Digital Multimedia 3rd edition is published by:

John Wiley & Sons Ltd.,
The Atrium,
Southern Gate,
Chichester
PO19 8SQ

The Publisher may be contacted at cs-books@wiley.co.uk or via their Web site www.wiley.com

The Authors may be contacted via their Web site www.macavon.co.uk or via the book's support site www.digitalmultimedia.org

Digital Multimedia

3rd edition

Nigel Chapman and Jenny Chapman © 2009

Published by John Wiley & Sons, Ltd.

The material in this sampler provides an indication of the content, style and teaching and learning features in the fully revised and re-illustrated 3rd edition of Digital Multimedia. We have included short excerpts from most chapters.

You can find a range of additional support materials, the full table of contents and preface from the book at the book's Web site, www.digitalmultimedia.org. Instructors who are affiliated to a recognized educational institution can request an evaluation copy of the book from our publishers, John Wiley & Sons, or by using the evaluation request form under Contact Us on the support site.

Suppose you had created a new computer program to display images of geological microscope slides interactively. Your program would offer people without a microscope the chance to examine rock samples under different lighting, rotate them and take measurements, almost as if they were working with the real thing. How would you provide the people who were going to use your program with instructions about how to operate it?

Until fairly recently, you probably would have had no hesitation in answering that question. You would write a manual. If your program was simple enough, you would just write a brief introduction and supplement it with a quick-reference card. If your program was more elaborate you might write a tutorial and a reference manual, and if appropriate, a developer's guide. However extensive the documentation needed to be, it would still have consisted of pages of text, like the one shown in Figure 1.1, composed and laid out in a way that best conveyed the information.

Measuring Angles

You can measure the angle between a reference point and some other position using the cross-hairs and the angle readouts.

1. Click on the button with an upright cross (+) on it to show the cross-hairs. This will cause a new button, labelled Set to appear, together with two text fields: the one to the left of the Set button (the base angle read-out) will be blank. The other (the angular difference readout) will show a copy of the current angle of rotation.
2. Use the slider or stepping arrows to rotate the slide to the position you want to use as the reference for your measurement.
3. Click the Set button. The current angle will be copied to the base angle readout and the angular difference readout will be set to zero.
4. Use the slider or stepping arrows to rotate the slide to the position where you want to measure the angle.
5. Read the angle in the angular difference readout.

Figure 1.1. ***Text***

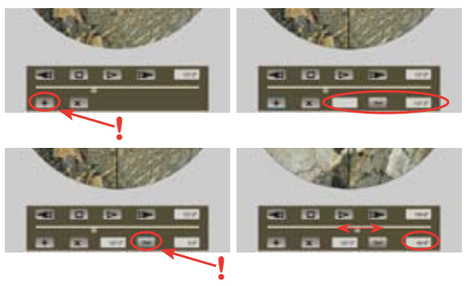


Figure 1.2. ***Images***

If your program had a graphical user interface (which in our example it certainly would have) you might think that the most direct way to convey your instructions was by using screenshots and other illustrations. A written manual often includes images. For a simple program, you might consider it best to dispense with all or most of the text and rely solely on pictures – such as those shown in Figure 1.2 – to provide the instructions, the way the manufacturers of flat-packed furniture do.

If you had the means, you could also consider creating an instructive Web site. Here you could combine text and images to be displayed in a Web browser. Where you needed to provide cross-references from one section of the manual to another you could use links, so that a visitor to the site could click on some text and go to the linked page which described the cross-referenced topic. Figure 1.3 shows what a page from such a site could look like. You might be able to take advantage of the possibilities offered by server-side computation to provide a community help service, allowing users of the program to add their own advice to the instructions you had supplied, for the benefit of other users.

Alternatively, you might choose to prepare a slide presentation in PowerPoint or some similar program. Again, you could combine textual instructions with relevant illustrations, but in this case the material would be divided into short pieces, intended to be shown in sequence one after another. You could present the slides to an audience or make them available for people to download and view on their own computers, stepping forwards and backwards to read and re-read each slide. Possibly, you might include animated transitions and effects between each slide, to emphasize the sequential development of the material and to add visual interest.

If you felt that the users of your program would be likely to understand information presented as video more easily than any other form of instruction, you could go beyond the slide presentation and create an instructional video. This could be made available either for distribution on DVD or for watching on a Web site using a video player plug-in, like the one shown in Figure 1.4. A video presentation can include dynamic screen recordings, showing exactly what happens on screen when you perform some operation such as measuring an angle of rotation. This type of screen recording could usefully be supplemented by a spoken commentary explaining what was happening.

Sound on its own would probably not be a very good medium for conveying how to use your program, but it could be used to provide supplementary tips. A sound recording in the form of a conversation between expert users of the program can be an effective means of conveying knowledge in an informal way that captures some of the character of personal conversations in which this sort of information is passed on. (We are assuming here that you are only concerned with instructing sighted users, as people who cannot see would not be able to examine rock samples in the way described, but of course for many other applications sound would play a much more important role, in providing an alternative mode of instruction for people who are blind or partially sighted. We will discuss this in much greater detail later on.)

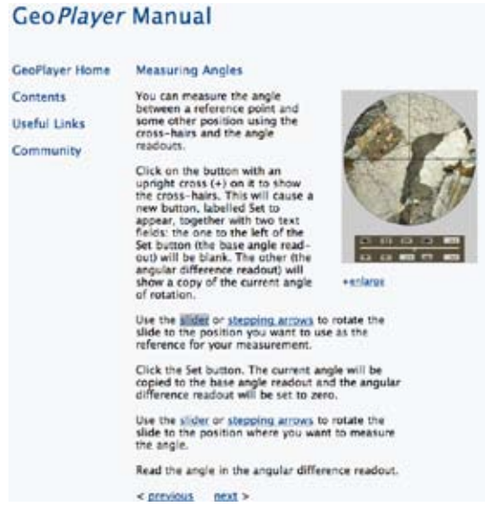


Figure 1.3. ***A Web page***

Figure 1.4. ***Video***

KEY POINTS

Drawing programs and vector graphics languages provide a basic repertoire of shapes that can easily be represented mathematically.

The commonest shapes are rectangles and squares, ellipses and circles, straight lines and Bézier curves.

Bézier curves are smooth curves that can be specified by an ordered set of control points; the first and last control points are the curve's end points.

A cubic Bézier curve has four control points: two end points and two direction points.

The sweep of a Bézier curve is determined by the length and direction of the direction lines between the end and direction points.

Cubic Bézier curves are drawn by dragging direction lines with a pen tool.

Quadratic Bézier curves only have a single direction point. They are the only Bézier curves supported by SWF. PDF and SVG provide both cubic and quadratic Bézier curves.

Bézier curve segments can be combined to make smooth paths.

A closed path joins up on itself, an open path does not.

If two curves join at a point and their direction lines through that point form a single line, the join will be smooth.

The points where curve segments join are the path's anchor points.

Apply a stroke to a path to make it visible, specifying the width and colour.

Fill closed paths with colours, gradients or patterns.

Use a fill rule to determine which points are inside a path.

Transformations

The objects that make up a vector image are stored in the form of a few values that are sufficient to describe them accurately: a line by its end points, a rectangle by its corners, and so on. The actual pixel values that make up the image need not be computed until it is displayed. It is easy to manipulate objects by changing these stored values. For example, if a line runs parallel to the *x*-axis from (4,2) to (10,2), all we need do in order to move it up by 5 units is add 5 to the *y* coordinates of its end points, giving (4,7) and (10,7), the end points of a line running parallel to the *x*-axis, but higher up. We have transformed the image by editing the model that is stored in the computer.

Affine Transformations

Only certain transformations can be produced by changing the stored values without altering the type of object. For instance, changing the position of just one corner of a rectangle would turn it into an irregular quadrilateral, so the simple representation based on the coordinates of the two corners could no longer be used. Transformations which preserve straight lines (i.e. which don't bend or break them) and keep parallel lines parallel are the only ones that can be used to maintain the fundamental shape of an object. Transformations that behave in this way are called ***affine transformations***.

The most important affine transformations are ***translation*** (moving the object in a straight line), ***scaling*** (changing its dimensions), ***rotation*** about a point, ***reflection*** about a line and ***shearing*** (a distortion of the angles of the axes of an object). These transformations are illustrated in Figure 3.25. Any modern drawing program will allow you to perform these transformations by direct manipulation of objects on the screen. For example, you would translate an object simply by dragging it to its new position.

Figure 3.25 illustrates another feature of all vector graphics programs. Several objects – in this case, four coloured squares – can be grouped and manipulated as a single entity. Grouping may be implemented entirely within the drawing program, as a convenience to designers; it is also supported within some graphics languages, including SVG. In a related feature, some programs and languages allow an object or a group of objects to be defined as a ***symbol***, which is a reusable entity.

Figure 3.25. ***An object being scaled, rotated, reflected, sheared and translated***

Instances of a symbol can be created, all of which refer to the original. If the symbol is edited, all instances of it are updated to reflect the changes. (Symbols behave much like pointers.)

Briefly, the operations which achieve the affine transformations are as follows.

Any translation can be done by adding a displacement to each of the x and y coordinates stored in the model of the object. That is, to move an object Δx to the right and Δy upwards, change each stored point (x,y) to $(x + \Delta x, y + \Delta y)$. Negative Δs move in the opposite direction.

Scaling is performed by multiplying coordinates by appropriate values. Different factors may be used to scale in the x and y directions: to increase lengths in the x direction by a factor of s_x and in the y direction by s_y, (x,y) must be changed to $(s_x x, s_y y)$. (Values of s_x or s_y less than one cause the object to shrink in the corresponding direction.) Thus, to double the size of an object, its stored coordinates must be multiplied by two. However, this has the effect of simultaneously displacing the object. (For example, if a unit square has its corners at (1,2) and (2,1), multiplying by two moves them to (2,4) and (4,2), which are the corners of a square whose side is of length 2, but it is now in the wrong place.) To scale an object in place, the multiplication must be followed by a suitable, easily computed displacement to restore it to its original position.

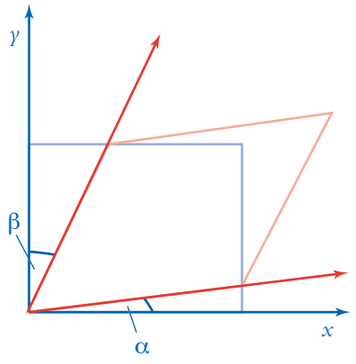


Figure 3.26. ***Skewed axes***

Rotation about the origin and reflection about an axis are simple to achieve. To rotate a point (x,y) around the origin in a clockwise direction by an angle θ, you transform it to the point $(x\cos\theta - y\sin\theta, x\sin\theta + y\cos\theta)$ (which you can prove by simple trigonometry if you wish).

To reflect it in the x-axis, simply move it to $(x,-y)$; in the y-axis, to $(-x,y)$.

When an object is sheared, it is as if we took the x-axis and skewed it upwards, through an angle α, say, and skewed the y-axis through an angle β (see Figure 3.26). You can show that the transformation can be achieved by moving (x, y) to $(x + y\tan\beta, y + x\tan\alpha)$.

Applying these operations to all the points of an object will transform the entire object. The more general operations of rotation about an arbitrary point and reflection in an arbitrary line require more complex, but conceptually simple, transformations. The details are left as an exercise.

IN DETAIL

Mathematicians will be aware that any combination of translation, scaling, rotation, reflection and skewing can be expressed in the form of a 3×3 transformation matrix

$$T = \begin{bmatrix} a & c & e \\ b & d & f \\ 0 & 0 & 1 \end{bmatrix}$$

To be able to express any transformation, including translation, as a matrix product, a point P = (*x*, *y*) is written as the column vector

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

(*P* is said to be specified using "homogeneous coordinates"), and the effect of applying the transformation is given by the product *T*•*P*. Since the bottom row of the transformation matrix is always the same, just six numbers are required to specify any transformation. This compact representation is often used internally by graphics systems to store transformation information, and can be specified explicitly, for example in SVG.

Distortion

Other, less structured, alterations to paths can be achieved by moving (i.e. changing the coordinates of) their anchor points and control points. This can be done by interactive manipulation in a drawing program. Anchor points and control points may also be added and deleted from paths, so that a designer or artist can fine-tune the shape of objects.

Some commonly required effects which fall between the highly structured transformations and the free manipulation of control points are provided by way of parameterized commands in Illustrator and similar programs. These are referred to as filters, by analogy with the filters available in bitmapped image manipulation programs. An object is selected and an operation is chosen from a menu of those available. The chosen filter is then applied to the selected object.

Figure 3.27 shows two examples of applying Illustrator's `Pucker & Bloat` filter to a simple shape. The result is achieved by turning each anchor point into a Bézier corner point, and then extending the direction lines either inwards, to give the puckering effect shown on the left, or outwards, to give

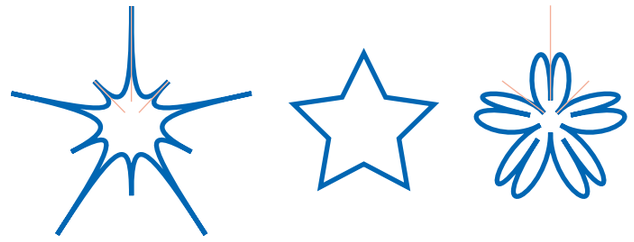


Figure 3.27. ***Pucker and bloat***

the bloating effect on the right. The extent of the distortion – the amount by which the direction lines are extended – is controlled by a slider when the filter is applied.

The thin red lines in Figure 3.27 are the direction lines for the two segments to the left and right of the vertex at the top of the original star. There are actually four of them: the two lines at the vertex itself lie on top of each other. (Remember that these are corner points.)

Other filters are available in Illustrator. Zig-zag adds extra anchor points between existing ones in a regular pattern; an option to the filter determines whether the segments of the resulting path should be straight lines, to produce a jagged result, or curves, which produces a smooth version of the effect. Roughening is similar, but adds the new anchor points in a pseudo-random fashion, so that the result is irregular. The tweak filter applies a proportional movement to all the anchor points and control points on a path. These filters are parameterized in values such as the maximum distance an anchor point may be moved. The parameters can be set via various controls presented in a dialogue when the filter is applied.

IN DETAIL

In Illustrator, distortions are applied as "effects", which do not actually add and modify the anchor points of the path they are applied to. It looks as if they are doing so, but the modification is "live": the actual path is not changed, but the effect is applied when it is displayed or exported to some other format. This means that the parameters can be changed after the initial application, and the effect can eaily be removed or temporarily disabled.

Evidently, these distortions are not affine transformations. By adding anchor points they can turn corner points into smooth points, straight lines into curves and vice versa, so straightness and parallelism are not preserved. The matrix representation cannot be used for such transformations.

The important thing to understand about transformations and distortions is that they are achieved simply by altering the coordinates of the defining points of objects, altering the stored model using nothing but arithmetical operations which can be performed efficiently. Although every pixel of the object must be transformed in the final displayed image, only the relatively few points that are needed to define the object within the model need to be recomputed beforehand. All the pixels will appear in the desired place when the changed model is rendered on the basis of these changed values.

Alterations to objects' appearance of the sort we will describe in Chapter 4, which rely on altering pixels, can only be achieved by rasterizing the objects, which destroys their vector characteristics.

KEY POINTS

Vector objects can be altered by changing the stored values used to represent them.

Affine transformations preserve straight lines and keep parallel lines parallel.

Translation, scaling, rotation, reflection and shearing are affine transformations, which can be performed by direct manipulation in vector drawing programs.

All five of these affine transformations can be defined by simple equations.

An entire object can be transformed by applying an affine transformation to each of its anchor points.

Several objects can be combined into a group, and transformed as a whole.

Less structured alterations to paths can be achieved by moving their anchor points and control points.

Existing anchor points can be moved and direction lines modified.

More structured distortions can be achieved using filters, which modify all a path's anchor points and control points systematically. Some filters add new anchor points.

The modifications implemented by distorting filters are not affine transformations.

3-D Graphics

Pictures on a screen are always two-dimensional, but this doesn't mean that the models from which they are generated need to be restricted to flat two-dimensional shapes. Models of three-dimensional objects correspond more closely to the way we perceive space. They enable us to generate two-dimensional pictures as perspective projections – or perhaps other sorts of projection – onto a plane, as if we were able to photograph the model. Sometimes, this may be easier than constructing the two-dimensional image from scratch, particularly if we can begin with a numerical description of an object's dimensions, as we might if we were designing some mechanical component, for example, or constructing a visualization on the basis of a simulation.

A three-dimensional model allows us to generate many different images of the same objects. For example, if we have a model of a house, we can produce a view of it from the front, from the back, from each side, from close up, far away, overhead, and so on, all using the same model. Figure 3.28 shows an example. If we were working in only two dimensions, each of these images would have to be drawn separately.†

† *Frank Lloyd Wright's "Heller House", modelled by Google, from Google 3D Warehouse.*

JPEG compression is highly effective when applied to the sort of images for which it is designed, i.e. photographic and scanned images with continuous tones. Such images can sometimes be compressed to as little as 5% of their original size without apparent loss of quality. Lossless compression techniques are nothing like as effective on this type of image. Still higher levels of compression can be obtained by using a lower quality setting, that is, by using coarser quantization that discards more information. When this is done the boundaries of the 8×8 squares to which the DCT is applied tend to become visible on screen, because the discontinuities between them mean that different frequency components are discarded in each square. At low compression levels (i.e. high quality settings) this does not matter, since enough information is retained for the common features of adjacent squares to produce appropriately similar results, but as more and more information is discarded, the common features become lost and the boundaries show up.

Such unwanted features in a compressed image are called ***compression artefacts***. Other artefacts may arise when an image containing sharp edges is compressed by JPEG. Here, the smoothing that is the essence of JPEG compression is to blame: sharp edges come out blurred. This is rarely a problem with the photographically originated material for which JPEG is intended, but it can be a problem if images created on a computer are compressed. In particular, if text, especially small text, occurs as part of an image, JPEG is likely to blur the edges, often making the text unreadable. For images with many sharp edges, JPEG compression should be avoided. Instead, images should be saved in a format such as PNG, which uses lossless LZ77 compression.

Figure 4.11 shows enlarged views of the same detail from our photograph, in its original form and having been JPEG-compressed using the lowest possible quality setting. The compression reduced its size from 24.3 MB to 160 kB, and even with such severe compression it is not easy to see the difference between the two images at normal size. However, when they are blown up as in Figure 4.11, you can see the edges of the 8×8 blocks – these are not the individual pixels, both images have the same resolution. You should also be able to see that some details, such as the brown specks in the yellow area, have been lost. These symptoms are typical of the way JPEG compression artefacts appear in heavily compressed photographic images.

IN DETAIL

Although a single cycle of JPEG compression and decompression at low settings may not cause much degradation of the image, repeated compression and decompression will do so. In general, therefore, if you are making changes to an image you should save working versions in some uncompressed format, and not keep saving JPEGs. However, it is possible to rotate an image through 90° without any loss, providing the image's dimensions are an exact multiple of the size of the blocks that are being compressed. This is probably why the dimensions of images from digital cameras are always multiples of 8: changing from landscape to portrait format and vice versa is such a rotation.

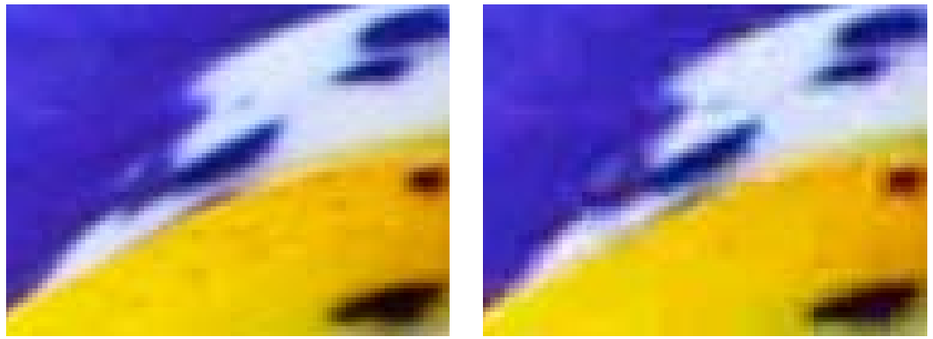


Figure 4.11. ***Original (left) and JPEG (right)***

JPEG2000

JPEG compression has been extremely successful: it is used for almost all photographic images on the Web, and by most low- to mid-range digital cameras for storing images. It is, however, by no means the best possible algorithm for performing image compression. Some of its shortcomings are a reflection of the limited processing power that was available on most computers at the time the JPEG standard was devised. Others arise from a failure to anticipate all the potential applications of compressed image files.

A successor to the JPEG standard has been developed in an attempt to overcome these shortcomings. It was adopted as an ISO standard in 2000, hence it is called ***JPEG2000***. Its aim is to improve on the existing DCT-based JPEG in several areas. These include providing better quality at high compression ratios (and thus low bit rates), incorporating lossless compression and alpha channel transparency within the single framework of JPEG2000, "region of interest" coding, where some parts of the image are compressed with greater fidelity than others, better progressive display and increased robustness in the face of transmission errors. Unlike the original JPEG standard, JPEG2000 also specifies a file format.

The basic structure of the compression process is the same. First, the image is divided into rectangular tiles, each of which is compressed separately. JPEG2000 tiles may be any size, up to the size of the entire image, so the artefacts seen in JPEG images at the edges of the 8×8 blocks can be reduced or eliminated by using bigger tiles. Next, a transform is applied to the data, giving a set of frequency coefficients. However, instead of the DCT, a transform based on diffferent principles, known as a ***Discrete Wavelet Transform (DWT)***, is used in JPEG2000.

It's quite easy to get an idea of what the DWT does by considering a simpler sort of wavelet, called the ***Haar wavelet***. (Actually, we are only going to consider a special case, and deal with that informally,

because even simple wavelets involve quite complicated mathematics.) To keep things extremely simple, consider a single row of an image with just four pixels, and suppose their values are 12, 56, 8 and 104. We can make a lower-resolution approximation to this row of pixels by taking the average of the first pair and the last pair of pixels, giving two new pixels, whose values are 34 and 56, but in doing so we have lost some information. One way (out of many) to retain that information is by storing the magnitude of the difference between each average and the pixels it was computed from: we can subtract this value from the average to get the first pixel, and add it to get the second. In our example, these ***detail coefficients***, as they are called, are 22 and 48. (34 − 22 = 12, 34 + 22 = 56, and so on.)

We can repeat this process, averaging the averages and computing a new detail coefficient. The new average is 45 and the detail coefficient is 11. Finally, we can combine this single average pixel value with all three detail coefficients, as the sequence 45, 11, 22, 48. It should be clear that the original four pixel values can be reconstructed from this sequence by reversing the process we used to arrive at it.

This exercise has not produced any compression, but – as with the DCT – it has rearranged the information about the image into a form where it is possible to isolate detail. You can think of the final sequence, which is the ***wavelet decomposition*** of the original, as comprising a value that describes the whole image coarsely – it is the average brightness of the whole image – together with a set of coefficients that can be used to add progressively more detail to the image. Using the first detail coefficient we can get a two-pixel image; using the other coefficients gets us back to the full four pixels. Each step in the reconstruction process doubles the resolution. If we want to compress the image by discarding detail, we just have to discard the later coefficients (or quantize them more coarsely than the earlier ones). This is essentially the way in which JPEG2000 compression works.

This transform could be applied to a complete two-dimensional image by first transforming all the rows and then – treating the matrix of values produced by the transform as a new image – transforming all the columns. Alternatively, the same result could be obtained by carrying out the first transform step on all the rows, and then on all the columns, before applying the second step to all the rows, and so on. If the process is carried out in this order, then after a step has been applied to both rows and columns the result will comprise a version of the whole image, at half the horizontal and vertical resolution of the version obtained at the previous step, in the top left quadrant of the matrix, together with detail coefficients in the other quadrants. The process is illustrated schematically in Figure 4.12.

The idea of encoding an image (or any function) at varying levels of resolution, embodied in the example we have just given, can be generalized, but the mathematics involved in doing so is by no

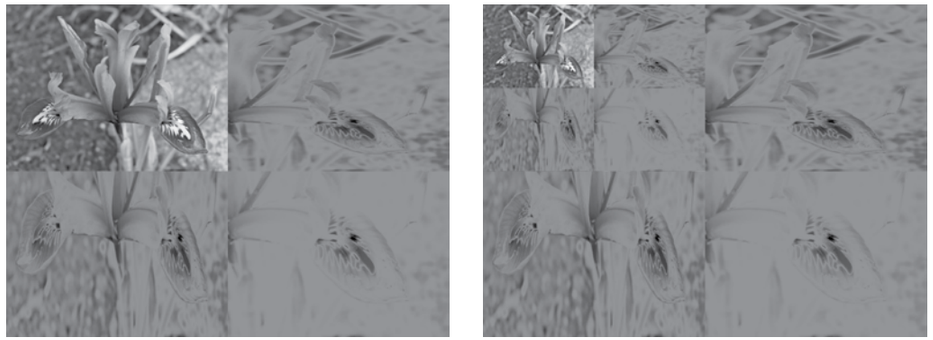


Figure 4.12. *Wavelet decomposition of an image*

means trivial. However, the subject is now well understood and many different ways of obtaining a wavelet decomposition are known. JPEG2000 specifies two: a reversible transform, which does not lose any information, and an irreversible transform, which does. Both of these can be implemented efficiently and relatively simply. The choice of transforms makes it possible for JPEG2000 to perform lossless as well as lossy compression within the context of a single algorithm.

After the wavelet decomposition has been computed, the coefficients are quantized using a quality setting to specify a step size – the difference between quantization levels. If this size is set to 1, no quantization occurs, so this step will also be lossless in that case.

Finally, the quantized coefficients are encoded using ***arithmetic coding***, a lossless compression algorithm that is more effective than Huffman coding.

The structure of the wavelet decomposition makes it easy to define a format for the data which allows an image to be displayed as a sequence of progressively better approximations, since each level of coefficients adds more detail to the image. This was considered a desirable property for images to be transmitted over networks. It also makes it possible to zoom into an image without loss of detail.

It has been estimated that decoders for JPEG2000 are an order of magnitude more complex than those for JPEG. As Figure 4.13 shows, the reward for the added complexity comes in the form of the extremely good quality that JPEG2000 can produce at high compression ratios. Here, the photograph has been compressed to roughly the same size as the JPEG version shown in Figure 4.11. This time, however, there are no block edges to be seen, and although some detail has been lost, the loss takes the form of a general softening of the image, which is more acceptable than the ugly artefacts produced in the JPEG.

Figure 4.13. *Original (left) and JPEG2000 (right)*

JPEG2000 compression is superior to JPEG, and the JPEG2000 file format supports some desirable features that JPEG files lack. However, at the time of writing, there is little or no support for JPEG2000 in digital cameras, Web browsers and graphics programs.

The main obstacle to more widespread support for JPEG2000 lies in the fact that JPEG is so thoroughly entrenched. Many millions of JPEG images are already available on the Web, there are many well-established and popular software tools for creating JPEG images and incorporating them into Web pages, most digital cameras (and even mobile phones) will generate JPEGs, and Web designers are familiar with JPEG.

There is therefore a lack of any perceived need for JPEG2000 and adoption of the new standard has been slow. Some influential institutions, including the Library of Congress and the Smithsonian Institution, do use JPEG2000 as an archival format, so it may be that instead of replacing JPEG as a Web image format, JPEG2000 will find a different niche.

IN DETAIL

In 2007, JPEG announced that it would be considering another format for standardization, in addition to the original JPEG and JPEG2000. JPEG XR is the name proposed for a standard version of Microsoft's HD Photo format (formerly Windows Media Photo). This is claimed to possess many of the advantages of JPEG2000, but appears to be better suited to implementation in digital cameras. HD Photo is based on a "lapped biorthogonal transform", which more closely resembles the DCT than DWT.

It is not yet clear whether JPEG XR will become a standard, or whether it will be adopted with any more enthusiasm than JPEG2000.

KEY POINTS

Images can be losslessly compressed using various methods, including run-length encoding (RLE), Huffman encoding and the dictionary-based LZ77, LZ78, LZW and deflate algorithms.

JPEG is the most commonly used lossy compression method for still images.

High-frequency information can be discarded from an image without perceptible loss of quality, because people do not perceive the effects of high frequencies in images very accurately.

The image is mapped into the frequency domain using the Discrete Cosine Transform (DCT).

The Discrete Cosine Transform is applied to 8×8 blocks of pixels.

Applying the DCT does not reduce the size of the data, since the array of frequency coefficients is the same size as the original pixel array.

The coefficients are quantized, according to a quantization matrix which determines the quality. The quantization discards some information.

After quantization there will usually be many zero coefficients. These are RLE-encoded, using a zig-zag sequence to maximize the length of the runs.

The non-zero coefficients are compressed using Huffman encoding.

Decompression is performed by reversing the process, using the Inverse DCT to recover the image from its frequency domain representation.

The decompressed image may exhibit compression artefacts, including blurring and visible edges at the boundaries between the 8×8 pixel blocks, especially at low quality settings.

JPEG2000 improves on JPEG in many areas, including image quality at high compression ratios. It can be used losslessly as well as lossily.

For JPEG2000 compression the image is divided into tiles, but these can be any size, up to the entire image.

A Discrete Wavelet Transform (DWT) is applied to the tiles, generating a wavelet decomposition, comprising a coarse (low resolution) version of the image and a set of detail coefficients that can be used to add progressively more detail to the image.

The DWT may be reversible (lossless) or irreversible (lossy).

The detail coefficients in the wavelet decomposition may be quantized and are then losslessly compressed using arithmetic encoding.

Consistent Colour

Colour adjustment is messy, and getting it wrong can cause irreparable damage to an image. It would be much better to get things right first time, but the varying colour characteristics of different monitors and scanners make this difficult. Some recent developments in software are aimed at compensating for these differences. They are based on the use of "profiles", describing how devices detect and reproduce colour.

We don't need much information to give a reasonable description of the colour properties of any particular monitor. We need to know exactly which colours the red, green and blue phosphors emit (the R, G and B chromaticities). These can be measured using a suitable scientific instrument, and then expressed in terms of one of the CIE device-independent colour spaces. We also need to know the maximum saturation each component is capable of, i.e. we need to know what happens when each electron beam is full on. We can deduce this if we can characterize the make-up and intensity of white, since this tells us what the (24-bit) RGB value (255,255,255) corresponds to. Again, the value of white – the monitor's ***white point*** – can be specified in a device-independent colour space.

IN DETAIL

You will sometimes see the white point specified as a "colour temperature", in degrees absolute. This form of specification is based on the observation that the spectral make-up of light emitted by a perfect radiator (a "black body") depends only on its temperature, so a black body temperature provides a concise description of an SPD. Most colour monitors for computers use a white point designed to correspond to a colour temperature of 9300 K. This is far higher than daylight (around 7500 K), or a conventional television monitor (around 6500 K), in order to generate the high light intensity required for a device that will normally be viewed under office lighting conditions. (Televisions are designed on the assumption that they will be watched in dimly lit rooms.) The "white" light emitted by a monitor when all its three colours are at full intensity is actually quite blue.

Computer monitors are not actually black bodies, and so their SPDs deviate from the shape of the black body radiation, which means that colour temperature is only an approximation of the characteristic of the white point, which is better specified using CIE colour values.

Finally, the most complex element in the monitor's behaviour is the relationship between the RGB values presented to it by the graphics controller, and the intensity of the light emitted in response. This relationship is not a simple linear one: the intensity of light emitted in response to an input of 100 is not 10 times that produced by an input of 10, which in turn is not 10 times

that produced by an input of 1. The ***transfer characteristic*** of a display – the relationship between the light intensity I emitted by the screen and the voltage V applied to the electron gun is often modelled by the transfer function $I = V^{\gamma}$, where γ is a constant. Thus, it is common to use the value of γ to characterize the response. Unfortunately, this model is not entirely accurate, and one of the sources of variability between monitors lies in the use of incorrect values for γ which attempt to compensate for errors in the formula. Another is the fact that some display controllers attempt to compensate for the non-linearity by adjusting values according to an inverse transfer function before they are applied to the electron guns, while others do not. In particular, Macintosh and Windows systems deal with the transfer characteristic differently, with the result that colours with the same RGB values look different on the two systems.

However, it is convenient to use a single number to represent the transfer characteristic, and γ serves this purpose reasonably well, and provides the last value normally used to model the behaviour of a monitor. In this context, it is usual to spell out the letter's name, and refer to the display's ***gamma***. Similar collections of values can be used to characterize the colour response of other types of device, including scanners and printers.

Armed with accurate device-independent values for red, green and blue chromaticities, and the white point and gamma of a particular monitor, it is possible to translate any RGB colour value into an absolute, device-independent, colour value in a CIE colour space that exactly describes the colour produced by that monitor in response to that RGB value. This is the principle behind the practice of ***colour management***.

In a typical situation calling for colour management an image is prepared using some input device. For simplicity, assume this is a monitor used as the display by a graphics editor, but the same principles apply to images captured by a digital camera or a scanner. The image will be stored in a file, using RGB values which reflect the way the input device maps colours to colour values – its ***colour space***. Later, the same image may be displayed on a different monitor. Now the RGB values stored in the image file are mapped by the output device, which probably has a different colour space from the input device. The colours which were stored in the input device's colour space are interpreted as if they were in the output device's colour space. In other words, they come out wrong. (See Figure 5.27.)

Figure 5.27. ***No colour management***

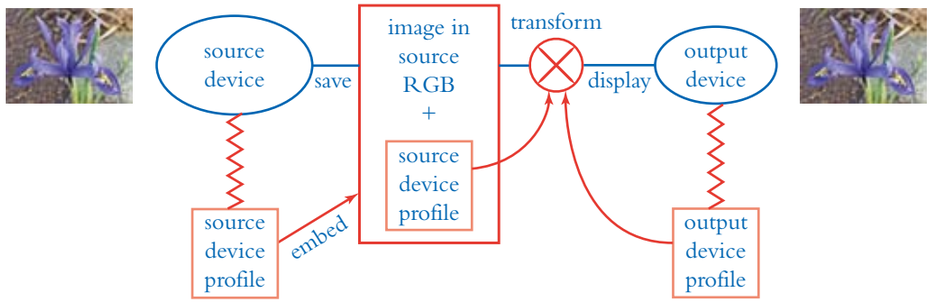


Figure 5.28. ***Colour management with embedded profiles***

One way of correcting this, as illustrated in Figure 5.28, is to embed information about the input device's colour space profile in the image file in the form of a ***colour profile***. Colour profiles are created by taking accurate measurements of a device's colour response, using standard techniques and colour targets. PSD, PDF, TIFF, JFIF, PNG and other types of file are able to accommodate such information, in varying degrees of detail. At the least, the R, G and B chromaticities, white point and gamma can be included in the file. Software on the machine being used to display the image can, in principle, use this information to map the RGB values it finds in the image file to colour values in a device-independent colour space. Then, using the device profile of the output monitor, it can map those device-independent values to the colour space of the output device, so that the colours are displayed exactly as they were on the input device. In practice, it is more likely that the two profiles would be combined and used to map from the input device colour space to the output device colour space directly.

It is possible, of course, that some of the colours in the input device's colour space are not available in the output device's colour space. For example, as we showed earlier, many colours in RGB lie outside the CMYK gamut, so they cannot be printed correctly. Colour management can only actually guarantee that the colours will be displayed exactly as they were intended within the capabilities of the output device. If software uses colour management consistently, the approximations made to accommodate a restricted colour gamut will be the same, and the output's colour will be predictable, at least.

An alternative way of using colour management software is to modify the colours displayed on a monitor using the profile of a different output device. In this way, colour can be accurately previewed, or "soft proofed". This mode of working is especially useful in pre-press work, where actually producing printed proofs may be expensive or time-consuming. For example, if you were preparing a book to be printed on a phototypesetter, you could use the phototypesetter's colour profile, in combination with the colour profile of your monitor, to make the monitor display the colours in the book as they would be printed on the phototypesetter.

To obtain really accurate colour reproduction across a range of devices, device profiles need to provide more information than simply the RGB chromaticities, white point and a single figure for gamma. In practice, for example, the gammas for the three different colours are not necessarily the same. As already stated, the actual transfer characteristics are not really correctly represented by gamma; a more accurate representation is needed. If, as well as displaying colours on a monitor, we also wished to be able to manage colour reproduction on printers, where it is necessary to take account of a host of issues, including the C, M, Y and K chromaticities of the inks, spreading characteristics of the ink, and absorbency and reflectiveness of the paper, even more information would be required – and different information still for printing to film, or video.

Since the original impetus for colour management software came from the pre-press and printing industries, colour management has already been developed to accommodate these requirements. The ***International Colour Consortium (ICC)*** has defined a standard device profile which supports extremely elaborate descriptions of the colour characteristics of a wide range of devices. ICC device profiles are used by colour management software such as Apple's ColorSync, the Adobe colour management system built into Photoshop and other Adobe programs, and the Kodak Precision Color Management System, to provide colour management services. Manufacturers of scanners, cameras, monitors and printers routinely produce ICC profiles of their devices.

Colour management is not much use unless accurate profiles are available. In fact, using an inaccurate profile can produce worse results than not using colour management at all. Unfortunately, no two devices are exactly identical and the colour characteristics of an individual device will change over time. Although a generic profile produced by the manufacturer for one line of monitors or scanners is helpful, to take full advantage of colour management it is necessary to calibrate individual devices, at relatively frequent intervals (once a month is often advised). Some high-end monitors are able to calibrate themselves automatically. For others, it is necessary to use a special measuring device in conjunction with software that displays a sequence of colour values and, on the basis of the measured output of the screen, generates an accurate profile.

You may wonder why the profile data is embedded in the file. Why is it not used at the input end to map the colour values to a device-independent form, such as L*a*b*, which can then be mapped to the output colour space when the image is displayed? The work is split between the two ends and no extra data has to be added to the file. The reason for not using this method is that most existing software does not work with device-independent colour values, so it could not display the images at all. If software ignores a device profile, things are no worse than they would have been if it was not there.

Clearly, though, it would be desirable to use a device-independent colour space for stored colour values. The ***sRGB (standard RGB)*** colour model attempts to provide such a space.

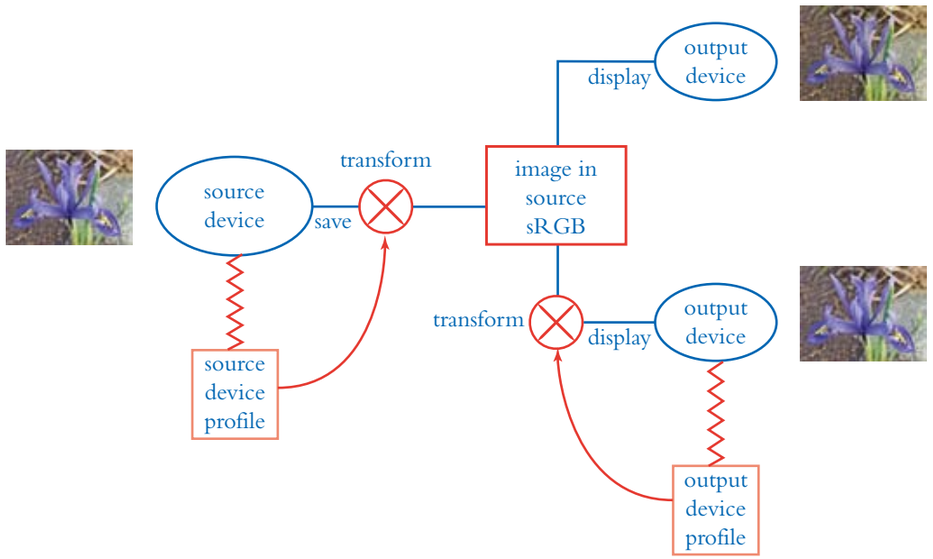


Figure 5.29. *Use of the sRGB colour space*

As you can guess, sRGB is an RGB colour model, and specifies standard values for the R, G and B chromaticities, white point and gamma. The standard values are claimed to be typical of the values found on most monitors (although many people claim that Adobe's standard RGB colour space is more widely applicable). As a result, if the display software is not aware of the sRGB colour space and simply displays the image without any colour transformation, it should still look acceptable. Figure 5.29 illustrates how sRGB can be used in colour management. With the software that transforms the image using the output device profile, colours should be accurately reproduced; if no transformation is applied, some colour shifts may occur, but these should not be as bad as they would have been if the image had been stored using the input device's colour space.

Use of sRGB colour is especially suitable for graphics on the World Wide Web, because there most images are only ever destined to be displayed on a monitor. Colour specifications in CSS are interpreted as sRGB values. Does it really matter if the colours in your image are slightly distorted when the image is displayed on somebody else's monitor? Consider online shopping catalogues or art gallery catalogues. For many images on the Web, accurate colour is important. (To take an extreme example, think about buying paint over the Internet.) One of the factors driving the development of colour management and its incorporation into Web browsers is the desire to facilitate online shopping. As well as the development of the sRGB colour space, this has led to the development of browser plug-ins providing full colour management facilities and increasingly, the incorporation of colour management directly into browsers.

KEY POINTS

The colour properties of a monitor can be roughly summarized by the R, G and B chromaticities, white point and gamma.

Gamma approximately models the relationship between RGB values and light intensity.

If an image is stored in one device's colour space and displayed on a device with a different colour space, colours will not be reproduced accurately.

If a colour profile that models the input device is embedded in the image file, it can be combined with a profile that models the output device to translate between the colour spaces and reproduce colours accurately.

Colours that are out of gamut should be reproduced consistently.

ICC colour profiles provide elaborate descriptions of the colour characteristics of a wide range of devices; they are used as a standard for colour management.

The success of colour management depends on having accurate profiles.

sRGB is intended as a standard device-independent colour space for monitors. It is used on the World Wide Web.

Exercises

Test Questions

1 What advantages are there to using images in greyscale instead of colour? Give some examples of applications for which greyscale is to be preferred, and some for which the use of colour is essential.

2 Is it true that any colour can be produced by mixing red, green and blue light in variable proportions?

3 What colours correspond to the eight corners of the cube in Figure 5.3?

4 Why do RGB colour values (r,g,b), with $r=g=b$ represent shades of grey?

5 Exactly how many distinct colours can be represented in 24-bit colour?

6 Explain carefully why the primary colours used in mixing pigments (paint or ink, for example) are different from those used in producing colours on a monitor.

Video Compression

The input to any video compression algorithm consists of a sequence of bitmapped images (the digitized video). There are two ways in which this sequence can be compressed: each individual image can be compressed in isolation, using the techniques introduced in Chapter 4, or subsequences of frames can be compressed by only storing the differences between them. These two techniques are usually called ***spatial compression*** and ***temporal compression***, respectively, although the more accurate terms ***intra-frame*** and ***inter-frame*** compression are also used, especially in the context of MPEG. Spatial and temporal compression are normally used together.

Since spatial compression is just image compression applied to a sequence of bitmapped images, it could in principle use either lossless or lossy methods. Generally, though, lossless methods do not produce sufficiently high compression ratios to reduce video data to manageable proportions, except on synthetically generated material (such as we will consider in Chapter 7), so lossy methods are usually employed. Lossily compressing and recompressing video usually leads to a deterioration in image quality, and should be avoided if possible, but recompression is often unavoidable, since the compressors used for capture are not the most suitable for delivery for multimedia. Furthermore, for post-production work, such as the creation of special effects, or even fairly basic corrections to the footage, it is usually necessary to decompress the video so that changes can be made to the individual pixels of each frame. For this reason it is wise – if you have sufficient disk space – to work with uncompressed video during the post-production phase. That is, once the footage has been captured and selected, decompress it and use uncompressed data while you edit and apply effects, only recompressing the finished product for delivery. (You may have heard that one of the advantages of digital video is that, unlike analogue video, it suffers no "generational loss" when copied, but this is only true for the making of exact copies.)

The principle underlying temporal compression algorithms is simple to grasp. Certain frames in a sequence are designated as ***key frames***. Often, key frames are specified to occur at regular intervals – every sixth frame, for example – which can be chosen when the compressor is invoked. These key frames are either left uncompressed, or more likely, only spatially compressed. Each of the frames between the key frames is replaced by a difference frame, which records only the differences between the frame which was originally in that position and either the most recent key frame or the preceding frame, depending on the sophistication of the decompressor.

For many sequences, the differences will only affect a small part of the frame. For example, Figure 6.5 shows part of two consecutive frames (de-interlaced), and the difference between them, obtained by subtracting corresponding pixel values in each frame. Where the pixels are identical, the result will be zero, which shows as black in the difference frame on the far right. Here, approximately 70% of the frame is black: the land does not move, and although the sea and clouds

Figure 6.5. ***Frame difference***

are in motion, they are not moving fast enough to make a difference between two consecutive frames. Notice also that although the girl's white over-skirt is moving, where part of it moves into a region previously occupied by another part of the same colour, there is no difference between the pixels. The cloak, on the other hand, is not only moving rapidly as she turns, but the shot silk material shimmers as the light on it changes, leading to the complex patterns you see in the corresponding area of the difference frame.

Many types of video footage are composed of large relatively static areas, with just a small proportion of the frame in motion. Each difference frame in a sequence of this character will have much less information in it than a complete frame. This information can therefore be stored in much less space than is required for the complete frame.

IN DETAIL

You will notice that we have described these compression techniques in terms of frames. This is because we are normally going to be concerned with video intended for progressively scanned playback on a computer. However, the techniques described can be equally well applied to fields of interlaced video. While this is somewhat more complex, it is conceptually no different.

Compression and decompression of a piece of video need not take the same time. If they do, the codec is said to be ***symmetrical***, otherwise it is ***asymmetrical***. In theory, this asymmetry could be in either direction, but generally it is taken to mean that compression takes longer – sometimes much longer – than decompression. This is acceptable, except during capture, but since playback must take place at a reasonably fast frame rate, codecs which take much longer to decompress video than to compress it are essentially useless.

Spatial Compression

The spatial element of many video compression schemes is based, like JPEG image compression, on the use of the Discrete Cosine Transform. The most straightforward approach is to apply JPEG compression to each frame, with no temporal compression. JPEG compression is applied to the three components of a colour image separately, and works the same way irrespective of the colour space used to store image data. Video data is usually stored using $Y'C_BC_R$ colour, with chrominance sub-sampling, as we have seen. JPEG compression can be applied directly to this data, taking advantage of the compression already achieved by this sub-sampling.

The technique of compressing video sequences by applying JPEG compression to each frame is referred to as ***motion JPEG*** or ***MJPEG*** (not to be confused with MPEG) compression, although you should be aware that, whereas JPEG is a standard, MJPEG is only a loosely defined way of referring to this type of video compression. MJPEG was formerly the most common way of compressing video while capturing it from an analogue source, and used to be popular in digital still image cameras that included primitive facilities for capturing video.

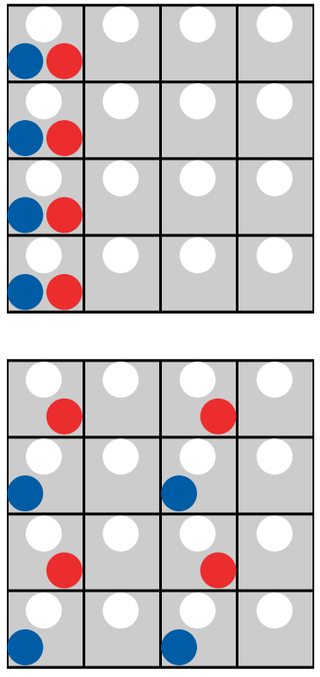


Figure 6.6. ***4:1:1 (top) and 4:2:0 chrominance sub-sampling***

Now that analogue video capture is rarely needed, the most important technology that uses spatial compression exclusively is DV. Like MJPEG, DV compression uses the DCT and subsequent quantization to reduce the amount of data in a video stream, but it adds some clever tricks to achieve higher picture quality within a constant data rate of 25 Mbits (3.25 Mbytes) per second than MJPEG would produce at that rate.

DV compression begins with chrominance sub-sampling of a frame with the same dimensions as CCIR 601. Oddly, the sub-sampling regime depends on the video standard (PAL or NTSC) being used. For NTSC (and DVCPRO PAL), 4:1:1 sub-sampling with co-sited sampling is used, but for other PAL DV formats 4:2:0 is used instead. As Figure 6.6 shows, the number of samples of each component in each 4×2 block of pixels is the same. As in still-image JPEG compression, blocks of 8×8 pixels from each frame are transformed using the DCT, and then quantized (with some loss of information) and run-length and Huffman encoded along a zig-zag sequence. There are, however, a couple of additional embellishments to the process.

First, the DCT may be applied to the 64 pixels in each block in one of two ways. If the frame is static, or almost so, with no difference between the picture in each field, the transform is applied to the entire 8×8 block, which comprises alternate lines from the odd and even fields.

However, if there is a lot of motion, so that the fields differ, the block is split into two 8×4 blocks, each of which is transformed independently. This leads to more efficient compression of frames with motion. The compressor may determine whether there is motion between the frames by using motion compensation (described below under MPEG), or it may compute both versions of the DCT and choose the one with the smaller result. The DV standard does not stipulate how the choice is to be made.

Second, an elaborate process of rearrangement is applied to the blocks making up a complete frame, in order to make best use of the space available for storing coefficients. A DV stream must use exactly 25 Mbits for each second of video; 14 bytes are available for each 8×8 pixel block. For some blocks, whose transformed representation has many zero coefficients, this may be too much, while for others it may be insufficient, requiring data to be discarded. In order to allow the available bytes to be shared between parts of the frame, the coefficients are allocated to bytes, not on a block-by-block basis, but within a larger "video segment". Each video segment is constructed by systematically taking 8×8 blocks from five different areas of the frame, a process called ***shuffling***. The effect of shuffling is to average the amount of detail in each video segment. Without shuffling, parts of the picture with fine detail would have to be compressed more highly than parts with less detail, in order to maintain the uniform bit rate. With shuffling, the detail is, as it were, spread about among the video segments, making efficient compression over the whole picture easier.

As a result of these additional steps in the compression process, DV is able to achieve better picture quality at 25 Mbits per second than MJPEG can achieve at the same data rate.

Temporal Compression

All modern video codecs use temporal compression to achieve either much higher compression ratios, or better quality at the same ratio, relative to DV or MJPEG. Windows Media 9, the Flash Video codecs and the relevant parts of MPEG-4 all employ the same broad principles, which were first expressed systematically in the MPEG-1 standard. Although MPEG-1 has been largely superseded, it still provides a good starting point for understanding the principles of temporal compression which are used in the later standards that have improved on it, so we will begin by describing MPEG-1 compression in some detail, and then indicate how H.264/AVC and other important codecs have enhanced it.

The MPEG-1 standard† doesn't actually define a compression algorithm: it defines a data stream syntax and a decompressor, allowing manufacturers to develop different compressors, thereby leaving scope for "competitive advantage in the marketplace". In practice, the compressor is fairly thoroughly defined implicitly, so we can describe MPEG-1 compression, which combines

† ISO/IEC 11172: "Coding of moving pictures and associated audio for digital storage media at up to about 1.5 Mbit/s."

temporal compression based on motion compensation with spatial compression based, like JPEG and DV, on quantization and coding of frequency coefficients produced by a discrete cosine transformation of the data.

A naïve approach to temporal compression consists of subtracting the value of each pixel in a frame from the corresponding pixel in the previous frame, producing a difference frame, as we did in Figure 6.5. In areas of the picture where there is no change between frames, the result of this subtraction will be zero. If change is localized, difference frames will contain large numbers of zero pixels, and so they will compress well – much better than a complete frame.

This frame differencing has to start somewhere, with frames that are purely spatially (intra-frame) compressed, so they can be used as the basis for subsequent difference frames. In MPEG terminology, such frames are called ***I-pictures***, where I stands for "intra". Difference frames that use previous frames are called ***P-pictures***, or "predictive pictures". P-pictures can be based on an earlier I-picture or P-picture – that is, differences can be cumulative.

Often, though, we may be able to do better, because pictures are composed of objects that move as a whole: a person might walk along a street, a football might be kicked, or the camera might pan across a landscape with trees. Figure 6.7 is a schematic illustration of this sort of motion, to demonstrate how it affects compression. In the two frames shown here, the fish swims from left to right. Pixels therefore change in the region originally occupied by the fish – where the background becomes visible in the second frame – and in the region to which the fish moves. The black area in the picture at the bottom left of Figure 6.7 shows the changed area which would have to be stored explicitly in a difference frame.

However, the values for the pixels in the area occupied by the fish in the second frame are all there in the first frame, in the fish's old position. If we could somehow identify the coherent area corresponding to the fish, we would only need to record its displacement together with the changed pixels in the smaller area shown at the bottom right of Figure 6.7. (The bits of weed and background in this region are not present in the first frame anywhere, unlike the fish.) This technique of incorporating a record of the relative displacement of objects in the difference frames is called ***motion compensation*** (also known as ***motion estimation***). Of course, it is now necessary to store the displacement as part of the compressed file. This information can be recorded as a ***displacement vector***, giving the number of pixels the object has moved in each direction.

If we were considering some frames of video shot under water showing a real fish swimming among weeds (or a realistic animation of such a scene) instead of these schematic pictures, the objects and their movements would be less simple than they appear in Figure 6.7. The fish's body would change shape as it propelled itself, the lighting would alter, the weeds would not stay still.

Attempting to identify the objects in a real scene and apply motion compensation to them would not work, therefore (even if it were practical to identify objects in such a scene).

MPEG-1 compressors do not attempt to identify discrete objects in the way that a human viewer would. Instead, they divide each frame into blocks of 16×16 pixels known as ***macroblocks*** (to distinguish them from the smaller blocks used in the DCT phase of compression), and attempt to predict the whereabouts of the corresponding macroblock in the next frame. No high-powered artificial intelligence is used in this prediction: all possible displacements within a limited range are tried, and the best match is chosen. The difference frame is then constructed by subtracting each macroblock from its predicted counterpart, which should result in fewer non-zero pixels, and a smaller difference frame after spatial compression.

The price to be paid for the additional compression resulting from the use of motion compensation is that, in addition to the difference frame, we now have to keep a record of the motion vectors describing the predicted displacement of macroblocks between frames. These can be stored relatively efficiently, however. The motion vector for a macroblock is likely to be similar

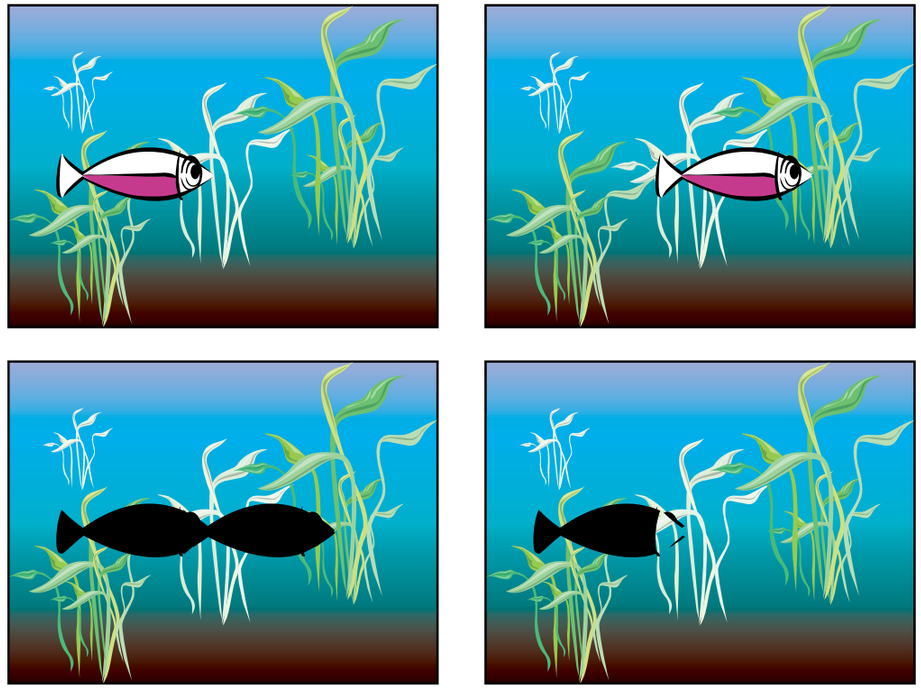


Figure 6.7. ***Motion compensation***

or identical to the motion vector for adjoining macroblocks (since these will often be parts of the same object), so, by storing the differences between motion vectors, additional compression, analogous to inter-frame compression, is achieved.

Although basing difference frames on preceding frames probably seems the obvious thing to do, it can be more effective to base them on following frames. Figure 6.8 shows why such backward

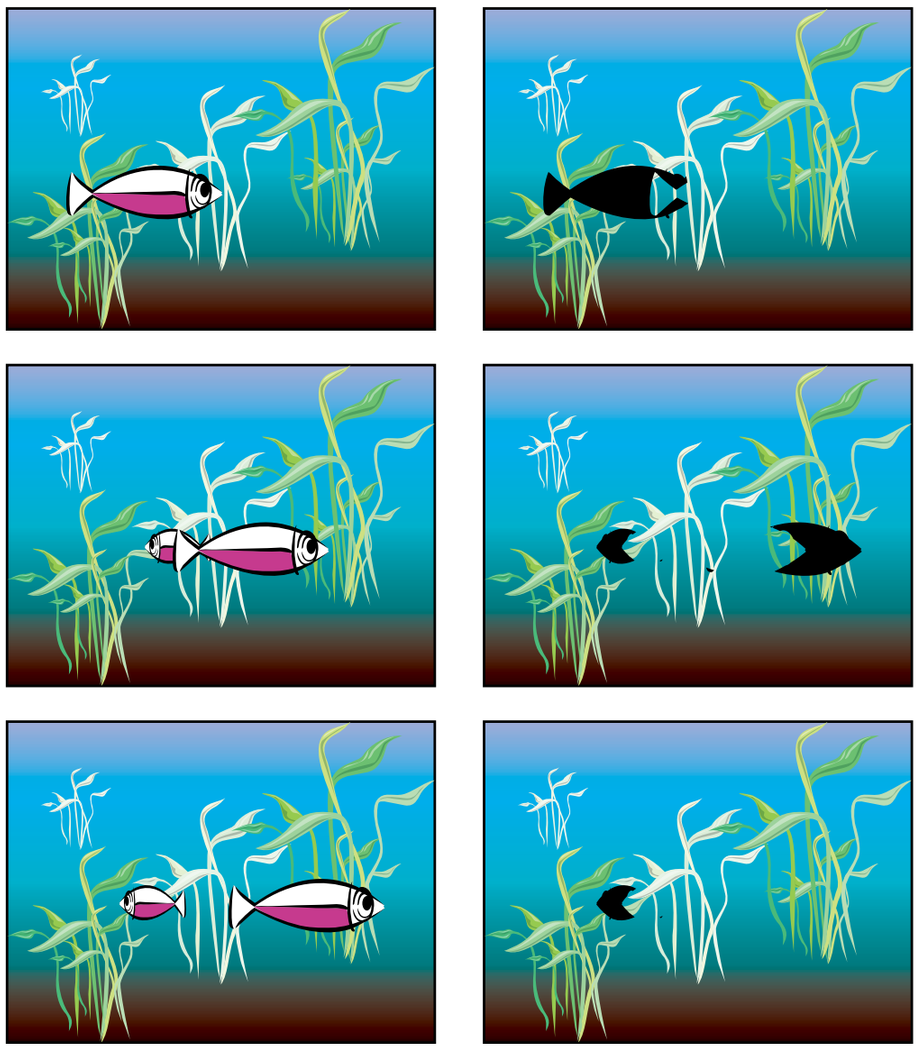


Figure 6.8. *Bi-directional prediction*

prediction can be useful. In the top frame, the smaller fish that is partially revealed in the middle frame is hidden, but it is fully visible in the bottom frame. If we construct an I-picture from the first two frames, it must explicitly record the area covered by the fish in the first frame but not the second, as before. If we construct the I-picture by working backwards from the third frame instead, the area that must be recorded consists of the parts of the frame covered up by either of the fish in the third frame but not in the second. Motion compensation allows us to fill in the bodies of both fish in the I-picture. The resulting area, shown in the middle of the right-hand column of Figure 6.8, is slightly smaller than the one shown at the top right. If we could use information from both the first and third frames in constructing the I-picture for the middle frame, almost no pixels would need to be represented explicitly, as shown at the bottom right. This comprises the small area of background that is covered by the big fish in the first frame and the small fish in the last frame, excluding the small fish in the middle frame, which is represented by motion compensation from the following frame. To take advantage of information in both preceding and following frames, MPEG compression allows for ***B-pictures***, which can use motion compensation from the previous or next I- or P-pictures, or both, hence their full name "bi-directionally predictive" pictures.

A video sequence can be encoded in compressed form as a sequence of I-, P- and B-pictures. It is not a requirement that this sequence be regular, but encoders typically use a repeating sequence, known as a ***Group of Pictures*** or ***GOP***, which always begins with an I-picture. Figure 6.9 shows a typical example. (You should read it from left to right.) The GOP sequence is `IBBPBB`. The diagram shows two such groups: frames 01 to 06 and frames 11 to 16. The arrows indicate the forward and bi-directional prediction. For example, the P-picture 04 depends on the I-picture 01 at the start of its GOP; the B-pictures 05 and 06 depend on the preceding P-picture 04 and the following I-picture 11.

All three types of picture are compressed using the MPEG-1 DCT-based compression method. Published measurements indicate that, typically, P-pictures compress three times as much as I-pictures, and B-pictures one and a half times as much as P-pictures. However, reconstructing B-pictures is more complex than reconstructing the other types, so there is a trade-off to be made between compression and computational complexity when choosing the pattern of a GOP.

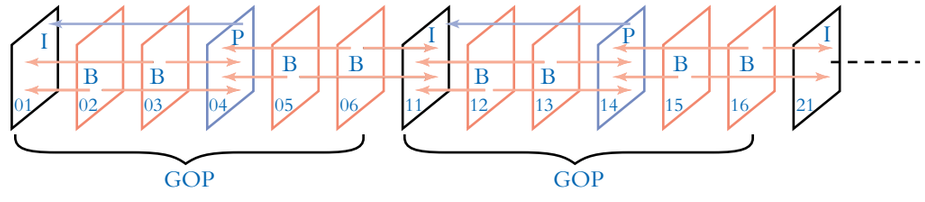


Figure 6.9. *An MPEG sequence in display order*

Exercises

Test Questions

1 What are the advantages and disadvantages of using a scanner or a digital stills camera to capture traditional art work as animation sequences?

2 When would it be appropriate to use an animated GIF for an animation sequence? What shortcomings of animated GIFs limit their usefulness?

3 What problems are associated with using linear methods to interpolate motion between key frames in animations? Explain how Bézier curves are used to overcome these problems.

4 Explain why bitmapped animations that use interpolation are no smaller than those that don't, but vector animations that use interpolation may be much smaller than those that don't.

5 Describe which properties of (a) a bitmapped animation and (b) a vector animation you could expect to be able to interpolate. Explain why there is a difference between the two.

6 Describe the motion of an object whose position is animated in After Effects using Bézier interpolation for the motion path, and linear interpolation for the velocity.

Discussion Topics

1 If an animation sequence is to be saved in a video format, what factors will influence your choice of codec? Under what circumstances would it be appropriate to treat the animated sequence exactly like a live-action video sequence?

2 The term "key frame" is used in connection with both animation and video. What are the similarities and differences between its meanings in the two contexts?

3 Will creating motion one frame at a time always produce a more convincing illusion of movement than using interpolation? Explain the reasons for your answer.

Practical Tasks

1 Create a short animation, similar to the bouncing ball example of Figure 7.8, which uses a motion path. Recreate the same animation, this time without using a motion path. Describe the difference between the two methods and the two results.

2 Flash has a `Trace Bitmap` command, which can be used to convert bitmapped images into vector graphics. Find out how this command works, and use it to convert a short video clip

that you import into Flash into a vector animation. Compare the size of the result with the original video. Experiment with changing parameters to the tracing operation, and see what effect they have on the appearance of the traced clip and the size of the final movie.

3 Create a very simple title for a video clip as a single image in a bitmapped graphics application such as Photoshop, and save it as a still image file. Using whatever tools are available (Photoshop Extended, Premiere, After Effects, etc.), create a pleasing 10-second title sequence by simply applying time-varying effects and filters to this single image. (If you want to go for a more sophisticated result, and have the necessary tools, you might create your original image on several layers and animate them separately.)

4 The countdown sequence illustrated in Figure 7.14 was created in After Effects. Create your own countdown that uses similar motion graphics in Flash.

Formats

Most of the development of digital audio has taken place in the recording and broadcast industries, where the emphasis is on physical data representations and data streams for transmission and playback. There are standards in these areas that are widely adhered to. The use of digital sound on computers is a much less thoroughly regulated area, where a wide range of incompatible proprietary formats and *ad hoc* standards can be found. Each of the three major platforms has its own sound file format: AIFF for MacOS, AU for other varieties of Unix, and WAV (or WAVE) for Windows, but support for all three is common on all platforms.

The standardizing influence of the Internet has been less pronounced in audio than it is in graphics. MP3 files have been widely used for downloading and storing music on computers and mobile music players. "Podcasts" typically use MP3 as the format for the audio that they deliver. The popularity of music-swapping services using MP3 led to its emergence as the leading audio format on the Internet, but QuickTime and Windows Media are used as container formats for audio destined for Apple's iPod music players and various devices that incorporate Windows Media technology. On Web pages, Flash movies are sometimes used for sound, because of the wide deployment of the Flash Player. It is possible to embed sound in PDF documents, but the actual playing of the sound is handled by other software, such as QuickTime, so MP3 is a good choice of format here, too, because it can be played on all the relevant platforms.

MP3 has its own file format, in which the compressed audio stream is split into chunks called "frames", each of which has a header, giving details of the bit rate, sampling frequency and other parameters. The file may also include metadata tags, oriented towards musical content, giving the title of a track, the artist performing it, the album from which it is taken, and so on. As we will describe later in this chapter, MP3 is primarily an encoding, not a file format, and MP3 data may be stored in other types of file. In particular, QuickTime may include audio tracks encoded with MP3, and Flash movies use MP3 to compress any sound they may include.

In Chapter 6, we explained that streamed video resembles broadcast television. Streamed audio resembles broadcast radio – that is, sound is delivered over a network and played as it arrives, without having to be stored on the user's machine first. As with video, this allows live transmission and the playing of files that are too big to be held on an average-sized hard disk. Because of the lower bandwidth required by audio, streaming is more successful for sound than it is for video. Streaming QuickTime can also be used for audio, on its own as well as accompanying video. QuickTime includes an AAC codec for high-quality audio. Windows Media audio can also be streamed. Both of these formats, as well as MP3, are used for broadcasting live concerts and for the Internet equivalent of radio stations.

KEY POINTS

Sounds are produced by the conversion of energy into vibrations in the air or some other elastic medium, which are detected by the ear and converted into nerve impulses which we experience as sound.

A sound's frequency spectrum is a description of the relative amplitudes of its frequency components.

The human ear can detect sound frequencies roughly in the range 20 Hz to 20 kHz, though the ability to hear the higher frequencies is lost as people age.

A sound's waveform shows how its amplitude varies over time.

Perception of sound has a psychological dimension.

CD audio is sampled at 44.1 kHz. Sub-multiples of this value may be used for low-quality digital audio. Some audio recorders use sampling rates that are multiples of 48 kHz.

Audio sampling relies on highly accurate clock pulses to prevent jitter.

Frequencies greater than half the sampling rate are filtered out to avoid aliasing.

CD audio uses 16-bit samples to give 65,536 quantization levels.

Quantization noise can be mitigated by dithering, i.e. adding a small amount of random noise which softens the sharp transitions of quantization noise.

Sound may be stored in AIFF, WAV or AU files, but on the Internet the MP3 format is dominant. MP3 data may be stored in QuickTime and Flash movies.

Processing Sound

With the addition of suitable audio input, output and processing hardware and software, a desktop computer can perform the functions of a modern multi-track recording studio. Such professional facilities are expensive and demanding on resources, as you would expect. They are also as complex as a recording studio, with user interfaces that are as intimidating to the novice as the huge mixing consoles of conventional studios. Fortunately, for multimedia, more modest facilities are usually adequate.

There is presently no single sound application that has the *de facto* status of a cross-platform desktop standard, in the way that Photoshop and Dreamweaver, for example, have in their respective fields. Several different packages are in use, some of which require special hardware support. Most of the well-known ones are biased towards music, with integrated support for MIDI sequencing (as described later in this chapter) and multi-track recording.

Several more modest programs, including some Open Source applications, provide simple recording and effects processing facilities. A specialized type of audio application has recently achieved some popularity among people who are not audio professionals. Apple's GarageBand and Adobe Soundbooth exemplify this type of program. They provide only primitive facilities for recording, importing and editing sound, and only a few of the effects that are found in professional software. Their novelty lies in facilities for creating songs. In the case of GarageBand, this is done by combining loops, which may either be recorded live instruments, or synthesized. In Soundbooth, templates consisting of several musical segments may be customized – for example by changing the orchestration or dynamics, or by rearranging the segments – to produce unique "compositions", which might serve as adequate soundtracks for corporate presentations, home videos and similar undemanding productions.

Video editing packages usually include some integrated sound editing and processing facilities, and some offer basic sound recording. These facilities may be adequate for multimedia production in the absence of special sound software, and are especially convenient when the audio is intended as a soundtrack to accompany picture.

Given the absence of an industry standard sound application for desktop use, we will describe the facilities offered by sound programs in general terms only, without using any specific example.

Recording and Importing Sound

Many desktop computers are fitted with built-in microphones, and it is tempting to think that these are adequate for recording sounds. It is almost impossible to obtain satisfactory results with these, however – not only because the microphones themselves are usually small and cheap, but because they are inevitably close to the machine's fan and disk drives, which means that they will pick up noises from these components. It is much better to plug an external microphone into a sound card, but if possible, you should do the actual recording using a dedicated device, such as a solid-state audio recorder, and a professional microphone, and capture it in a separate operation. Compression should be avoided at this stage. Where sound quality is important, or for recording music to a high standard, it will be necessary to use a properly equipped studio. Although a computer can form the basis of a studio, it must be augmented with microphones and other equipment in a suitable acoustic environment, so it is not really practical for a multimedia producer to set up a studio for one-off recordings. It may be necessary to hire a professional studio, which offers the advantage that professional personnel will generally be available.

Before recording, it is necessary to select a sampling rate and sample size. Where the sound originates in analogue form, the choice will be determined by considerations of file size and bandwidth, which will depend on the final use to which the sound is to be put, and the facilities available for sound processing. As a general rule, the highest possible sampling rate and sample size

should be used, to minimize deterioration of the signal when it is processed. If a compromise must be made, the effect on quality of reducing the sample size is more drastic than that of reducing the sampling rate. The same reduction in size can be produced by halving the sampling rate or halving the sample size, but the former is the better option. If the signal is originally a digital one – the digital output from a solid-state recorder, for example – the sample size should be matched to the incoming rate, if possible.

A simple calculation suffices to show the size of digitized audio. The sampling rate is the number of samples generated each second, so if the rate is r Hz and the sample size is s bits, each second of digitized sound will occupy $rs/8$ bytes. Hence, for CD quality, $r = 44.1 \times 10^3$ and $s = 16$, so each second occupies just over 86 kbytes (86×1024 bytes), each minute roughly 5 Mbytes. These calculations are based on a single channel, but audio is almost always recorded in stereo, so the estimates should be doubled. Conversely, where stereo effects are not required, the space occupied can be halved by recording in mono.

The most vexatious aspect of recording is getting the levels right. If the level of the incoming signal is too low, the resulting recording will be quiet, and more susceptible to noise. If the level is too high, ***clipping*** will occur – that is, at some points, the amplitude of the incoming signal will exceed the maximum value that can be recorded. The value of the corresponding sample will be set to the maximum, so the recorded waveform will apparently be clipped off straight at this threshold. (Figure 8.11 shows the effect on a pure sine wave.) The result is heard as a particularly unpleasant sort of distortion.

Ideally, a signal should be recorded at the highest possible level that avoids clipping. Sound applications usually provide level meters, so that the level can be monitored, with clipping alerts. Where the sound card supports it, a gain control can be used to alter the level. If this is not available, the only option is to adjust the output level of the equipment from which the signal originates.

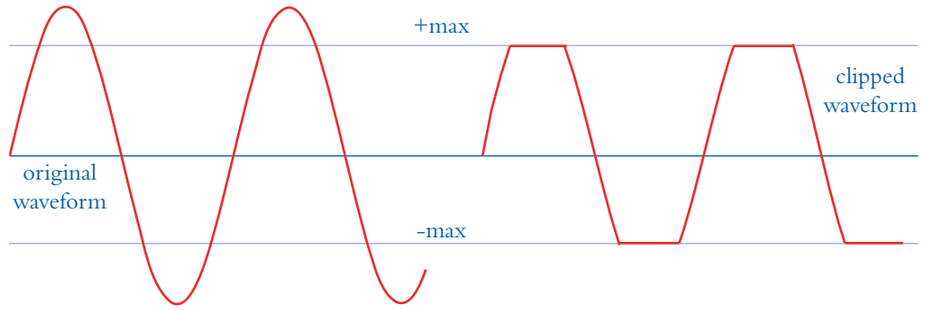


Figure 8.11. ***Clipping***

Text has a dual nature: it is a visual representation of language, and a graphic element in its own right. Text in digital form must also be a representation of language – that is, we need to relate bit patterns stored in a computer's memory or transmitted over a network to the symbols of a written language (either a natural one or a computer language). When we consider the display of stored text, its visual aspect becomes relevant. We then become concerned with such issues as the precise shape of characters, their spacing, and the layout of lines, paragraphs and larger divisions of text on the screen or page. These issues of display are traditionally the concern of the art of typography. Much of the accumulated typographical practice of the last several centuries can be adapted to the display of the textual elements of multimedia.

In this chapter, we consider how the fundamental units of written languages – characters – can be represented in a digital form, and how the digital representation of characters can be turned into a visual representation for display and laid out on the screen. We will show how digital font technology and markup make it possible to approximate the typographical richness of printed text in the textual components of multimedia.

Character Sets

In keeping with text's dual nature, it is convenient to distinguish between the lexical content of a piece of text and its appearance. By content we mean the characters that make up the words and other units, such as punctuation or mathematical symbols. (At this stage we are not considering "content" in the sense of the meaning or message contained in the text.) The appearance of the text comprises its visual attributes, such as the precise shape of the characters, their size, and the way the content is arranged on the page or screen. For example, the content of the following two sentences from the short story *Jeeves and the Impending Doom* by P.G. Wodehouse is identical, but their appearance is not:

```
The Right Hon was a tubby little chap who looked as if he had
been poured into his clothes and had forgotten to say 'When!'
```

THE RIGHT HON was a tubby little chap who looked as if he had been *poured* into his clothes and had forgotten to say 'When!'

We all readily understand that the first symbol in each version of this sentence is a capital T, even though one is several times as large as the other, is much darker, has some additional strokes, and extends down into the line below. To express their fundamental identity, we distinguish between

an abstract character and its graphic representations, of which there is a potentially infinite number. Here, we have two graphic representations of the same abstract character (the letter T).

As a slight over-simplification, we could say that the content is the part of a text that carries its meaning or semantics, while the appearance is a surface attribute that may affect how easy the text is to read, or how pleasant it is to look at, but does not substantially alter its meaning. In the example just given, the fixed-width, typewriter-like font of the first version clearly differs from the more formal book font used for most of the second, but this and the initial dropped capital and use of different fonts do not alter the joke. Note, however, that the italicization of the word "poured" in the second version does imply an emphasis on the word that is missing in the plain version (and also in the original story), although we would normally consider italicization an aspect of the appearance like the use of the small caps for "Right Hon". So the distinction between appearance and content is not quite as clear-cut as one might think, but it is useful because it permits a separation of concerns between these two qualities that text possesses.

Abstract characters are grouped into ***alphabets***. Each particular alphabet forms the basis of the written form of a certain language or group of languages. We consider any set of distinct symbols to be an alphabet, but we do not define "symbol". In the abstract, an alphabet can be any set at all, but in practical terms, the only symbols of interest will be those used for writing down some language. This includes the symbols used in an ideographic writing system, such as those used for Chinese and Japanese, where each character represents a whole word or concept, as well as the phonetic letters of Western-style alphabets, and the intermediate syllabic alphabets, such as Korean Hangul. In contrast to colloquial usage, we include punctuation marks, numerals and mathematical symbols in an alphabet, and treat upper- and lower-case versions of the same letter as different symbols. Thus, for our purposes, the English alphabet includes the letters A, B, C, …, Z and a, b, c, … ,z, but also punctuation marks, such as comma and exclamation mark, the digits 0, 1, …, 9, and common symbols such as + and =.

To represent text digitally, it is necessary to define a mapping between (abstract) characters in some alphabet and values that can be stored in a computer system. As we explained in Chapter 2, the only values that we can store are bit patterns, which can be interpreted as integers to base 2, so the problem becomes one of mapping characters to integers. As an abstract problem this is trivial: any mapping will do, provided it associates each character of interest with exactly one number. Such an association is called – with little respect for mathematical usage – a ***character set***; its domain (the alphabet for which the mapping is defined) is called the ***character repertoire***. For each character in the repertoire, the character set defines a ***code value*** in its range, which is sometimes called the set of ***code points***. The character repertoire for a character set intended for written English text would include the 26 letters of the alphabet in both upper- and lower-case forms, as well as the 10 digits and the usual collection of punctuation marks. The character repertoire

for a character set intended for Russian would include the letters of the Cyrillic alphabet. Both of these character sets could use the same set of code points; provided that it was not necessary to use both character sets simultaneously (for example, in a bilingual document), a character in the English alphabet could have the same code value as one in the Cyrillic alphabet. The character repertoire for a character set intended for the Japanese Kanji alphabet must contain at least the 1945 ideograms for common use and 166 for names sanctioned by the Japanese Ministry of Education, and could contain over 6000 characters. Consequently, the Japanese Kanji alphabet requires far more distinct code points than an English or Cyrillic character set.

The mere existence of a character set is adequate to support operations such as editing and searching of text, since it allows us to store characters as their code values, and to compare two characters for equality by comparing the corresponding integers; it only requires some means of input and output. In simple terms, this means that it is necessary to arrange that when a key is pressed on a keyboard, or the equivalent operation is performed on some other input device, a command is transmitted to the computer, causing the bit pattern corresponding to the character for that key to be passed to the program currently receiving input. Conversely, when a value is transmitted to a monitor or other output device, a representation of the corresponding character should appear.

There are advantages to using a character set with some structure to it, instead of a completely arbitrary assignment of numbers to abstract characters. In particular, it is useful to use integers within a comparatively small range that can easily be manipulated by a computer. It can be helpful, too, if the code values for consecutive letters are consecutive numbers, since this simplifies some operations on text, such as sorting.

Standards

The most important consideration concerning character sets is standardization. Transferring text between different makes of computer, interfacing peripheral devices from different manufacturers and communicating over networks are everyday activities. Continual translation between different manufacturers' character codes would not be acceptable, so a standard character code is essential. The following description of character code standards is necessarily somewhat dry, but an understanding of them is necessary if you are to avoid the pitfalls of incompatibility and the resulting corruption of texts. Unfortunately, standardization is never a straightforward business, and the situation with respect to character codes remains somewhat unsatisfactory.

ASCII (American Standard Code for Information Interchange) was the dominant character set from the 1970s into the early twenty-first century. It uses 7 bits to store each code value, so there is a total of 128 code points. The character repertoire of ASCII only comprises 95 characters, however. The values 0 to 31 and 127 are assigned to ***control characters***, such as form-feed, carriage return and delete, which have traditionally been used to control the operation of output devices. The control characters are a legacy from ASCII's origins in early teletype character sets. Many of them no longer have any useful meaning, and are often appropriated by application programs for their own purposes. Figure 9.1 shows the ASCII character set. (The character with code value 32 is a space.)

32		33	!	34	"	35	#
36	$	37	%	38	&	39	'
40	(	41	)	42	*	43	+
44	,	45	-	46	.	47	/
48	0	49	1	50	2	51	3
52	4	53	5	54	6	55	7
56	8	57	9	58	:	59	;
60	<	61	=	62	>	63	?
64	@	65	A	66	B	67	C
68	D	69	E	70	F	71	G
72	H	73	I	74	J	75	K
76	L	77	M	78	N	79	O
80	P	81	Q	82	R	83	S
84	T	85	U	86	V	87	W
88	X	89	Y	90	Z	91	[
92	\	93	]	94	^	95	_
96	`	97	a	98	b	99	c
100	d	101	e	102	f	103	g
104	h	105	i	106	j	107	k
108	l	109	m	110	n	111	o
112	p	113	q	114	r	115	s
116	t	117	u	118	v	119	w
120	x	121	y	122	z	123	{
124	|	125	}	126	~		

Figure 9.1. ***The printable ASCII characters***

American English is one of the few languages in the world for which ASCII provides an adequate character repertoire. Attempts by the standardization bodies to provide better support for a wider range of languages began when ASCII was adopted as an ISO standard (ISO 646) in 1972. ISO 646 incorporates several national variants on the version of ASCII used in the United States, to accommodate, for example, some accented letters and national currency symbols.

A standard with variants is no real solution to the problem of accommodating different languages. If a file prepared in one country is sent to another and read on a computer set up to use a different national variant of ISO 646, some of the characters will be displayed incorrectly. For example, a hash character (#) typed in the United States would be displayed as a pound sign (£) in the UK (and vice versa) if the British user's computer used the UK variant of ISO 646. (More likely, the hash would display correctly, but the Briton would be unable to type a pound sign, because it is more convenient to use US ASCII (ISO 646-US) anyway, to prevent such problems.)

A better solution than national variants of the 7-bit ISO 646 character set lies in the provision of a character set with more code points, such that the ASCII character repertoire is mapped to the values 0–127, thus assuring compatibility, and additional symbols required outside the USA or for

160		161	¡	162	¢	163	£
164	¤	165	¥	166	¦	167	§
168	¨	169	©	170	ª	171	«
172	¬	173	-	174	®	175	¯
176	°	177	±	178	²	179	³
180	´	181	µ	182	¶	183	·
184	¸	185	¹	186	º	187	»
188	¼	189	½	190	¾	191	¿
192	À	193	Á	194	Â	195	Ã
196	Ä	197	Å	198	Æ	199	Ç
200	È	201	É	202	Ê	203	Ë
204	Ì	205	Í	206	Î	207	Ï
208	Ð	209	Ñ	210	Ò	211	Ó
212	Ô	213	Õ	214	Ö	215	×
216	Ø	217	Ù	218	Ú	219	Û
220	Ü	221	Ý	222	Þ	223	ß
224	à	225	á	226	â	227	ã
228	ä	229	å	230	æ	231	ç
232	è	233	é	234	ê	235	ë
236	ì	237	í	238	î	239	ï
240	ð	241	ñ	242	ò	243	ó
244	ô	245	õ	246	ö	247	÷
248	ø	249	ù	250	ú	251	û
252	ü	253	ý	254	þ	255	ÿ

Figure 9.2. ***The top part of the ISO Latin1 character set***

specialized purposes are mapped to other values. Doubling the set of code points was easy: the seven bits of an ASCII character are invariably stored in an 8-bit byte. It was originally envisaged that the remaining bit would be used as a parity bit for error detection. As data transmission became more reliable, and superior error checking was built in to higher-level protocols, this parity bit fell into disuse, effectively becoming available as the high-order bit of an 8-bit character.

Predictably, the different manufacturers each developed their own incompatible 8-bit extensions to ASCII. These all shared some general features: the lower half (code points 0–127) was identical to ASCII; the upper half (code points 128–255) held accented letters and extra punctuation and mathematical symbols. Since a set of 256 values is insufficient to accommodate all the characters required for every alphabet in use, each 8-bit character code had different variants; for example, one for Western European languages, another for languages written using the Cyrillic script, and so on. (Under MS-DOS and Windows, these variants are called "code pages.")

Despite these commonalities, the character repertoires and the code values assigned by the different manufacturers' character sets are different. For example, the character é (e with an acute accent) has the code value 142 in the Macintosh Standard Roman character set, whereas it has the code value 233 in the corresponding Windows character set, in which 142 is not assigned as the value for any character; 233 in Macintosh Standard Roman, on the other hand, is É. Because the repertoires of the character sets are different, it is not even always possible to perform a translation between them, so transfer of text between platforms is problematical.

Clearly, standardization of 8-bit character sets was required. During the 1980s the multi-part standard ISO 8859 was produced. This defines a collection of 8-bit character sets, each designed to accommodate the needs of a group of languages (usually geographically related). The first part of the standard, ISO 8859-1, is usually referred to as ***ISO Latin1***, and covers most Western European languages. Like all the ISO 8859 character sets, the lower half of ISO Latin1 is identical to ASCII (i.e. ISO 646-US); the code points 128–159 are mostly unused, although a few are used for various diacritical marks. Figure 9.2 shows the 96 additional code values provided for accented letters and symbols. (The character with code value 160 is a "non-breaking" space.)

IN DETAIL

The Windows Roman character set (Windows 1252) is sometimes claimed to be the same as ISO Latin1, but it uses some of the code points between 128 and 159 for characters which are not present in ISO 8859-1's repertoire.

Other parts of ISO 8859 are designed for use with Eastern European languages, including Czech, Slovak and Croatian (ISO 8859-2 or Latin2), for languages that use the Cyrillic alphabet (ISO 8859-5), for modern Greek (ISO 8859-7), Hebrew (ISO 8859-8), and others – there is a total of 10 parts to ISO 8859 with more projected, notably an ISO Latin0, which includes the Euro currency symbol.

ISO 8859-1 has been used extensively on the World Wide Web for pages written in languages that use the alphabet it supports, but manufacturers' proprietary non-standard character sets have remained in use. There is a fundamental problem with 8-bit character sets, which has prevented ISO 8859's universal adoption: 256 is not enough code points – not enough to represent ideographically based alphabets, and not enough to enable us to work with several languages at a time (unless they all happen to use the same variant of ISO 8859). Newer standards that are not restricted to so few code points are rendering ISO 8859 obsolete.

Unicode and ISO 10646

The only possible solution to the problem of insufficient code points is to use more than one byte for each code value. A 16-bit character set has 65,536 code points – putting it another way, it can accommodate 256 variants of an 8-bit character set simultaneously. Similarly, a 24-bit character set can accommodate 256 16-bit character sets, and a 32-bit character set can accommodate 256 of those. ISO (in conjunction with the IEC) set out to develop a 32-bit ***Universal Character Set (UCS)***, designated ISO 10646, structured in this way: a collection of 2^{32} characters can be arranged as a hypercube (a four-dimensional cube) consisting of 256 groups, each of which consists of 256 planes of 256 rows, each comprising 256 characters (which might be the character repertoire of an 8-bit character set). The intention was to organize the immense character repertoire allowed by a 32-bit character set with alphabets distributed among the planes in

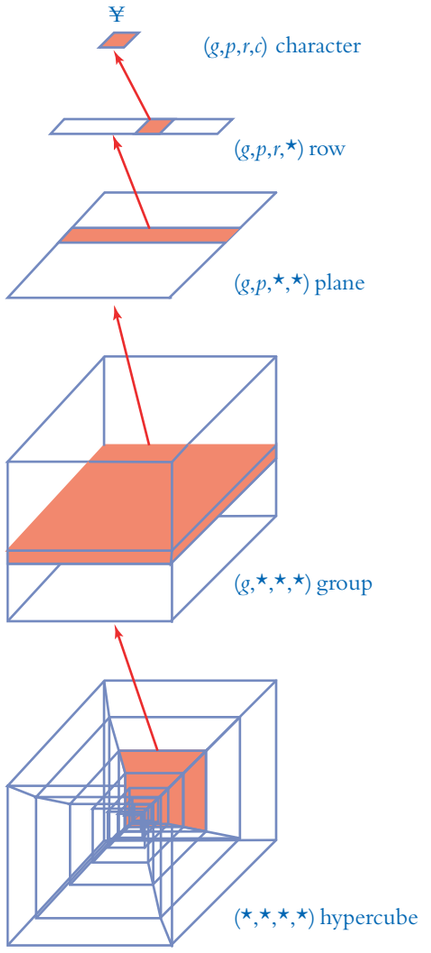


Figure 9.3. ***The structure of ISO 10646***

a linguistically sensible way, so that the resulting character set would have a clear logical structure. Each character can be identified by specifying its group *g*, its plane *p*, and a row *r* and column *c* (see Figure 9.3). Each of *g*, *p*, *r* and *c* is an 8-bit quantity, which can fit in one byte; four bytes thus identify a unique character, so, inverting our viewpoint, the code value for any character is the 32-bit value which specifies its position within the hypercube.

To make the structure of the character set evident, we usually write code points as quadruples (*g*,*p*,*r*,*c*), which can be considered as the coordinates of a point in a four-dimensional space. By extension, such a quadruple also identifies a subset of the character set using a ⋆ to denote all values in the range 0–255. Thus (0,0,0,⋆) is the subset with all but the lowest-order byte zero. In ISO 10646 this subset is identical to ISO Latin1.

At the same time as ISO was developing this elegant framework for its character set, an industry consortium was working on a 16-bit character set, known as *Unicode*. As we noted above, a 16-bit character set has 65,536 code points. This is not sufficient to accommodate all the characters required for Chinese, Japanese and Korean scripts in discrete positions. These three languages and their writing systems share a common ancestry, so there are thousands of identical ideographs in their scripts. The Unicode committee adopted a process they called ***CJK consolidation***,† whereby characters used in writing Chinese, Japanese and Korean are given the same code value if they look the same, irrespective of which language they belong to, and whether or not they mean the same thing in the different languages. There is clearly a cultural bias involved here, since the same process is not applied to, for example, upper-case A and the Greek capital alpha, which are identical in appearance but have separate Unicode code values. The pragmatic justification is that, with Chinese, Japanese and Korean, thousands of characters are involved, whereas with the European and Cyrillic languages, there are relatively few. Furthermore, consolidation of those languages would interfere with compatibility with existing standards.

† *Some documents use the name "Han unification" instead.*

Unicode provides code values for all the characters used to write contemporary "major" languages, as well as the classical forms of some languages. The alphabets available include Latin, Greek, Cyrillic, Armenian, Hebrew, Arabic, Devanagari, Bengali, Gurmukhi, Gujarati, Oriya, Tamil, Telugu, Kannada, Malayalam, Thai, Lao, Georgian and Tibetan, as well as the Chinese, Japanese and Korean ideograms and the Japanese and Korean phonetic and syllabic scripts. Unicode also includes punctuation marks, technical and mathematical symbols, arrows and the miscellaneous symbols usually referred to as "dingbats" (pointing hands, stars, and so on). In addition to the accented letters included in many of the alphabets, separate diacritical marks (such as accents and tildes) are available and a mechanism is provided for building composite characters by combining these marks with other symbols. (This not only provides an alternative way of making accented letters, it also allows for the habit mathematicians have of making up new symbols by decorating old ones.)

In Unicode, code values for nearly 39,000 symbols are provided, leaving some code points unused. Others are reserved for the UTF-16 expansion method (described briefly later on), while a set of 6400 code points is reserved for private use, allowing organizations and individuals to define codes for their own use. Even though these codes are not part of the Unicode standard, it is guaranteed that they will never be assigned to any character by the standard, so their use will never conflict with any standard character, although it might conflict with those of other individuals.

Unicode is restricted to characters used in text. It specifically does not attempt to provide symbols for music notation or other symbolic writing systems that do not represent language.

Unicode and ISO 10646 were brought into line in 1991 when the ISO agreed that the plane (0,0,⋆,⋆), known as the ***Basic Multilingual Plane (BMP)***, should be identical to Unicode. ISO 10646 thus utilizes CJK consolidation, even though its 32-bit code space does not require it to do so. The overwhelming advantage of this arrangement is that the two standards are compatible (and the respective committees have pledged that they will remain so). To understand how it is possible to take advantage of this compatibility, we must introduce the concept of a character set ***encoding***.

An encoding is another layer of mapping, which transforms a code value into a sequence of bytes for storage and transmission. When each code value occupies exactly one byte it might seem that the only sensible encoding is an identity mapping where each code value is stored or sent as itself in a single byte. Even in this case, though, a more complex encoding may be required. Because 7-bit ASCII was the dominant character code for such a long time, there are network protocols which assume that all character data is ASCII and remove or mangle the top bit of any 8-bit byte. To avoid this it may be necessary to encode 8-bit characters as sequences of 7-bit characters.

XHTML Elements and Attributes

We are now in a position to describe the XHTML tags that can be used to mark up text, and the CSS properties that can be used to control its layout. Although we are describing these particular languages, you should appreciate that the underlying principles of markup and layout apply to any system of text preparation.

Please note that the account which follows is not exhaustive. The scope of this book does not allow us to provide a full tutorial on XHTML and CSS, or a definitive reference guide. More details can be found in *Web Design: A Complete Introduction*, or in the detailed reference material listed on the supporting Web site.

The HTML 4.0 specification – and thus the XHTML 1.0 specification – defines 91 elements, of which 10 are deprecated, since there are now preferred ways of achieving the same effect. (Many attributes are also deprecated, even for elements which are not.) Only a few of these elements are concerned purely with text layout. Those that are can conveniently be divided into ***block-level*** and ***inline*** elements. Block-level elements are those which are normally formatted as discrete blocks, such as paragraphs – i.e. their start and end are marked by line breaks. Inline elements do not cause such breaks; they are run in to the surrounding text. Thus, the distinction corresponds to the general distinction between block and inline formatting described in Chapter 9.

IN DETAIL

There are three DTDs for XHTML 1.0: Strict, Transitional and Frameset. The Strict DTD excludes the deprecated elements and attributes, whereas the Transitional DTD, which is intended as a temporary expedient to make it easier to transform older HTML documents to XHTML, permits the deprecated features to be used.

The Frameset DTD includes an additional feature that allows a Web page to be created from a set of independent documents. The page is divided into "frames", one for each document, which can be updated independently. Frames cause usability problems, and their use has declined as CSS features can be used to achieve the most common layouts that frames were formerly used for.

The Strict DTD should always be used unless there are compelling reasons to use one of the others.

The most frequently used block-level textual element is the paragraph (p) element, which we have looked at already. Other block-level elements concerned purely with text layout include level 1 to level 6 headers, with element names h1, h2, …, h6, br which causes a line break, and hr the horizontal rule (straight line) element, which is sometimes used as a visual separator. The blockquote element is used for long quotations, which are normally displayed as indented paragraphs. Note, though,

that using blockquote as a way of producing an indented paragraph is an example of the sort of structural markup abuse that should be avoided: markup is not intended to control layout. This being so, the pre element, which is used for "pre-formatted" text and causes its content to be displayed exactly as it is laid out, is something of an anomaly, yet may be useful when the other available elements do not serve and elaborate stylesheet formatting is not worthwhile.

The only elaborate structures that XHTML supports as block-level elements are lists and tables. Tables are relatively complex constructions (as they must be, to accommodate the range of layouts commonly used for tabulation), but since their use is somewhat specialized we omit any detailed description. Lists, in contrast, are quite simple. XHTML provides three types: "ordered" lists, in the form of ol elements, "unordered" lists, ul elements, and "definition" lists, dl elements. Both ol and ul elements contain a sequence of list items (li elements), which are laid out appropriately, usually as separate blocks with hanging indentation. The difference is that, by default, user agents will automatically number the items in an ordered list. The items in an unordered list are marked by some suitable character, often a bullet. The distinction is somewhat arbitrary: all lists are ordered, in the sense that the items appear in a definite order. CSS rules can be used to number items automatically or insert bullets in front of them in either kind of list, but lists are often laid out and styled in a completely different way. If the list is being used structurally as a container for a sequence of items, it is conventional to use a ul element, with ol being reserved for lists where numbering is part of the semantics, such as a list of the 10 best-selling books on multimedia.

The items of a dl element are somewhat different, in that each consists of two elements – a term (dt) and a definition (dd). The intended use of a dl is, as its name suggests, to set lists of definitions. Typically each item consists of a term being defined, which will often be exdented, followed by its definition. Figure 10.3 shows the default appearance of lists produced by the following XHTML fragment. Note that a list item element can contain a list, giving nested lists.

- first item, but not numbered 1;
- second item, but not numbered 2;
- the third item contains a list, this time a numbered one:
  1. first numbered sub-item;
  2. second numbered sub-item;
  3. third numbered sub-item;
- fourth item, but not numbered 4;

ONE
    the first cardinal number;
TWO
    the second cardinal number;
THREE
    the third cardinal number

Figure 10.3. ***Default display of XHTML lists in a browser***

```
<ul>
   <li>first item, but not numbered 1;</li>
   <li>second item, but not numbered 2;</li>
   <li>the third item contains a list, this time a numbered one:
   <ol>
      <li>first numbered sub-item;</li>
      <li>second numbered sub-item;</li>
```

```
        <li>third numbered sub-item;</li>
    </ol></li>
    <li>fourth item, but not numbered 4;</li>
</ul>
<dl>
    <dt>ONE</dt><dd>the first cardinal number;</dd>
    <dt>TWO</dt><dd>the second cardinal number;</dd>
    <dt>THREE</dt><dd>the third cardinal number</dd>
</dl>
```

The most abstract block-level element is div, which simply identifies a division within a document that is to be treated as a unit. Usually, a division is to be formatted in some special way. The class attribute is used to identify types of division, and a stylesheet can be used to apply formatting to everything that falls within any division belonging to that class. We will see some examples in the following sections. Even in the absence of a stylesheet, classes of divisions can be used to express the organizational structure of a document. However, div elements should not be over-used; applying rules to other elements using contextual selectors (which we will describe later) is often more efficient.

Inline elements are used to specify formatting of phrases within a block-level element. It might seem that they are therefore in conflict with the intention of structural markup. However, it is possible to identify certain phrases as having special significance that should be expressed typographically without compromising the principle of separating structure from appearance. Examples of elements that work in this way are em for emphasis, and strong for strong emphasis. Often the content of these elements will be displayed by a visual user agent as italicized and bold text, respectively, but they need not be. In contrast, the i and b elements explicitly specify italic and bold text. These two elements are incompatible with structural markup and should be avoided (especially since a stylesheet can be used to change their effect).

There is an inline equivalent to div: a span element identifies a sequence of inline text that should be treated in some special way. In conjunction with the class attribute, span can be used to apply arbitrary formatting to text.

All the elements we have described can possess a class attribute, which permits subsetting. Additionally, each may have an id attribute, which is used to specify an identifier for a particular occurrence of the element. For example,

```
<ul id="navigation">
```

is the start tag of a list identified as navigation. The values of id attributes must be unique within

a single document, so that each value identifies exactly one element. This identifier can be used in various ways, one of which is in a CSS selector, where it must be prefixed by a # symbol instead of the dot used for classes. For example,

```
#navigation { text-indent: 6pc; }
```

will cause the list with its id set to navigation to be displayed with a special indent.

One important collection of elements, which we will not consider in detail here, is concerned with the construction of forms for data entry. XHTML provides a form element, within which you can use several special elements for creating controls, such as check boxes, radio buttons and text fields. We will look at how the data entered in such a form may be used as the input to a program running on a server in Chapter 16. For a more thorough description, consult *Web Design: A Complete Introduction*.

KEY POINTS

Block-level elements are normally formatted as discrete blocks; inline elements are run in to the surrounding text.

The block-level elements in XHTML include p (paragraph), h1–h6 (headings), br (line break), hr (horizontal rule), blockquote and pre (pre-formatted).

Unordered, ordered and definition lists are marked up as ul, ol or dl elements, which contain li elements (ul or ol) or pairs of dt (term) and dd (definition) elements.

Divisions of a document that should be treated as a unit are identified by div elements.

Inline elements include em (emphasis) and strong; span is used to identify arbitrary inline divisions.

Any element may have a class attribute, and/or an id attribute with a unique identifying value.

CSS Properties

CSS can be used to transform the sparse text markup provided in XHTML into an expressive and flexible formatting system. Again, note that the principles of layout – the values that can be changed and the ways they can be combined, for example – apply to many text preparation systems. Paragraph and character styles in a word processor or a DTP system such as InDesign perform a very similar function to that of stylesheets and are applied to the text of a document via its markup in a similar way. In turn, these formatting operations closely resemble the tasks that have been performed in setting type by hand for hundreds of years.

Headings stand out and are noticed because of a lack of similarity with the body text, but at the same time they are visually linked with it by their colour and proximity. In the gallery section towards the bottom of the page, symmetry is used for the arrows linking to the previous and next pages, and similarity of colour is used to indicate which is (and is not) the current page. (This is further reinforced by the use of a box around the current page number – an absence of similarity makes this stand out, as we will discuss shortly.) Finally, the images themselves appear as a group, reinforcing the idea of a gallery, because of their proximity and similarity of size and labelling.

It is necessary to ensure that any of the visual components of an interface or Web page which are linked conceptually will also seem linked visually, in the sort of way just described. Groupings will depend upon the nature of each case, but they will always be required. In the case of our geological slides application for example, illustrated in Figure 11.1 (and elsewhere in the book), it is essential to group the player controls together in the manner familiar from physical media players in order for the user to be able to find and use them quickly. The chaotic version of the interface illustrated in Figure 11.1 shows that it is not sufficient simply to have the controls available somewhere in the interface.

Gestalt principles should also be applied when designing the presentation of information or feedback to the user. Again, we saw in the chaotic version of the interface illustrated on the right of Figure 11.1 that presenting the readouts for angle of rotation and so on in different styles, sizes and colours would tend to mislead the user, suggesting importance or distinction for particular elements where none existed. We will discuss this phenomenon shortly.

IN DETAIL

The navbar is an excellent example of gestalt principles in operation which also illustrates the relationship between document structure and visual structure.

The very earliest Web sites had no navbars; HTML had (and XHTML 1.0 has) no element type suitable for grouping together a set of links. But designers seeking to make the links that define the top-level structure of a site conspicuous and easy to find soon came up with the idea of grouping them together in a distinctive style and a consistent position. This is a classic application of the gestalt principles of visual design.

This purely visual grouping led to the emergence of the navbar as a concept, which then led Web designers to use ad hoc markup to delimit navbars: first, a table, then a div element, finally a consensus has emerged that navbars should be marked up as unordered lists. They aren't, though. It is only with the inclusion of the nl (navigation list) and nav elements in the XHTML 2 and HTML 5 proposals, respectively, that markup has been provided to specify navbars as structural elements.

Visual Hierarchy

Visual hierarchy is concerned with the way in which particular elements may dominate a visual field. Having considered how the gestalt principles of visual perception explain perceived grouping, it is not difficult to understand that if we partially disrupt that grouping we will alter how a design is perceived and interpreted. Effective visual communication often depends upon inverting the gestalt principles so as to deliberately destroy grouping in order to make something stand out or to overturn an inherent dominance in a visual field. It is usually the anomalies in a design that create visual hierarchy. However, this principle can only be applied successfully within a structured whole. As we have already seen (in the example on the right of Figure 11.1), without structure there is simply chaos.

The simplest kind of visual hierarchy to understand is concerned with size, but it is important to realize that although size may be used, visual hierarchy does not always depend upon it. In any visual hierarchy there may be a range of different levels of prominence – a hierarchy is a structure which may contain many levels, although it does not necessarily have to contain more than two.

Figure 11.9 shows two simple examples using text alone. The text is identical in each case, so if we could not see the difference in presentation we would not realize that there was any distinction between the two. When we see the examples laid out in this way, however, we do not interpret them in the same way. In the left-hand example there is no visual guide to interpretation – all the words are similar in size (as well as font and colour) so we perceive them as a group in which each has equal importance, and interpretation is dependent upon the meaning of the words alone. In the right-hand example, though, we infer an emphasis on certain words (and the meaning of those words) from the fact that they are written in a larger (or smaller) size than the others. Certain words stand out and others recede precisely because of a lack of similarity. In this case visual hierarchy is established through the use of size alone, but the hierarchy is complex. For example, it would seem that tigers may be scarier than bears, perhaps, and both of them scarier than lions,

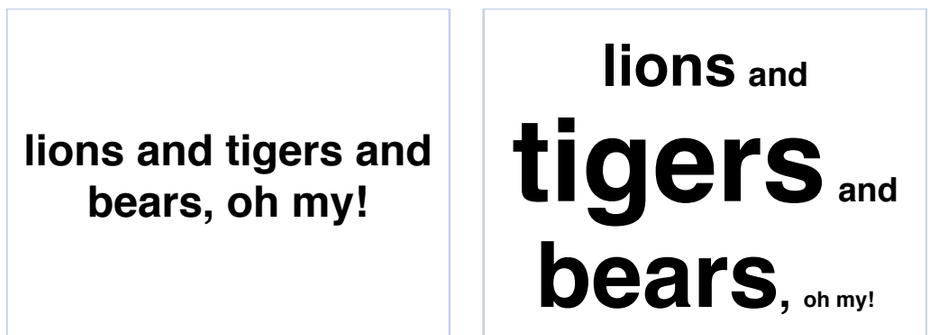


Figure 11.9. ***Expressing hierarchical emphasis through type size***

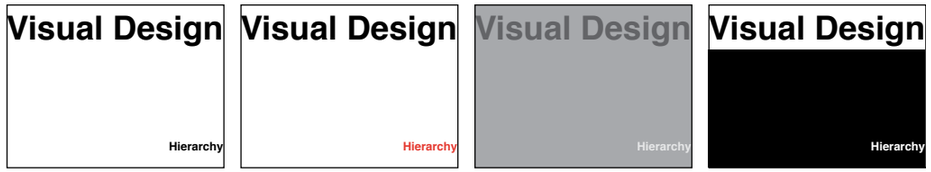


Figure 11.10. ***Visual hierarchy is not necessarily determined by size***

but at the same time the phrase "Oh my!" also stands out because it is so tiny. (We might infer from this that it is therefore spoken in a tiny, scared voice.)

Figure 11.10 takes the concept a little further, although it is still based only on text. Here, visual hierarchy is altered in the different examples simply by changing the colour of the text and background. On the far left, where all the text is black and the background uniformly white, size alone determines visual hierarchy – although even here the setting apart of the word "Hierarchy" helps it be noticed (by avoiding proximity and therefore avoiding grouping). In the second example from the left, changing the colour of the word "Hierarchy" to red immediately draws attention to it, so that it now challenges the dominance of the heading "Visual Design". This is achieved by emphasizing dissimilarity (inverting similarity), as well as by using a colour which tends to stand out. (We discuss colour in visual design in more depth in the next section.)

In the next example, the uniformly grey background and the use of greys for the text undermines hierarchy by blurring the distinction between figure and ground (which is very marked in the other examples) and by re-asserting similarity. The result is a much less striking design, in which there is no clear dominance. Finally, in the example on the far right, the background is divided into two areas of unequal size, with very high tonal contrast between each other and the text written upon them. This makes the strongest design of all (if a little lacking in subtlety), in which there is a more complex relationship between the visual elements. In this case, the figure–ground distinction is particularly prominent. The large area of black ground behind the small white text helps to compensate for the text's lack of size, whereas the small area of white ground around

Figure 11.11. ***Visual hierarchy in an image***

the large black text helps to focus attention on that text. The result is a closer balance and a less obvious visual hierarchy.

Visual hierarchy of course applies to all visual fields, not just to designs using text. Analysis of all the ways in which hierarchy may be achieved in images and graphic design would require far more space than is available here, but one simple example of visual hierarchy in an image is shown in Figure 11.11. The image on the left is a simple, unaltered greyscale photograph. Nothing has been done to it to change any hierarchy inherent in the original. In this case, therefore, the foghorn tends to be visually dominant, as it is so clearly distinguished from its ground (the sky) and lacks proximity to the other elements. In the example on the right, however, we have undermined this "natural" hierarchy simply by colouring one element in the image bright red. As a result this element becomes visually dominant – going to the top of the visual hierarchy – by virtue of its extreme dissimilarity from the rest of the elements in the image.

KEY POINTS

Gestalt principles of visual perception are derived from the theories of gestalt psychology. They are concerned with how the human brain tends to organize the visual information that reaches it through the eyes.

Perception of patterns and structures may be determined by the grouping of objects in a visual field.

The gestalt principles of proximity, similarity, symmetry, figure/ground and closure determine our recognition of grouping.

Non-symbolic ordering based on the gestalt principles is the foundation of structure in visual design.

The precise appearance and arrangement of objects may lead to one principle dominating the others.

Ignoring gestalt principles frequently results in a confusing design.

The component parts of an interface or Web page should usually be organized according to gestalt principles.

Navbars on Web pages, and the conventional arrangement of controls on media players illustrate the application of gestalt principles to multimedia design.

Visual hierarchy describes the dominance of one or more elements in the visual field. Like other hierarchies, it may have many levels.

Visual hierarchy may be achieved by applying gestalt principles "inversely", in order to disrupt grouping and make one or more elements appear dominant.

Visual hierarchy is not necessarily determined by size.

Flash UI Components

It has always been possible to implement data entry controls in Flash by drawing something that looks like a conventional control, turning it into a movie clip and using some scripts to make its behaviour match its appearance. There are two drawbacks to creating controls in this way. One is that much of the scripting must be repeated for every control; the other is that it offers unbounded scope for designing the appearance of controls. This may sound like an advantage, but as we mentioned at the beginning of this chapter, one of the keys to usability is familiarity. Users need to be able to look at a control and know with reasonable certainty what it does. This was not the case with some of the controls created by imaginative designers in the early days of Flash.

Recent versions of Flash support the use of reusable ***components***, which are described in the documentation as "movie clips with parameters that allow you to modify their appearance and behavior". In particular, a collection of user interface (UI) components is distributed with the Flash program. In effect, they are ready-made interface elements which have a standard appearance and behaviour. Individual instances of a component will differ in some respects: for instance, each time a "combo box" component is used to create a pop-up menu, it will have its own set of menu items.

Figure 12.18 shows how some UI components could be used to recreate part of the survey form we implemented in XHTML in the preceding section. (The dropped-down Flash menu is obscuring the Dreamweaver menu.) It is possible to customize the appearance of the components, but if the defaults are used the appearance will be identical on all platforms and in all browsers. While consistency may be considered desirable in many ways, it means that Flash-based user interfaces never look quite right on any platform.

Figure 12.18. *Flash text field, label and combo box components*

There are Flash UI components equivalent to all the XHTML control elements, except that every button must be created from the `Button` component, with its behaviour determined by the script that is attached to it. There are no distinct submit, reset and file upload buttons. Pop-up menus are provided by the `ComboBox` component, which is slightly more general than XHTML's `select` element. If the component is made editable, which is done by setting a property in the Flash environment, a text box is displayed above the menu. A user can either select an item from the menu or type something else into the text box, as shown in Figure 12.19, where the user has typed the value `FreeBSD` instead of choosing one of the menu items. Radio buttons, check boxes, text fields and text areas all have their own components, and there is a component that can be used to attach labels to controls. (Flash only has extremely primitive support for styling text: deprecated HTML appearance tags, such as `<font>` can be included in the label text, otherwise a script must be used to apply the styling.)

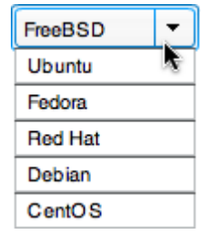


Figure 12.19. *A* ComboBox

The repertoire of Flash UI components extends beyond the set of standard XHTML input elements. Components are available for creating progress bars and sliders. A `ColorPicker` component can be used to create a primitive swatch selector with a field for entering hexadecimal colour codes and the `NumericStepper` component is available for creating fields for entering numbers: the small arrows at the right add one to, or subtract one from, the number shown in the field. As well as these components, which are illustrated in Figure 12.20, there are several others for displaying and editing data in various ways. There are also video player components, as we mentioned earlier.

IN DETAIL

The Draft HTML 5 specification includes new types of input element and other elements and attributes which will add much of the functionality of the standard Flash UI components to HTML. The standard is under development and is unlikely to be finalized or implemented universally before 2010.

There are significant differences between the way Flash UI components and XHTML control elements are used. Trivially, XHTML tags can be written by hand and styled explicitly with CSS rules, whereas Flash interfaces are usually built visually on the stage. This is a somewhat illusory distinction, though, because XHTML is often created in visual authoring programs, like

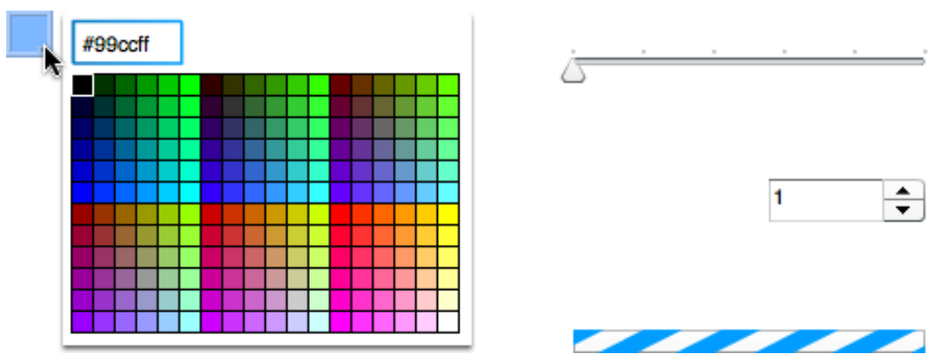


Figure 12.20. ColorPicker, Slider, NumericStepper *and* ProgressBar *components*

Dreamweaver, and Flash movies can be created outside the Flash authoring environment, using ActionScript and an XML-based layout language. However, there is a real distinction between the two in the way in which they are displayed. A Flash movie's appearance is fixed: components will always be the same size and occupy the same positions. As we explained in Chapter 10, text on a Web page can reflow, or change its size, which may radically change the layout and appearance of the page. Forms are always difficult to lay out well, but accommodating the dynamic appearance of Web pages makes them harder still.

More significantly, Flash UI components don't do anything useful without some scripting. Of course, a Web form on its own is useless, too, but it will always submit its data – any processing can be confined to a script on the server, which will often be designed and written almost independently of the form. A form built from Flash UI components must be supplemented by some ActionScript, even to make it send its data to a server. Often, data derived from the components is used by scripts within the Flash movie to perform non-trivial computation, the results of which are displayed in the form itself.

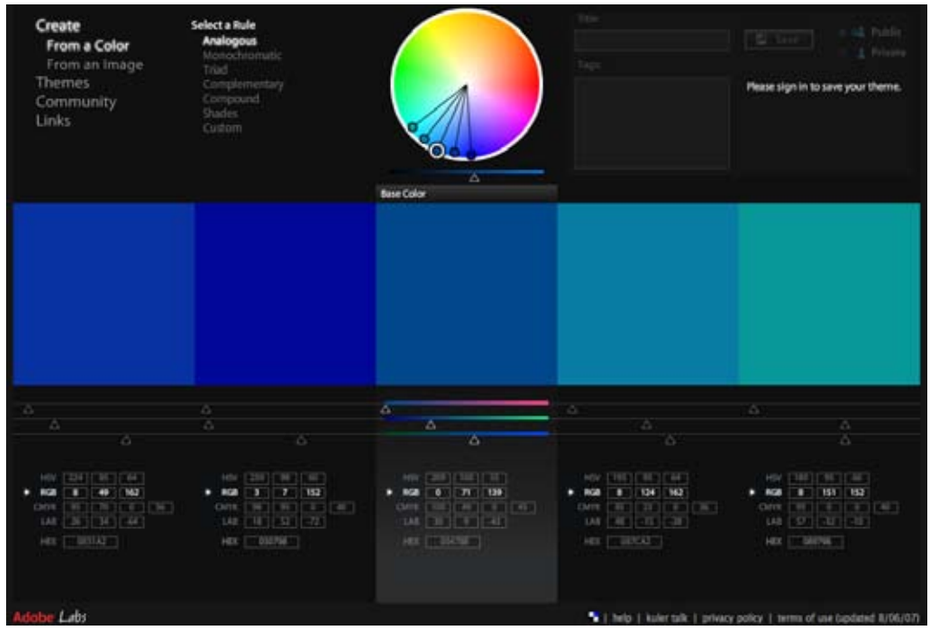
XHTML forms are created using markup. Flash UI components are more akin to the user interface widgets provided by the APIs of an operating system. They can be laid out in a visual design environment, but they need programming before they will do anything. In both cases, there may be additional programs running on a server that process data sent from a form.

Multimedia Interfaces

Despite our earlier remarks about familiarity, there are occasions when the standard controls are not adequate or appropriate, and some special device for interaction is needed. In such cases, it is necessary to strike a balance between convention and innovation. A new application may call for a new interface. If it does something that nothing else does, this will be inevitable. However, people are rarely willing to read a manual or watch a video to find out how to operate the controls. If they can't at least work out the basics straight away, they won't use the application.

In trying to use a new program or Web site, people will rely on their past experience of physical objects and of other programs and Web sites as their main guides. Almost nobody in the developed world needs to read a manual to find out how to operate the QuickTime Player, because almost everybody has used a video player, DVD or CD player. In fact the controls are different – they are not physical buttons at all. However, by using recognizable elements and symbols, with a familiar purpose, and by making sure the relationships between them are clear and understandable, interfaces can be built that, when taken as a whole, are new.

Flash can be used as a means of creating interfaces that are not constrained to a set of standard controls. Any movie clip may have scripts which respond to events attached to it, as we will



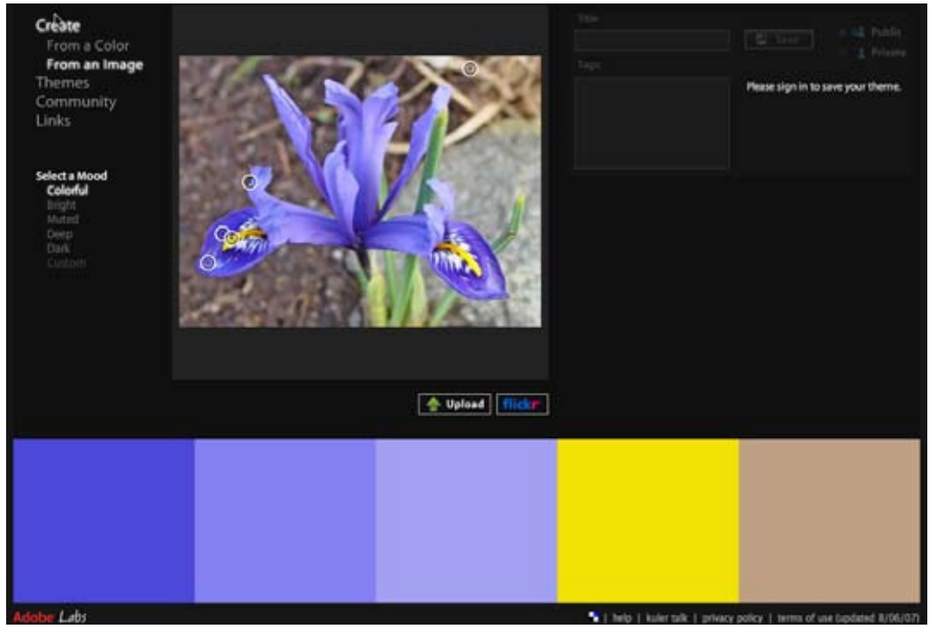


Figure 12.21. ***Kuler***

demonstrate in Chapter 14, so it is possible to create "controls" with any appearance whatsoever. Movie clips have a timeline, so controls can easily be animated. Objects can be created or modified by scripts, so interface elements created in this way can change dynamically. Completely innovative user interfaces can be made relatively easily.

A pioneering example of the use of Flash to create a multimedia interface is Adobe's ***Kuler*** application. Kuler is an online application for creating and sharing colour schemes – that is, sets of colours which will go well together in a design. Figure 12.21 shows the interfaces it provides for creating colour schemes. (Unfortunately, the greyscale background used on the Kuler site, while good for providing a neutral context for the colour schemes, makes the text on the screenshots hard to read.)

At the top left of the page, under the heading Create, you can choose between the options From a Color and From an Image. The top screenshot shows the interface for creating a scheme from a colour. You begin by choosing a base colour, which is the starting point for the colour scheme. This can be done using the slider controls or by entering values in one of several colour spaces in the fields below the central colour swatch. These controls are Flash components, customized for this application. They resemble controls used in the standard interface to Adobe Creative Suite applications, so their operation is clear to anybody who has worked with those programs (which most people using Kuler probably will have done).

Above the base colour swatch is a colour wheel, with circles attached to radial lines on it that correspond to the five colours in the colour scheme. (Figure 12.22 shows the colour wheel in more detail.) To the left of the colour wheel is a menu headed Select a Rule. By choosing a colour harmony rule from this menu, you can generate a set of colours based on the rule. You can adjust the colours in the scheme by dragging the circles in the colour wheel. When you do so, the harmonic relationships between the components are preserved. As we suggested in Chapter 5, the mechanical application of rules may not always yield the best results when choosing colours, so you can also choose the Custom rule and adjust each colour by hand, either by dragging in the colour wheel or by using the controls below each swatch.

Figure 12.22. ***Colour wheel controls in Kuler***

It should be clear that some complex computation of colour values is occurring behind this interface. The use of a specialized multimedia interface is entirely

appropriate here, as it allows direct manipulation of visual objects corresponding to the colour data that is being calculated. The use of non-standard controls, specifically the colour dots on the wheel, is justified, because the operations being carried out are themselves non-standard. However, the use of single dots on a colour wheel to select HSB colours is well established, so it is easy for users familiar with colour wheels in image processing applications to appreciate how Kuler's grouped controls work as an extension of the more familiar colour picker.

Figure 12.23. ***Selecting colour from an image in Kuler***

The lower screenshot in Figure 12.21 shows Kuler's alternative way of creating colour schemes, starting with an uploaded image. In this case, instead of choosing a colour rule, you can choose a "mood": Colorful, Bright, Muted, Deep or Dark, and Kuler will pick colours from the image that its algorithm considers to have the quality appropriate to the chosen mood. The pixels from which the colours are taken are shown with circles around them, as the detail in Figure 12.23 shows. In this case, it is not possible to adjust the colour scheme, except by moving individual colour selectors to a different place on the image by hand. The resemblance between these circles and the ones used on the colour wheel suggests that such movements are possible. The colour scheme's swatches change as the circles are moved, to provide feedback that this is happening.

We have stressed the use of Flash for creating multimedia interfaces because it is presently the only means of doing so relatively easily, in a platform-independent way, which can be embedded in Web pages to provide the interface to Web applications. For desktop applications, the Adobe Integrated Runtime (AIR) may be used to run programs with Flash-based interfaces on any platform.

However, Flash is not the only possibility for creating multimedia interfaces. The major desktop operating systems provide extensive APIs for creating, displaying and manipulating images and time-based media. Using a system programming language like C++ or Objective-C, a programmer can create interfaces every bit as media-rich as anything made in Flash. The result will probably be more efficient and is more likely to match other interfaces on the system, but programming at this level is harder and more time-consuming than writing ActionScript.

The technology defined by Web standards should provide an alternative means of creating interfaces to Web applications. Scripts triggered by events can be attached to any document element, and they can dynamically rewrite the page or alter stylesheets to hide or reveal elements, change their appearance or move them. JavaScript is not as powerful a language as ActionScript, though,

and Web browsers have many inconsistencies in how they implement it, which makes JavaScript programming more difficult and less reliable than using ActionScript. Furthermore, the Document Object Model (DOM), which defines how JavaScript can interact with page elements, does not provide as much control over document elements as ActionScript does over movie clips. New Web technologies, such as SVG and HTML 5, promise more opportunities, but their development and adoption so far has been very slow.

JavaScript libraries, including Prototype, script.aculo.us, jQuery, SproutCore, Adobe's Spry, the Yahoo! Interface Library and a growing list of others, are available to make it easy to use the existing capabilities of JavaScript and the DOM to implement dynamic interfaces. These libraries do not enable you to do anything that you couldn't do otherwise, using your own scripts, but they remove the need for repetitive programming of common tasks, particularly in respect of ensuring compatibility with all browsers. Their facilities are usually packaged in code that makes them easy to reuse – some of the libraries we have mentioned could be described as frameworks. Many of the effects they enable depend on a technique known as AJAX, in which data is retrieved from the server by a script, instead of by loading a new page into the browser in the usual way. The retrieved data is typically used to rewrite the current page. We will describe AJAX in a little more detail in Chapter 16; for a detailed example, see *Web Design: A Complete Introduction*.

KEY POINTS

By embedding controls in multimedia, we can provide ways of interacting with data or computation.

Standard dialogue controls can often be used for such purposes.

XHTML provides input elements, for text fields, check boxes, radio buttons, etc., textarea for multiple lines of text and select and option elements for pop-up menus and lists. These elements are used within a form to send data to a script on the server.

Flash UI components provide the same controls, plus a few others. They must be combined with some ActionScript to do anything useful.

Flash movie clips and ActionScript can be used to create interfaces that are not restricted to using standard controls. Flash-based interfaces can be used in a Web browser or in desktop applications using AIR.

Multimedia applications which do something new may require innovative interface design. Users will draw on existing experience when trying to use new interfaces, so familiar features and ideas should be used where possible.

JavaScript and other Web-standard technology can be used to program multimedia interfaces, but the possibilities are less extensive than those which Flash offers. JavaScript libraries are used to make the task simpler.

Exercises

Test Questions

1 What controls would you provide for the user if you were designing a video player program? Are any of them redundant? If so, should they be omitted?

2 What are the two ways of underlining a link with CSS? What are the relative advantages and disadvantages of each method?

3 Why would it usually be inadvisable to change the font size of link text when the cursor moves over it? Are there any circumstances in which changing the font size would be an acceptable form of rollover effect?

4 Give three examples of the use of dragging to interact with images.

5 Are there any rollover effects that can be implemented on a Web page using JavaScript but not using CSS alone? Consider only cases where the effect is applied to the element that the cursor rolls over, not remote rollovers.

6 In an XHTML form, which type of control would you use for each of the following questions?

(a) What is your email address?

(b) What is your country of residence?

(c) How many people live at your present address?

(d) What sex are you (male or female)?

(e) What party did you vote for at the last general election?

Which Flash UI components would you use for the same questions?

7 Give three reasons why Flash UI components should be used in preference to purpose-built movie clips for implementing controls. What factors might make you choose to implement your own Flash controls instead of using UI components?

People who have good vision, hearing and motor control, and have no cognitive disabilities, tend to take these faculties for granted and often forget that not everybody shares their good fortune. They also often fail to realize that their present condition is not likely to last forever. In Chapter 1, we mentioned that most media are visual in nature, and we have made an implicit assumption when describing interactivity that a mouse or other pointing device is used for input in conjunction with a keyboard. But for many people, seeing, hearing and using a mouse and keyboard are difficult or impossible, because they suffer from some congenital condition or have had an illness or accident, or are simply suffering from the common effects of advancing age.

It may be tempting to think that – although this may be unfortunate for those concerned – there is nothing to be done about it. This is not true. As we will describe shortly, technology is available to allow people with disabilities to perceive and operate computers, even if the normal means of doing so are denied them. However, the creator of a Web page or an application with a multimedia interface may have to make some extra effort to ensure that people who depend on such technology to use computers are not excluded.

Background

If a program or Web page is equally usable by everybody, irrespective of any physical or mental limitations they may suffer from at the time, it is said to be ***accessible***. Accessibility has received most attention in the context of the World Wide Web. W3C's ***Web Accessibility Initiative (WAI)*** has produced sets of guidelines to help Web designers produce accessible pages. Recognizing that the Web is not restricted to the standard technologies, such as XHTML and JavaScript, but may include Flash, PDF and other media, WAI has tried in the most recent version of the guidelines to incorporate principles that apply to all multimedia. We will therefore base our description of accessibility on the Web and WAI's guidelines, but we will concentrate on the broad concepts which are more widely applicable. If you only need specific guidance on creating accessible Web pages, consult Chapter 9 of *Web Design: A Complete Introduction* or some of the specialized works listed on this book's support site.

Problems with Access

Figure 13.1 summarizes some of the problems that people may experience when using the Web and multimedia. The range is broad, and each may create different barriers. However, none of these problems makes it impossible to use a computer. This is fortunate, as many conditions can be mitigated by using computers and the Internet. For example, people who cannot see may not be able to go out shopping alone, but they ought to be able to shop using e-commerce Web sites.

	Typical Conditions	Problems with Multimedia	Assistive Technology
Vision	Blindness	Inability to perceive graphical interface	Screen readers, Braille displays
	Low vision	Difficulty seeing and reading	Screen magnifiers
	Colour defects	Inability to perceive information represented by colour	Browser option to set stylesheets
Hearing	Deafness Tinnitus	Inability to perceive information in sound	Signing avatars
Movement	Repetive Strain Injuries Limb injuries Effects of stroke Cerebral palsey	Inability to use pointing device and/or conventional keyboard	Alternative devices simulating keyboard input, voice input
Cognition	Dyslexia	Difficulty perceiving information conveyed in text	Screen readers
	Attention deficit disorders Lack of sleep	Difficulty concentrating	
	Autism Down's syndrome Effects of stroke Alzheimer's disease	Difficulty understanding content, problems with orientation and navigation	
Age-related	Presbyopia	Difficulty reading small text	Controls to increase text sizes
	Loss of coordination	Difficulty using pointing device	
	Short-term memory loss	Loss of orientation	

Figure 13.1. ***Some conditions affecting accessibility***

They will have to rely on the aid of assistive technology such as screen readers, so Web designers must ensure that they do nothing to erect additional barriers to accessibility.

Making Web sites and programs with multimedia interfaces accessible can assist a wide range of people, but it most obviously and effectively helps people with physical and mental disabilities. If this doesn't make you feel a social obligation to maximize accessibility, you should be aware that in many countries there are legal requirements to do so. Legislation forbidding discrimination against people with disabilities is increasingly common around the world. Although there is considerable variation among the laws in different countries, generally, where disability legislation is in force, it is considered to apply to digital services and sources of information, including Web sites.

It is not possible to arrive at an accurate value for the number of people in the world who have difficulties that interfere with their use of computers. Not all problems are reported; the only figures that are readily available are those for people who are officially registered as disabled, but definitions of what qualifies as a disability vary from country to country. A broad picture can be obtained for some conditions, though.

There were, for instance, 364,615 people in the UK who were registered as "severely sight impaired" (blind) or "sight impaired" (partially sighted) at the end of March 2006. RNIB (the Royal National Institute for the Blind) estimated that about two million people in the UK have significant sight loss. In the United States, according to the latest census statistics, 1.5 million people over the age of 15 suffered from blindness.

An often-cited statistic is that 1 in 12 men and 1 in 200 women suffer from some defect in their colour vision, usually an inability to distinguish clearly between reds and greens. We will discuss how this may interfere with the perception of user interface elements later in this chapter. Since colour defects are largely genetically determined, their prevalence varies among different populations, and the figures quoted only apply to people of European origin. People originating from other parts of the world do not suffer as commonly from colour defects: among Asians the figure is estimated at 1 in 20 males, and among people of African origin it is as low as 3%, and the common form of deficiency is different from that found among Europeans.

Determining how many people suffer from repetitive strain injuries (more accurately known as cumulative stress disorders) is problematical, because of the difficulty in diagnosing the condition accurately – or even defining it. The US Bureau of Labor Statistics has reported that 60% of all reported occupational illnesses are RSIs; in 2001, a study reported that 10% of Canadians (an estimated 2.3 million people) had suffered some form of RSI "serious enough to limit their normal activities" in the preceding year. Both these studies concluded that the incidence of RSI was increasing.

The developed world presently has an aging population. That is, the proportion of people over the age of 60 is increasing steadily. With increasing longevity comes an increase in age-related problems, such as deteriorating vision, arthritis, loss of muscular strength and coordination and lapses in short-term memory.

The problems people with disabilities experience when accessing multimedia are related to problems potentially experienced by people using mobile devices. A mobile phone user may disable the loading of images when using the phone to access the Web. They are therefore in the same position with respect to those images as a blind person: if information is conveyed solely by images, they won't see it. Mobile phones may be used in bright sunlight, so that colours are hard to see on the screen: the user is temporarily colour-blind, as other people are permanently. It may be hard to hear when a device is used in a crowded place, so the phone user shares the problems of somebody who is hard of hearing. The absence of a mouse and proper keyboard on most mobile devices may make selecting objects on the screen as difficult for the phone user as it is for somebody with restricted mobility using a conventional computer. It follows that measures to enhance accessibility for people with disabilities often provide benefits for users of mobile devices too.

Such measures may also be appreciated by people with no actual problems. A survey carried out on behalf of Microsoft in 2003 estimated that 40% of computer users of working age used some of the built-in accessibility options or utilities on their computers. Interestingly, not all of these people absolutely needed to do so; many suffered no difficulties, or only slight ones that did not prevent their using the standard interface. However, they found it more convenient or comfortable to use options for changing the display or mouse behaviour. Accessibility will enhance their experience, allowing them to use their computers in ways that they prefer.

Hence, even if we add together all of the people who might possibly be suffering from some permanent or temporary condition that interferes with their use of computers, we still underestimate the potential benefits of accessibility.

Assistive Technology

To understand what is required to enhance accessibility, you need to know something about the devices that people use to help them overcome their limitations. ***Assistive technologies*** are software or hardware products that provide alternative forms of input and output for people who cannot use the conventional mouse, keyboard and screen. These products allow some users to interact with computers in ways that they would otherwise find difficult or impossible.

Among assistive technologies, the devices used by people who are blind or have very limited vision probably present the biggest challenge in connection with multimedia. These devices alter what psychologists call the "modality" of the interaction with the computer. This means that the

information is transmitted via a different sense – hearing or touch instead of sight – and processed in the manner appropriate to that sense.

Screen readers are programs that use speech synthesis to speak text. Simple screen readers in effect scan a window, reading whatever they encounter in the order it appears. Simple programs of this type are sometimes called "screen scrapers", since they just take the text off the screen. More sophisticated programs work at a deeper level, rendering actual data as spoken text. These work particularly well in the context of the Web, where they are able to interpret the structural information contained in XHTML documents and use it to create a more meaningful verbal rendering of the page. For instance, if a page is laid out in two columns, using absolutely positioned `div` elements, a screen scraper might read straight across the columns, whereas a screen reader that could interpret the markup would distinguish the contents of the two `div` elements correctly.

As well as their obvious benefits to blind people, screen readers are also of use to people who are dyslexic or illiterate.

IN DETAIL

Screen readers may be independent programs, they may be built in to the operating system, or, for the Web, they may be implemented as browser extensions. The leading screen reader program, which is often taken as a *de facto* standard of screen readers' behaviour, is called JAWS. Both JAWS and its leading competitor, Window Eyes, only run under Microsoft Windows. (And both are very expensive.) Less powerful screen reading capabilities are built into Mac OS X in the form of VoiceOver, while FireVox is a free extension that adds screen reading to the Firefox browser on Windows, MacOS X and Linux.

Braille is a system of representing text as patterns of raised dots that blind people can learn to read with their fingers. ***Refreshable Braille displays*** use pins that can be dynamically raised or lowered to present a changing Braille representation of the text on a computer screen. Such displays are particularly useful to people who are both blind and deaf, and cannot therefore use a screen reader.

Both screen readers and refreshable Braille displays alter their users' perception of multimedia in two ways. First, they reduce it entirely to text, so that any information that is conveyed by images alone is lost. Second, they make everything into a time-based experience; when the elements of a Web page are read out or translated into Braille in order over time, for example, it makes sense to talk about the page's duration. This can be not only time-consuming but tedious; if every page of a Web site has a navbar across the top, a blind person using a screen reader might have to wait to hear all the navbar's links on each page before getting to that page's main content.

People with poor eyesight who can nevertheless see to some extent may need to increase the size of text on their screens in order to read it. The huge numbers of middle-aged and elderly people who suffer – as almost everybody over a certain age does – from the hardening of their eyes' lenses known as presbyopia find it difficult to focus at short distances, so they cannot read small print or text in a small font on a screen. For them, increasing the font size in their browser by a factor of up to two is usually sufficient to make text readable without the use of reading glasses (which are not very satisfactory for use with a computer screen). Almost all Web browsers now provide a means of doing this. The effect on the layout of pages not designed to accommodate changes in font size may be highly disruptive. As an alternative to increasing the size of text, most browsers allow the entire page to be magnified. This avoids any problems with layout, but means that horizontal scroll bars will often be needed, which interferes with usability.

People with more severely impaired vision may need to magnify the contents of their screen to a much greater extent. Screen magnifiers perform this function. As Figure 13.2 shows, extreme magnification may lead to an almost complete loss of context, especially on pages with a lot of empty space. (Both shots in this figure show the same area of the screen, at normal magnification on the left, zoomed to the maximum amount available by way of the Mac OS X Universal Access Zoom feature on the right.) A certain amount of trial and error will be needed for the person viewing this page to get back to the top navbar, for example.

Mice, trackballs and trackpads are among the main causes of RSI in computer users, so there are many people who must avoid using these devices. In that case – and if the injury does not prevent it – RSI sufferers use the keyboard for all input. Blind people cannot use a pointing device, since they are unable to see what they are pointing at, so they also rely on keyboard input. Many physical and cognitive disorders can make it difficult for people to point accurately with a mouse.

Figure 13.2. ***Zooming in with a screen magnifier can lead to a loss of context***

Names, Variables and Assignment

Our informal description of objects showed that it is necessary to give names to objects if we are to work with them. The host objects provided in JavaScript and ActionScript are given names by the host system. For example, we already mentioned that a script running in a Web browser has access to an object called `document`. There is no need for a programmer to do anything to create this object or give it its name. In Flash, symbol instances can be given names in the Properties panel when they are placed on the stage, and the names can be used to refer to the corresponding objects within a script.

As we explained earlier, these objects belong to classes. Their classes also need names, and the names of the built-in classes (such as `MovieClip`) are also provided by the system.

The names that can be used for objects and classes – and other names that you can use in a program – are subject to some restrictions. They must consist only of letters (upper- or lower-case), digits, underscore (_) or $ symbols, and they cannot begin with a digit. Note that this does not allow names to contain spaces.

Certain conventions are used to restrict the form of names further. Names beginning with underscores and $ symbols are usually only used by libraries and machine-generated code. Names of methods and properties defined in scripts are usually written in lower-case, with underscores being used to separate multiple words. Class names, and the methods in classes defined by APIs, are conventionally written in "camel-case". That is, when the name consists of more than one word there are no spaces between words, but each word starts with a capital letter. Class names consisting of just a single word begin with an initial capital letter (which means they are just a degenerate case of the general rule).

IN DETAIL

The ECMAScript standard's rules that define the allowable forms for names are more complicated than we have stated, because they allow the use of Unicode characters beyond the ASCII subset. However, it would be unwise to rely on any implementation supporting any non-ASCII characters in names.

Some names which look as if they ought to be legal are not allowed to be used to name objects and so on, because they are ***reserved words***, which are used by the language as part of its syntax. We will mention some reserved words as we go along.

Various sorts of entity which need names can appear in a program besides classes, objects, methods and properties. ***Variables*** are the most common of these.

Variables are containers – or "locations", in the jargon of programming languages – that have a name (often called an identifier) and can hold a value. The value stored in a location can be changed by the operation of ***assignment***. Storing a value in a location allows it to be referred to using the variable's name. Variable names follow the same convention as method names.

Values can be stored in variables using the assignment operator, written as `=`. (This is a potential pitfall if you are not used to programming and expect the `=` sign to mean equality, as it does in mathematics. In ECMAScript – and many other programming languages – equality is written `==`.) So, if `the_amount` is a variable

```
the_amount = 149
```

will store the number 149 in it.

A single number is not the only thing that can be assigned. Strings, written between " signs, like attribute values in XHTML, can also be assigned, and so can Booleans (which may be true or false). As well as values of these "primitive" types, objects belonging to any host class or class you have defined yourself can be stored in variables.

Expressions of any of these types may be constructed using operators. For numbers, the conventional arithmetic operations are provided (`+`, `-`, `*`, `/`, `%`, where * is multiplication and `%` is remainder after division); for strings, the only operator is concatenation, written as `+`, which sticks two strings together; for numbers, the logical and, or and not operations are available, written `&&`, `||` and `!`.

Variables can be used in expressions. When the name of a variable appears in an expression, its current value is substituted. For example, after the assignment just shown, the expression `the_amount + 1` has the value 150. In most scripts, method calls and references to objects' properties are commonly used in expressions, as you will see later.

Expressions can be assigned to variables. We might write

```
the_amount = the_amount + 1
```

which would store 150 in the variable. That is, we can change the value stored in a variable as a program's execution proceeds.

The names of objects' properties can be used in the same way as variables, but they must be accessed through an object using the dot notation we described earlier. For instance, if `tf` is a `TextFormat` object, we can set its `align` property using an assignment such as

```
tf.align = "right"
```

Assignments are one of the sorts of ***statement*** that ECMAScript provides. Statements are the primitive pieces of executable code from which programs are constructed.

In early versions of ECMAScript and in current versions of JavaScript, if you want to introduce a variable into your program you just use it. Generally, it's a good idea to assign something to it as the first thing you do, because otherwise it holds the special value `undefined`, and trying to use it will always be a mistake.

In JavaScript, you can, if you like, explicitly declare a variable, by using the reserved word `var` in front of the initial assignment, as in

```
var the_start_angle = 0
```

This makes it clear that you are using a new variable, but doesn't do anything else. In ActionScript, though, things are different. You must declare all your variables before you use them. Furthermore, when you do so you must specify what type of value will be stored in them, by adding a colon followed by a class name after the variable's name, as in

```
var the_start_angle:Number = 0
```

The class names `Number`, `String` and `Boolean` are used for the primitive types of value. The name of any class can be used instead, if you are storing objects in your variable.

As we mentioned earlier, some properties of host objects are "read-only" and cannot be changed. Sometimes you want to create a variable whose value cannot be changed. This can be done in ActionScript (but not JavaScript) by declaring variables using the word `const` instead of `var`:

```
const MAUVE:Number = 0xFF7FFF
```

By convention, the names of constants are written entirely in upper case, as we have done here. You are most likely to see constants as properties of certain host objects. This example also demonstrates that numerical values can be written in hexadecimal notation by prefixing them with `0x`.

Flow of Control

If you write a series of assignments one after the other on separate lines, they will be executed in order when your script runs. (If you like, you can add a semi-colon after each assignment to terminate the statement, but it is not necessary. You only need to use semi-colons to terminate statements if you want to place two statements on the same line.)

Most useful computation demands that statements be executed in a more elaborate order than a straight sequence. The first requirement is to be able to execute a statement if and only if some condition is true. You use a ***conditional statement*** for this purpose. It takes the form

```
if ( E )
    S1
else
    S2
```

where `if` and `else` are reserved words, `E` is an expression and S_1 and S_2 are statements – either single statements, such as an assignment, or blocks, consisting of a sequence of statements enclosed between curly brackets { and }. The indentation is used to show the structure of the conditional; this is a convention not a requirement. The effect of the conditional statement is as you might expect: if `E` is true, S_1 is executed, otherwise S_2 is executed. This implies that `E` should be something whose value is either true or false – that is, a Boolean expression.

Boolean values are most commonly created by comparing variables with other values. For example, `x > 0` has the value `true` when (the value stored in) `x` is positive but `false` otherwise. ECMAScript has a full range of comparison operators, but owing to the restricted character set available on keyboards, the operators only approximate the conventional mathematical symbols. They are `<`, `<=` (for ≤, less than or equal), `==` (equal, as noted earlier), `!=` (not equal), `>=` (for ≥, greater than or equal) and `>`. These operators can be applied to numerical values with the expected meaning, but also to strings, when a dictionary-order comparison is performed. `"digital" >= "multimedia"` is `false`, for example, because "digital" comes before "multimedia".

You can combine conditions using the Boolean operators we listed earlier. It is advisable to use brackets in complicated cases. You can also assign the result of a comparison to a variable, as a way of remembering it. Subsequently, the variable name can be used on its own in any context where you could use a comparison.

IN DETAIL

It is not necessary to put curly brackets round a single statement when you use it as an alternative in a conditional (or most other contexts where you can use a single statement or a block), but it does no harm to put them, and a lot of people prefer to do so. We prefer to omit them unless they are needed, but there is no special virtue in this.

As an example of using a conditional statement, suppose that you wish to compute a payment, perhaps a commission. Say the payment is 10% of some total amount, except that if the amount is less than 10 (euros, dollars, zlotys, or whatever), no payment is made. Assume that the total is held

in a variable called amount, declared elsewhere, and we want to compute the payment and store it in a variable called payment. This can be achieved in JavaScript in the following way:

```
var commission = amount * 0.1     // 10% = 0.1
if (commission < 10)
    payment = 0
else
    payment = commission
```

(In ActionScript, we would have had to add :Number after commission on the first line.)

The characters from // to the end of the first line are a ***comment*** – that is, text which has no effect on the computation, but serves as an annotation for the benefit of people reading the script. You won't see many comments of this sort in this chapter, because the accompanying text does the job of explanation here. However, large or subtle scripts benefit from the use of meaningful comments, and you should develop the habit of adding them, if only to help you remember what you have done when you come back to it after an interruption.

If there isn't anything to be done when E is false you can leave out the else S_2 clause from a conditional statement. For example, you may have realized we could have assigned the commission provisionally to payment and then reset it if it was too small. This would use a single-branched conditional, like this:

```
var commission = amount * 0.1
var payment = commission
if (payment < 10) payment = 0
```

As well as saving a bit of code, this allows us to declare payment only when we have a meaningful value to assign to it. (If you like to keep your code short, you can eliminate the variable commission, too. We leave that to you for now.)

You use a conditional statement when you need to make a choice, as in the example just given. ***Loops*** are used when you need to repeat an operation. As a simple example, suppose you wished to replicate a string a certain number of times, and that variables s and repetitions held the string and the replication count, respectively. So, if the value of s was "ECMA" and that of repetitions was 4, you would want to produce "ECMAECMAECMAECMA". You could accomplish this operation by setting a variable ss to the empty string, and then sticking s onto the end of it four times. You could just write four assignment statements, but if you wanted to change the value of repetitions, you would need to change the number of assignments. If the value of repetitions depended on a user's input, you could not do this.

Loops allow you to use the value of a variable (or any expression) to control the number of times something is done. ECMAScript has several sorts of loop. We will only describe one – the ***for loop*** – which captures a common pattern, where the loop is controlled by a counter whose value is increased every time the loop is executed.

The form of a for loop is illustrated by the following JavaScript code to replicate a string:

```
ss = ""
for (i = 0; i < repetitions; i = i + 1)
    ss = ss + s
```

The loop is introduced by the reserved word for. All the book-keeping concerned with iteration is kept together at the top of the loop inside the brackets. First comes the initialization, which is performed once before the loop. Next comes the condition which determines how many times the loop is executed: the condition is evaluated every time round the loop and the loop is only repeated again if it is true. The third component is the increment, which is performed at the end of each iteration.

After all these components comes the loop's body, which is executed repeatedly as long as the condition remains true. Note that in this case the condition depends on the value of i, which is changed in the increment, so there is reason to suppose that the value of the condition will eventually become false. If the condition depended only on values that were not changed in the increment or the loop's body, the loop would run on forever.

IN DETAIL

The increment in a for loop does not have to consist of adding one to a variable but it often does. JavaScript has a special shorthand for this operation: ++i is equivalent to i = i + 1. This shorthand is itself a contraction of a more general shorthand. We could have written the same operation as i += 1. In general, any assignment statement where a variable is combined with some value using a single operator and immediately assigned back to the same variable can be contracted: x = x @ v can be written x @= v, where @ stands for any binary operator. We would normally have written the example loop as:

```
ss = ""
for (i = 0; i < repetitions; ++i)
    ss += s
```

These shorthands are very useful and are used by most experienced programmers, but they may be confusing to newcomers, so we will avoid them in our examples. You are likely to encounter them if you read many actual programs, though.

XML (eXtensible Markup Language) is the basis for XHTML and a host of other languages that have been proposed, if not actually used, for marking up different types of content on the Web. XML is the format used to deliver RSS feeds and podcasts. Almost all the formats and languages proposed for implementing the W3C's "Semantic Web" are based on XML. In particular, XML can be used as a concrete syntax for RDF (Resource Description Framework), which is intended to provide a standard for metadata on the Web. It is also used as a format for exchanging data between Web applications, and has found many applications off the Web. XML is used to define ODF (OpenDocument file format) and Microsoft's OOXML (Office Open XML), both of which are used as formats for office documents, and provides the syntax for Apple's property list files, used to record preferences on Mac OS X. Adobe used XML to define MXML, the layout language for Flex, and XFL, for exchanging Flash documents.

At its simplest, XML can be used as a markup language, like XHTML, to apply tags to documents so as to identify their structural elements. Unlike XHTML, though, XML does not provide a fixed set of elements. In effect, it lets you make up your own. For simple purposes, such as exchanging data between a blogging application and a desktop blog editor, this is adequate, but for more important tasks it is helpful to be able to impose constraints on which elements may be used, what attributes they may have and which elements may be contained in others. This allows a program that processes the XML data to verify that it is correctly formed and includes everything it should and nothing it shouldn't.

A formal definition of a set of elements and their attributes, together with constraints on the way they may be combined, can be created in the form of an XML ***Document Type Definition (DTD)***. In effect, a DTD defines a specialized markup language. XHTML is defined by a DTD, for instance. You can therefore consider XML not just as a markup language, but also as a markup ***metalanguage*** – a language for defining markup languages.

Languages defined by XML are often called ***XML-based languages***. They all share the basic notation for writing tags and entities that we described for XHTML, but have different sets of elements and their own rules about how these elements can be used. The common syntax makes it feasible to mix elements from several XML-based languages in the same document.

XML 1.0 was adopted as a World Wide Web Consortium Recommendation early in 1998. It was intended to provide a new foundation for the Web, which would be built by mixing XML-based languages including XHTML, SVG, MathML (Math Markup Language) and SMIL (Synchronized Multimedia Integration Language). Metadata would be added using RDF and the semantics would be described for processing by machines using OWL (Web Ontology Language). Improved linking would be provided by XLink and better forms by XForms. XSL (the eXtensible Stylesheet Language) would allow radical restructuring of documents to be performed, which in turn would allow more extensive control over appearance than CSS provides. XML would even bootstrap itself: DTDs would be replaced by "schemas" which would define XML-based languages using XML syntax.

Things haven't worked out that way, though. The XML-based Web has been met by a mixture of indifference and incomprehension by the majority of the Web design community, and by positive hostility from the Web browser makers. The leading proposal for a successor to XHTML 1.0 is HTML 5, which can be "serialized" as XML, but can also be serialized using a custom syntax based on HTML 4, which is recommended instead. XForms has attracted little attention and HTML 5 includes an alternative extension to the existing form elements. The metadata formats considered essential to the Semantic Web are only used in small specialized communities. The XML-based languages specifically concerned with the subject matter of this book are being little used. SVG is finally being implemented, at least partially, in most browsers, but continues to be overshadowed by Flash as a vector format. SMIL's only popular application to date has been the embedding of advertisements in media streams.

Despite all this, XML is worth knowing about. Certain features of XHTML only make sense if you understand the relationship between XHTML and XML. Some XML-based languages for Web metadata are likely to become increasingly important, even if others continue to be seen as irrelevant. XML's role in Web services, allowing data to be exchanged between Web applications, is firmly established. Although the dramatic change to a Better Web built on XML may not have occurred as foreseen, XML's other uses continue to grow. Furthermore, because XML is easy to read, looking at an XML-based language, such as SVG, can provide insight into the way media are represented.

Syntax and DTDs

To begin with, you will get quite a long way if you think of XML as being the same as XHTML except that you can make up your own tags and attribute names. In fact, it would be more accurate to say that XHTML is XML with a fixed repertoire of tags and attributes. Therefore, you already know that tags are written between angle brackets, as in `<tag>`, and, unless the element is empty, must be matched by a closing tag, which has a / before the element name, as in `</tag>`. Attributes' values are written enclosed in double quotes and assigned to attributes following the element name using an = sign. Empty elements can be written like a start tag with a / before the closing >. Elements must be properly nested. Character and entity references beginning with

an & and ending with a ; can be used for certain characters that are hard to type or, like <, are reserved for a special purpose.

Well-Formed Documents

Simply following the rules we just summarized allows you to write what are called ***well-formed*** documents. Being well-formed simply means that a document obeys the rules of XML syntax that allow it to be parsed correctly. For many purposes, this is adequate: it lets us create documents whose structure is expressed with markup tags, which can be processed by a computer program.

The following XML document shows you what well-formed XML looks like.

```
<books>
    <book id = "dmt">
        <title>Digital Media Tools</title>
        <author>Nigel Chapman</author>
        <author>Jenny Chapman</author>
        <price sterling="34.99" euro="47.30" />
        <publisher>John Wiley &amp;amp; Sons</publisher>
        <numberinstock current="1" ordered="6" />
    </book>

    <book id = "perl">
        <title>Perl: The Programmer's Companion</title>
        <author>Nigel Chapman</author>
        <price sterling="38.99" euro="52.70" />
        <publisher>John Wiley &amp;amp;Sons</publisher>
        <numberinstock current="0" ordered="0" />
    </book>

    <book id = "dmm">
        <title>Digital Multimedia</title>
        <author>Nigel Chapman</author>
        <author>Jenny Chapman</author>
        <price sterling="29.95" euro="45" />
        <publisher>John Wiley &amp;amp;Sons</publisher>
        <numberinstock current="12" ordered="20" />
    </book>
</books>
```


The document provides a list of books, with their basic bibliographical details. It might be part of the stock inventory of an online bookshop, so the data also includes the current prices in two major currencies and a record of the stock situation.

The XML markup imposes a structure on the data, but it has no semantics. We have chosen element names that make it clear to an English-speaking reader what each element is supposed to represent, but the actual names have no formal significance. We could just as easily have used `aardvark` elements to record the authors' names. The meaning only resides in what is done to the document by software.

A little thought will tell you that there is considerable scope for choice in representing a particular set of data as XML. In particular, there is no clear-cut criterion for deciding whether to use an element with an attribute to record values such as, in this document, the publisher's name, or an element whose content is the value. For example, `<publisher>John Wiley &amp; Sons</publisher>` or `<publisher company="John Wiley &amp; Sons" />`.

Here, we have used a mixture: the price and number in stock are recorded using attributes; this has made it easy for us to combine two different values in both cases: the price in sterling or in euros, the number in stock currently and in a few days' time. Using attributes avoids multiplying the number of elements. On the other hand, placing the author in the content of an element has made it possible to have two author elements for the jointly written books. This would help indexing and searching software to identify the book from either author. The decision to use element content to record the values of the remaining fields is somewhat arbitrary.

We showed in Chapter 14 how the structure of an XHTML document could be displayed as a tree. The structure which XML markup imposes on any document can equally well be represented in the form of a tree, sometimes called a ***structure model*** in this context. The structure model is essentially an abstract representation of the way in which document elements are ordered and contained within each other. Figure 15.1 is a picture of the structure model of the document for books shown earlier. (Compare it with Figure 14.2.)

The methods of the Core DOM can be used to traverse any XML document's tree structure. Unlike the methods of the HTML DOM, those of the Core DOM do not embody any assumptions about which elements are being used, or what attributes they have. They allow a program to traverse and alter the tree of any XML document in a consistent manner. This means that if data is formatted as XML the repetitive operations used to extract individual items, such as the price of a particular book, can be performed using library routines that build a tree and implement the DOM methods, leaving only the operations specific to the particular application to be implemented from scratch.

KEY POINTS

The Internet is a global network of networks, communicating via TCP/IP.

ADSL, cable modems and the 3G mobile phone networks are among the technologies used for broadband connections to the Internet.

Online distribution of multimedia is based on the client/server model of distributed computation. Servers listen for requests from clients and send responses, providing some data or service to the client.

Protocols are sets of rules governing the interactions between servers and clients.

HTTP (Hypertext Transfer Protocol) is a simple protocol designed for the fast transmission of hypermedia between Web servers and clients (e.g. browsers).

DNS (Domain Name Service) translates domain names to numerical IP addresses.

Protocols are organized into layers, with each layer providing services to the layer above, which are implemented using the services of the layer below.

TCP/IP networks are packet-switched, so messages are multiplexed.

IP (Internet Protocol) only provides a mechanism for getting datagrams from their source to their destination through a network of networks.

Each host is identified by a unique IP address.

TCP (Transmission Control Protocol) is layered on top of IP to provide reliable delivery of sequenced packets, using a system of acknowledgements with a sliding window of unacknowledged packets.

TCP uses transport addresses, consisting of an IP address and a port number, to provide connections between programs running on different hosts.

UDP (User Datagram Protocol) only tries its best to deliver datagrams. It does not offer reliable delivery, so it has less overhead than TCP, which makes it more suitable for streamed video and audio.

RTP (Real-Time Transport Protocol) runs on top of UDP, adding features for synchronization, sequencing and identifying different payloads.

Multicasting may be used to send the same data to many users: a single packet is sent, and is duplicated along the way whenever routes to different users diverge.

Multimedia applications such as live video streaming are suited to multicast.

For multicast, hosts must be assigned to host groups. Certain IP addresses identify groups instead of single hosts.

Delivering Multimedia

The network and transport protocols we have described do no more than deliver packets of data – more or less reliably – to their designated destinations. Higher-level protocols must run on top of them to provide services suitable for distributed multimedia applications. We will describe two such protocols: HTTP, which is the basis of the World Wide Web, and ***RTSP (Real Time Streaming Protocol)***, a protocol designed to control streamed media. Our description is not exhaustive, but is intended to show what is involved in mapping the requirements of some kinds of distributed multimedia applications on to the transport facilities provided by the protocols described in preceding sections.

HTTP

Interaction between a Web client and server over HTTP takes the form of a disciplined conversation, with the client sending requests which are met by responses from the server. The conversation is begun by the client – it is a basic property of the client/server model that servers do nothing but listen for requests, except when they are computing a response.

To start things off, the client opens a TCP connection to a server. The identity of the server is usually extracted from a URL. As we explained previously, the domain name in the URL is translated to an IP address by DNS. This translation is done transparently, as far as HTTP is concerned, so HTTP can always work in terms of names. By default, Web servers listen to port 80, so this will normally be the port number used when the connection is opened.

Originally, prior to version 1.1 of HTTP, a connection was used for one request and its response only. That is, the client opened a TCP connection and sent one request; the server sent a response to that request and closed the connection. This way of using connections is very efficient from the server's point of view. As soon as it has sent the response and closed the connection it can forget about the transaction and get on with something else, without having to keep the connection open and wait for further requests to come over it, or close the connection if it times out. The disadvantage is that accesses to Web servers tend to come in clusters. If a page contains 10 images most browsers will make 11 requests in rapid succession – one for the page itself and one for each image – so 11 connections between the same pair of hosts must be opened and closed. For technical reasons, to help minimize packet loss, a TCP/IP connection starts out at a slow rate and gradually works up to the best speed it can attain with acceptable losses in the existing state of the network. Hence, in our example, the data for the page and its images will not be sent as fast over the 11 connections as it would over a single connection, since each new connection has to start over and work up to speed. HTTP version 1.1 has therefore introduced the possibility of a persistent connection, which must be explicitly closed by the client (although the server will close it after a specified time has elapsed without requests).

Nevertheless, the structure of an HTTP session retains the form of a sequence of requests that evoke responses. No state information is retained in the server, which is therefore unaware of any logical connections between any requests.

Each HTTP message (request or response) consists of a string of 8-bit characters, so it can be treated as text by programs that read HTTP messages (and can be read by humans if they have a program that can eavesdrop on them). Messages conform to a simple rigid structure, consisting of an initial line (the ***request line*** for a request, the ***status line*** for a response) containing the essential message, followed by one or more headers, containing various parameters and modifiers. These may be followed by the message ***body***, which contains data – such as the contents of a file being sent by the server – if there is any. Headers are separated from the data by a blank line. (The line terminator is always the combination of a carriage return followed by a linefeed, irrespective of the conventions of the operating system running either the server or client.)

A request line comprises three elements. The ***method*** is a name identifying the service being requested: the most commonly used methods are GET, which is used to request a file or other resource, and POST, which is used to send data from a form. The ***identifier*** comes next, and tells the server which resource is being requested, for example by giving the path name of a file. Finally, the HTTP ***version*** indicates which protocol version the client is using.

For example, if a user clicked on a link with its href attribute pointing to the URL http://www.webdesignbook.org/Info/index.html, their Web browser would connect to the host with name www.webdesignbook.org, using port 80 so that communication would go to the Web server on that machine. It would then send an HTTP request, whose request line was:

```
GET /Info/index.html HTTP/1.1
```

The headers that may follow a request line all take the form of a header name followed by a colon and some arguments. For example, the following two headers can be found after the request line shown above:

```
Host: www.webdesignbook.org
User-Agent: Mozilla/5.0 (Macintosh; U; PPC Mac OS X 10.5;en-GB; rv:1.9) ↩
Gecko/2008061004 Firefox/3.0
```

(The User-Agent header has been split over two lines to fit on the page here, as indicated by the ↩ symbol.)

The Host header tells the server the host name that the request is directed at. This is necessary, because nowadays it is common for several host names to correspond to a single IP address, using

a mechanism known as "virtual hosting", which is implemented by most important Web servers, and this might make request identifiers ambiguous. The User-Agent header identifies the Web browser (or other user agent) that is making the request. This should allow the server to make allowances for any known problems with this particular browser, if it chooses to. However, it is common for Web browsers to send User-Agent headers that falsely identify them, so the information in this header is often unreliable.

One of the most commonly seen headers in GET requests is Accept. Its arguments indicate – using Internet media types – the range of types of data that the browser can deal with. For example,

```
Accept: image/gif, image/x-xbitmap, image/jpeg
```

is sent by a browser to indicate that it is able and willing to display GIF, JPEG and X bitmap images. Browsers may send several Accept headers instead of combining them. Web servers will only send them data that is of an acceptable type. In a media type used in this way, a * can be used as a wildcard character, so

```
Accept: image/*
```

indicates that the browser will accept any image format, and

```
Accept: */*
```

that it will accept any type of data for which there is an Internet media type.

Two similar headers, Accept-Charset and Accept-Language, are used by browsers to inform servers about the character sets and languages they will accept. The following would be typical of the headers sent by a browser set up in an English-speaking country where British usage is preferred:

```
Accept-Language: en-gb,en
Accept-Charset: UTF-8
```

Since a GET request does not send any data, its body is empty – the message terminates with the blank line.

The first line of an HTTP server's response is the status line, indicating how it coped with the request. This line begins with the protocol version, telling the client which HTTP version the server is using. Next comes a numerical status code, whose meaning is defined by the HTTP standard, followed by a short phrase, explaining to human readers what the code means. If all goes well, the code will be 200, which means OK, as shown:

```
HTTP/1.1 200 OK
```

Just as the client told the server who it was in the User-Agent header, the server introduces itself in the Server header. It also dates and times the response, and uses the Content-type header to inform the client about the media type of the data being returned. For example:

```
Server: Apache/2.2.0 (Fedora)
Date: Mon, 16 Jun 2008 15:17:06 GMT
Content-Type: text/html; charset=UTF-8
```

Typically, a server's response does contain some data. In the case of a response to a GET request, for example, this will be the contents of the file that was requested. In this example response, after any other headers there would be a blank line, followed by the XHTML document itself – the contents of a text file, encoded using UTF-8, containing the markup and text of the Web page in question. Any images and other embedded material would not be returned at this point, but the browser would send additional requests when it encountered markup referring to an external resource.

For instance, if the document that was returned in response to the request we have just described included a style element that referenced a stylesheet called styles.css, the browser would fetch it by sending another request, which included the following:

```
GET /styles.css HTTP/1.1
Host: www.webdesignbook.org
User-Agent: Mozilla/5.0 (Macintosh; U; PPC Mac OS X 10.5; en-GB; rv:1.9)↩
Gecko/2008061004 Firefox/3.0
Accept: text/css,*/*
Accept-Language: en-gb,en
```

The server should send a response with the following headers (plus some others):

```
HTTP/1.x 200 OK
Date: Mon, 16 Jun 2008 15:17:06 GMT
Server: Apache/2.2.0 (Fedora)
Content-Type: text/css
```

Again, the headers will be followed by a blank line and then the stylesheet itself. Similar requests will be made for the images in the page, and similar responses will be used to return them. The interaction is illustrated in Figure 16.5.

We will describe some other noteworthy headers shortly.

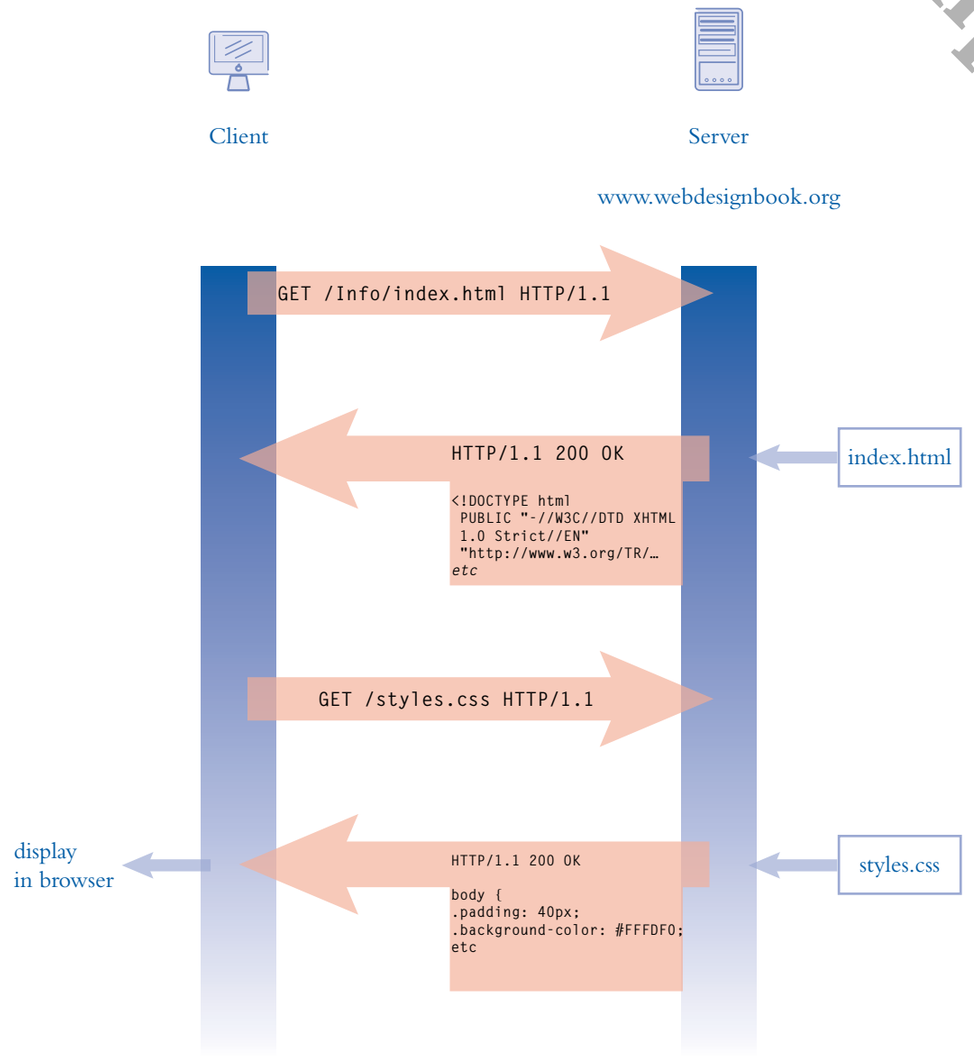


Figure 16.5. ***HTTP requests and responses***

The status code in a response is not always 200, unfortunately. The HTTP 1.1 standard defines many 3-digit return codes. These are divided into groups, by their first digit. Codes less than 200 are informative; the only such codes presently defined are 100, which means "continuing", and is only applicable to persistent connections, and 101, which is sent when the server switches to a different protocol, for example to stream some real-time data.