \text{if } K = 3; \\ (Z, X, V), & \text{if } K = 4; \\ (V, X, Y), & \text{if } K = 5 \end{cases} \tag{3.23}$$

where $\lfloor x \rfloor$ means the floor of x, see Appendix B. In Table 3.3 the HSV values of some RGB colors are shown.

3.3.3 The YUV and YC_bC_r Color Representations

A number of other color representations exist, but those mentioned above are those most often applied in image processing. One exception, however, is the color representations used for transmission, storage, and compression of image and video. These representations all have a similar structure, which is presented in this section.

In the early days of TV only monochrome screens were available and hence only intensity information was transmitted from the TV stations. RGB cameras captured RGB signals, but converted them into luminance values, denoted Y, before transmit-

ting them. Knowledge of human perception was taken into account when defining the weights used for the conversion, see Sect. 3.2.2:

$$Y = W_R \cdot R + W_G \cdot G + W_B \cdot B \quad Y \in [0, 255] \tag{3.24}$$

where $W_R + W_G + W_B = 1$.

As the color screen technology matured, a need for transmitting color signals arose. Two requirements were set up when defining how to transmit color signals: 1) The signal should be compatible with the already existing signals used for monochrome screens and 2) the decoding on the receiver side should be as simple as possible. From this it followed that the color information was transmitted as *weighted difference signals* with respect to Y:

$$X_1 = \frac{W_{X1}}{1 - W_B} \cdot (B - Y) \quad X_1 \in [-W_{X1} \cdot 255, W_{X1} \cdot 255] \tag{3.25}$$

$$X_2 = \frac{W_{X2}}{1 - W_R} \cdot (R - Y) \quad X_2 \in [-W_{X2} \cdot 255, W_{X2} \cdot 255] \tag{3.26}$$

where W_{X1} and W_{X2} are weight factors, W_R and W_B are from Eq. 3.24, and X_1 and X_2 encode the blue and red information, respectively. The green information can then be inferred from Y, X_1 and X_2. Note that when no color is present, i.e. $R = G = B$, we have $X_1 = 0$ and $X_2 = 0$, see Appendix F. This means that X_1 and X_2 need not be send.

So, by transmitting (Y, X_1, X_2) a monochrome receiver can simply show Y, while a color receiver can decode (R, G, B) and show a color signal using the following equations, see Appendix F for details:

$$R = Y + X_2 \cdot \frac{1 - W_R}{W_{X2}} \tag{3.27}$$

$$G = Y - X_1 \cdot \frac{W_B \cdot (1 - W_B)}{W_{X1} \cdot W_G} - X_2 \cdot \frac{W_R \cdot (1 - W_R)}{W_{X2} \cdot W_G} \tag{3.28}$$

$$B = Y + X_1 \cdot \frac{1 - W_B}{W_{X1}} \tag{3.29}$$

Note that since all the weights are known in advance the conversion becomes rather simple.

One of the most well known color spaces using this principle is the YUV color space. The YUV color space is for example used in most European TV transmission standards. YUV uses the weights: $W_R = 0.299$, $W_G = 0.587$, $W_B = 0.114$, $W_{X1} = 0.436$, and $W_{X2} = 0.615$, and has the conversion listed below, see Appendix F for details. In Table 3.3 the YUV values of some RGB values are shown.

$$\begin{bmatrix} Y \\ U \\ V \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.147 & -0.289 & 0.436 \\ 0.615 & -0.515 & -0.100 \end{bmatrix} \cdot \begin{bmatrix} R \\ G \\ B \end{bmatrix} \quad \begin{matrix} Y \in [0, 255] \\ U \in [-111, 111] \\ V \in [-157, 157] \end{matrix} \tag{3.30}$$

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1.000 & 0.000 & 1.140 \\ 1.000 & -0.395 & -0.581 \\ 1.000 & 2.032 & 0.000 \end{bmatrix} \cdot \begin{bmatrix} Y \\ U \\ V \end{bmatrix} \quad \begin{matrix} R \in [0, 255] \\ G \in [0, 255] \\ B \in [0, 255] \end{matrix} \tag{3.31}$$

Another well know color space using this principle is the $\mathrm{YC}_b\mathrm{C}_r$ color space, which is used in for example JPEG and MPEG. $\mathrm{YC}_b\mathrm{C}_r$ uses the weights: $W_R = 0.299$, $W_G = 0.587$, $W_B = 0.114$, $W_{X1} = 0.5$, and $W_{X2} = 0.5$, and has the conversions listed below. See Appendix F for details.

$$\begin{bmatrix} Y \\ C_b \\ C_r \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.169 & -0.331 & 0.500 \\ 0.500 & -0.419 & -0.081 \end{bmatrix} \cdot \begin{bmatrix} R \\ G \\ B \end{bmatrix} + \begin{bmatrix} 0 \\ 128 \\ 128 \end{bmatrix} \quad \begin{matrix} Y \in [0, 255] \\ C_b \in [0, 255] \\ C_r \in [0, 255] \end{matrix} \tag{3.32}$$

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1.000 & 0.000 & 1.403 \\ 1.000 & -0.344 & -0.714 \\ 1.000 & 1.773 & 0.000 \end{bmatrix} \cdot \begin{bmatrix} Y \\ C_b - 128 \\ C_r - 128 \end{bmatrix} \quad \begin{matrix} R \in [0, 255] \\ G \in [0, 255] \\ B \in [0, 255] \end{matrix} \tag{3.33}$$

Note that 128 is added/subtracted in order to bring the values into the range [0, 255]. Note also the simplicity of the conversions compared to those for HSI and HSV. In Table 3.3 the $\mathrm{YC}_b\mathrm{C}_r$ values of some RGB values are shown.

3.4 Further Information

When reading literature on color spaces and color processing it is important to realize that a number of different terms are used.[6] Unfortunately, some of these terms are used interchangeably even though they might have different physical/perceptual/technical meanings. We therefore give a guideline to some of the terms you are likely to encounter when reading literature on colors:

Chromatic Color All colors in the RGB color cube except those lying on the gray-line spanned by $(0, 0, 0)$ and $(255, 255, 255)$.

Achromatic Color The colorless values in the RGB cube, i.e., all those colors lying on the gray-line. The opposite of chromatic color.

Shades of gray The same as achromatic color.

Intensity The average amount of energy, i.e., $(R + G + B)/3$.

Brightness The amount of light perceived by a human.

Lightness The amount of light perceived by a human.

Luminance The amount of light perceived by a human. Note that when you venture into the science of color understanding, the luminance defines the amount of emitted light.

Luma Gamma-corrected luminance.

[6] When going into color perception and color understanding even more terms are added to the vocabulary.

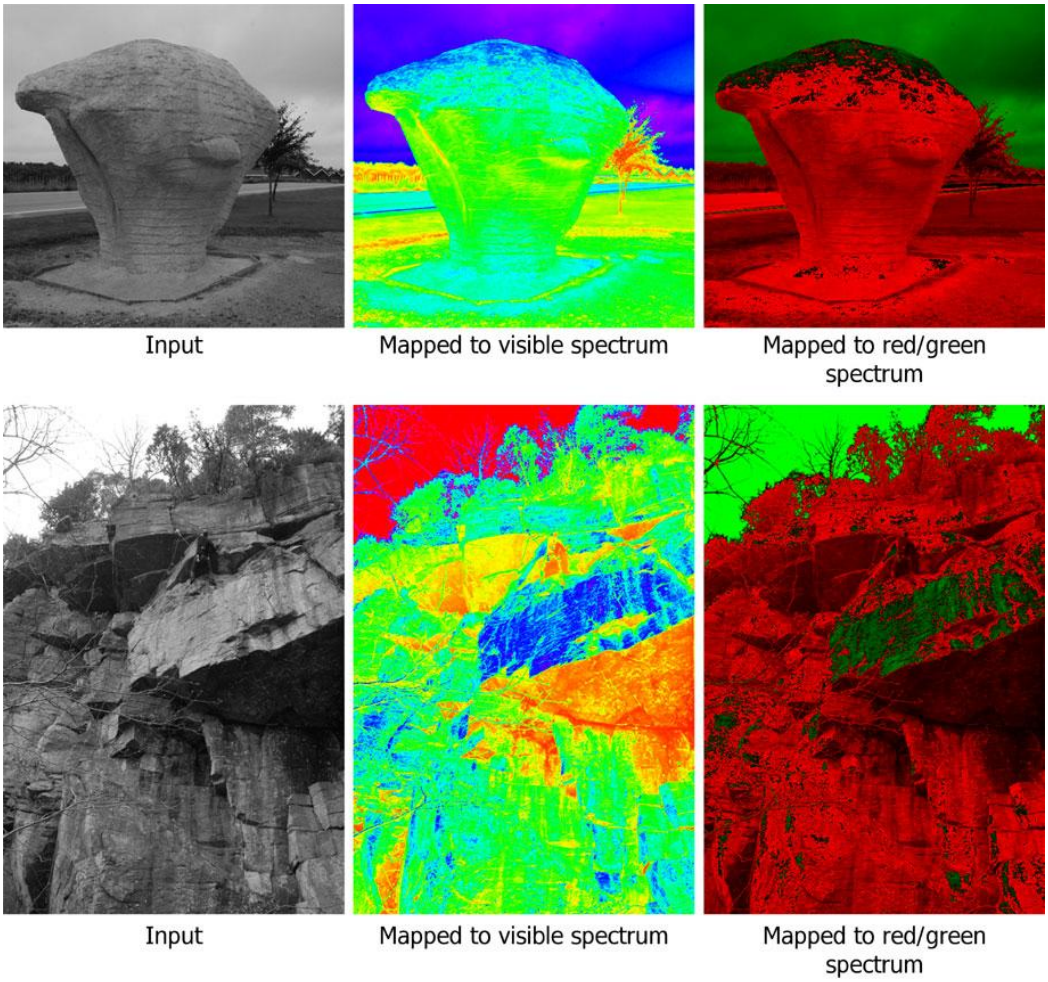


Fig. 3.12 Examples of pseudo color mapping

Shade Darkening a color. When a subtractive color space is applied, different shades (darker nuances) of a color are obtained by mixing the color with different amounts of black.

Tint Lightening a color. When a subtractive color space is applied, different tints (lighter nuances) of a color are obtained by mixing the color with different amounts of white.

Tone A combination of shade and tint, where gray is mixed with the input color.

’(**denoted prime**) The primed version of a color, i.e., R’, means that the value has been gamma-corrected.

Sometimes a gray-scale image is mapped to a color image in order to enhance some aspect of the image. As mentioned above a true color image cannot be reconstructed from a gray-level image. We therefore use the term *pseudo color* to underline that we are not talking about a true RGB image. How to map from gray-scale to color can be done in many different ways. In Fig. 3.12 and Fig. 2.22 examples are illustrated.

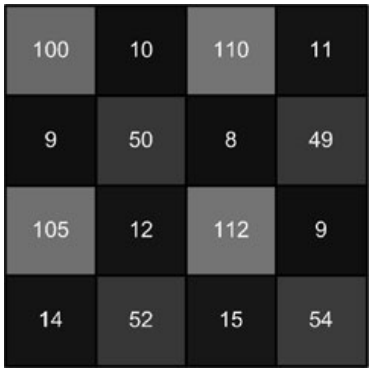


Fig. 3.13 A color image captured by a Bayer pattern

3.5 Exercises

Exercise 1: Explain the following concepts: rods, cones, achromatic, chromaticity plane, additive colors, subtractive colors, color spaces.

Exercise 2: How many different 512×512 color (24-bit) images can be constructed?

Exercise 3: The image in Fig. 3.13 was captured by a Bayer pattern sensor. Use demosaicing to convert the image into an RGB image.

Exercise 4: An RGB image is converted into a gray-scale image so that the cyan color is enhanced. What are the weight factors for R, G, and B, respectively?

Exercise 5: Is the RGB pixel $(R, G, B) = (42, 42, 42)$ located on the gray-vector?

Exercise 6: An RGB image is converted into a gray-scale image. During the conversion $W_B = 0$ and the two remaining colors are weighted equally. A pixel in the gray-scale image has the value 100. How much green was present in the corresponding RGB pixel when we know that $R = 20$?

Exercise 7: Convert the RGB pixel $(R, G, B) = (20, 40, 60)$ into (r, g, b), (r, g, I), (H, S, I), (H, S, V), (Y, U, V), and (Y, C_b, C_r), respectively.

Exercise 8: Show that $r + g + b = 1$.

Additional exercise 1: How is color represented in HTML?

Additional exercise 2: What is the "red-eye effect" in pictures and what can be done about it?

Additional exercise 3: What is white balance?

Additional exercise 4: What is color blindness?

Point Processing

4

Sometimes when people make a movie they lower the overall intensity in order to create a special atmosphere. Some overdo this and the result is that the viewer cannot see anything except darkness. What do you do? You pick up your remote and adjust the level of the light by pushing the brightness button. When doing so you actually perform a special type of image processing known as *point processing*.

Say we have an input image $f(x, y)$ and wish to manipulate it resulting in a different image, denoted the *output image* $g(x, y)$. In the case of changing the brightness in a movie, the input image will be the one stored on the DVD you are watching and the output image will be the one actually shown on the TV screen. Point processing is now defined as an operation which calculates the new value of a pixel in $g(x, y)$ based on the value of the pixel *in the same position* in $f(x, y)$ and some operation. That is, the values of a pixel's neighbors in $f(x, y)$ have no effect whatsoever, hence the name point processing. In the forthcoming chapters the neighbor pixels *will* play an important role. The principle of point processing is illustrated in Fig. 4.1. In this chapter some of the most fundamental point processing operations are described.

4.1 Gray-Level Mapping

When manipulating the brightness by your remote you actually change the value of b in the following equation:

$$g(x, y) = f(x, y) + b \tag{4.1}$$

Every time you push the '+' brightness button the value of b is increased and vice versa. The result of increasing b is that a higher and higher value is added to each pixel in the input image and hence it becomes brighter. If $b > 0$ the image becomes brighter and if $b < 0$ the image becomes darker. The effect of changing the brightness is illustrated in Fig. 4.2.

T.B. Moeslund, *Introduction to Video and Image Processing*,
Undergraduate Topics in Computer Science,
DOI 10.1007/978-1-4471-2503-7_4, © Springer-Verlag London Limited 2012

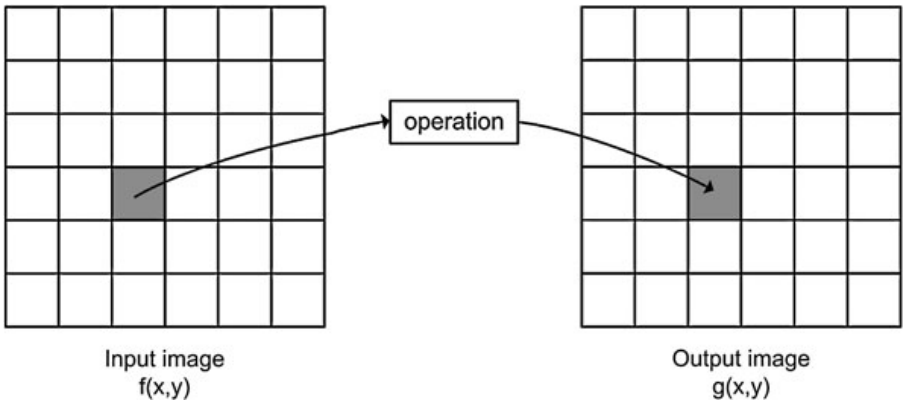


Fig. 4.1 The principle of point processing. A pixel in the input image is processed and the result is stored at the same position in the output image

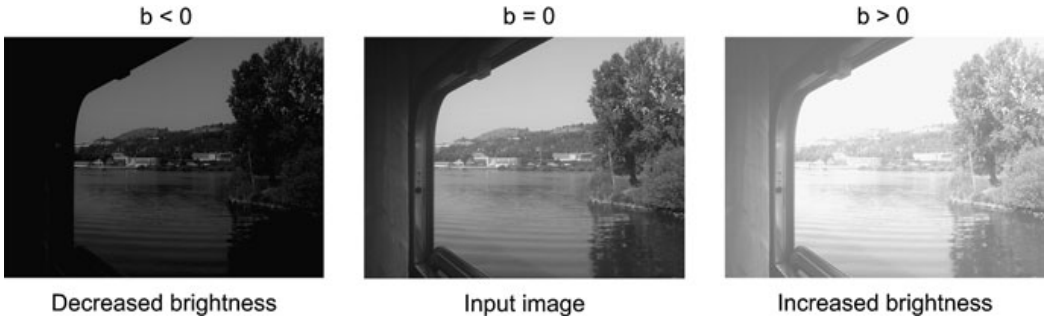


Fig. 4.2 If b in Eq. 4.1 is zero, the resulting image will be equal to the input image. If b is a negative number, then the resulting image will have decreased brightness, and if b is a positive number the resulting image will have increased brightness

An often more convenient way of expressing the brightness operation is by the use of a graph, see Fig. 4.3. The graph shows how a pixel value in the input image (horizontal axis) maps to a pixel value in the output image (vertical axis). Such a graph is denoted *gray-level mapping*. In the first graph, the mapping does absolutely nothing, i.e., $g(142, 42) = f(142, 42)$. In the next graph all pixel values are increased ($b > 0$), hence the image becomes brighter. This results in two things: i) no pixel will be completely dark in the output and ii) some pixels will have a value above 255 in the output image. The latter is no good due to the upper limit of an 8-bit image and therefore all pixels above 255 are set equal to 255 as illustrated by the horizontal part of the graph. When $b < 0$ some pixels will have negative values and are therefore set equal to zero in the output as seen in the last graph.

Just like changing the brightness on your TV, you can also change the contrast. The contrast of an image is a matter of how different the gray-level values are. If we look at two pixels next to each other with values 112 and 114, then the human eye has difficulties distinguishing them and we will say there is a *low* contrast. On the other hand if the pixels are 112 and 212, respectively, then we can easily distinguish

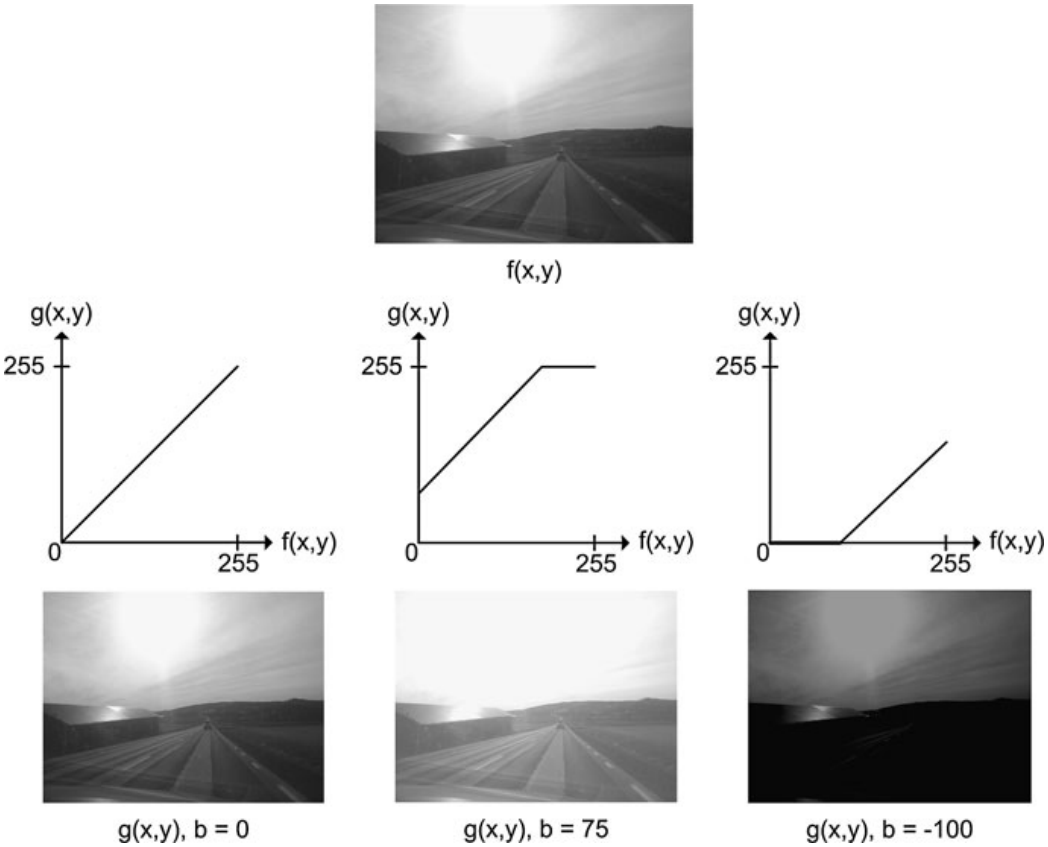


Fig. 4.3 Three examples of gray-level mapping. The top image is the input. The three other images are the result of applying the three gray-level mappings to the input. All three gray-level mappings are based on Eq. 4.1

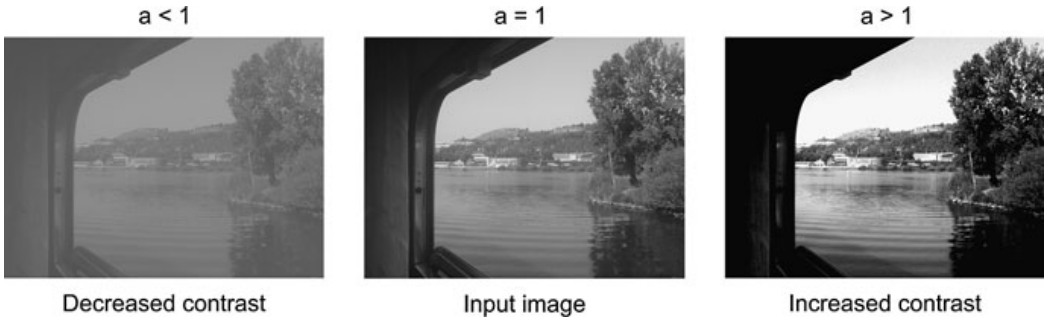


Fig. 4.4 If a in Eq. 4.2 is one, the resulting image will be equal to the input image. If a is smaller than one then the resulting image will have decreased contrast, and if a is higher than one then the resulting image will have increased contrast

them and we will say the contrast is *high*. The contrast of an image is changed by changing the slope of the graph[1]:

[1] In practice the line is not rotated around $(0, 0)$ but rather around the center point $(127, 127)$, hence $b = 127(1 - a)$. However, for the discussion here it suffice to say that $b = 0$ and only look at the slope.

$$g(x, y) = a \cdot f(x, y) \tag{4.2}$$

If $a > 1$ the contrast is increased and if $a < 1$ the contrast is decreased. For example when $a = 2$ the pixels 112 and 114 will get the values 224 and 228, respectively. The difference between them is increased by a factor 2 and the contrast is therefore increased. In Fig. 4.4 the effect of changing the contrast can be seen.

If we combine the equations for brightness, Eq. 4.1, and contrast, Eq. 4.2, we have

$$g(x, y) = a \cdot f(x, y) + b \tag{4.3}$$

which is the equation of a straight line. Let us look at an example of how to apply this equation. Say we are interested in a certain part of the input image where the contrast might not be sufficient. We therefore find the range of the pixels in this part of the image and map them to the entire range, $[0, 255]$ in the output image. Say that the minimum pixel value and maximum pixel values in the input image are 100 and 150, respectively. Changing the contrast then means to say that all pixel value below 100 are set to zero in the output and all pixel values above 150 are set to 255 in the output image. The pixels in the range $[100, 150]$ are then mapped to $[0, 255]$ using Eq. 4.3 where a and b are defined as follows:

$$a = \frac{255}{f_2 - f_1}, \qquad b = -a \cdot f_1 \tag{4.4}$$

where $f_1 = 100$ and $f_2 = 150$.

4.2 Non-linear Gray-Level Mapping

Gray-level mapping is not limited to linear mappings as defined by Eq. 4.3. In fact the designer is free to define the gray-level mapping as she pleases as long as there is one and only one output value for each input value. Often the designer will utilize a well defined equation/graph as opposed to defining a new one. Below three of the most common *non-linear mapping* functions are presented.

4.2.1 Gamma Mapping

In many cameras and display devices (flat panel televisions for example) it is useful to be able to increase or decrease the contrast in the dark gray levels and the light gray levels individually since humans have a non-linear perception of contrast. A commonly used non-linear mapping is gamma mapping, which is defined for

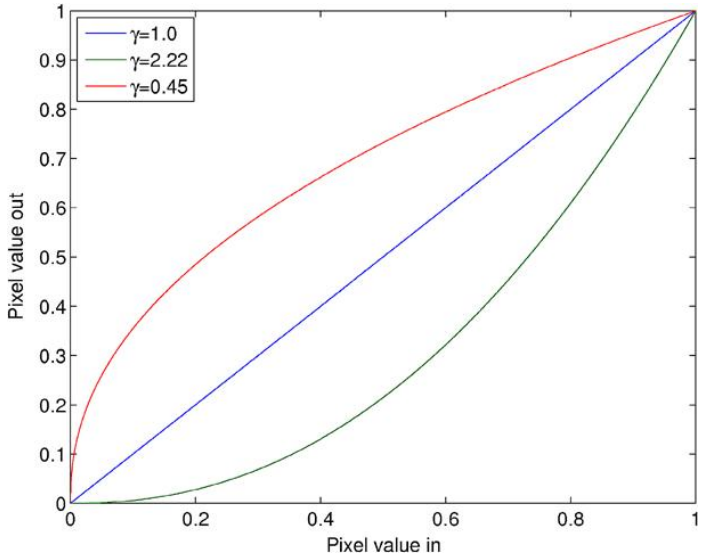


Fig. 4.5 Gamma-mapping curves for different gammas

positive γ as

$$g(x, y) = f(x, y)^{\gamma} \tag{4.5}$$

Some gamma-mapping curves are illustrated in Fig. 4.5. For $\gamma = 1$ we get the identity mapping. For $0 < \gamma < 1$ we increase the dynamics in the dark areas by increasing the mid-levels. For $\gamma > 1$ we increase the dynamics in the bright areas by decreasing the mid-levels. The gamma mapping is defined so that the input and output pixel values are in the range $[0, 1]$. It is therefore necessary to first transform the input pixel values by dividing each pixel value with 255 before the gamma transformation. The output values should also be scaled from $[0, 1]$ to $[0, 255]$ after the gamma transformation.

A concrete example is given. A pixel in a gray-scale image with value $v_{\text{in}} = 120$ is gamma mapped with $\gamma = 2.22$. Initially, the pixel value is transformed into the interval $[0, 1]$ by dividing with 255, $v_1 = 120/255 = 0.4706$. Secondly, the gamma mapping is performed $v_2 = 0.4706^{2.22} = 0.1876$. Finally, it is mapped back to the interval $[0, 255]$ giving the result $v_{\text{out}} = 0.1876 \cdot 255 = 47$. Examples are illustrated in Fig. 4.6.

Fig. 4.6 Gamma mapping to the *left* with $\gamma = 0.45$ and to the *right* with $\gamma = 2.22$. In the *middle* the original image

4.2.2 Logarithmic Mapping

An alternative non-linear mapping is based on the logarithm operator. Each pixel is replaced by the logarithm of the pixel value. This has the effect that low intensity pixel values are enhanced. It is often used in cases where the dynamic range of the image is too great to be displayed or in images where there are a few very bright spots on a darker background. Since the logarithm is not defined for 0, the mapping is defined as

$$g(x, y) = c \cdot \log\bigl(1 + f(x, y)\bigr) \tag{4.6}$$

where c is a scaling constant that ensures that the maximum output value is 255. It is calculated as

$$c = \frac{255}{\log(1 + v_{\max})} \tag{4.7}$$

where $v_{\max}$ is the maximum pixel value in the input image.

The behavior of the logarithmic mapping can be controlled by changing the pixel values of the input image using a linear mapping before the logarithmic mapping. The logarithmic mapping from the interval $[0, 255]$ to $[0, 255]$ is seen in Fig. 4.7. This mapping will clearly stretch the low intensity pixels while suppressing the contrast in high intensity pixels. An example is illustrated in Fig. 4.7.

4.2.3 Exponential Mapping

The exponential mapping uses a part of the exponential curve. It can be expressed as

$$g(x, y) = c \cdot \left(k^{f(x,y)} - 1\right) \tag{4.8}$$

where k is a parameter that can be used to change of shape of the transformation curve and c is a scaling constant that ensures that the maximum output value is 255. It is calculated as

$$c = \frac{255}{k^{v_{\max}} - 1} \tag{4.9}$$

where $v_{\max}$ is the maximum pixel value in the input image. k is normally chosen as a number just above 1. This will enhance details in the bright areas while decreasing detail in the dark areas. An example is illustrated in Fig. 4.7.

Please note that both linear and non-linear gray-level mapping can also be applied to color images. This is simply done by performing gray-level mapping on each of the three color channels.

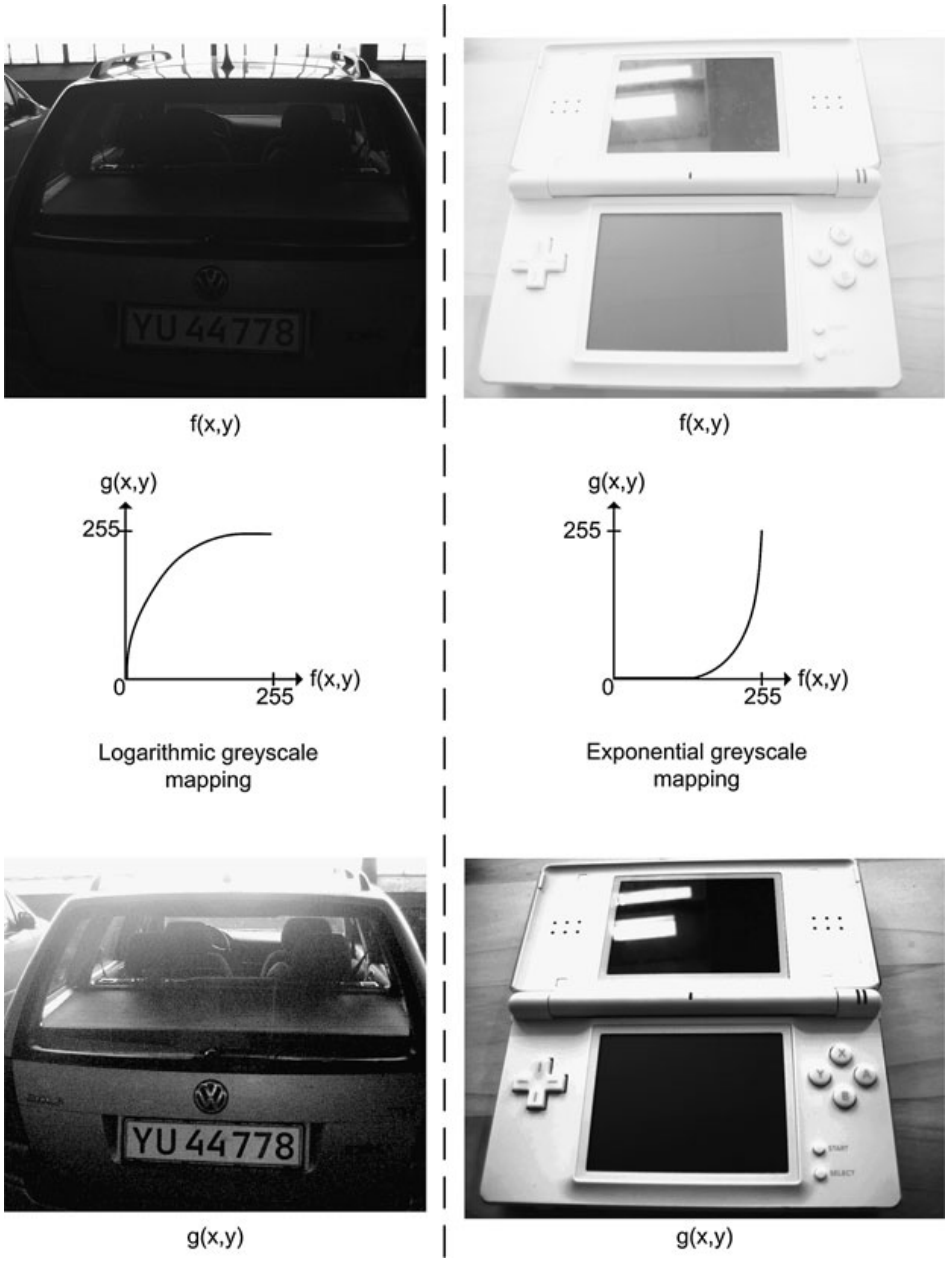


Fig. 4.7 Examples of logarithmic and exponential gray-level mappings. Logarithmic mapping is useful for bringing out details in dark images and exponential mapping is useful for bringing out details in bright images

4.3 The Image Histogram

So now we know how to correct images using gray-level mapping, but how can we tell if an image is too dark or too bright?

The obvious answer is that we can simply look at the image. But we would like a more objective way of answering this question. Moreover, we are also interested in a method enabling a computer to automatically assess whether an image is too

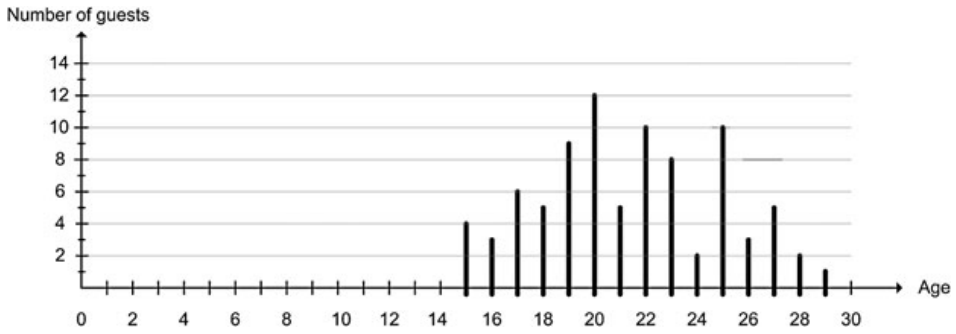


Fig. 4.8 A histogram showing the age distribution of the guests at a party. The horizontal axis represents age and the vertical axis represents the number of guests

dark, too bright or has too low a contrast, and automatically correct the image using gray-level mapping. To this end we introduce a simple but powerful tool namely *the image histogram*. Everybody processing images should always look at the histogram of an image before processing it—and so should you!

A histogram is a graphical representation of the frequency of events. Say you are at a party together with 85 other guests. You could then ask the age of each person and plot the result in a histogram, as illustrated in Fig. 4.8. The horizontal axis represents the possible ages and the vertical axis represents the number of people having a certain age. Each column is denoted a *bin* and the height of a bin corresponds to the number of guests having this particular age. This plot is the histogram of the age distribution among the guests at the party. If you divide each bin with the total number of samples (number of guests) each bin now represents the fraction of guests having a certain age—multiply by 100% and you have the numbers in percentages. We can for example see that 11.6% of the guests are 25 years old. In the rest of this book we will denote the vertical axis in a histogram by *frequency*, i.e., the number of samples.

We now do exactly the same for the pixel values of an image. That is, we go through the entire image pixel-by-pixel and count how many pixels have the value 0, how many have the value 1, and so on up to 255. This results in a histogram with 256 bins and this is the image histogram.

If the majority of the pixels in an image have low values we will see this as most high bins being to the left in the histogram and can thus conclude that the image is dark. If most high bins are to the right in the histogram, the image will be bright. If the bins are spread out equally, the image will have a good contrast and vice versa. See Fig. 4.9.

Note that when calculating an image histogram the actual position of the pixels is not used. This means i) that many images have the same histogram and ii) that an image cannot be reconstructed from the histogram. In Fig. 4.10 four images with the same histogram are shown.

We can of course also calculate the histogram of a color image. This is done separately for each color channel. An example is shown in Fig. 4.11.

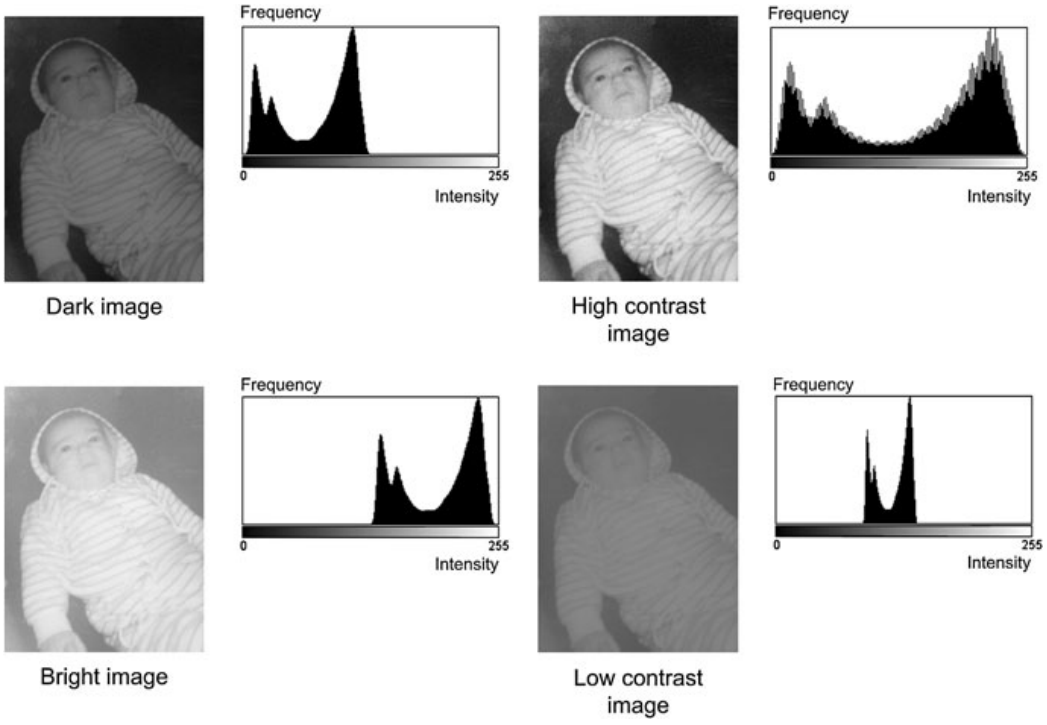


Fig. 4.9 Four images and their respective histograms

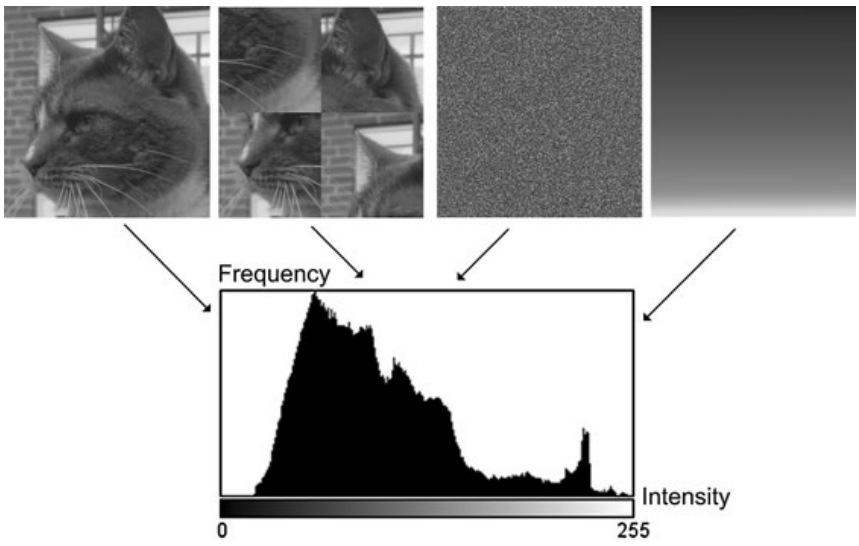


Fig. 4.10 Four images with the exact same histogram

4.3.1 Histogram Stretching

Armed with this new tool we now seek a method to automatically correct the image so that it is neither too bright nor too dark and does not have too low contrast. In terms of histograms, this means that the histogram should start at 0 and end at 255.

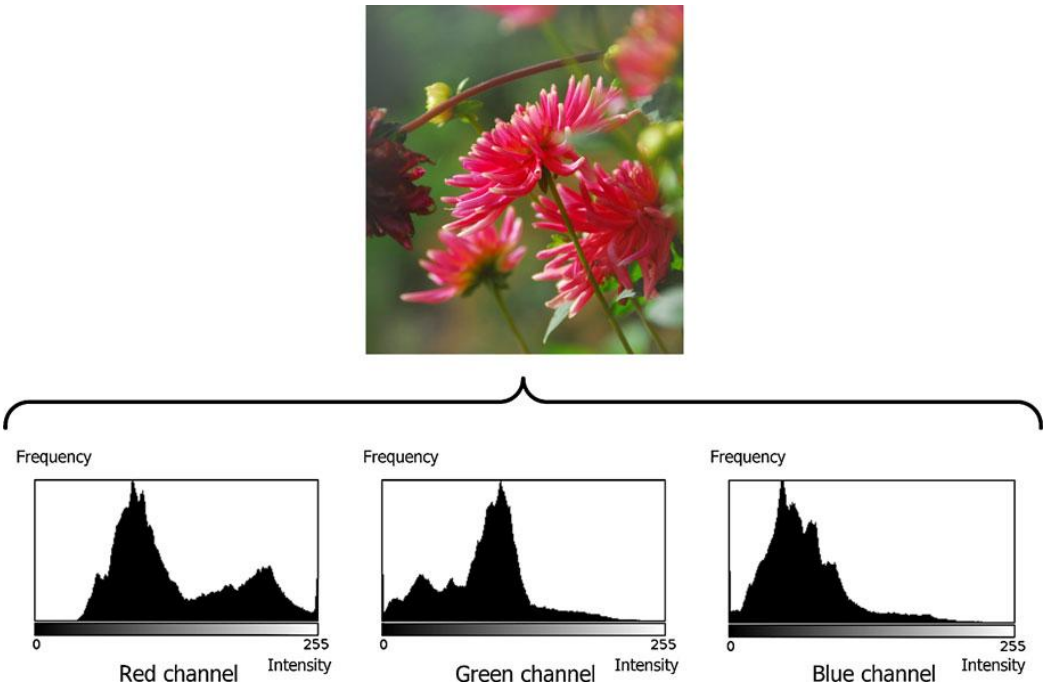


Fig. 4.11 The histograms of a color image

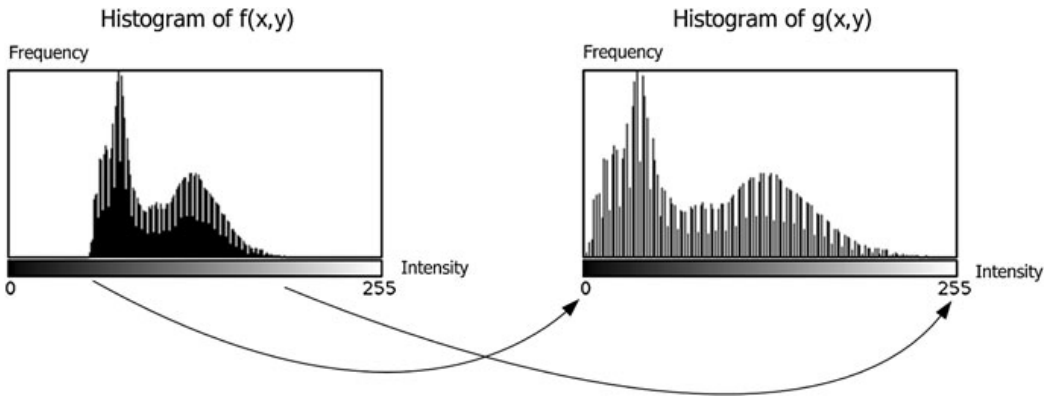


Fig. 4.12 The concept of histogram stretching

We obtain this by mapping the left-most non-zero bin in the histogram to 0 and the right-most non-zero bin to 255, see Fig. 4.12.

We can see that the histogram has been stretched so that the very dark and very bright values are now used. It should also be noted that the distance between the different bins is increased, hence the contrast is improved. This operation is denoted *histogram stretching* and the algorithm is exactly the same as Eq. 4.3 with a and b defined as in Eq. 4.4. f_1 is the left-most non-zero bin in the histogram and f_2 is the right-most non-zero bin in the histogram of the input image.

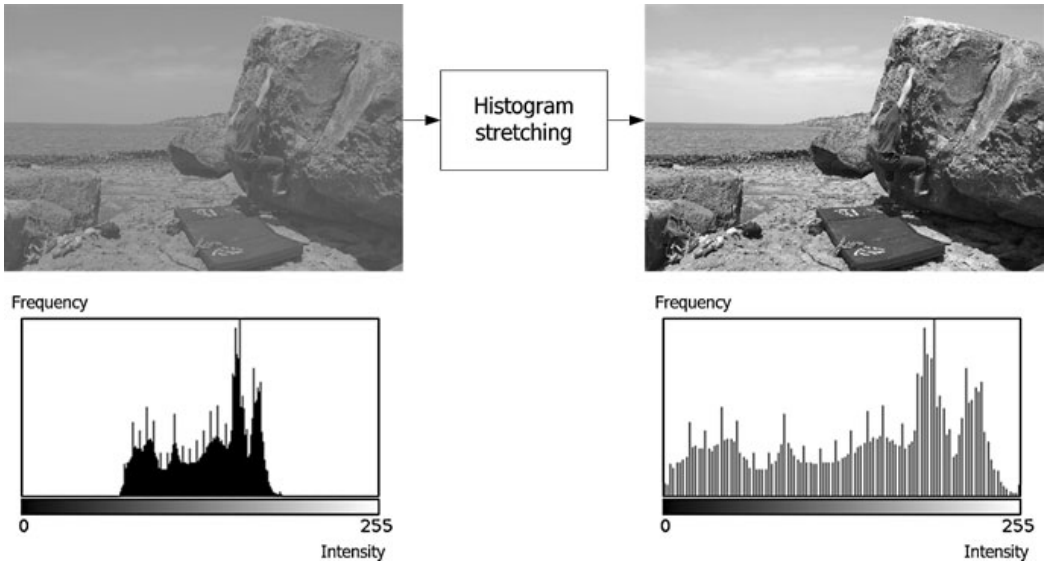


Fig. 4.13 An example of histogram stretching

Conceptually it might be easier to appreciate the equation if we rearrange Eq. 4.3:

$$g(x,y) = \frac{255}{f_2 - f_1} \cdot f(x,y) - f_1 \cdot a \quad \Leftrightarrow \tag{4.10}$$

$$g(x,y) = \frac{255}{f_2 - f_1} \cdot \big(f(x,y) - f_1\big) \tag{4.11}$$

First the histogram is shifted left so that f_1 is located at 0. Then each value is multiplied by a factor a so that the maximum value $f_2 - f_1$ becomes equal to 255. In Fig. 4.13 an example of histogram stretching is illustrated.

If just one pixel has the value 0 and another 255, histogram stretching will not work, since $f_2 - f_1 = 255$. A solution is *modified histogram stretching* where small bins in the histogram are removed by changing their values to those of larger bins. But if a significant number of pixels with very small and very high values exit, we still have $f_2 - f_1 = 255$, and hence the histogram (and image) remains the same, see Fig. 4.15. A more robust method to improve the histogram (and image) is therefore to apply *histogram equalization*.

4.3.2 Histogram Equalization

Histogram equalization is based on non-linear gray-level mapping using a *cumulative histogram*.

Table 4.1 A small histogram and its cumulative histogram. i is the bin number, $H[i]$ the height of bin i, and $C[i]$ is the height of the ith bin in the cumulative histogram

i	0	1	2	3
$H[i]$	1	5	0	7
$C[i]$	1	6	6	13

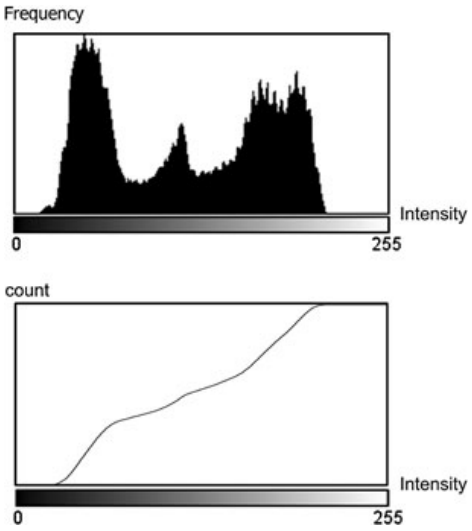


Fig. 4.14 An example of a cumulative histogram. Notice how the tall bins in the ordinary histogram translate into steep slopes in the cumulative histogram

Imagine we have a histogram $H[i]$ where i is a bin number (between 0 and 255) and $H[i]$ is the height of bin i. The cumulative histogram is then defined as

$$C[j] = \sum_{i=0}^{j} H[i] \tag{4.12}$$

In Table 4.1 a small example is provided.

In Fig. 4.14 a histogram is shown together with its cumulative histogram. Where the histogram has high bins, the cumulative histogram has a steep slope and where the histogram has low bins, the cumulative histogram has a small slope. The idea is now to use the cumulative histogram as a gray-level mapping. So the pixel values located in areas of the histogram where the bins are high and dense will be mapping to a wider interval in the output since the slope is above 1. On the other hand, the regions in the histogram where the bins are small and far apart will be mapped to a smaller interval since the slope of the gray-level mapping is below 1.

For this to work in practice we need to ensure that the y-axis of the cumulative histogram is in the range [0, 255]. This is simply done by first dividing each value on the y-axis with *count*, i.e., the total number of pixels in the image, and then multiply with 255. In Fig. 4.15 the effect of histogram equalization is illustrated.

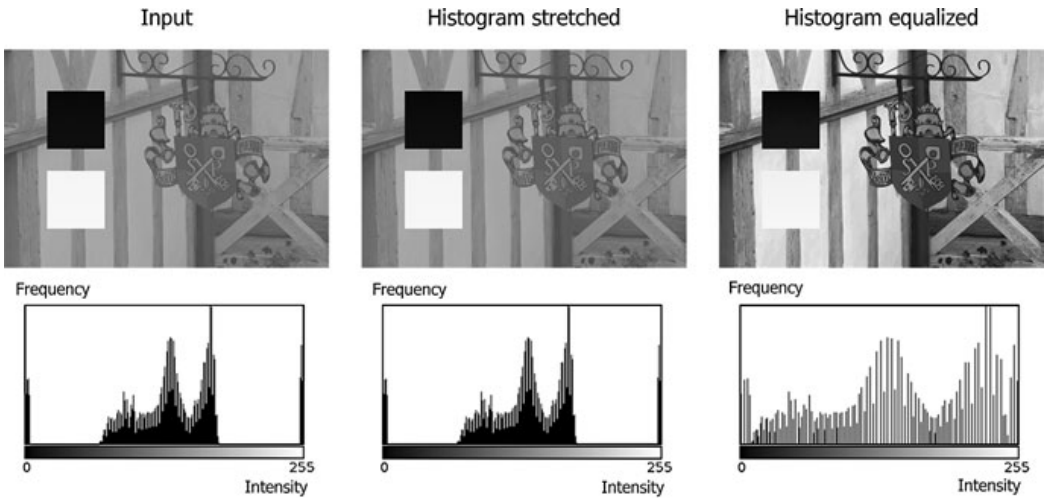


Fig. 4.15 The effect of histogram stretching and histogram equalization on an input image with both very high and very low pixel values

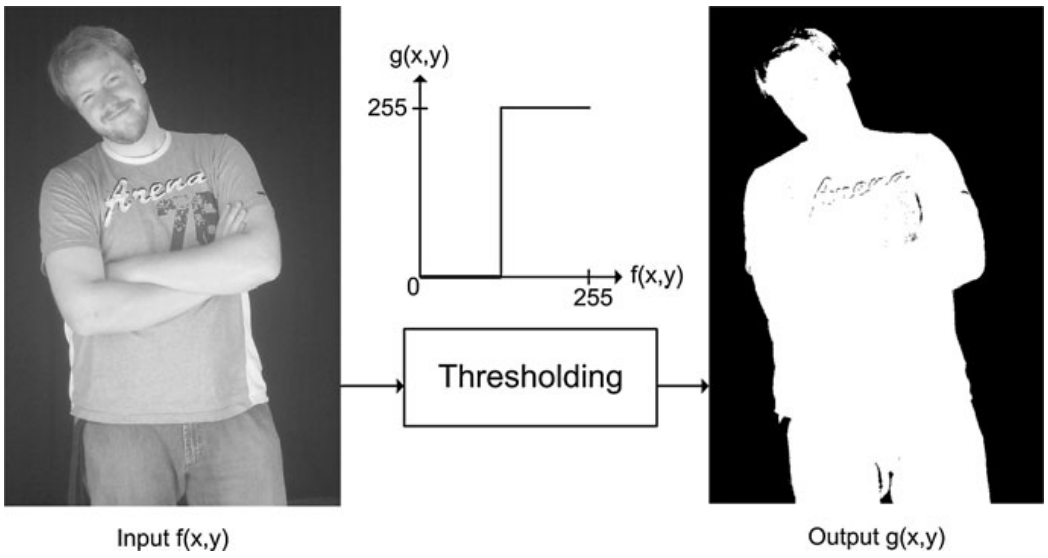


Fig. 4.16 An example of thresholding. Notice that it is impossible to define a perfect silhouette with the thresholding algorithm. This is in general the case

4.4 Thresholding

One of the most fundamental point processing operations is *thresholding*. Thresholding is the special case when $f_1 = f_2$ in Eq. 4.11. Mathematically this is undefined, but in practice it simply means that all input values below f_1 are mapped to zero in the output and all input values above f_1 are mapped to 255 in the output. This means that we will only have completely black and completely white pixel values in the output image. Such an image is denoted a *binary image*, see Fig. 4.16, and this representation of an object is denoted the *silhouette* of the object.

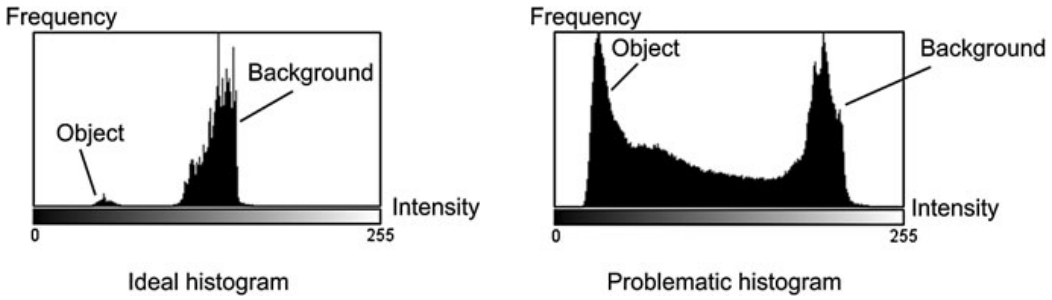


Fig. 4.17 Ideal histogram: a clear definition of object and background. Problematic histogram: the distinction between the object and the background is harder, if not impossible

One might argue that we loose information when doing this operation. However, imagine you are designing a system where the goal is to find the position of a person in a sequence of images and use that to control some parameter in a game. In such a situation all you are interested in is the position of the person and nothing more. In this case, thresholding in such a manner that the person is white and the rest is black, would be exactly what we are interested in. In fact, we can say we have removed the redundant information or eliminated noise in the image.

Thresholding is normally not described in terms of gray-level mapping, but rather as the following *segmentation* algorithm:

$$\begin{aligned} &\text{if } f(x,y) \leq T \quad \text{then } g(x,y) = 0 \\ &\text{if } f(x,y) > T \quad \text{then } g(x,y) = 255 \end{aligned} \tag{4.13}$$

where T is the threshold value. We might of course also reverse the equalities so that every pixel below the threshold value is mapped to white and every pixel above the threshold value is mapped to black.

In many image processing systems, thresholding is a key step to segmenting the foreground (information) from the background (noise). To obtain a good thresholding the image is preferred to have a histogram which is bi-modal. This means that the histogram should consist of two "mountains" where one mountain corresponds to the background pixels and the other mountain to the foreground pixels. Such a histogram is illustrated to the left in Fig. 4.17. In an ideal situation like the one shown to the right, deciding the threshold value is not critical, but in real life the two mountains are not always separated so nicely and care must therefore be taken when defining the correct threshold value.

In situations where you have influence on the image acquisition process, keep this histogram in mind. In fact, one of the sole purposes of image acquisition is often to achieve such a histogram. So it is often beneficial to develop your image processing algorithms and your setup (camera, optics, lighting, environment) in parallel.

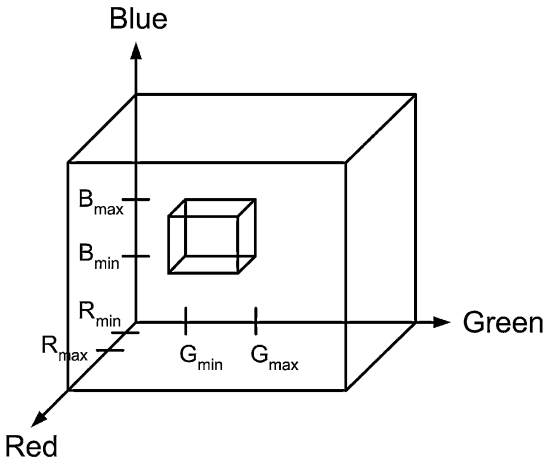


Fig. 4.18 The box is defined by the threshold values. The box indicates the region within the RGB color cube where object pixels lie

4.4.1 Color Thresholding

Color thresholding can be a powerful approach to segmenting objects in a scene. Imagine you want to detect the hands of a human for controlling some interface. This can be done in a number of ways, where the easiest might be to ask the user to wear colored gloves. If this is combined with the restriction that the particular color of the gloves is neither present in the background nor on the rest of the user, then by finding all pixels with the color of the gloves we have found the hands. This operates similarly to the thresholding operation described in Eq. 4.13. The difference is that each of the color values of a pixel is compared to two threshold values, i.e., in total six threshold values. If each color value for a pixel is within the threshold values, then the pixel is set to white (foreground pixel) otherwise black (background pixel). The algorithm looks as follows for each pixel:

$$
\begin{aligned}
&\textbf{If} \\
&\quad R > R_{\min} \quad \text{and} \quad R < R_{\max} \quad \text{and} \\
&\quad G > G_{\min} \quad \text{and} \quad G < G_{\max} \quad \text{and} \\
&\quad B > B_{\min} \quad \text{and} \quad B < B_{\max} \\
&\textbf{Then} \quad g(x, y) = 255 \\
&\textbf{Else} \quad g(x, y) = 0
\end{aligned}
\tag{4.14}
$$

where (R, G, B) are the RGB values of the pixel being processed and $R_{\min}$ and $R_{\max}$ define the range of acceptable values of red in order to accept the current pixel as belonging to an object of interest (similarly for green and blue).

The algorithm actually corresponds to defining a box in the RGB color space and classifying a pixel as belonging to an object if it is within the box and otherwise classifying it as background. This is illustrated in Fig. 4.18.

One problem with color thresholding is its sensitivity to changes in the illumination. Say you have defined your threshold values so that the system can detect the two gloved hands. If someone increases the amount of light in the room, the color

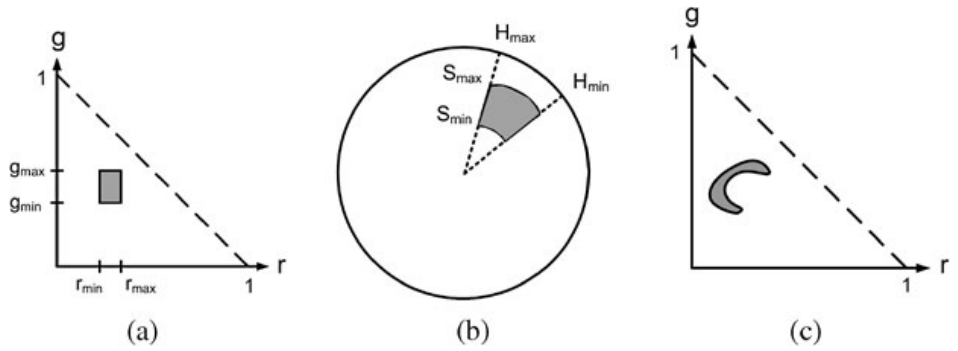


Fig. 4.19 The two gray shapes in figure (**a**) and (**b**) are defined by threshold values and indicate the regions within the two color spaces where object pixels lie. (**a**) The rg-color space. (**b**) The hs-color space. (**c**) An example of a shape that is not well defined by threshold values. Instead a LUT should be applied

will stay the same, but the intensity will change. To handle such a situation, you need to increase/decrease the threshold values accordingly. This will result in the box in Fig. 4.18 being larger and hence the risk of including non-glove pixels will increase. In the worst case, the box will be as large as the entire RGB color cube.

The solution is to convert the RGB color image into a representation where the color and intensity are separated, and then do color thresholding on only the colors, e.g., rg-values or hs-values. The thresholds can now be more tight, hence reducing the risk of false classification. In Fig. 4.19 the equivalent of Fig. 4.18 is shown for rg- and hs-representations, respectively. Regardless of which color representation is applied, the problem of choosing proper threshold values is the same. Please consult Appendix C regarding this matter.

Sometimes we can find ourselves in a situation where the colors of an object are not easily described by a few threshold values. In Fig. 4.19(c) this is illustrated by the banana-shaped region. If you fit a box to this shape (by using four thresholds values) you will clearly include non-object pixels and hence have an incorrect segmentation of the object. The solution is to define a *look-up-table* (LUT). A LUT is a table containing the color values belonging to the object of interest (in some color space). These values can be found in a training phase by manually inspecting the object of interest in a number of different images. Normally the values are considered as an image and a morphologic closing operation, see Chap. 6, is performed to obtain a smooth and coherent shape. During run-time Eq. 4.14 is replaced by a function that takes the value of a pixel and test if this value is present in the LUT. If not, the corresponding output pixel is set to black, otherwise it is set to white.

No matter which color space you use for thresholding it is often a good idea to also do some thresholding on the intensity values. If you look at the color cube you can see that all possible colors will have a vector starting in $(0, 0, 0)$. This means that the vectors will lie in the vicinity of $(0, 0, 0)$ and the practical meaning of this is that it is hard to distinguish colors when the intensity is low. Therefore it is often a good idea not to process the colors of pixels with low intensity values. Likewise, color pixels with a very high intensity might also be problematic to process. Say we

have a pixel with the following RGB values (255, 250, 250). This will be interpreted as very close to white and hence containing no color. But it might be that the real values are (10000, 250, 250). You have no way of knowing, since the red value is saturated in the image acquisition process. So the red pixel is incorrectly classified as (close to) white. In general you should try to avoid saturated pixels in the image acquisition process, but when you do encounter them, please take great care before using the color of such a pixel. In fact, you are virtually always better off ignoring such pixels.

4.4.2 Thresholding in Video

When you need to threshold a single image you can simply try all possible threshold values and see which one provides the best result. When you built a system that operates on live input video the situation is different. Imagine you have constructed a setup with a camera and some lighting etc. You connect a monitor and look at the images being captured by the camera. If nothing is happening in the images (static scene) the images will seem to be exactly the same. But they are not. For example, if the camera is mounted on a table which moves slightly whenever someone is walking nearby, the images will change slightly. Another typical situation is the fact that most indoor lighting is powered by an alternating light source, for example 50 Hz, meaning that the level of illumination changes rapidly over time. Such changes can often not be detected by simply looking at the scene. But if you subtract two consecutive images[2] and display the result, you can experience this phenomena. If the images are in fact exactly the same, then the output image (after image subtraction) should only contain zeros, hence be black. The more non-zero pixels you have in the output image the more "noise" is present in your setup. Another way of illustrating such small changes is to calculate and visualize the histogram for each image. No matter what, it is always a good idea to use one of these methods to judge the uncertainties in your image acquisition/setup.

Due to these uncertainties you always need to *learn* the threshold values when processing video. In this context, learning means to evaluate what the right threshold value is in different situations and then select a representative value, see Appendix C. Approaching the threshold value selection like this will help in many situation. But if you have a scenario where the lighting can change significantly, then you need a different approach.

A significant change is especially observed when sunlight enters the scene, either because the system operates outside or due to windows in the room where the setup is located. When a cloud passes in front of the sun an abrupt change can be seen in the images. Even without clouds, the changing position (and intensity) of the sun during the day can also result in large changes accumulating over time. Further abrupt changes appear due to the auto gain being enabled, see

[2] How to subtract images is explained in Sect. 4.6. This technique plays a major role in Chap. 8.

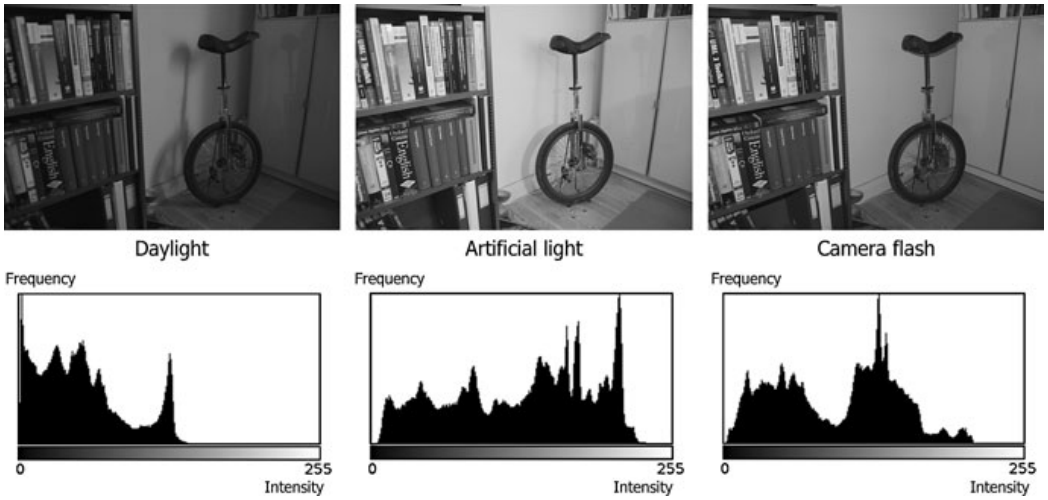


Fig. 4.20 Three images of the same scene with different illuminations and hence different histograms

Chap. 2. Imagine a white object is entering a scene where the background is dark. As more and more of the object becomes visible in the scene the auto gain function will decrease the brightness accordingly in order to keep the overall brightness constant. This means that the threshold value needs to be changed from image to image and often rather significantly. Such significant changes can sometimes be handled by preforming a histogram stretching/equalization. This only works when the changes result in a shifted histogram (making the image brighter or darker) without changing the structure of the histogram. An example of a changed structure is when light from multiple windows illuminate the objects in the scene differently over time. In Fig. 4.20 examples of different illuminations of the same scene are shown.

Automatic Thresholding: Global Method

As mentioned above, thresholding is based on the notion that an image consists of two groups of pixels; those from the object of interest (foreground) and those from the background. In the histogram these two groups of pixels result in two "mountains" denoted modes. We want to select a threshold value somewhere between these two modes. Automatic methods for doing so exist and they are based on analyzing the histogram, i.e. all pixels are involved, and hence denoted a *global* method. The idea is to try all possible threshold values and for each, evaluate if we have two good modes. The threshold value producing the best modes is selected. Different definitions of "good modes" exits and here we describe the one suggested by Otsu [14].

The method evaluates Eq. 4.15 for each possible threshold value T and select the T where $C(T)$ is minimum. The reasoning behind the equation is that the correct threshold value will produce two narrow modes, whereas an incorrect threshold value will produce (at least one) wide mode. The narrowness of a mode can be measured by the variance σ^2, see Appendix C for a definition of σ^2. So the smaller the variances the better. To balance the measure, each variance is weighted by the

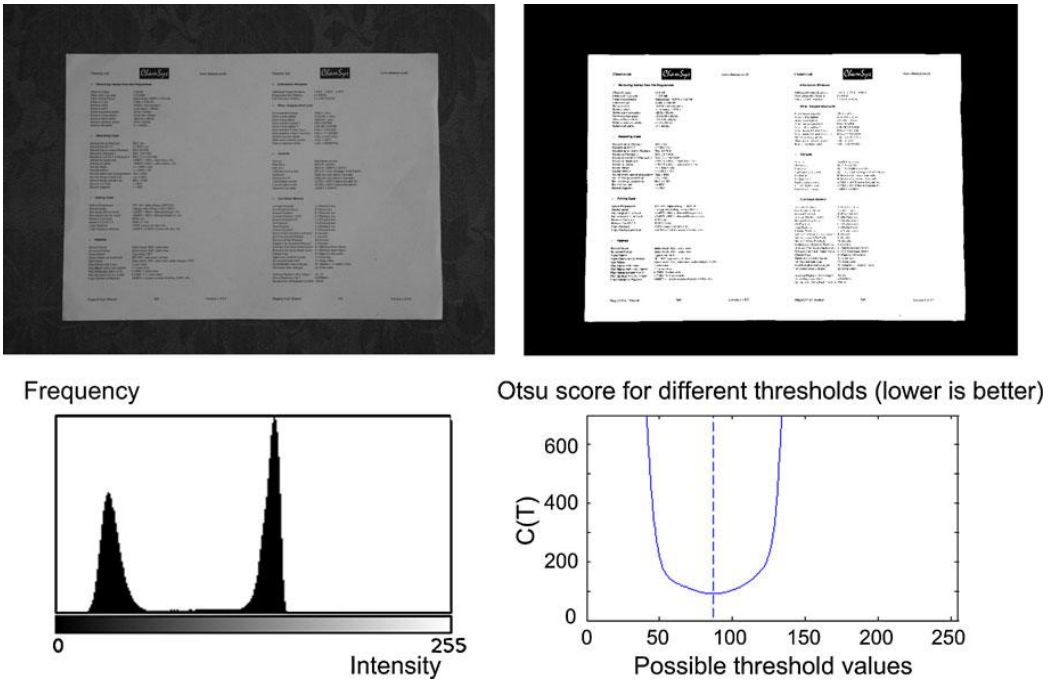


Fig. 4.21 Global automatic thresholding. *Top left*: Input image. *Top right*: Input image thresholded by the value found by Otsu's method. *Bottom left*: Histogram of input image. *Bottom right*: $C(T)$ as a function of T. See text. The *vertical dashed line* illustrates the minimum value, i.e., the selected threshold value

number of pixels used to calculate it. A very efficient implementation is described in [14]. The method works very well in situations where two distinct modes are present in the histogram, see Fig. 4.21, but it can also produce good results when the two modes are not so obvious.

$$C(T) = M_1(T) \cdot \sigma_1^2(T) + M_2(T) \cdot \sigma_2^2(T) \tag{4.15}$$

where $M_1(T)$ is the number of pixels to the left of T and $M_2(T)$ is the rest of the pixels in the image. $\sigma_1^2(T)$ and $\sigma_2^2(T)$ are the variances of the pixels to the left and right of T, respectively.

Automatic Thresholding: Local Method

In Fig. 4.23 an image with non-even illuminating is shown. The consequence of this type of illumination is that an object pixel in one part of the image is identical to a background pixel in another part of the image. The image can therefore not be thresholded using a single (global) threshold value, see Fig. 4.23. But if we crop out a small area of the image and look at the histogram, we can see that two modes are present and that this image can easily be thresholded, see Fig. 4.22. From this follows that thresholding is possible locally, but not globally.

We can view thresholding as a matter of finding object pixels and these are per definition different from background pixels. So if we had an image of the background, we could then subtract it from the input image and the object pix-

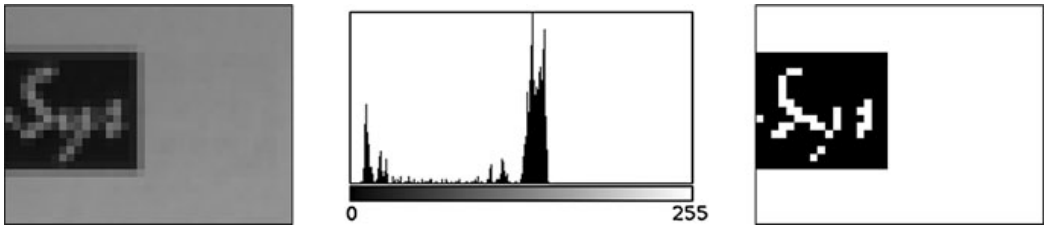


Fig. 4.22 *Left*: Cropped image. *Center*: Histogram of input. *Right*: Thresholded image

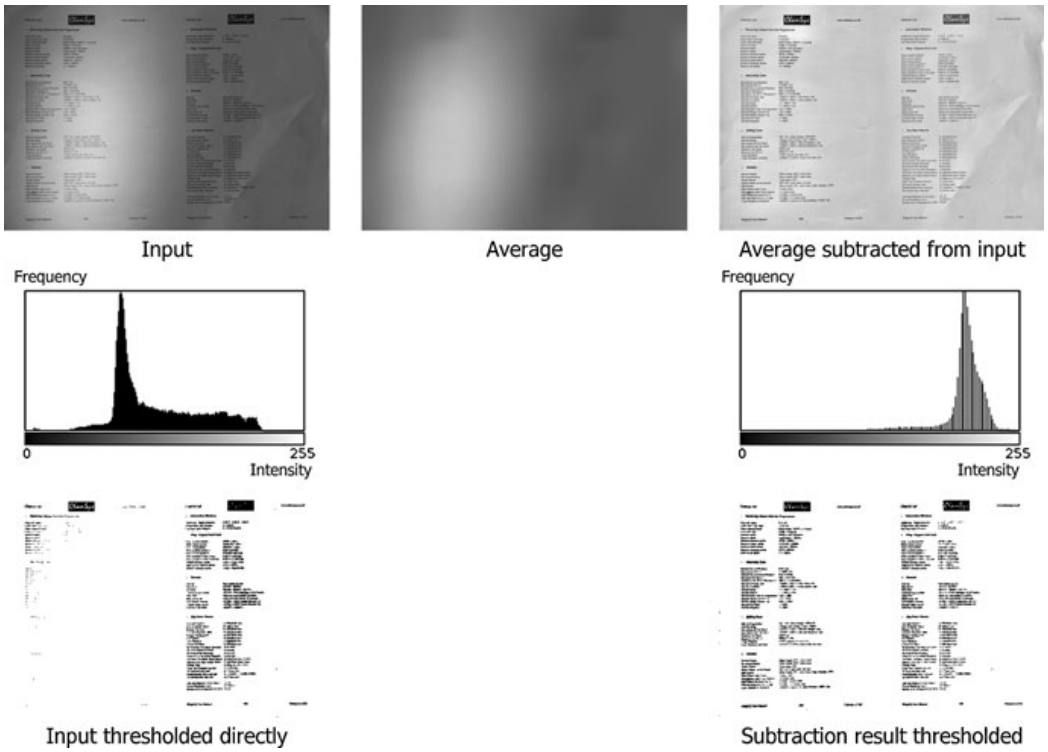


Fig. 4.23 Local automatic thresholding. *Top row*: *Left*: Input image. *Center*: Mean version of input image. *Right*: Mean image subtracted from input. *Center row*: Histograms of input and mean image subtracted from input. *Bottom row*: Thresholded images

els would stand out. We can estimate a background pixel by calculating the average of the neighboring pixels.[3] Doing this for all pixels will result in an estimate of the background image, see Fig. 4.23. We now subtract the input and the background image and the result is an image with a more even illumination where a global threshold value can be applied, see Fig. 4.23.[4] Depending on the situation this could either be a fixed threshold value or an automatic value as describe above.

[3]How to calculate the average is discussed in the next chapter.

[4]In the subtraction process both positive and negative values can appear. Since we are only interested in the difference we take the absolute value.

The number of neighborhood pixels to include in the calculation of the average image depends on the nature of the uneven illumination, but in general it should be a very high number. The method assumes the foreground objects of interest are small compared to the background. The more this assumption is violated, the worse the method performs.

4.5 Logic Operations on Binary Images

After thresholding we have a binary image consisting of only white pixels (255) and black pixels (0). We can combine two binary images using logic operations. The basic logic operations are NOT, AND, OR, and XOR (exclusive OR). The NOT operation do not combine two images but only works on one at a time. NOT simply means to invert the binary image. That is, if a pixel has the value 0 in the input it will have the value 255 in the output, and if the input is 255 the output will be 0. The three other basic logic operations combine two images into one output. Their operations are described using a so-called *truth table*. Below the three truth tables are listed.

(a) Truth table for AND

AND		Input 2	
		0	255
Input 1	0	0	0
	255	0	255

(b) Truth table for OR

OR		Input 2	
		0	255
Input 1	0	0	255
	255	255	255

(c) Truth table for XOR

XOR		Input 2	
		0	255
Input 1	0	0	255
	255	255	0

A truth table is interpreted in the following way. The left-most column contains the possible values a pixel in image 1 can have. The topmost row contains the possible values a pixel in image 2 can have. The four remaining values are the output values. From the truth tables we can for example see that 255 AND $0 = 0$, and 0 OR $255 = 255$. In Fig. 4.24 a few other examples are shown. Note that from a programming point of view white can be represented by 1 and only one byte is then required to represent each pixel. This can save memory and speed up the implementation.

4.6 Image Arithmetic

Instead of combining an image with a scalar as in Eq. 4.1, an image can also be combined with another image. Say we have two images of equal size, $f_1(x, y)$ and $f_2(x, y)$. These are combined pixel-wise in the following way:

$$g(x, y) = f_1(x, y) + f_2(x, y) \tag{4.16}$$

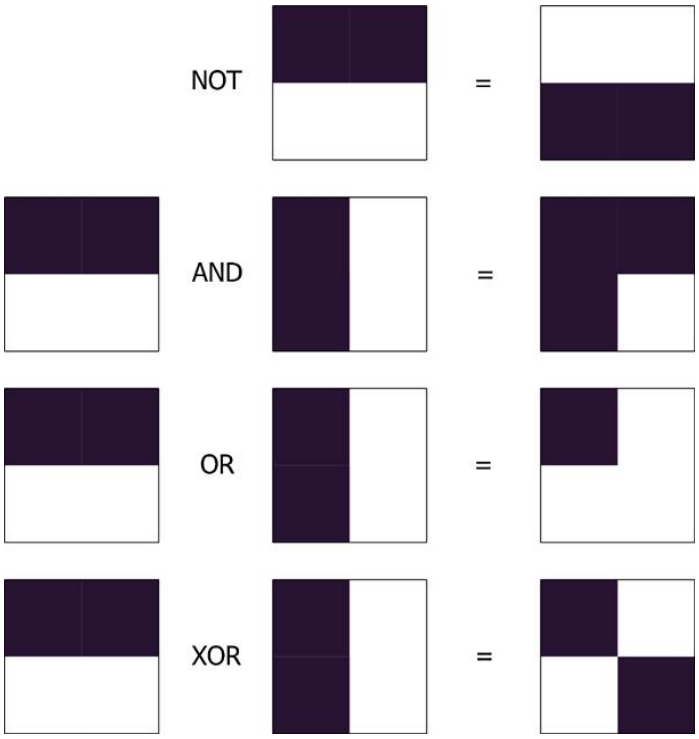


Fig. 4.24 Different logic operations

Other arithmetic operations can also be used to combine two images, but most often addition or subtraction are the ones applied. No matter the operation image arithmetic works equally well for gray-scale and color images.

When adding two images some of the pixel values in the output image might have values above 255. For example if $f_1(10, 10) = 150$ and $f_2(10, 10) = 200$, then $g(10, 10) = 350$. In principle this does not matter, but if an 8-bit image is used for the output image, then we have the problem known as *overflow*. That is, the value cannot be represented. A similar situation can occur for image subtraction where a negative number can appear in the output image. This is known as *underflow*.

One might argue that we could simply use a 16 or 32-bit image to avoid these problems. However, using more bit per pixel will take up more space in the computer memory and require more processing power from the CPU. When dealing with many images, e.g., video data, this can be a problem.

The solution is therefore to use a temporary image (16-bit or 32-bit) to store the result and then map the temporary image to a standard 8-bit image for further processing. This principle is illustrated in Fig. 4.25.

This algorithm is the same as used for histogram stretching except that the minimum value can be negative:

1. Find the minimum number in the temporary image, f_1
2. Find the maximum number in the temporary image, f_2
3. Shift all pixels so that the minimum value is 0: $g_i(x, y) = g_i(x, y) - f_1$
4. Scale all pixels so that the maximum value is 255: $g(x, y) = g_i(x, y) \cdot \frac{255}{f_2 - f_1}$

where $g_i(x, y)$ is the temporary image.

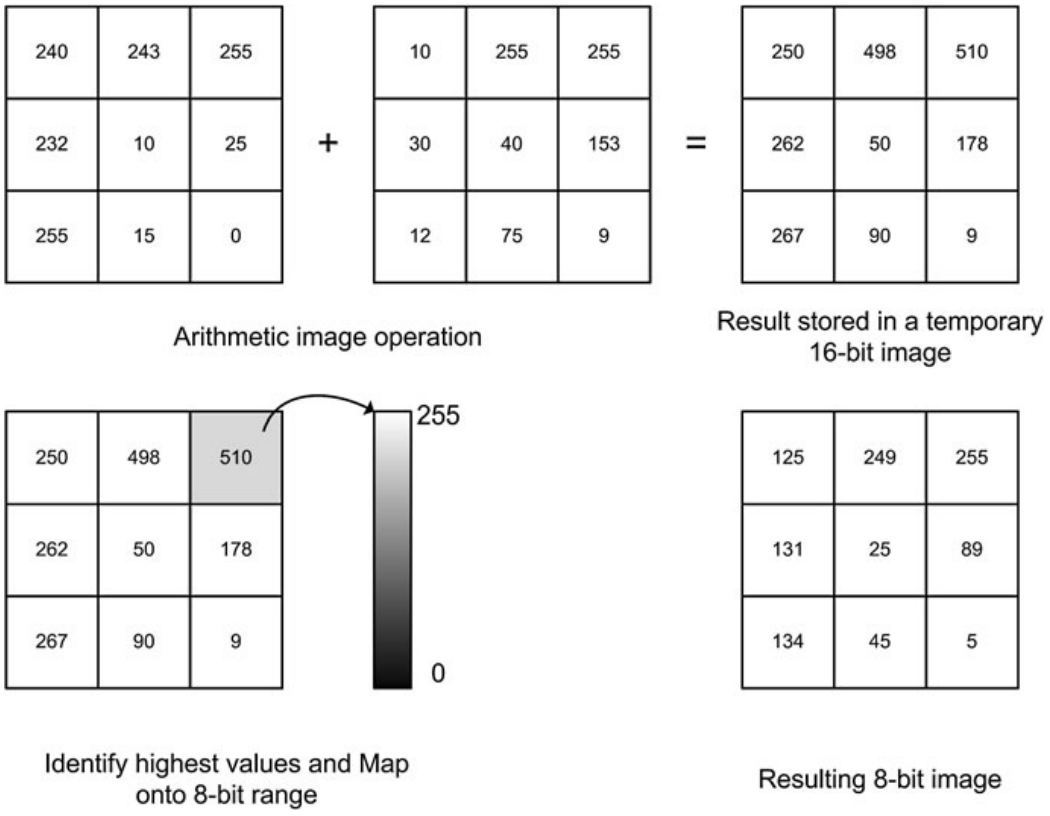


Fig. 4.25 An example of overflow and how to handle it. The addition of the images produces values above the range of the 8-bit image, which is handled by storing the result in a temporary image. In this temporary image the highest value is identified, and used to scale the intensity values down into the 8-bit range. The same approach is used for underflow. This approach also works for images with both over- and underflow

Image arithmetic has a number of interesting usages and here two are presented. In Chap. 8 we present another one, which is related to video processing.

The first one is simply to invert an image. That is, a black pixel in the input becomes a white pixel in the output etc. The equation for image inversion is defined in Eq. 4.17 and an example is illustrated in Fig. 4.26.

$$g(x, y) = 255 - f(x, y) \tag{4.17}$$

Another use of image arithmetic is *alpha blending*. Alpha blending is used when mixing two images, for example gradually changing from one image to another image. The idea is to extend Eq. 4.16 so that the two images have different importance. For example 20% of $f_1(x, y)$ and 80% of $f_2(x, y)$. Note that the sum of the two percentages should be 100%. Concretely, the equation is rewritten as

$$g(x, y) = \alpha \cdot f_1(x, y) + (1 - \alpha) \cdot f_2(x, y) \tag{4.18}$$

Fig. 4.26 Input image and inverted image

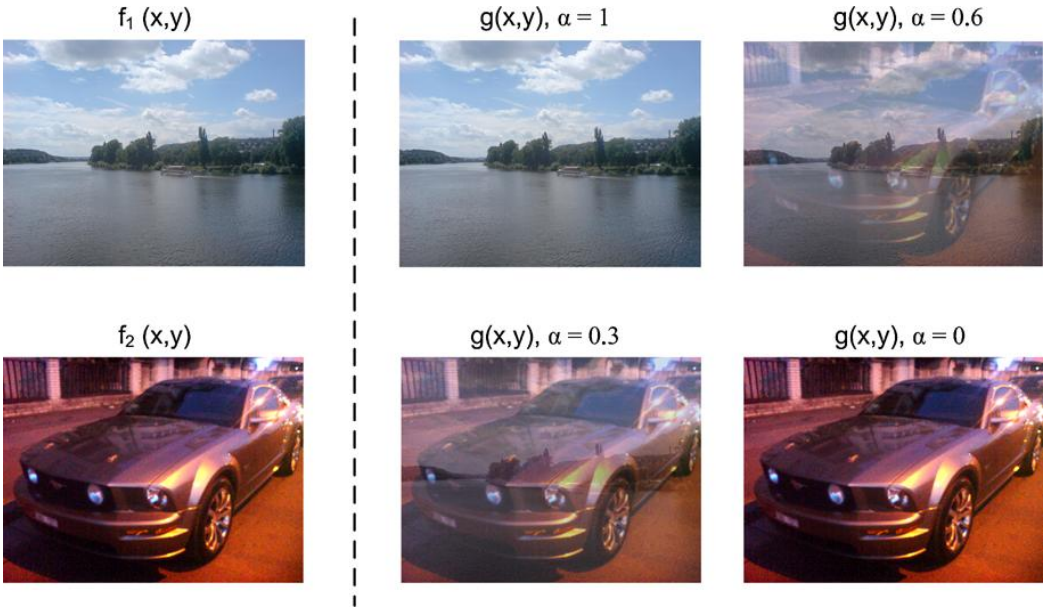


Fig. 4.27 Examples of alpha blending, with different alpha values

where $\alpha \in [0, 1]$ and α is the Greek letter "alpha", hence the name alpha blending. If $\alpha = 0.5$ then the two images are mixed equally and Eq. 4.18 has the same effect as Eq. 4.16. In Fig. 4.27, a mixing of two images is shown for different values of α.

In Eq. 4.18, α is the same for every pixel, but it can actually be different from pixel to pixel. This means that we have an entire image (with the same size as $f_1(x, y)$, $f_2(x, y)$ and $g(x, y)$) where we have α-values instead of pixels: $\alpha(x, y)$. Such an "α-image" is often referred to as an *alpha-channel*. This can for example be used to define the transparency of an object.

4.7 Programming Point Processing Operations

When implementing one of the point processing operations in software the following is done.

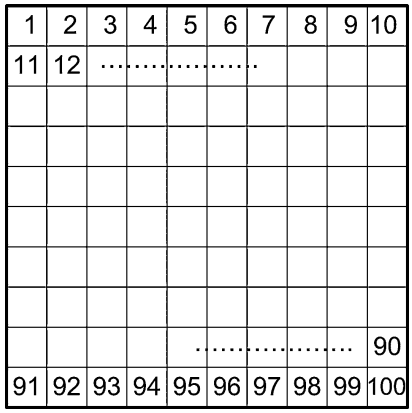


Fig. 4.28 The order in which the pixels are visited. Illustrated for a 10×10 image

Remember that each pixel is individually processed meaning that it does not matter in which order the pixels are processed. However, we follow the order illustrated in Fig. 4.28. Starting in the upper-left corner we move from left to right and from top to bottom ending in the lower-right corner.[5]

Note that this order corresponds to the way the coordinate system is defined, see Fig. 2.19. The reason for this order is that it corresponds to the order in which the pixels from the camera are sent to the memory of the computer. Also the same order the pixels on your TV are updated. Physically the pixels are also stored in this order meaning that your algorithm is faster when you process the pixels in this order due to memory access time.

In terms of programming the point processing operations can be implemented as illustrated below—here exemplified in C-code:

```
for (y = 0; y < M; y = y + 1)
{
  for (x = 0; x < N; x = x + 1)
  {
     temp = GetPixel(input, x, y);
     value = Operation(temp);
     SetPixel(output, x, y, value);
  }
}
```

where M is the height of the image and N is the width of the image. *GetPixel* is a function, which returns the value of the pixel at position (x, y) in the image called *input*. The function *SetPixel* changes the value of the pixel at position (x, y) in the image called *output* to *value*. Note that the two functions are not built-in C-functions. That is, you either need to write them yourself or include a library where they (or similar functions) are defined.

[5]Note that the above order of scanning through the image and the code example is general and used for virtually all methods, operations and algorithms presented in this book.

The programming example primarily consists of two FOR-loops which go through the image, pixel-by-pixel, in the order illustrated in Fig. 4.28. For each pixel, a point processing operation is applied.

Below we show what the C-code would look like if the operation in Eq. 4.3 were implemented.

```
for (y = 0; y < M; y = y + 1)
{
  for (x = 0; x < N; x = x + 1)
  {
     value = a * GetPixel(input, x, y) + b;
     SetPixel(output, x, y, value);
  }
}
```

where a and b are defined beforehand.

Below we show what the C-code would look like if the operation in Eq. 4.13 were implemented.

```
for (y = 0; y < M; y = y + 1)
{
  for (x = 0; x < N; x = x + 1)
  {
     if (GetPixel(input, x, y) > T)
       SetPixel(output, x, y, 255);
     else
       SetPixel(output, x, y, 0);
  }
}
```

where T is defined beforehand.

4.8 Further Information

Thresholding is a key method in many video processing systems. Please remember that there is a direct relationship between your image acquisition process, your setup and your choice of threshold value. If the methods described in this chapter are not sufficient, please bear in mind that other methods for especially automatic thresholding exist.

A very popular use of color thresholding is to segment objects (especially people) by placing them in front of a unique colored background. The object pixels are then found as those pixels in the image which do *not* have this unique color. This principle is denoted *chroma-keying* and used for special effects in many movie productions as well as in TV weather-forecasts, etc. In the latter example the host appears to

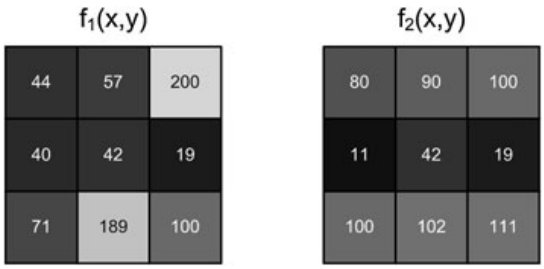


Fig. 4.29 Two gray-scale images

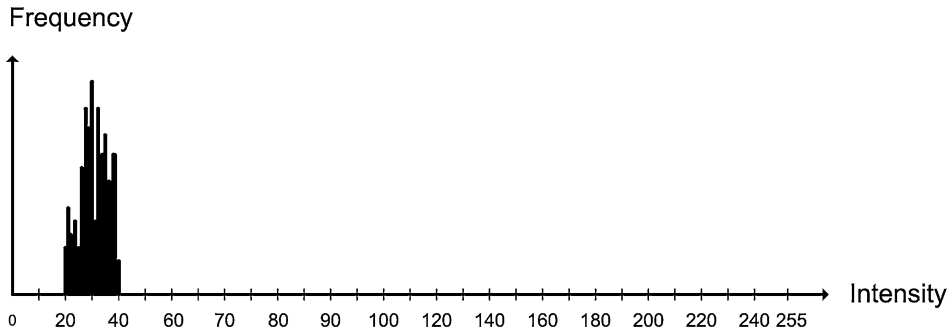


Fig. 4.30 The histogram of an image

be standing in front of a weather map. In reality the host is standing in front of a green or blue screen and the background pixels are then replaced by pixels from the weather map. Obviously, this only works when the color of the host's clothing is different from the unique color used for covering the background.

When you as a designer have the freedom of defining the colors to be recognized you can use the HSI color representation to select the most optimal colors. If you only need one color, then you are free to choose, but when more colors are to be thresholded, optimal basically means to pick colors most different and hence avoid overlap. Looking at the HS circle in Fig. 3.11 you can see that the angle between two colors should be 180° in order to minimize the risk of overlap. With three colors you need to have 120° between the colors etc. Obviously this approach assumes you can construct all possible color, which might not be realistic in a real-life situation.

4.9 Exercises

Exercise 1: Explain the following concepts: point processing, brightness, contrast, gray-level mapping, image histogram, thresholding, logic operations.

Exercise 2: A linear gray-level mapping is performed on image $f_1(x, y)$ in Fig. 4.29 where $a = 1$ and $b = 15$. What is the output value of $f_1(2, 2)$?

Exercise 3: A gamma gray-level mapping is performed on image $f_1(x, y)$ in Fig. 4.29 where $\gamma = 0.45$. What is the output value of $f_1(2, 2)$?

Exercise 4: A logarithmic gray-level mapping is performed on image $f_1(x, y)$ in Fig. 4.29. What is the output value of $f_1(2, 2)$?

Exercise 5: Given a histogram, how can the original gray-scale image be recreated?

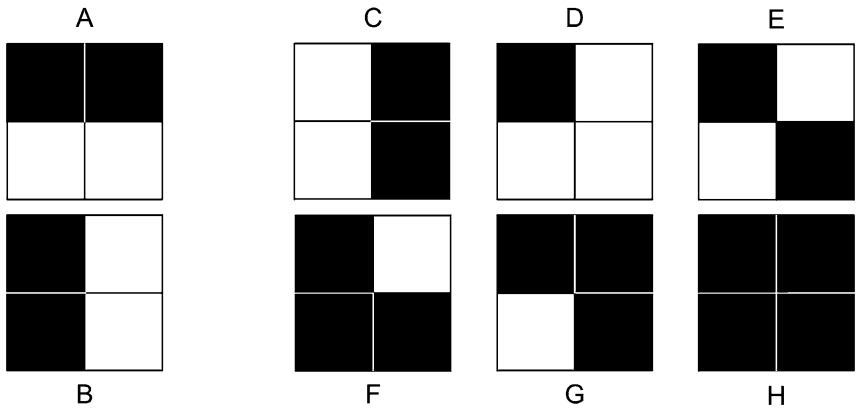


Fig. 4.31 Eight binary images

Exercise 6: Look at the histogram in Fig. 4.30. Does it come from a dark or bright image?

Exercise 7: Look at the histogram in Fig. 4.30. Does it come from an image with high or low contrast?

Exercise 8: Histogram stretching is performed on the histogram in Fig. 4.30. After histogram stretching a pixel has the value 128. What value did this pixel have before histogram stretching?

Exercise 9: Calculate the cumulative histogram of image $f_2(x, y)$ in Fig. 4.29.

Exercise 10: How will the threshold algorithm look like if two threshold values are used instead of just one?

Exercise 11: Explain the two automatic thresholding methods and discuss their differences.

Exercise 12: Given the two binary images A and B in Fig. 4.31. How can logic operations be applied to generate the binary images: C, D, E, F, G, and H?

Exercise 13: The two images $f_1(x, y)$ and $f_2(x, y)$ in Fig. 4.29 are added together. Calculate the 8-bit output image?

Exercise 14: The image $f_2(x, y)$ in Fig. 4.29 is inverted and alpha blended with $f_1(x, y)$ where $\alpha = 0.5$. Calculate the output image.

Additional exercise 1: Describe the motivation for using gamma-correction in image capturing and visualization.

Additional exercise 2: Find and describe alternative automatic thresholding methods.