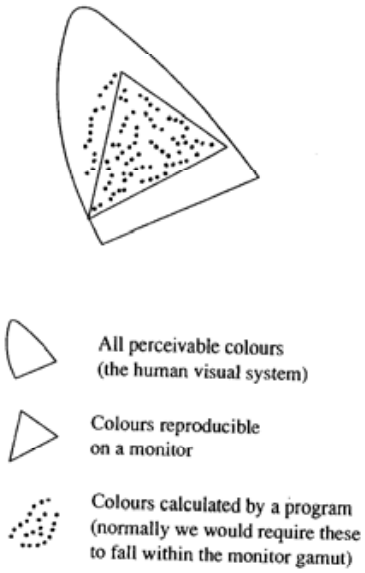


Figure 15.1
The hierarchy of colour sets relevant to computer imagery. A colour is a two-dimensional point in this space.

space in which reside all the colours that we have an interest in. First of all we need to define the hierarchy of colour sets that we will be referring to. These are:

(1) The set of all colours perceivable by human beings with normal colour vision.

(2) The set of colours that can be displayed by a monitor screen or captured by an input device. This is a subset of (1) for reasons that will become clear in the course of this chapter.

(3) The set of colours that can be calculated by a graphics program and stored in a frame memory. For a 24-bit system (16 million colours) this will generally be a subset of (1) but a superset of (2). That is unless we take special precautions we may generate colours that are outside the display gamut or range.

The hierarchy is illustrated (Figure 15.1) in a cross-section of a three-dimensional colour space that will be explained later.

15.2 Colour and three-dimensional space

Why is colour a three-component vector? Again we have to bear in mind that colour is a human sensation. Traditionally we describe colours in words, usually by allusion to common objects 'apple green' or 'blood red' etc. More precisely, colour is communicated in the painting and dyeing industry by the production of charts of sample colours. The numerical specification of colour has a long

Volkswagen 1011 - Part 5 of 6

history that began with Isaac Newton, but it was only in the twentieth century that numerical systems became important industrially.

The answer to the question 'why is colour specified by three numerical labels?' is that we have three different types of cone in our retinas which have different sensitivities to different wavelengths (Figure 15.7(a)). Light can be specified physically as a spectral power distribution or SPD – the objective measurement of light energy as a function of wavelength – and we should be able to categorize the effect of any SPD on a human observer by three weights – the relative response of the three different types of cone. And so it happens that we can visually match a sample colour by additively mixing three coloured lights. We can, for example, match a sample or target colour by controlling the three intensities of a red, green and a blue light. However, note the important point that in matching with primary colours red, green and blue we are not basing the labelling of an SPD on the cone spectral sensitivity curves, but are using the human vision system to match colours with a mix of primaries. To do this for all colours on a wavelength-by-wavelength basis leads to spectral sensitivity curves that our retinas would have if the cones responded maximally to these colours. The reason for this somewhat convoluted approach is that we can derive these functions easily from colour matching experiments; precise knowledge of the actual spectral sensitivity curves of the retina was harder to come by.

Thus numerical specification of colour is by a triple of primary colours. Most, but not all, perceivable colours can be produced by additively mixing appropriate amounts of three primary colours (red, green and blue, for example). If we denote a colour by **C**, we have:

$$\mathbf{C} = r\mathbf{R} + g\mathbf{G} + b\mathbf{B}$$

where r, g and b are the relative weights of each primary required to match the colour to be specified. The important point here is that this system, even though it is not specifying information related directly to the SPD of the colour, is saying that a colour **C** can be specified by a numerical triple because if a matching experiment was performed an observer would choose the components r, g, b to match or simulate the colour **C**.

In a computer graphics monitor a colour is produced by exciting triples of adjacent dots made of red, green and blue phosphors. The dots are small and the eye perceives the triples as a single dot of colour. Thus we specify or label colours in reality using three primaries and the production of colours on a monitor is also specified in a similar way. However, note the important distinction that colour on a monitor is not produced by mixing the radiation from three light sources but by placing the light sources in close proximity to each other.

Unfortunately in computer graphics this three-component specification of colour together with the need to produce a three-component RGB signal for a monitor has led to a widely held assumption that light–object interaction need only be evaluated at three points in the spectrum. This is the 'standard' RGB paradigm that tends to be used in Phong shading, ray tracing and radiosity. If it is intended to simulate accurately the interaction of light with objects in a scene,

then it is necessary to evaluate this interaction at more than three wavelengths; otherwise aliasing will result in the colour domain because of undersampling of the light distribution and object reflectivity functions. Of course, aliasing in the colour domain simply consists of a shift in colour away from a desired effect and in this sense it is invisible. (This is in direct contrast to spatial domain aliasing which produces annoying and disturbing visual artefacts.) Colours in most computer graphics applications are to a great extent arbitrary and shifts due to inaccurate simulation in the colour domain are generally not important. It is only in applications where colour is a subtle part of the simulation, say, for example, in interior design, that these effects have to be taken into account.

Given that we can represent or describe the sensation of colour, as far as colour matching experiments are concerned, with numeric labels, we now face the question: which numbers shall we use? This heralds the concept of different colour spaces or domains.

It may be as we suggested in the previous section, that a calculation or rendering domain be a wavelength or spectral space. Eventually, however, we need to produce an image in $RGB_{monitor}$ space to drive a particular monitor. What about the storage and communication of images? Here we need a universal standard. $RGB_{monitor}$ spaces, as we shall see, are particular to devices. These devices have different gamuts or colour ranges all of which are subsets of the set of perceivable colours. A universal space will be device independent and will embrace all perceivable colours. Such a space exists and is known as the CIE XYZ standard. A CIE triple is a unique numeric label associated with any perceivable colour.

Another requirement in computer graphics is a facility that allows a user to manipulate and design using colour. It is generally thought that an interface that allows a user to mix primary colours is anti-intuitive and spaces that are inclined to perceptual sensations such as hue, saturation and lightness are preferred in this context.

We now list the main colour spaces used in computer imagery.

(1) CIE XYZ space: the dominant international standard for colour specification. A colour is specified as a set of three tri-stimulus values or artificial primaries XYZ.

(2) Variations or transformations of CIE XYZ space (such as CIE xyY space) that have evolved over the years for different contexts. These are transforms of CIE XYZ that better reflect some detail in the perception of colour, for example, perceptual linearity.

(3) Spectral space: in image synthesis light sources are defined in this space as n wavelength samples of an intensity distribution. Object reflectivity is similarly defined. A colour specified on a wavelength-by-wavelength basis is how we measure colour with a device such as a spectrophotometer. As we have pointed out, this does not necessarily relate to our perception of an SPD as one colour or another. We synthesize an image at n wavelengths and then need to 'reduce' this to three components for display.

(4) RGB space: the 'standard' computer graphics paradigm for Phong shading. This is just a three-sample version of spectral space, light sources and object reflectivity are specified as three wavelengths: Red, Green and Blue. We understand the primaries R, G and B to be pure or saturated colours.

(5) $RGB_{monitor}$ space: a triple in this space produces a particular colour on a particular display. In other words it is the space of a display. The same triple may not necessarily produce the same colour sensation on different monitors because monitors are not calibrated to a single standard. Monitor RGBs are not pure or saturated primaries because the emission of light from an excited phosphor exhibits a spectral power distribution over a band of frequencies. If the usual three-sample approach is used in rendering then usually whatever values are calculated in RGB space are assumed to be weights in $RGB_{monitor}$ space. If an n sample calculation has been performed then a device-dependent transformation is used to produce a point in $RGB_{monitor}$ space.

(6) HSV space: a non-linear transformation of RGB space enabling colour to be specified as Hue, Saturation and Value.

(7) YIQ space: a non-linear transformation of RGB space used in analogue TV.

We will now deal with the issues surrounding these colour spaces. We will start with RGB space because it is the most familiar and easiest to use. We will then look at certain problems that lead us on to consideration of CIE space.

15.2.1 RGB space

Given the subtle distinction between (4) and (5) above we now describe RGB space as a general concept. This model is the traditional form of colour specification in computer imagery. It enables, for example, diffuse reflection coefficients in shading equations to be given a value as a triple (R, G, B). In this system (0, 0, 0) is black and (1, 1, 1) is white. Colour is labelled as relative weights of three primary colours in an additive system using the primaries Red, Green and Blue. The space of all colour available in this system is represented by the RGB cube (Figure 15.2 and Figure 15.3 (Colour Plate)). Important points concerning RGB space are:

(1) It is perceptually non-linear. Equal distances in the space do not in general correspond to perceptually equal sensations. A step between two points in one region of the space may produce no perceivable difference; the same increment in another region may result in a noticeable colour change. In other words, the same colour sensation may result from a multiplicity of RGB triples. For example, if each of RGB can vary between 0 and 255, then over 16 million unique RGB codes are available.

(2) Because of the non-linear relationship between RGB values and the intensity produced at each phosphor dot (see Section 15.5), low RGB values produce

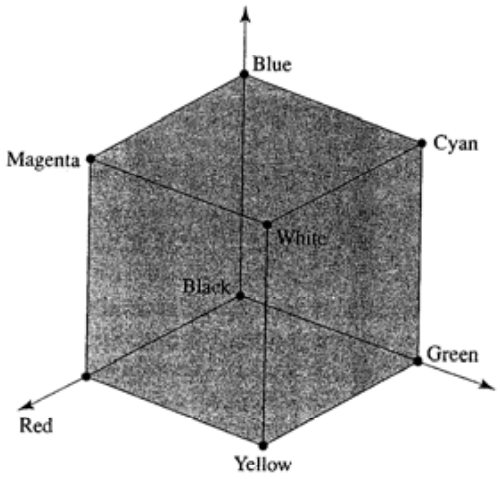


Figure 15.2
The RGB colour solid. See also Figure 15.3 (Colour Plate).

small changes in response on the screen. As many as 20 steps may be necessary to produce a 'just noticeable difference' at low intensities; whereas a single step may produce a perceivable difference at high intensities.

(3) The set of all colours produced on a computer graphics monitor, the RGB space, is always a subset of the colours that can be perceived by humans. This is not peculiar to RGB space. Any set of three visible primaries can only produce through additive mixing of a subset of the perceivable colour set.

(4) It is not a good colour description system. Without considerable experience, users find it difficult to give RGB values to colours known by label. What is the RGB value of 'medium brown'? Once a colour has been chosen it may not be obvious how to make subtle changes to the nature of the colour. For example, changing the 'vividness' of a chosen colour will require unequal changes in the RGB components.

15.2.2 The HSV single hexcone model

The H(ue) S(aturation V(alue) or single hexcone model was proposed by A.R. Smith in 1978 (Smith 1978). Its purpose is to facilitate a more intuitive interface for colour than the selection of three primary colours. The colour space has the shape of a hexagonal cone or hexcone. The HSV cone is a non-linear transformation of the RGB cube and although it tends to be referred to as a perceptual model, it is still just a way of labelling colours in the monitor gamut space. Perceptual in this context means the attributes that are used to represent the colour are more akin to the way in which we think of colour; it does not mean that the space is perceptually linear. The perceptual non-linearity of RGB space is carried over into HSV space; in particular, perceptual changes in hue are distinctly non-linear in angle.

It can be employed in any context where a user requires control or selection of a colour or colours on an aesthetic or similar basis. It enables control over the range or gamut of an RGB monitor using the perceptually based variables Hue, Saturation and Value. This means that a user interface can be constructed where the effect of varying one of the three qualities is easily predictable. A task such as make a colour brighter, paler or more yellow is far easier when these perceptual variables are employed, than having to decide on what combinations of RGB changes are required.

The HSV model is based on polar coordinates rather than Cartesian and H is specified in degrees in the range 0 to 360. One of the first colour systems based on polar coordinates and perceptual parameters was that due to Munsell. His colour notation system was first published in 1905 and is still in use today. Munsell called his perceptual variables Hue, Chroma and Value and we can do no better than reproduce his definition for these. Chroma is related to saturation – the term that appears to be preferred in computer graphics.

Munsell's definitions are:

- Hue: 'It is that quality by which we distinguish one colour family from another, as red from yellow, or green from blue or purple.'
- Chroma: 'It is that quality of colour by which we distinguish a strong colour from a weak one; the degree of departure of a colour sensation from that of a white or grey; the intensity of a distinctive hue; colour intensity.'
- Value: 'It is that quality by which we distinguish a light colour from a dark one.'

The Munsell system is used by referring to a set of samples – the Munsell Book of Colour. These samples are in 'just discriminable' steps in the colour space.

The HSV model relates to the way in which artists mix colours. Referring to the difficulty of mentally imagining the relative amounts of R, G and B required to produce a single colour, Smith says:

> Try this mixing technique by mentally varying RGB to obtain pink or brown. It is not unusual to have difficulty. . . . the following [HSV] model mimics the way an artist mixes paint on his palette: he chooses a pure hue, or pigment and lightens it to a tint of that hue by adding white, or darkens it to a shade of that hue by adding black, or in general obtains a tone of that hue by adding some mixture of white and black or grey.

In the HSV model, varying H corresponds to selecting a colour. Decreasing S (desaturating the colour) corresponds to adding white. Decreasing V (devaluing the colour) corresponds to adding black. The derivation of the transform between RGB and HSV space is easily understood by considering a geometric interpretation of the hexcone. If the RGB cube is projected along its main diagonal onto a plane normal to that diagonal, then a hexagonal disc results.

The following correspondence is then established between the six RGB vertices and the six points of the hexcone in the HSV model:

RGB		HSV
(100)	red	(0, 1, 1)
(110)	yellow	(60, 1, 1)
(010)	green	(120, 1, 1)
(011)	cyan	(180, 1, 1)
(001)	blue	(240, 1, 1)
(101)	magenta	(300, 1, 1)

where H is measured in degrees. This hexagonal disc is the plane containing V = 1 in the hexcone model. For each value along the main diagonal in the RGB cube (increasing blackness) a contained sub-cube is defined. Each sub-cube defines a hexagonal disc. The stack of all hexagonal discs makes up the HSV colour solid.

Figure 15.4 shows the HSV single hexcone colour solid and Figure 15.5 (Colour Plate) is a further aid to its interpretation showing slices through the achromatic axis. The right-hand half of each slice is the plane of constant H and the left-hand half that of H + 180.

Apart from perceptual non-linearity another subtle problem implicit in the HSV system is that the attributes are not themselves perceptually independent. This means that it is possible to detect an apparent change in Hue, for example, when it is the parameter Value that is actually being changed.

Finally, perhaps the most serious departure from perceptual reality resides in the geometry of the model. The colour space labels all those colours reproducible on a computer graphics monitor and implies that all colours on planes of constant V are of equal brightness. Such is not the case. For example, maximum intensity blue has a lower perceived brightness than maximum intensity yellow. We conclude from this that because of the problems of perceptual non-linearity and the fact that different hues at maximum V exhibit different perceptual values, representing a monitor gamut with any 'regular' geometric solid such as a cube or a hexcone is only an approximation to the sensation of colour and this fact means that we have to consider perceptually based colour spaces.

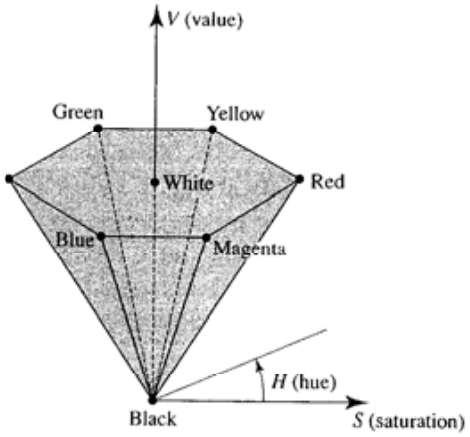


Figure 15.4
HSV single hexcone colour solid. See also Figure 15.5 (Colour Plate).

A simpler way of expressing this fact is to reiterate that colour is a perceptual sensation and cannot be accurately labelled by dividing up the RGB voltage levels of a monitor and using this scale as a colour label. This is essentially what we are doing with both the RGB and the HSV model and the association of the word 'perceptual' with the HSV model is unfortunate and confusing.

15.2.3 YIQ space

YIQ space is a linear transformation of RGB space that is the basis for analogue TV. Its purpose is efficiency in terms of bandwidth usage (compared with the RGB form) and to maintain compatibility for black and white TV (all the information required for black and white reception is contained in the Y component).

$$\begin{bmatrix} Y \\ I \\ Q \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.144 \\ 0.596 & -0.275 & -0.321 \\ 0.212 & -0.523 & 0.311 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

Note that the constant matrix coefficients mean that the transformation assumes that the RGB components are themselves defined with respect to a standard (in this case an NTSC definition). The Y component is the same as the CIE Y primary (see Section 15.3.1) and is called luminance. Colour information is 'isolated' in the I and Q components (equal RGB components will result in zero I and Q values). The bandwidth optimization comes about because human beings are more sensitive to changes in luminance than to changes in colour in this sense. We can discriminate spatial detail more finely in grey scale changes than in colour changes. Thus, a lower bandwidth can be tolerated for the I and Q components resulting in a bandwidth saving over using RGB components.

Colour representations where the colour and luminance information are separated are important in image processing where we may want to operate on image structure without affecting the colour of the image.

15.3 Colour, information and perceptual spaces

We now come to consider the use of perceptual spaces in computer imagery. In particular we shall look at the CIE XYZ space – an international numerically based colour labelling system first introduced in 1931 and derived from colour matching experiments.

To deal with colour reality we need to manipulate colours in a space that bears some relationship to perceptual experience. We have already alluded to applications where such considerations may be important. For example, in CAAD for interiors, the design of fabrics or the finish on such expensive consumer durables as cars, it will be necessary for computer graphics to move out of the arbitrary RGB domain into a space where colour is accurately simulated. Of course, in

attempting to transmit an illusion of reality in a computer graphics simulation there are many other factors involved – surface texture, the macroscopic nature of the colour (metallic paint or ordinary gloss paint, for example) and geometrical accuracy, but at the moment in computer graphics it is the case that the RGB triple is the *de facto* standard for rendering.

Colour is used much in visualization applications to communicate numerical information. This has a long history. Possibly the most familiar manifestation is a coloured terrain map. Here colours are chosen to represent height. Traditionally colours are chosen with green representing low heights. Heights from 0 to 100 m may be represented by lightening shades of green through to yellow. Darkening shades of brown may represent the range 1000 to 3000 m. Above 3000 m there are usually two shades of purple, and white is reserved for 6000 m and above.

This technique has been used in image processing and computer graphics where it is called pseudo-colour enhancement. It is used most commonly to display a function of two variables, $f(x, y)$, in two space where before such a function would have been displayed using 'iso-f'contours. In pseudo-colour enhancement a deliberately restricted colour list (of, say, 10 colours) is chosen and the value of f is mapped into the nearest colour. The function appears like a terrain map with islands of one colour against a background of another.

In computer graphics and image processing the most popular mapping of $f(x, y)$ into colour has been some variation of the rainbow colours with red used to represent high or hot and blue used for low intensity or cold – in other words a path around the outer edge of HSV space. One of the problems with this mapping is that depending on the number of colour steps used, transitions between different colours appear as false contours. Violent colour discontinuities appear in the image where the function f is continuous. There is a contradiction here: we need these apparent discontinuities to highlight the shape of the function but they can easily be interpreted as transitions in the function where no transition exists. This is particularly true in non-mathematical images which are not everywhere continuous to start with. Natural discontinuities may exist in the function anyway, say in a medical image made up of the response of a device to different tissue. The appearance of false contours in such an image may be undesirable.

Thus, whether the contours add to or subtract from the perception of the nature and shape of f depends in the end on the image context. The effect of false contours is easily diminished by adding more colours to the mapping but this may have the effect of making the function more difficult to interpret.

The use of perceptual colour spaces in the context of numerical information is extremely important. If an accurate association between colours and numeric information is required, then a perceptually linear colour scale should be used. We discussed in Section 15.2.1 the perceptual non-linearity of RGB space and it is apparent that unless this factor is dealt with, it will interfere with the association of a colour with a numeric value. There is no good reason, apart from

cultural associations like the example of the terrain map coding in cartography, why a hue circle should be used as a pseudo-colour scale.

The use of pseudo-colour in two space to display functions of two spatial variables has been around for many years. The last ten years have seen an increasing application of three-dimensional computer graphics techniques in the visualization of scientific results and simulations (an area that has been awarded the acronym ViSC). The graphics techniques used are mainly animation, volume rendering (both dealt with elsewhere in this text) and the use of pseudo-colour in three space, which we will now examine.

Figure 13.1 (Colour Plate) illustrates an application. It shows an isosurface extracted from a Navier–Stokes simulation of a reverse flow pipe combustor. In this simulation the primary gas flow is from left to right. Air is forced into the chamber under compression at the left, and dispersed by two fans. Eight fuel jets, situated radially approximately halfway along the combustor, are directed in such a way as to send the fuel mixture in a spiralling path towards the front of the chamber. Combustible mixing takes place in the central region and thrust is created at the exhaust outlet on the right. The isosurfaces shown connect all points where the net flow along the long axis is zero – a zero velocity surface.

Such an isosurface can be displayed by using conventional three-dimensional rendering techniques as the illustration demonstrates. In the second illustration we have sought to superimpose a pseudo-colour that represents temperature. A spectral colour path, from blue to magenta, around the circumference of the HSV cone is used.

Thus, in the same three-dimensional image we are trying to represent two functions simultaneously. First, the shape of an isosurface and, second, the temperature at every point on the isosurface. Perceptual problems arise in this case because we are using colour to represent both shape and temperature, whereas normally the colour is experienced as an association with a single phenomenon. For example, it tends to be difficult in such representations, to interpret the shape of the isosurface in regions of rapidly varying hue or temperature. Nevertheless representational schemes like this are becoming commonplace in visualization techniques. They represent a kind of summary of complex data that, prior to the use of three-dimensional computer graphics, could only be examined one part at a time. For example, the simulation in the illustration may have been investigated by using a rotating cross-section. This leaves the difficult task of building up a three-dimensional picture of the data to the brain of the viewer.

15.3.1 CIE XYZ space

We have discussed in previous sections that we need spectral space to try to simulate reality. This implies that we need a way of 'reducing' or converting spectral space calculations for a monitor display. Also, we saw that we need perceptual colour spaces for choosing mappings for pseudo-colour enhancement. Another

raison d'être for perceptual colour spaces in computer graphics is for the storage and the communication of files within the computer graphics community and for communication between computer graphicists and industries that use colour.

The CIE standard allows a colour to be specified as a numeric triple (X, Y, Z). CIE XYZ space embraces all colours perceivable by human beings and it is based on experimentally determined colour matching functions. Thus, unlike the three previous colour spaces, it is not a monitor gamut space.

The basis of the standard, adopted in 1931, was colour matching experiments where a user controls or weights three primary light sources to match a target monochromatic light source. The sources used were almost monochromatic and were **R** = 700 nm, **G** = 546.1 nm and **B** = 435.8 nm. In other words the weights in:

$$\mathbf{C} = r\mathbf{R} + g\mathbf{G} + b\mathbf{B}$$

are determined experimentally.

The result of such experiments can be summarized by colour matching functions. These are shown in Figure 15.6(b) and show the amounts of red, green and blue light which when additively mixed will produce in a standard observer a monochromatic colour whose wavelength is given by λ. That is:

$$\mathbf{C}_\lambda = r(\lambda) + g(\lambda) + b(\lambda)$$

For any colour sensation **C** which exhibits an SPD $P(\lambda)$, r, g and b are given by:

$$r = k \int_\lambda P(\lambda) r(\lambda) d(\lambda)$$

$$g = k \int_\lambda P(\lambda) g(\lambda) d(\lambda)$$

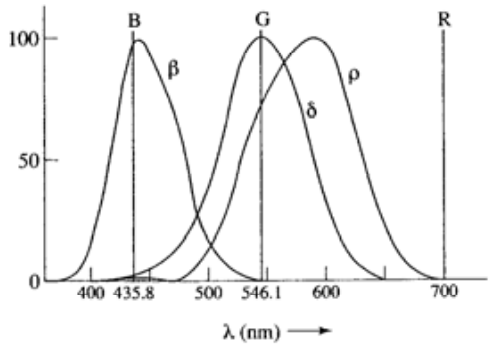
$$b = k \int_\lambda P(\lambda) b(\lambda) d(\lambda)$$

Thus, we see that colour matching functions reduce a colour **C**, with any shape of spectral energy distribution to a triple rgb. At this stage we should make the extremely important point that the triple rgb bears no relationship whatever to a triple RGB specified in the aforementioned (computer graphics) system. As we discussed in Section 15.2, computer graphicists understand the triple RGB to be three samples of the SPD of an illuminant or three samples of the reflectivity function of the object which are linearly combined in rendering models to produce a calculated RGB for reflected light. In other words, we can render by working with three samples or we can extend our approach to working with *n* samples. In contrast the triple rgb is *not* three samples of an SPD but the values obtained by integrating the product of the SPD and each matching function. In other words, it is a specification of the SPD as humans see it (in terms of colour matching) rather than as a spectrophotometer would see it.

There is, however, a problem in representing colours with an additive primary system which is that with positive weights, only a subset of perceivable colours can be described by the weights (r, g, b). The problem arises out of the fact that when two colours are mixed the result is a less saturated colour. It is impossible

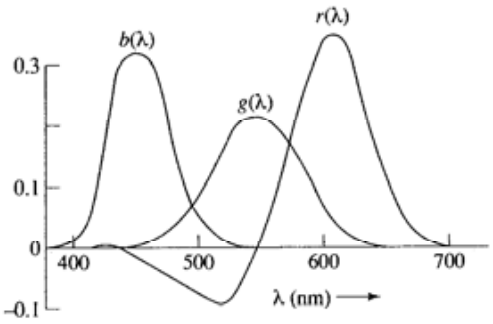
Figure 15.6
The 'evolution' of the CIE colour matching functions.

Spectral sensitivity curves of the ρ, δ and β cones in the retina and their relationship to the monochromatic colours:
red = 700 nm
green = 546.1 nm
blue = 435.8 nm



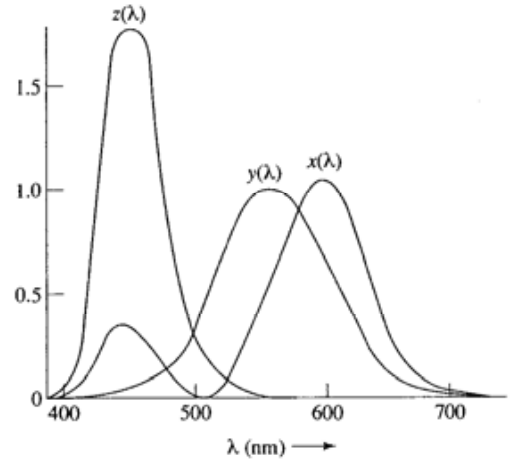
(a)

RGB colour matching functions for the CIE 1931 Standard Colourimetric Observer



(b)

CIE matching functions for the CIE 1931 Standard Colourimetric Observer



(c)

to form a highly saturated colour by superimposing colours. Any set of three primaries forms a bounded space outside of which certain perceivable highly saturated colours exist. In such colours a negative weight is required.

To avoid negative weights the CIE devised a standard of three supersaturated (or non-realizable) primaries X, Y and Z, which, when additively mixed, will produce all perceivable colours using positive weights. The three corresponding matching functions $x(\lambda)$, $y(\lambda)$ and $z(\lambda)$ shown in Figure 15.6(c) are always positive. Thus we have:

$$X = k \int_{\lambda} P(\lambda)x(\lambda)d(\lambda)$$

$$Y = k \int_{\lambda} P(\lambda)y(\lambda)d(\lambda)$$

$$Z = k \int_{\lambda} P(\lambda)z(\lambda)d(\lambda)$$

where:

$k = 680$ for self-luminous objects

The space formed by the XYZ values for all perceivable colours is CIE XYZ space. The matching functions are transformations of the experimental results. In addition the $y(\lambda)$ matching function was defined to have a colour matching function that corresponded to the luminous efficiency characteristic of the human eye, a function that peaks at 550 nm (yellow-green).

The shape of the CIE XYZ colour solid is basically conical with the apex of the cone at the origin (Figure 15.7). Also shown in this space is a monitor gamut which appears as a parallelepiped. If we compare this space to HSV space we can

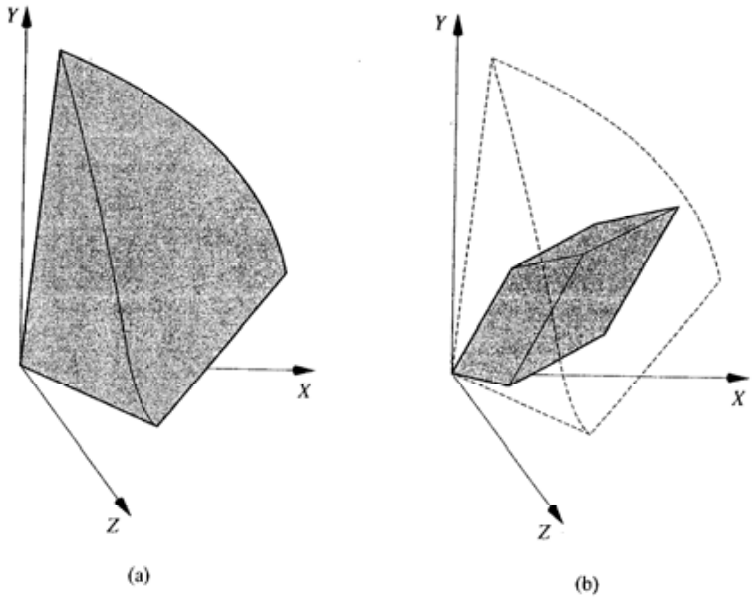


Figure 15.7
(a) CIE XYZ solid.
(b) A typical monitor gamut in CIE XYZ space.

view the solid as distorted HSV space. The black point is at the origins and the HSV space is deformed to embrace all colours and to encompass the fact that the space is based on perceptual measurements. If we consider, for example, the outer surface of the deformed cone, this is made of rays that emanate from the origin terminating on the edge of the cone. Along any ray is the set of colours of identical chromaticity (see the next section). If a ray is moved in towards the white point, situated on the base of the deformed cone then we desaturate the set of colours specified by the ray. Within this space, the monitor gamut is a deformed (sheared and scaled) cube, forming a subset of the volume of perceivable colours.

15.3.2 CIE xyY space

An alternative way of specifying the (X, Y, Z) triple is (x, y, Y) where (x, y) are known as chromaticity coordinates:

$$x = \frac{X}{X + Y + Z}$$

$$y = \frac{Y}{X + Y + Z}$$

Plotting x against y for all visible colours yields a two-dimensional (x, y) space known as the CIE chromaticity diagram.

The wing-shaped CIE chromaticity diagram (Figure 15.8) is extensively used in colour science. It encompasses all the perceivable colours in two-dimensional space by ignoring the luminance Y. The locus of the pure saturated or spectral colours is formed by the curved line from blue (400 nm) to red (700 nm). The straight line between the end points is known as the purple or magenta line. Along this line is located the purples or magentas. These are colours whose perceivable sensation cannot be produced by any single monochromatic stimulus, and which cannot be isolated from daylight.

Also shown in Figure 15.8 is the gamut of colours reproducible on a computer graphics monitor from three phosphors. The monitor gamut is a triangle formed by drawing straight lines between three RGB points. The RGB points are contained within the outermost curve of monochromatic or saturated colours. Examination of the emission characteristics of the phosphors will reveal a spread about the dominant wavelength which means that the colour contains white light and is not saturated. When, say, the blue and green phosphors are fully excited their emission characteristics add together into a broader band meaning that the resultant colour will be less saturated than blue or green.

The triangular monitor gamut in CIE xy space is to be found in most texts dealing with colour science in computer graphics, but it is somewhat misleading. The triangle is actually the projection out of CIE xyY space of the monitor gamut, with the vertices formed from phosphor vertices that each have a different luminance. Figure 15.9 shows the general shape of monitor gamut in CIE

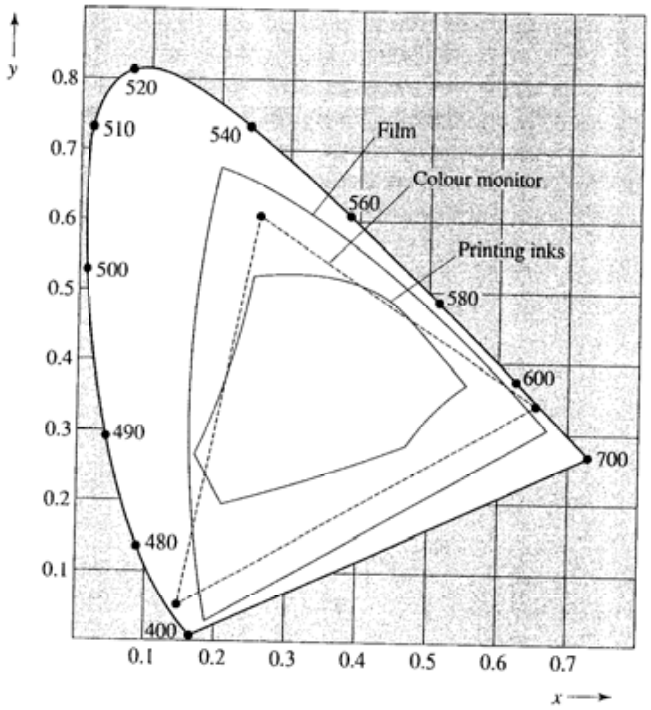


Figure 15.8
CIE chromaticity diagram showing typical gamuts for colour film, colour monitor and printing inks.

xyY space and Figure 15.10 (Colour Plate) shows three slices through the space. The geometric or shape transformation from the scaled and sheared cube in XYZ space to the curvilinear solid (with six faces) in xyY space is difficult to interpret. For example, one edge of the cube maps to a single point.

There are a number of important uses of the CIE chromaticity diagram. We give one important example. It can be used to compare the gamut of various display devices. This is important in computer graphics when an image is eventually to be reproduced on a number of different devices. Figure 15.8 shows a CIE chromaticity diagram with the gamut of a typical computer graphics monitor together with the gamut for modern printing inks. The printing ink gamut is enclosed within the monitor gamut, which is itself enclosed by the gamut for

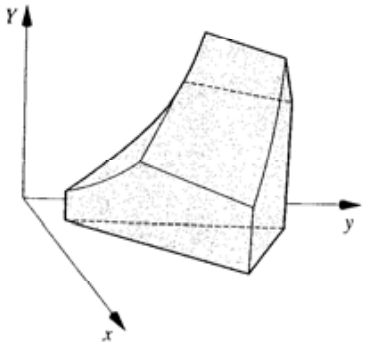


Figure 15.9
Monitor gamut in CIE xyY space (see also Figure 15.10 (Colour Plate)).

colour film. This means that some colours attainable on film are not reproducible on a computer graphics monitor, and certain colours on a monitor cannot be reproduced by printing. The gamut of display devices and reproduction techniques is always contained by the gamut of perceivable colours – the saturated or spectral colours being the most difficult to reproduce. However, this is not generally a problem because spectral and near spectral colours do not tend to occur naturally. It is the relative spread of device gamuts that is important rather than the size of any gamut with respect to the visual gamut.

15.4 Rendering and colour spaces

We have discussed reasons for the lack of accurate colours in computer graphics and now look at one of these reasons in more detail – colour aliasing is invisible.

Physics tells us that the light reflected from a surface, as a function of wavelength, is the product of the wavelength-dependent surface reflectance function and the spectral energy distribution function of the light source. If we simply evaluate this product at three wavelengths (the RGB Phong shading model discussed in Chapter 6) then clearly, because of the gross undersampling, we will not produce a result that simulates the real characteristic. What happens is that the three-sample approach will produce a colour shift away from the real colour. However, this shift is in most contexts completely invisible because we have no expectations of what particular colour should emerge from a computer graphics model anyway. A wrong colour does not necessarily look wrong.

To try to simulate real colour interaction numerically we can simply expand our three-sample rendering approach to n samples and work in spectral space, sampling the light source distribution function and the reflectivity of the object at appropriate wavelength intervals.

We look at three approaches which are summarized in Figure 15.11. The first – the *de facto* standard approach to rendering – takes no account of colour except in the most approximate way. The illuminant SPD is sampled at three wavelengths, or more usually arbitrarily specified as 1, 1, 1 for white light. Similarly the reflectivity of the object is specified at each of the R, G and B wavelengths. Three rendering equations/models are applied and the calculated RGB intensities are fed directly to the monitor without further alteration. This method produces works with input values that are arbitrary in the sense that a user may want to render a dark red object, but may not be concerned with specifying the colour of the object and illuminant to any degree of accuracy. Only three rendering equations are used.

The second approach applies the rendering equations in spectral space for a set of wavelengths ($n = 9$ appears to be a good compromise). Here the rendering cost is at least a factor of three greater than the 'arbitrary' colour method. The output from the renderer is a sampled intensity function and this must be transformed into (three-sample) $RGB_{monitor}$ space for display. The implication here is that if we have gone to the trouble to render at n wavelengths then we wish to display the

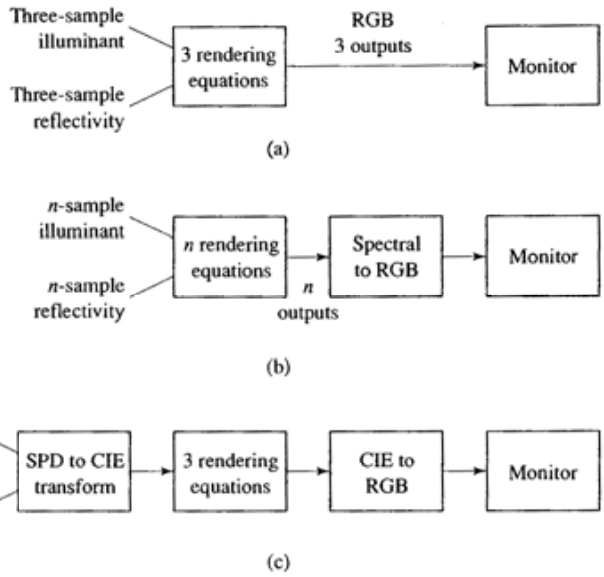


Figure 15.11
Rendering strategies and colours. (a) 'Standard' rendering for 'arbitrary' colour applications. (b) Spectral space rendering for colour-sensitive applications. (c) CIE space rendering for colour-sensitive applications.

result as accurately as possible and we need certain monitor parameters to be able to derive the spectral-to-$RGB_{monitor}$ transformation (see Section 15.5.2).

In the final approach we render in CIE space. This means specifying the SPD illuminant as CIE XYZ values using the matching functions. However, we have the problem of the surface reflectivity. What values do we use for this? This is a subtle point and the reader is referred to the paper by Borges (1991) which addresses exactly this issue. Here, we can note that we can simply express the reflectivity function as a CIE XYZ triple and use this in a three equation rendering approach. The output from the renderer is a CIE XYZ triple and we then require a CIE-to-$RGB_{monitor}$ transform to display the result.

The difference between an image produced by 'spectral rendering' and 'RGB rendering' is shown (to within the limits of the reproduction process) in Figure 15.12 (Colour Plate) for a ray tracer.

We must remember that we are only attending to a single aspect in the simulation of reality – which is the prevention of erroneous colour shifts due to undersampling in spectral space. Colour is also determined by the local reflection model itself. Defects in the accuracy with which the reflection model simulates reality still exist. We cannot overcome these simply by extending the number of samples in spectral space.

15.5 Monitor considerations

15.5.1 $RGB_{monitor}$ space and other monitor considerations

Serious use of colour in computer imagery needs careful attention to certain aspects of the display monitor. Computer graphics monitors are not standardized

and the application of the same RGB triple to different monitors will produce different colours on the screen. The most important factors are:

(1) Colour on a monitor is not produced by the superposition mixing of three lights, but relies on the eye to spatially mix the tiny light sources produced by three phosphor dots. There is nothing that we can do about this. One of the consequences is that saturated colours are not displayed at their full brightness – an area of pure red is only one-third red and two-thirds black. This means that, for example, even as we as human beings seem to compensate for this effect, taking photographs directly from the screen produces poor results.

(2) Different monitors are manufactured with phosphors that have different spectral energy distributions. For example, different phosphors are used to achieve different persistences (the length of time a phosphor glows after being activated). This can be corrected by a linear transformation as we demonstrate in the next section.

(3) The relationship between the RGB values applied to the monitor and the intensity of light produced on the screen is non-linear. The cure for this, gamma correction – a non-linear transformation – is described in the next section.

(4) In image synthesis, shading equations can produce colours that are outside the gamut of the monitor – undisplayable colours. We have somehow to clip these colours or bring them back into the monitor gamut. This is also a non-linear operation.

15.5.2 Monitor considerations – different monitors and the same colour

Contexts in which real colours are produced in computer imagery are, for example, rendering in spectral space and using perceptual space mapping. With spectral space we can produce a CIE XYZ triple from our final set of results. CIE XYZ space is used as a final standard and we need a device-specific transformation to go from CIE XYZ space to the particular $RGB_{monitor}$ space.

We can write:

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} X_r & X_g & X_b \\ Y_r & Y_g & Y_b \\ Z_r & Z_g & Z_b \end{bmatrix} \begin{bmatrix} R_m \\ G_m \\ B_m \end{bmatrix}$$

$$= \boldsymbol{T} \begin{bmatrix} R_m \\ G_m \\ B_m \end{bmatrix}$$

where $\boldsymbol{T}$ is particular to a monitor and a linear relationship is assumed between the outputs from the phosphors and the RGB values. If $\boldsymbol{T}_1$ is the transformation for monitor 1 and $\boldsymbol{T}_2$ the transformation for monitor 2, then $\boldsymbol{T}_2^{-1}\,\boldsymbol{T}_1$ converts the

RGB values of monitor 1 to those for monitor 2. $\boldsymbol{T}$ can be calculated in the following way. We define:

$$D_r = X_r + Y_r + Z_r$$
$$D_g = X_g + Y_g + Z_g$$
$$D_b = X_b + Y_b + Z_b$$

giving:

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} D_r x_r & D_g x_g & D_b x_b \\ D_r y_r & D_g y_g & D_b y_b \\ D_r z_r & D_g z_g & D_b z_b \end{bmatrix} \begin{bmatrix} R_m \\ G_m \\ B_m \end{bmatrix}$$

where:

$$x_r = X_r/D_r \quad Y_r = Y_r/D_r \quad z_r = Z_r/D_r \text{ etc.}$$

Writing the coefficients as a product of two matrices we have:

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} x_r & x_g & x_b \\ y_r & y_g & y_b \\ z_r & z_g & z_b \end{bmatrix} \begin{bmatrix} D_r & 0 & 0 \\ 0 & D_g & 0 \\ 0 & 0 & D_b \end{bmatrix} \begin{bmatrix} R_m \\ G_m \\ B_m \end{bmatrix}$$

where the first matrix is the chromaticity coordinates of the monitor phosphor. We now specify that equal RGB voltages of (1, 1, 1) should produce the alignment white:

$$\begin{bmatrix} X_w \\ Y_w \\ Z_w \end{bmatrix} = \begin{bmatrix} x_r & x_g & x_b \\ y_r & y_g & y_b \\ z_r & z_g & z_b \end{bmatrix} \begin{bmatrix} D_r \\ D_g \\ D_b \end{bmatrix}$$

For example, with standard white D_{65} we have:

$$x_w = 0.313 \quad y_w = 0.329 \quad z_w = 0.358$$

and scaling the white point to give unity luminance yields:

$$X_w = 0.951 \quad Y_w = 1.0 \quad Z_w = 1.089$$

Example chromaticity coordinates for an interlaced monitor (long persistence phosphors) are:

	x	y
red	0.620	0.330
green	0.210	0.685
blue	0.150	0.063

Using these we have:

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} 0.584 & 0.188 & 0.179 \\ 0.311 & 0.614 & 0.075 \\ 0.047 & 0.103 & 0.939 \end{bmatrix} \begin{bmatrix} R_m \\ G_m \\ B_m \end{bmatrix}$$

Inverting the coefficient matrix gives:

$$\begin{bmatrix} R_m \\ G_m \\ B_m \end{bmatrix} = \begin{bmatrix} 2.043 & -0.568 & -0.344 \\ -1.036 & 1.939 & 0.043 \\ 0.011 & -0.184 & 1.078 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

The significance of the negative components is that RGB space is a subset of XYZ space, and XYZ colours that lie outside the monitor gamut will produce negative RGB values.

15.5.3 Monitor considerations – colour gamut mapping

Monitor gamuts generally overlap and colours that are available on one monitor may not be reproducible on another. This is manifested by RGB values that are less than zero, or greater than one, after the transformation $\boldsymbol{T}_2^{-1}\boldsymbol{T}_1$ has been applied. This problem may also arise in rendering. In accurate colour simulation, using real colour values, it is likely that colour triples produced by the calculation may lie outside the monitor gamut. In other words the image gamut may be, in general, greater than the monitor gamut. This problem is even greater in the case of hard copy devices such as printers which have smaller gamuts than monitors.

The goal of the process is to compress the image gamut until it just fits in the device gamut in such a way that the image quality is maintained. This will generally depend on the content of the image and the whole subject area is still a research topic. There are, however, a number of simple strategies that we can adopt. The process of producing a displayable colour from one that is outside the gamut of the monitor is called 'colour clipping'.

Clearly we could adopt a simple clamping approach and limit out of range values. Better strategies are suggested by Hall (1989). Undisplayable colours fall into one of two categories:

(1) Colours that have chromaticities outside the monitor gamut (negative RGB values).

(2) Colours that have displayable chromaticities, but intensities outside the monitor gamut (RGB values greater than one).

Any correction results in a shift or change from the calculated colour and we can select a method depending on whether we wish to tolerate a shift in hue, saturation and/or value.

For the first category the best approach is to add white to the colour or to desaturate it until it is displayable. This maintains the hue or dominant wavelength and lightness at the cost of saturation. In the second case there are a number of possibilities. The entire image can be scaled until the highest intensity is in range; this has an effect similar to reducing the aperture in a camera. Alternatively the chromaticity can be maintained and the intensity scaled. Finally, the dominant hue and intensity can be maintained and the colour desaturated by adding white.

15.5.4

Monitor considerations – gamma correction

All of the foregoing discussion has implicitly assumed that there is a linear relationship between the actual RGB values input to a monitor and the intensity produced on the screen. This is not the case. That we need to maintain linearity comes from the fact that as far as possible we require a person viewing, say, a TV image of a scene on a monitor, to see the colour relationships as he perceives them from the scene. This implies that the end-to-end response of the TV system should be linear (Figure 15.13(a)). In a TV system gamma correction is applied at the camera (for reasons that also have to do with coding the signal optimally for noise) to pre-compensate for the monitor non-linearity. This is shown in Figure 15.13(b) which shows gamma correction introduced in the camera compensating for the non-linear relationship at the monitor. A computer graphics system (Figure 15.13(c)) is analogous to a TV camera with a linear intensity characteristic because the rendering calculations are linear. Because of this gamma correction is required after the calculation and this is usually implemented in the form of a look-up table.

Now consider the details. The red intensity, for example, produced on a monitor screen by an input value of R'_i is:

$$R_m = K(R'_i)^{\gamma_r}$$

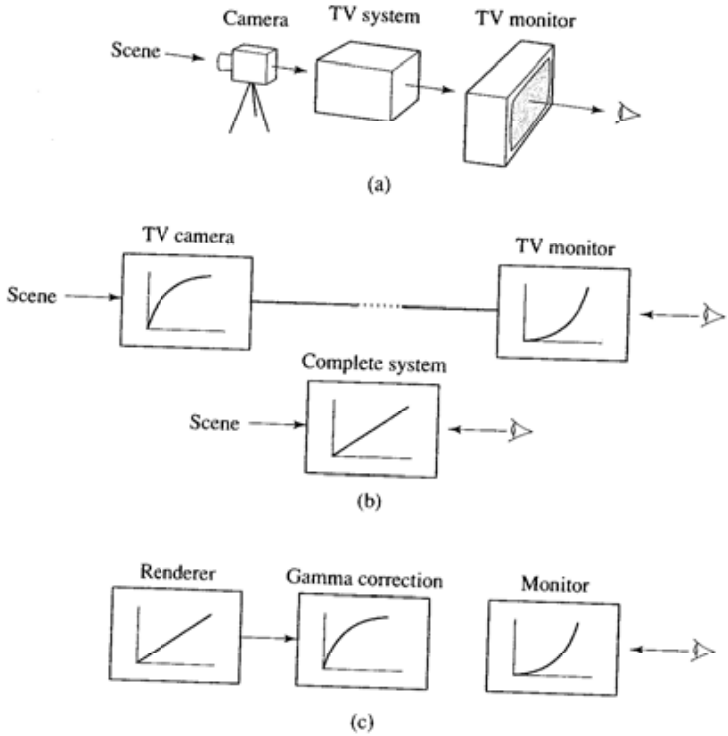


Figure 15.13
Gamma correction.
(a) A viewer should ideally see the same colours on a TV monitor as if he or she were viewing the scene.
(b) Gamma correction is applied in a TV camera.
(c) Computer graphics system.

where γ_r is normally in the range 2.3 to 2.8. The goal of the process is to linearize the relationship between the RGB values produced by the program and if γ_r, γ_g and γ_b are known then so-called gamma correction can be applied to convert the program value R_i to the value that when plugged into the above equation will result in a linear relationship. That is:

$$R'_i = k(R_i)^{1/\gamma_r}$$

An inexpensive method for determining γ is given in a paper by Cowan (1983). The two relationships are shown in Figure 15.14. The second graph is easily incorporated in a video look-up table. Note that the price paid for gamma correction is a reduction in the dynamic range. For example, if k is chosen such that 0 maps to 0 and 255 to 255 then 256 intensity levels are reduced to 167. This can cause banding and it is better to perform the correction in floating point and then to round.

Using a monitor with uncorrected gamma results in both intensity and chromaticity shifts away from the colour calculated by the program. Consider, for example, the triple (0, 255, 127). If this is not gamma corrected the

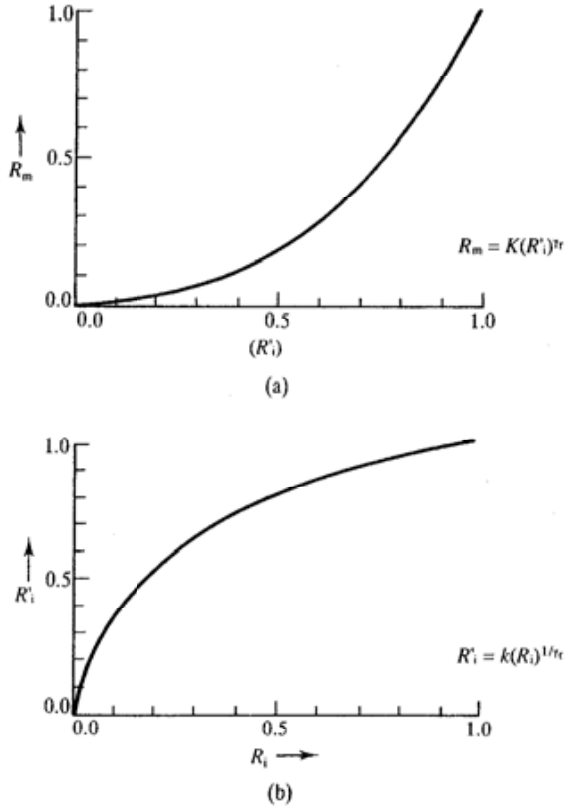


Figure 15.14
Gamma correction.
(a) Intensity as a function of applied voltage values.
(b) Corrected values as a function of applied ones.

display will decrease the blue component, leaving the red and green components unchanged.

Gamma correction leaves zero and maximum intensities unchanged and alters the intensity in mid-range. A 'wrong' gamma that occurs either because gamma correction has not been applied or because an inaccurate value of gamma has been used in the correction will always result in a wrong image with respect to the calculated colour.

16 Image-based rendering and photo-modelling

Introduction

A new field with many diverse approaches, image-based rendering (IBR) is difficult to categorize. The motivation for the name is that most of the techniques are based on two-dimensional imagery, but this is not always the case and the way in which the imagery is used varies widely amongst methods. A more accurate common thread that runs through all the methods is pre-calculation. All methods make cost gains by pre-calculating a representation of the scene from which images are derived at run-time. IBR has mostly been studied for the common case of static scenes and a moving view point, but applications for dynamic scenes have been developed.

There is, however, no debate concerning the goal of IBR which is to decouple rendering time from scene complexity so that the quality of imagery, for a given frame time constraint, in applications like computer games and virtual reality can be improved over conventionally rendered scenes where all the geometry is reinserted into the graphics pipeline whenever a change is made to the view point. It has emerged, simultaneously with LOD approaches (see Chapter 2) and scene management techniques, as an effective means of tackling the dependency of rendering time on scene complexity.

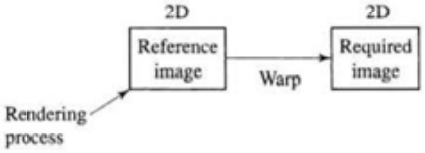
We will also deal with photo-modelling in this chapter. This is related to image-based rendering because many image-based rendering schemes were designed to operate with photo-modelling. The idea of photo-modelling is to capture the real-world complexity and at the same time retain the flexibility advantages of three-dimensional graphics.

16.1 Reuse of previously rendered imagery – two-dimensional techniques

We begin by considering methods that rely on the concept of frame coherence and reuse of already rendered imagery in some way. Also, as the title of the section implies, we are going to consider techniques that are essentially two-dimensional. Although the general topic of image-based rendering, of course, itself implies two-dimensional techniques there has be some use of the depth information associated with the image, as we shall see in future sections. The distinction is that with techniques which we categorize as two-dimensional we do not operate with detailed depth values, for example, a value per pixel. We may only have a single depth value associated with the image entity as is implied by visibility ordering in image layers (see Section 16.2.2).

A useful model of an image-based renderer is to consider a required image being generated from a source or reference image – rendered in the normal way – by warping the reference image in image space (Figure 16.1). In this section we shall consider simple techniques based on texture mapping that can exploit the hardware facilities available on current 3D graphics cards. The novel approach here is that we consider rendered objects in the scene as texture maps, consider a texture map as a three-dimensional entity and pass it through the graphics pipeline. The common application of such techniques is in systems where a viewer moves through a static environment.

To a greater or lesser extent all such techniques involve some approximation compared with the projections that are computed using conventional techniques and an important part of such methods is determining when it is valid to reuse previously generated imagery and when new images must be generated.



IBR as a process that produces an image by warping a reference image

Figure 16.1
Planar imposters and image warping.

16.1.1 Planar impostors or sprites

Impostor is the name usually given to an image of an object that is used in the form of a texture map – an entity we called a billboard in Chapter 8. In Chapter 8 the billboard was an object in its own right – it was a two-dimensional entity inserted into the scene. Impostors are generalizations of this idea. The idea is that because of the inherent coherence in consecutive frames in a moving view point sequence, the same impostor can be reused over a number of frames until an error measure exceeds some threshold. Such impostors are sometimes qualified by the adjective dynamic to distinguish them from pre-calculated object images that are not updated. A planar sprite is used as a texture map in a normal rendering engine. We use the adjective planar to indicate that no depth information is associated with the sprite – just as there is no depth associated with a texture map (although we retain depth information at the vertices of the rectangle that contains the sprite). The normal (perspective) texture mapping in the renderer takes care of warping the sprite as the view point changes.

There are many different possible ways in which sprites can be incorporated into a rendering sequence. Schaufler's method (Schaufler and Sturzlinger 1996) is typical and for generating an impostor from an object model it proceeds as follows. The object is enclosed in a bounding box which is projected onto the image plane resulting in the determination of the object's rectangular extent in screen space – for that particular view. The plane of the impostor is chosen to be that which is normal to the view plane normal and passes through the centre of the bounding box. The rectangular extent in screen space is initialized to transparent and the object rendered into it. This is then treated as a texture map and placed in the texture memory. When the scene is rendered the object is treated as a transparent polygon and texture mapped. Note that texture mapping takes into account the current view transformation and thus the impostor is warped slightly from frame to frame. Those pixels covered by the transparent pixels are unaffected in value or z depth. For the opaque pixels the impostor is treated as a normal polygon and the Z-buffer updated with its depth.

In Maciel and Shirley (1995) 'view-dependent impostors' are pre-calculated – one for each face of the object's bounding box. Space around the object is then divided into view point regions by frustums formed by the bounding box faces and its centre. If an impostor is elected as an appropriate representation then whatever region the current view point is in determines the impostor used.

16.1.2 Calculating the validity of planar impostors

As we have implied, the use of impostors requires an error metric to be calculated to quantify the validity of the impostor. Impostors become invalid because we do not use depth information. At some view point away from the view point from which the impostor was generated the impostor is perceived for what it is – a flat image embedded in three-dimensional space – the illusion is destroyed.

The magnitude of the error depends on the depth variation in the region of the scene represented by the impostor, the distance of the region from the view point and the movement of the view point away from the reference position from which the impostor was rendered. (The distance factor can be gainfully exploited by using lower resolution impostors for distant objects and grouping more than one object into clusters.) For changing view point applications the validity has to be dynamically evaluated and new impostors generated as required.

Shade *et al.* (1996) use a simple metric based on angular discrepancy. Figure 16.2 shows a two-dimensional view of an object bounding box with the plane of the impostor shown in bold. $\boldsymbol{v}_0$ is the view point for the impostor rendering and $\boldsymbol{v}_1$ is the current view point. $\boldsymbol{x}$ is a point or object vertex which coincides with $\boldsymbol{x}'$ in the impostor view. Whenever the view point changes from $\boldsymbol{v}_0$, $\boldsymbol{x}$ and $\boldsymbol{x}'$ subtend an angle θ and Shade *et al.* calculate an error metric which is the maximum angle over all points $\boldsymbol{x}$.

Schaufler and Sturzlinger's (1996) error metric is based on angular discrepancy related to pixel size and the consideration of two worst cases. First, consider the angular discrepancy due to translation of the view point parallel to the impostor plane (Figure 16.3(a)). This is at a maximum when the view point moves normal to a diagonal of a cube enclosing the bounding box with the impostor plane coincident with the other diagonal. When the view point moves to $\boldsymbol{v}_1$ the points $\boldsymbol{x}'$, $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ should be seen as separate points. The angular discrepancy due to this component of view point movement is then given by the angle θ_{trans} between the vectors $\boldsymbol{v}_1\boldsymbol{x}_1$ and $\boldsymbol{v}_1\boldsymbol{x}_2$. As long as this is less that the angle subtended by a pixel at the view point this error can be tolerated. For a view point moving towards the object we consider the construction in Figure 16.3(b). Here the worst case is the corner of the front face of the cube. When the view point moves in to $\boldsymbol{v}_1$ the points $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ should be seen as separate and the angular discrepancy is given as θ_{size}. An impostor can then be used as:

$$\text{use\_imposter} := (\theta_{trans} < \theta_{screen}) \text{ or } (\theta_{size} < \theta_{screen})$$

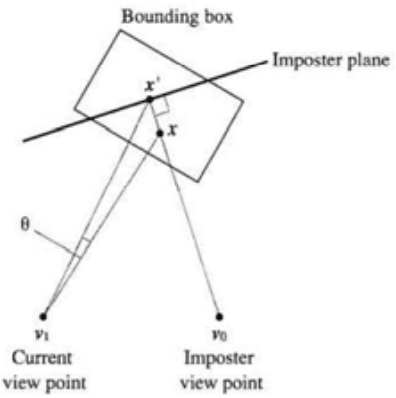


Figure 16.2
Angular discrepancy of an impostor image (after Shade *et al.* (1996)).

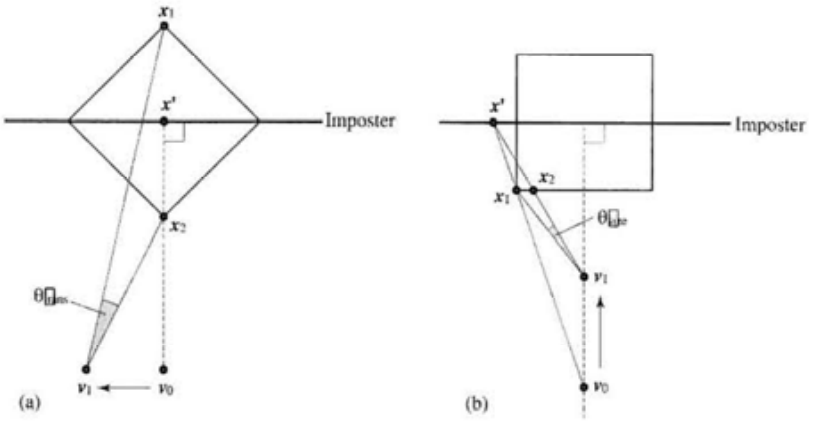


Figure 16.3
Schaufler's worst case angular discrepancy metric (after Shaufler (1996)).
(a) Translation of view point parallel to an impostor.
(b) Translation of view point towards an impostor plane.

where:

$$\theta_{screen} = \frac{\text{field of view}}{\text{screen resolution}}$$

The simplest way to use impostors is to incorporate them as texture maps in a normal rendering scheme exploiting texture mapping hardware.

So far we have said nothing about what makes up an impostor and the assumption has been that we generate an image from an object model. Shade *et al.* (1996) generalize this concept in a scheme called Hierarchical Image Caching and generate impostors from the entire contents of nodes in a BSP tree of the scene combining the benefits of this powerful scene partitioning method with the use of pre-rendered imagery. Thus, for example, distant objects that require infrequent updates can be grouped into clusters and a single impostor generated for the cluster. The algorithm thus operates on and exploits the hierarchy of the scene representation. Objects may be split over different leaf nodes and this leads to the situation of a single objects possessing more than one impostor. This causes visual artefacts and Shade *et al.* (1996) minimize this by ensuring that the BSP partitioning strategy splits as few objects as possible and by 'inflating' the geometry slightly in leaf regions so that the impostors overlap to eliminate gaps in the final image that may otherwise appear.

16.2 Varying rendering resources

16.2.1 Priority rendering

An important technique that has been used in conjunction with 2D imagery is the allocation of different amounts of rendering resources to different parts of the image. An influential (hardware) approach is due to Regan and Pose (1994).

They allocated different frame rates to objects in the scene as a function of their distance from the view point. This was called priority rendering because it combined the environment map approach with updating the scene at different rates. They use a six-view cubic environment map as the basic pre-computed solution. In addition, a multiple display memory is used for image composition and on the fly alterations to the scene are combined with pre-rendered imagery.

The method is a hybrid of a conventional graphics pipeline approach with an image-based approach. It depends on dividing the scene into a priority hierarchy. Objects are allocated a priority depending on their closeness to the current position of the viewer and their allocation of rendering resources and update time are determined accordingly. The scene is pre-rendered as environment maps and, if the viewer remains stationary, no changes are made to the environment map. As the viewer changes position the new environment map from the new view point is rendered according to the priority scheme.

Regan and Pose (1994) utilize multiple display memories to implement priority rendering where each display memory is updated at a different rate according to the information it contains. If a memory contains part of the scene that is being approached by a user then it has to be updated, whereas a memory that contains information far away from the current user position can remain as it is. Thus overall different parts of the scene are updated at different rates – hence priority rendering. Regan and Pose (1994) use memories operating at 60, 30, 15, 7.5 and 3.75 frames per second. Rendering power is directed to those parts of the scene that need it most. At any instant the objects in a scene would be organized into display memories according to their current distance from the user. Simplistically the occupancy of the memories might be arranged as concentric circles emanating from the current position of the user. Dynamically assigning each object to an appropriate display memory involves a calculation which is carried out with respect to a bounding sphere. In the end this factor must impose an upper bound on scene complexity and Regan and Pose (1994) report a test experiment with a test scene of only 1000 objects. Alternatively objects have to be grouped into a hierarchy and dealt with through a secondary data structure as is done in some speed-up approaches to conventional ray tracing.

16.2.2 Image layering

Lengyel and Snyder (1997) generalized the concept of impostors and variable application of rendering resources calling their technique 'coherent image layers'. Here the idea is again to devote rendering resources to different parts of the image according to need expressed as different spatial and/or temporal sampling rates. The technique also deals with objects moving with respect to each other. This is done by dividing the image into layers. (This is, of course, an old idea; since the 1930s cartoon production has been optimized by dividing the image into layers which are worked on independently and composed into a final film.) Thus fast-moving foreground objects can be allocated more resources than slow-moving background objects.

Another key idea of Lengyel and Snyder's work is that any layer can itself be decomposed into a number of components. The layer approach is taken into the shading itself and different resources given to different components in the shading. A moving object may consist of a diffuse layer plus a highlight layer plus a shadow layer. Each component produces an image stream and a stream of two-dimensional transformations representing its translation and warping in image space. Sprites may be represented at different resolutions to the screen resolution and may be updated at different rates. Thus sprites have different resolution in *both* space and time.

A sprite in the context of this work is now an 'independent' entity rather than being a texture map tied to an object by the normal vertex/texture coordinate association. It is also a pure two-dimensional object – not a two-dimensional part (a texture map) of a three-dimensional object. Thus as a sprite moves the appropriate warping has to be calculated.

In effect the traditional rendering pipeline is split into 'parallel' segments each representing a different part of the image (Figure 16.4). Different quality settings can be applied to each layer which manifests in different frame rates and different resolutions for each layer. The layers are then combined in the compositor with transparency or alpha in depth order.

A sprite is created as a rectangular entity by establishing a sprite rendering transform $\boldsymbol{A}$ such that the projection of the object in the sprite domain fits tightly in a bounding box. This is so that points within the sprite do not sample non-object space. The transform $\boldsymbol{A}$ is an affine transform that maps the sprite onto the screen and is determined as follows. If we consider a point in screen space p_s then we have:

$$p_s = \boldsymbol{T}p$$

where p is the equivalent object point in world space and $\boldsymbol{T}$ is the concatenation of the modelling, viewing and projection transformations.

We require an $\boldsymbol{A}$ such that (Figure 16.5):

$$p_s = \boldsymbol{A}^{-1}\boldsymbol{A}\boldsymbol{T}p = \boldsymbol{A}q$$

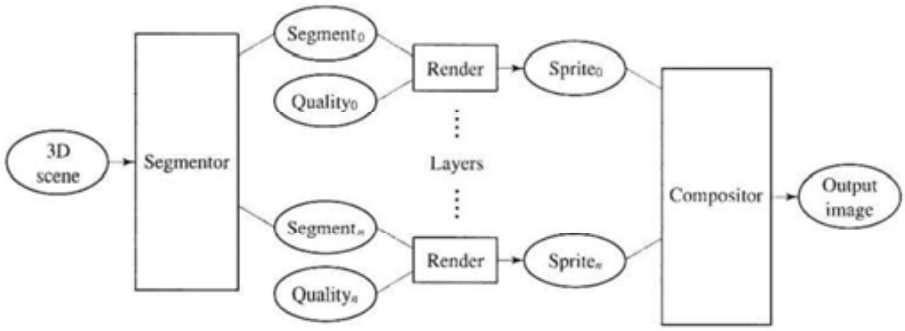


Figure 16.4
The layer approach of Lengyel and Snyder (after Lengyel and Snyder (1997)). Rendering resources are allocated to perceptually important parts of the scene (layers). Slowly changing layers are updated at a lower frame rate and at lower resolution.

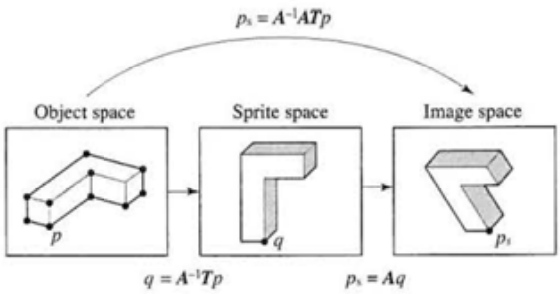


Figure 16.5
The sprite rendering transform $\boldsymbol{A}$.

where q is a point in sprite coordinates and:

$$\boldsymbol{A} = \begin{bmatrix} a & b & t_x \\ c & d & t_y \end{bmatrix}$$

Thus, an affine transformation is used to achieve an equivalent warp that would occur due to a conventional transformation $\boldsymbol{T}$.

The transform $\boldsymbol{A}$ – a 2 × 3 matrix – is updated as an object undergoes rigid motion and provides the warp necessary to change the shape of the sprite in screen space due to the object motion. This is achieved by transforming the points of a characteristic polyhedron (Figure 16.6) representing the object into screen space for two consecutive time intervals using $\boldsymbol{T}_{n-1}$ and $\boldsymbol{T}_n$ and finding the six unknown coefficients for $\boldsymbol{A}$. Full details of this procedure are given in Lengyel and Snyder (1997).

Calculating the validity of layers

As any sequence proceeds, the reusability of the layers needs to be monitored. In Section 16.1.2 we described a simple geometric measure for the validity of sprites. With image layers Lengyel and Snyder (1997) develop more elaborate criteria based upon geometric, photometric and sampling considerations. The geometric and photometric tests measure the difference between the image due to the layer or sprite and what the image should be if it were conventionally rendered.

A geometric error metric (Lengyel and Snyder call the metrics fiducials) is calculated from:

$$F_{\text{Geometric}} = \max_i \|P_i - \boldsymbol{A}p'_i\|$$

where $\boldsymbol{A}p'_i$ is a set of characteristic points in the layer in the current frame warped into their position from the previous frame and p_i the position the points actually occupy. (These are always transformed by $\boldsymbol{T}$, the modelling, viewing and perspective transform in order to calculate the warp. This sounds like a circular argument but finding $\boldsymbol{A}$ (previous section) involves a best fit procedure. Remember that the warp is being used to approximate the transformation $\boldsymbol{T}$.) Thus a threshold can be set and the layer considered for re-rendering if this is exceeded.

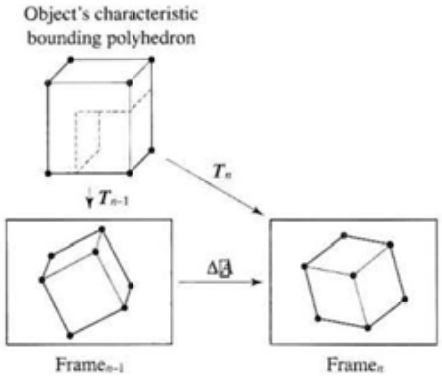


Figure 16.6
The effect of the rigid motion of the points in the bounding polyhedron in screen space is expressed as a change in the affine transform $\boldsymbol{A}$ (after Lengyel and Snyder (1997)).

For changes due to relative motion between the light source and the object represented by the layer, the angular change in $\boldsymbol{L}$, the light direction vector from the object, can be computed.

Finally, a metric associated with the magnification/minification of the layer has to be computed. If the relative movement between a viewer and object is such that layer samples are stretched or compressed then the layer may need to be re-rendered. This operation is similar to determining the depth parameter in mip-mapping and in this case can be computed from the 2×2 sub-matrix of the affine transform.

After a frame is complete a regulator considers resource allocation for the next frame. This can be done either on a 'budget-filling' basis where the scene quality is maximized or on a threshold basis where the error thresholds are set to the highest level the user can tolerate (freeing rendering resources for other tasks). The allocation is made by evaluating the error criteria and estimating the rendering cost per layer based on the fraction of the rendering budget consumed by a particular layer. Layers can then be sorted in a benefit/cost order and re-rendered or warped by the regulator.

Ordering layers in depth

So far nothing has been said about the depth of layers – the compositor requires depth information to be able to generate a final image from the separate layers. Because the method is designed to handle moving objects the depth order of layers can change and the approach is to maintain a sorted list of layers which is dynamically updated. The renderer produces hidden surface eliminated images within a layer and a special algorithm deals with the relative visibility of the layers as indivisible entities. A Kd tree is used in conjunction with convex polyhedra that bound the geometry of the layer and an incremental algorithm (fully described in Snyder (1998)) is employed to deal with occlusion without splitting.

16.3 Using depth information

16.3.1 Three-dimensional warping

As we have already mentioned, the main disadvantage of planar sprites is that they cannot produce motion parallax and they produce a warp that is constrained by a threshold beyond which their planar nature is perceived.

We now come to consider the use of depth information which is, of course, readily available in synthetic imagery. Although the techniques are now going to use the third dimension we still regard them as image-based techniques in the sense that we are still going to use, as source or reference, rendered images albeit augmented with depth information.

Consider, first, what depth information we might employ. The three commonest forms in order of their storage requirements are: using layers or sprites with depth information (previous section), using a complete (unsegmented) image with the associated Z-buffer (in other words one depth value per pixel) and a layered depth image or LDI. An LDI is a single view of a scene with multiple pixels along each line of sight. The amount of storage that LDIs require is a function of the depth complexity of the average number of surfaces that project onto a pixel.

We begin by considering images complete with depth information per pixel – the normal state of affairs for conventionally synthesized imagery. It is intuitively obvious that we should be able to generate or extrapolate an image at a new view point from the reference image providing that the new view point is close to the reference view point. We can define the pixel motion in image space as the warp:

$$I(x, y) \rightarrow I'(x', y')$$

which implies a reference pixel will move to a new destination. (This is a simple statement of the problem which ignores important practical problems that we shall address later.) If we assume that the change in the view point is specified by a rotation $\boldsymbol{R} = [r_{ij}]$ followed by a translation $\boldsymbol{T} = (\Delta x, \Delta y, \Delta z)^{\mathrm{T}}$ of the view coordinate system (in world coordinate space) and that the internal parameters of the viewing system/camera do not change – the focal length is set to unity – then the warp is specified by:

$$x' = \frac{(r_{11}x + r_{12}y + r_{13})Z(x, y) + \Delta x}{(r_{31}x + r_{32}y + r_{33})Z(x, y) + \Delta z}$$

$$y' = \frac{(r_{21}x + r_{22}y + r_{23})Z(x, y) + \Delta y}{(r_{31}x + r_{32}y + r_{33})Z(x, y) + \Delta z} \qquad [16.1]$$

where:

$Z(x, y)$ is the depth of the point $\boldsymbol{P}$ of which (x, y) is the projection.

This follows from:

$$x' = \frac{x_{v'}}{z_{v'}} \qquad y' = \frac{y_{v'}}{z_{v'}}$$

where $(x_{v'}, y_{v'}, z_{v'})$ are the coordinates of the point $\boldsymbol{P}$ in the new viewing system. A visualization of this process is shown in Figure 16.7.

We now consider the problems that occur with this process. The first is called image folding or topological folding and occurs when more than one pixel in the reference image maps into position (x', y') in the extrapolated image (Figure 16.8(a)). The straightforward way to resolve this problem is to calculate $Z(x', y')$ from $Z(x, y)$ but this requires an additional rational expression and an extra Z-buffer to store the results.

McMillan (1995) has developed an algorithm that specifies a unique evaluation order for computing the warp function such that surfaces are drawn in a back-to-front order thus enabling a simple painter's algorithm to resolve this visibility problem. The intuitive justification for this algorithm can be seen by considering a simple special case shown in Figure 16.9. In this case the view point has moved to the left so that its projection in the image plane of the reference view coordinate system is outside and to the left of the reference view window. This fact tells us that the order in which we need to access pixels in the reference is from right to left. This then resolves the problem of the leftmost pixel in the reference image overwriting the right pixel in the warped image. McMillan shows that the accessing or enumeration order of the reference image can be reduced to nine cases depending on the position of the projection of the new view point in the reference coordinate system. These are shown in Figure 16.10. The general case, where the new view point stays within the reference view window divides the image into quadrants. An algorithm structure that utilizes this method to resolve depth problems in the many-to-one case is thus:

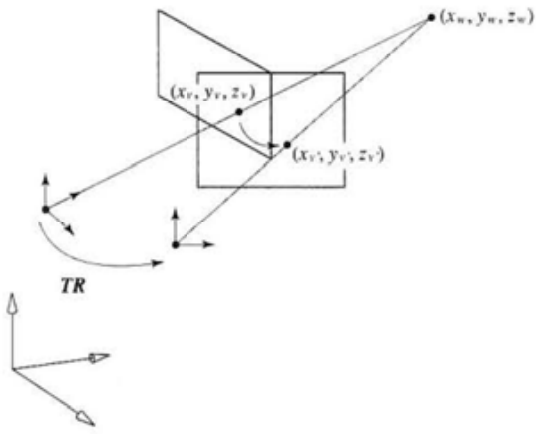


Figure 16.7
A three-dimensional warp is calculated from rotation $\boldsymbol{R}$ and translation $\boldsymbol{T}$ applied to the view coordinate system.

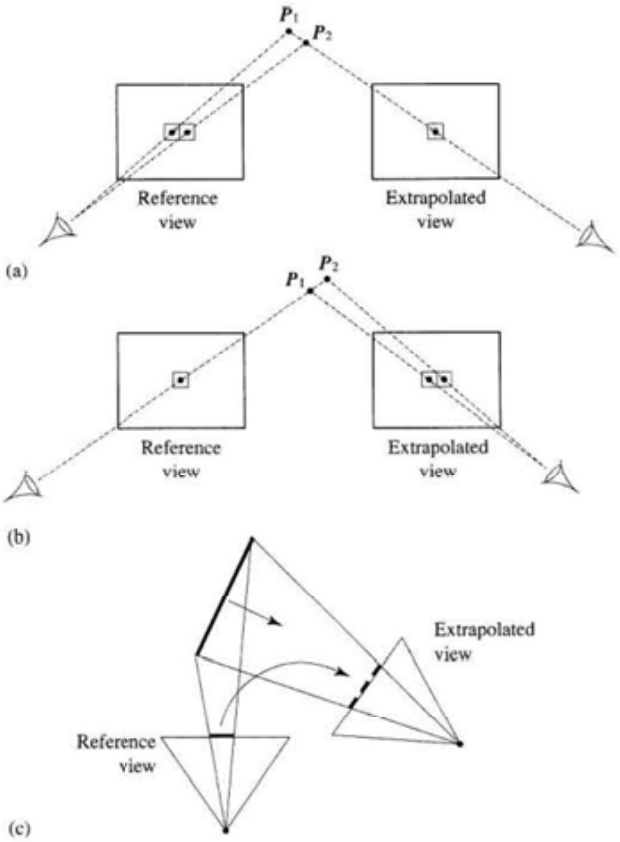


Figure 16.8
Problems in image warping. (a) Image folding: more than one pixel in the reference view maps into a single pixel in the extrapolated view. (b) Holes: information occluded in the reference view is required in the extrapolated view. (c) Holes: the projected area of a surface increases in the extrapolated view because its normal rotates towards the viewing direction. (d) See Colour Plate section.

(1) Calculate the projection of the new view point in the reference coordinate system.

(2) Determine the enumeration order (one out of the nine cases shown in Figure 16.10) depending on the projected point.

(3) Warp the reference image by applying Equation 16.1 and writing the result into the frame buffer.

The second problem produced by image warping is caused when occluded areas in the reference image 'need' to become visible in the extrapolated image (Figure 16.8(b)) producing holes in the extrapolated image. As the figure demonstrates, holes and folds are in a sense the inverse of each other, but where a deterministic solution exists for folds no theoretical solution exists for holes and a heuristic needs to be adopted – we cannot recover information that was not there in the first place. However, it is easy to detect where holes occur. They are simply unassigned pixels in the extrapolated image and this enables the problem to be localized and the most common solution is to fill them in with colours from

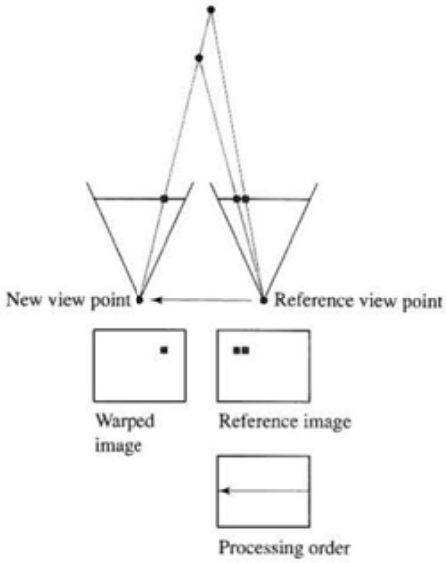


Figure 16.9
The view point translates to the left so that the projection of the new view point in the image plane of the reference view coordinate system is to the left of the reference view window. The correct processing order of the reference pixels is from right to left.

neighbouring pixels. The extent of the holes problem depends on the difference between the reference and extrapolated view points and it can be ameliorated by considering more than one reference image, calculating an extrapolated image from each and compositing the result. Clearly if a sufficient number of reference images are used then the hole problem will be eliminated and there is no need for a local solution which may insert erroneous information.

A more subtle reason for unassigned pixels in the extrapolated image is apparent if we consider surfaces whose normal rotates towards the view direction in

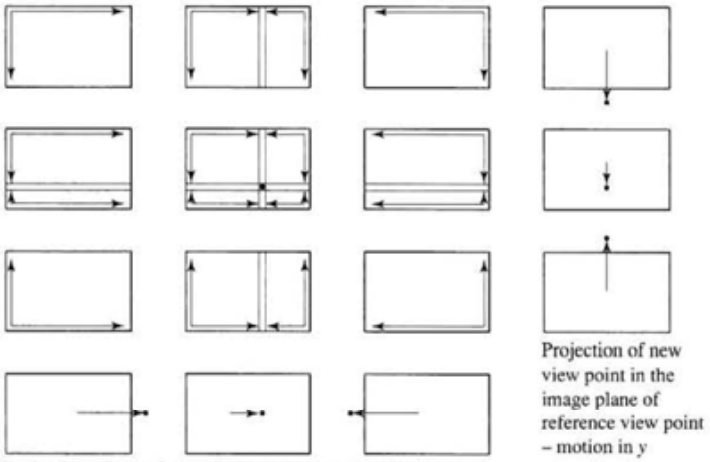


Figure 16.10
A visualization of McMillan's priority algorithm indicating the correct processing order as a function of view point motion for nine cases (after McMillan (1995)).

the new view system (Figure 16.8(c)). The projected area of such a surface into the extrapolated image plane will be greater than its projection in the reference image plane and for a one-to-one forward mapping holes will be produced. This suggests that we must take a rigorous approach to reconstruction in the interpolated image. Mark *et al.* (1997) suggest calculating the appropriate dimension of a reconstruction kernel, for each reference pixel as a function of the view point motion but they point out that this leads to a cost per pixel that is greater than the warp cost. This metric is commonly known as splat size (Chapter 13) and its calculation is not straightforward for a single reference image with Z depth only for visible pixels. (A method that stores multiple depth values for a pixel is dealt with in the next section.)

The effects of these problems on an image are shown in Figure 16.8(d) (Colour Plate). The first two images show a simple scene and the corresponding Z-buffer image. The next image shows the artefacts due to translation (only). In this case these are holes caused by missing information and image folding. The next image shows artefacts due to rotation (only) – holes caused by increasing the projected area of surfaces. Note how these form coherent patterns. The final image shows artefacts caused by both rotation and translation.

Finally, we note that view-dependent illumination effects will not in general be handled correctly with this simple approach. This, however, is a problem that is more serious in image-based modelling methods (Section 16.6). As we have already noted in image warping we must have reference images whose view point is close to the required view point.

16.3.2 Layered depth images (LDIs)

Many of the problems encountered in the previous section disappear if our source imagery is in the form of an LDI (Shade *et al.* 1998). In particular we can resolve the problem of holes where we require information in the extrapolated image in areas occluded in the source or reference image. An LDI is a three-dimensional data structure that relates to a particular view point and which samples, for each pixel, all the surfaces and their depth values intersected by the ray through that pixel (Figure 16.11). (In practice, we require a number of LDIs to represent a scene and so can consider a scene representation to be four-dimensional – or the same dimensionality as the light field in Section 16.5.) Thus, each pixel is associated with an array of information with a number of elements or layers that is determined by the number of surfaces intersected. Each element contains a colour, surface normal and depth for surface. Clearly this representation requires much more storage than an image plus Z-buffer but this requirement grows only linearly with depth complexity.

In their work Shade *et al.* (1998) suggest two methods for pre-calculating LDIs for synthetic imagery. First, they suggest warping n images rendered from different view points into a single view point. During the warping process if more than one pixel maps into a single LDI pixel then the depth values associated

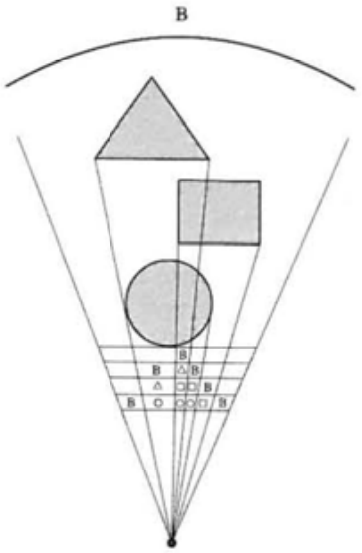


Figure 16.11
A representation of a layered depth image (LDI).

with each source view are compared and enable the layers to be sorted in depth order.

An alternative approach which facilitates a more rigorous sampling of the scene is to use a modified ray tracer. This can be done simplistically by initiating a ray for each pixel from the LDI view point and allowing the rays to penetrate the object (rather than being reflected or refracted). Each hit is then recorded as a new depth pixel in the LDI. All of the scene can be considered by pre-calculating six LDIs each of which consists of a 90° frustum centred on the reference view point. Shade *et al.* (1998) point out that this sampling scheme is not uniform with respect to a hemisphere of directions centred on the view point. Neighbouring pixel rays project a smaller area onto the image plane as a function of the angle between the image plane normal and the ray direction and they weight the ray direction by the cosine of that angle. Thus, each ray has four coordinates: two pixel coordinates and two angles for the ray direction. The algorithm structure to calculate the LDIs is then:

(1) For each pixel, modify the direction and cast the ray into the scene.

(2) For each hit: if the intersected objects lies within the LDI frustum it is re-projected through the LDI view point.

(3) If the new hit is within a tolerance of an existing depth pixel the colour of the new sample is averaged with the existing one; otherwise a new depth pixel is created.

During the rendering phase, an incremental warp is applied to each layer in back to front order and images are alpha blended into the frame buffer without the need for Z sorting. McMillan's algorithm (see Section 16.3.1) is used to ensure

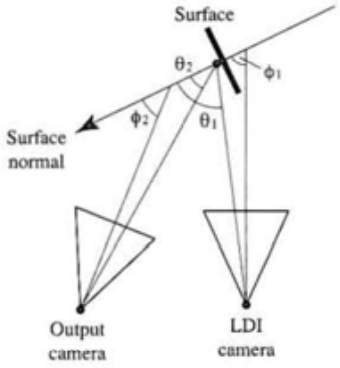


Figure 16.12
Parameters used in splat size computation (after Shade *et al.* (1998)).

that the pixels are selected for warping in the correct order according to the projection of the output camera in the LDI's system.

To enable splat size computation Shade *et al.* (1998) use the following formula (Figure 16.12):

$$\text{size} = \frac{d_1^2 \cos\theta_2 \, \text{res}_2 \tan\frac{\text{fov}_1}{2}}{d_2^2 \cos\theta_1 \, \text{res}_1 \tan\frac{\text{fov}_2}{2}}$$

where:

size is the dimension of a square kernel (in practice this is rounded to 1, 3, 5 or 7)
the angles θ are approximated as the angles ϕ, where ϕ is the angle between the surface normal and the z axis of the camera system
fov is the field of view of a camera
res = $w*h$ (the width and height of the LDI)

16.4 View interpolation

View interpolation techniques can be regarded as a subset of 3D warping methods. Instead of extrapolating an image from a reference image, they interpolate a pair of reference images. However, to do this three-dimensional calculations are necessary. In the light of our earlier two-dimensional/three-dimensional categorization they could be considered a two-dimensional technique but we have decided to emphasize the interpolation aspect and categorize them separately.

Williams and Chen (1993) were the first to implement view interpolation for a walkthrough application. This was achieved by pre-computing a set of reference images representing an interior – in this case a virtual museum. Frames required in a walkthrough were interpolated at run time from these reference frames. The interpolation was achieved by storing a 'warp script' that specifies

the pixel motion between reference frames. This is a dense set of motion vectors that relates a pixel in the source image to a pixel in the destination image. The simplest example of a motion field is that due to a camera translating parallel to its image plane. In that case the motion field is a set of parallel vectors – one for each pixel – with a direction opposite to the camera motion and having a magnitude proportional to the depth of the pixel. This pixel-by-pixel correspondence can be determined for each pair of images since the three-dimensional (image space) coordinates of each pixel is known, as is the camera or view point motion. The determination of warp scripts is a pre-processing step and an interior is finally represented by a set of reference images together with a warp script relating every adjacent pair. For a large scene that requires a number of varied walkthroughs the total storage requirement may be very large; however, any derived or interpolated view only requires the appropriate pair of reference images and the warp script.

At run time a view or set of views between two reference images is then reduced to linear interpolation. Each pixel in both the source and destination images is moved along its motion vector by the amount given by linearly interpolating the image coordinates (Figure 16.13). This gives a pair of interpolated images. These can be composited and using a pair of images in this way reduces the hole problem. Chen and Williams (1993) fill in remaining holes with a procedure that uses the colour local to the hole. Overlaps are resolved by using a Z-buffer to determine the nearest surface, the z values being linearly interpolated along with the (x, y) coordinates. Finally, note that linear interpolation of the motion vectors produces a warp which will not be exactly the same as that produced if the camera was moved into the desired position. The method is only exact from the special case of a camera translating parallel to its image plane. Williams and Chen (1993) point out that a better approximation can be obtained by quadratic or cubic interpolation in the image plane.

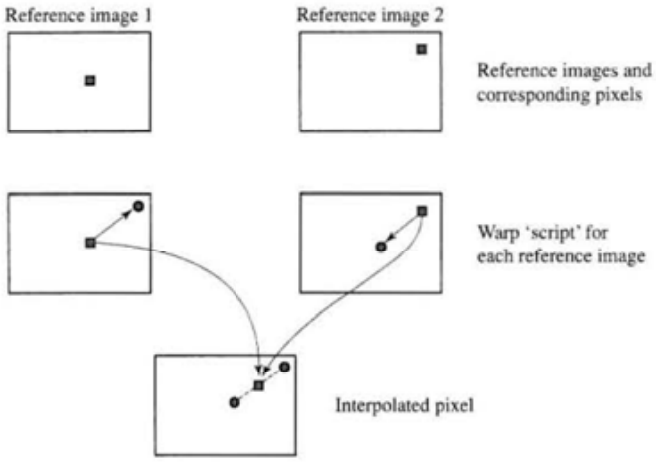


Figure 16.13
Simple view interpretation: a single pair of corresponding pixels define a path in image space from which an interpolated view can be constructed.

16.4.1 View morphing

Up to now we have considered techniques that deal with a moving view point and static scenes. In a development that they call view morphing Seitz and Dyer (1996) address the problem of generating in-between images where non-rigid transformations have occurred. They do this by addressing the approximation implicit in the previous section and distinguish between 'valid' and 'non-valid' in-between views.

View interpolation by warping a reference image into an extrapolated image proceeds in two-dimensional image plane space. A warping operation is just that – it changes the shape of the two-dimensional projection of objects. Clearly the interpolation should proceed so that the projected shape of the objects in the reference projection is consistent with their real three-dimensional shape. In other words, the interpolated view must be equivalent to a view that would be generated in the normal way (either using a camera or a conventional graphics pipeline) by changing the view point from the reference view point to that of the interpolated view. A 'non-valid' view means that the interpolated view does not preserve the object shape. If this condition does not hold then the interpolated views will correspond to an object whose shape is distorting in real three-dimensional space. This is exactly what happens in conventional image morphing between two shapes. 'Impossible', non-existent or arbitrary shapes occur as in-between images because the motivation here is to appear to change one object into an entirely different one. The distinction between valid and invalid view interpolation is shown in Figure 16.14.

An example where linear interpolation of images produces valid interpolated views is the case where the image planes remain parallel (Figure 16.15). Physically, this situation would occur if a camera was allowed to move parallel to its image plane (and optionally zoom in and out). If we let the combined viewing and perspective transformations (see Chapter 5) be $\boldsymbol{V}_0$ and $\boldsymbol{V}_1$ for the two reference images then the transformation for an in-between image can be obtained by linear interpolation:

$$\boldsymbol{V}_i = (1 - s)\ \boldsymbol{V}_0 + s\boldsymbol{V}_1$$

If we consider a pair of corresponding points in the reference images $\boldsymbol{P}_0$ and $\boldsymbol{P}_1$ which are projections of world space point $\boldsymbol{P}$, then it is easily shown (see Seitz and Dyer (1996)) that the projection of point $\boldsymbol{P}$ from the intermediate (interpolated) view point is given by linear interpolation:

$$\begin{aligned}\boldsymbol{P}_i &= \boldsymbol{P}_0(1 - s) + \boldsymbol{P}_1 s \\ &= \boldsymbol{V}_i \boldsymbol{P}\end{aligned}$$

In other words linear interpolation of pixels along a path determined by pixel correspondence in two reference images is exactly equivalent to projecting the scene point that resulted in these pixels through a viewing and projective transformation given by an intermediate camera position, provided parallel views are maintained, in other words using the transformation $\boldsymbol{V}_i$, which would

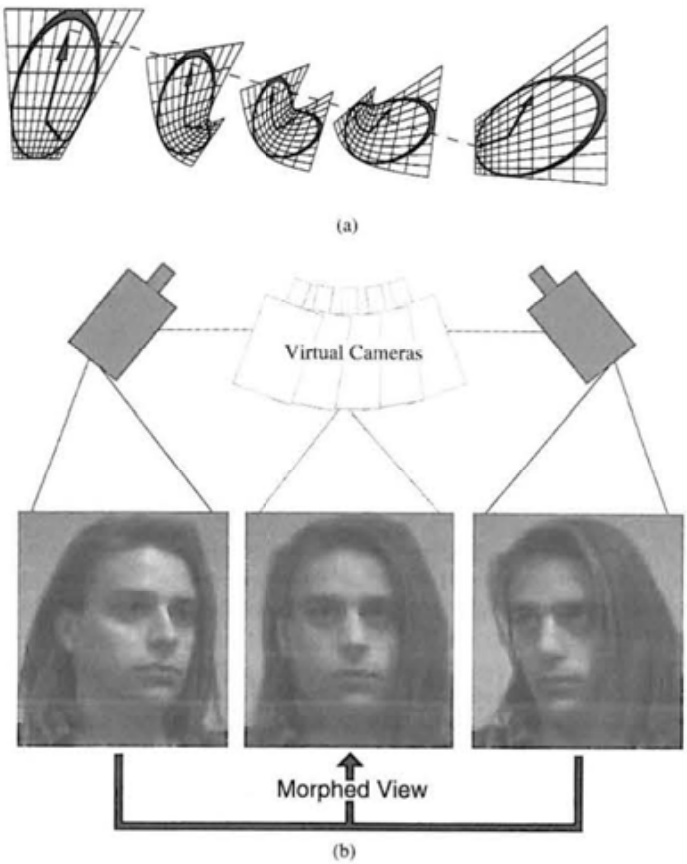


Figure 16.14
Distinguishing between valid and invalid view interpolation. In (a), using a standard (morphing) approach of linear interpolation produces gross shape deformation (this does not matter if we are morphing between two different objects – it becomes part of the effect). (b) The interpolated (or morphed view) is consistent with object shape. (Courtesy of Steven Seitz.)

be obtained if $\boldsymbol{V}_0$ and $\boldsymbol{V}_1$ were linearly interpolated. Note also that we are interpolating views that would correspond to those obtained if we had moved the camera in a straight line from $\boldsymbol{C}_0$ to $\boldsymbol{C}_1$. In other words the interpolated view corresponds to the camera position:

$$\boldsymbol{C}_i = (sC_x, sC_y, 0)$$

If we have reference views that are not related in this way then the interpolation has to be preceded (and followed) by an extra transformation. This is the general situation where the image planes of the reference views and the image plane of the required or interpolated view have no parallel relationship. The first transformation, which Seitz and Dyer call a 'prewarp', warps the reference images so that they appear to have been taken by a camera moving in a plane parallel to its image plane. The pixel interpolation, or morphing, can the be performed as in the previous paragraph and the result of this is postwarped to form the final interpolated view, which is the view required from the virtual camera position.

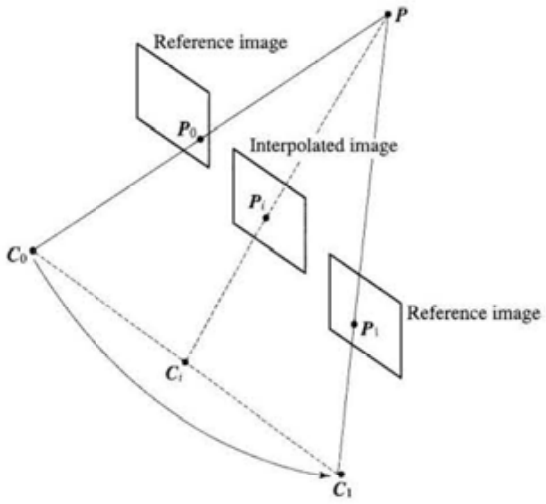
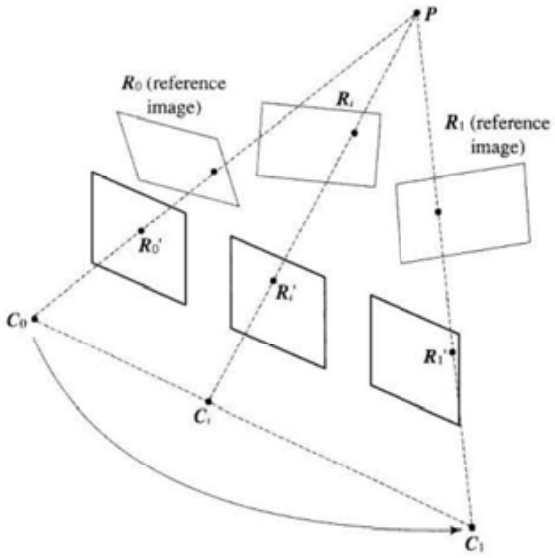
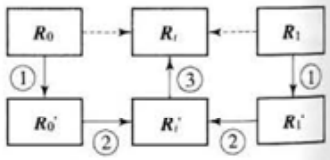


Figure 16.15
Moving the camera from $\mathbf{C}_0$ to $\mathbf{C}_1$ (and zooming) means that the image planes remain parallel and $\boldsymbol{P}_i$ can be linearly interpolated from $\boldsymbol{P}_0$ and $\boldsymbol{P}_1$ (after Seitz and Dyer (1996)).

A simple geometric illustration of the process is shown in Figure 16.16. Here $\boldsymbol{R}_0$ and $\boldsymbol{R}_1$ are the references images. Prewarping these to $\boldsymbol{R}_0'$ and $\boldsymbol{R}_1'$ respectively means that we can now linearly interpolate these rectified images to produce $\boldsymbol{R}_i'$. This is then postwarped to produce the required $\boldsymbol{R}_i$. An important consequence of this method is that although the warp operation is image based we require knowledge of the view points involved to effect the pre- and post-warp transformations. Again this has ramifications for the context in which the method is





Three-step transformation sequence

Figure 16.16
Prewarping reference images, interpolating and postwarping in view interpolation (after Seitz and Dyer (1996)).

used, implying that in the case of photographic imagery we have to record or recover the camera view points.

The prewarping and postwarping transformations are derived as follows. First, it can be shown that any two perspective views that share the same centre of projection are related by a planar projective transformation – a 3 × 3 matrix obtained from the combined viewing perspective transformation $\boldsymbol{V}$. Thus $\boldsymbol{R}_0$ and $\boldsymbol{R}_1$ are related to $\boldsymbol{R}_0'$ and $\boldsymbol{R}_1'$ by two such matrices $\boldsymbol{T}_0$ and $\boldsymbol{T}_1$. The procedure is thus as follows:

(1) Prewarp $\boldsymbol{R}_0$ and $\boldsymbol{R}_1$ using $\boldsymbol{T}_0^{-1}$ and $\boldsymbol{T}_1^{-1}$ to produce $\boldsymbol{R}_0'$ and $\boldsymbol{R}_1'$.

(2) Interpolate to calculate $\boldsymbol{R}_i'$, $\boldsymbol{C}_i$ and $\boldsymbol{T}_i$.

(3) Apply $\boldsymbol{T}_i$ to $\boldsymbol{R}_i'$ to give image $\boldsymbol{R}_i$.

16.5 Four-dimensional techniques – the Lumigraph or light field rendering approach

Up to now we have considered systems that have used a single image or a small number of reference images from which a required image is generated. We have looked at two-dimensional techniques and methods where depth information has been used – three-dimensional warping. Some of these methods involve pre-calculation of a special form of rendered imagery (LDIs) others post-process a conventionally rendered image. We now come to a method that is an almost total pre-calculation technique. It is an approach that bears some relationship to environment mapping. An environment map caches all the light rays that arrive at a single point in the scene – the source or reference point for the environment map. By placing an object at that point we can (approximately) determine those light rays that arrive at the surface of the object by indexing into the map. This scheme can be extended so that we store in effect an environment map for every sampled point in the scene. That is, for each point in the scene we have knowledge of all light rays arriving at that point. We can now place an object at any point in the scene and calculate the reflected light. The advantage of this approach is that we now minimize most of the problems related to three-dimensional warping at the cost of storing a vast amount of data.

A light field is a similar approach. For each and every point of a region in the scene in which we wish to reconstruct a view we pre-calculate and store or cache the radiance in every direction at that point. This representation is called a light field or Lumigraph (Levoy and Hanrahan 1996; Gortler *et al.* 1996) and we construct for a region of free space by which is meant a region free of occluders. The importance of free space is that it reduces the light field from a five-dimensional to a four-dimensional function. In general, for every point (x, y, z) in scene space we have light rays travelling in every direction (parametrized by two angles) giving a five-dimensional function. In occluder free space we can assume (unless there is atmospheric interaction) that the radiance along a ray is constant. The two 'free space scenes' of interest to us are: viewing an object from

anywhere outside its convex hull and viewing an environment such as a room from somewhere within its (empty) interior.

The set of rays in any region in space can be parametrized by their intersection with two parallel planes and this is the most convenient representation for a light field (Figure 16.17(a)). The planes can be positioned anywhere. For example, we can position a pair of planes parallel to each face of a cube enclosing an object and capture all the radiance information due to the object (Figure 16.7(b)). Reconstruction of any view of the object then consists of each pixel in the view plane casting a ray through the plane pair and assigning $L(s, t, u, v)$ to that pixel (Figure 16.7(c)). The reconstruction is essentially a resampling process and unlike the methods described in previous sections it is a linear operation.

Light fields are easily constructed from rendered imagery. A light field for a single pair of parallel planes placed near an object can be created by moving the camera in equal increments in the (s, t) plane to generate a series of sheared perspective projections. Each camera point (s, t) then specifies a bundle of rays arriving from every direction in the frustum bounded by the (u, v) extent. It could be argued that we are simply pre-calculating every view of the object that we require at run time; however, two factors mitigate this brute-force approach. First, the resolution in the (s, t) plane can be substantially lower than the resolution in the (u, v) plane. If we consider a point on the surface of the object coincidence, say, with the (u, v) plane, then the (s, t) plane contains the reflected

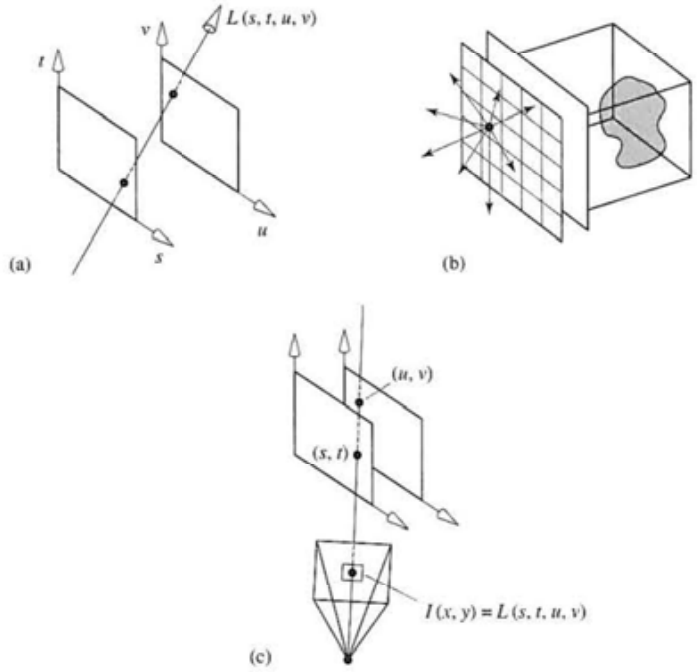


Figure 16.17
Light field rendering using parallel plane representation for rays. (a) Parametrization of a ray using parallel planes. (b) Pairs of planes positioned on the face of a bounding cube can represent all the radiance information due to an object. (c) Reconstruction for a single pixel $I(x,y)$.

light in every direction (constrained by the (s, t) plane extent). By definition, the radiance at a single point on the surface of an object varies slowly with direction and a low sampling frequency in the (s, t) plane will capture this variation. A higher sampling frequency is required to calculate the variation as a function of position on the surface of the object. Second, there is substantial coherence exhibited by a light field. Levoy and Hanrahan (1996) report a compression ratio of 118:1 for a 402 Mb light field and conclude that given this magnitude of compression the simple (linear) re-sampling scheme together with simplicity advantages over other IBR methods make light fields a viable proposition.

16.6 Photo-modelling and IBR

Another distinguishing factor in IBR approaches is whether they work only with computer graphics imagery (where depth information is available) or whether they use photographs as the source imagery. Photography has the potential to solve the other major problem with scene complexity – the modelling cost. Real world detail, whose richness and complexity eludes even the most elaborate photo-realistic renderers, is easily captured by conventional photographic means. The idea is to use IBR techniques to manipulate the photographs so that they can be used to generate an image from a view point different from the camera view point.

Photographs have always been used in texture mapping and this classical tool is still finding new applications in areas which demand an impression of realism that would be unobtainable from conventional modelling techniques, except at great expense. A good example is facial animation where a photograph of a face is wrapped onto a computer graphics model or structure. The photo-map provides the fine level of detail, necessary for convincing and realistic expressions, and the underlying three-dimensional model is used as a basis for controlling the animation.

In building geometric representations from photographs, many of the problems that are encountered are traditionally part of the computer vision area but the goals are different. Geometric information recovered from a scene in a computer vision context usually has some single goal, such as collision avoidance in robot navigation or object recognition, and we are usually concerned in some way with reducing the information that impinges on the low-level sensor. We are generally interested in recovering the shape of an object without regard to such irrelevant information as texture; although we may use such information as a device for extracting the required geometry, we are not interested in it *per se*. In modelling a scene in detail, it is precisely the details such as texture that we are interested in, as well as the pure geometry.

Consider first the device of using photography to assist in modelling. Currently available commercial photo-modelling software concentrates on extracting pure geometry using a high degree of manual intervention. Common approaches use a pre-calibrated camera, knowledge of the position of the

camera for each shot and a sufficient number of shots to capture the structure of the building, say, that is being modelled. Extracting the edges from the shots of the building enables a wireframe model to be constructed. This is usually done semi-automatically with an operator matching corresponding edges in the different projections. It is exactly equivalent to the shape from stereo problem using feature correspondence except that now we use a human being instead of a correspondence-establishing algorithm. We may end up performing a large amount of manual work on the projections, as much work as would be entailed in using a conventional CAD package to construct the building. The obvious potential advantage is that photo-modelling offers the possibility of automatically extracting the rich visual detail of the scene, as well as the geometry.

It is interesting to note that in modelling from photographs approaches, the computer graphics community has side-stepped the most difficult problems that are researched in computer vision by embracing some degree of manual intervention. For example, the classical problem of correspondence between images projected from different view points is solved by having an operator manually establish a degree of correspondence between frames which can enable the success of algorithms that establish detailed pixel-by-pixel correspondence. In computer vision such approaches do not seem to be considered. Perhaps this is due to well-established traditional attitudes in computer vision which has tended to see the imitation of human capabilities as an ultimate goal, as well as constraints from applications.

Using photo-modelling to capture detail has some problems. One is that the information we obtain may contain light source and view-dependent phenomena such as shadows and specular reflections. These would have to be removed before the imagery could be used generate the simulated environment from any view point. Another problem of significance is that we may need to warp detail in a photograph to fit the geometric model. This may involve expanding a very small area of an image. Consider, for example, a photograph – taken from the ground – of high building with a detailed facade. Important detail information near the top of the building may be mapped into a small area due to the projective distortion. In fact, this problem is identical to view interpolation.

Let us now consider the use of photo-modelling without attempting to extract the geometry. We simply keep the collected images as two-dimensional projections and use these to calculate new two-dimensional projections. We never attempt to recover three-dimensional geometry of the scene (although it is necessary to consider the three-dimensional information concerning the projections). This is a form of image-based rendering and it has something of a history.

Consider a virtual walk through an art gallery or museum. The quality requirements are obvious. The user needs to experience the subtle lighting conditions designed to best view the exhibits. These must be reproduced and sufficient detail must be visible in the paintings. A standard computer graphics approach may result in using a (view-independent) radiosity solution for the rendering together with (photographic) texture maps for the paintings. The

radiosity approach, where the expensive rendering calculations are performed once only to give a view-independent solution may suffice in many contexts in virtual reality, but it is not a general solution for scenes that contain complex geometrical detail. As we know, a radiosity rendered scene has to be divided up into as large elements as possible to facilitate a solution and there is always a high cost for detailed scene geometry.

This kind of application – virtual tours around buildings and the like – has already emerged with the bulk storage freedom offered by videodisk and CD-ROM. The inherent disadvantage of most approaches is that they do not offer continuous movement or walkthrough but discrete views selected by a user's position as he (interactively) navigates around the building. They are akin to an interactive catalogue and require the user to navigate in discrete steps from one position to the other as determined by the points from which the photographic images were taken. The user 'hops' from view point to view point.

An early example of a videodisk implementation is the 'Movie Map' developed in 1980 (Lippman 1980). In this early example the streets of Aspen were filmed at 10-foot intervals. To invoke a walkthrough, a viewer retrieved selected views from two videodisk players. To record the environment four cameras were used at every view point – thus enabling the viewer to pan to the left and right. The example demonstrates the trade-off implicit in this approach – because all reconstructed views are pre-stored the recording is limited to discrete view points.

An obvious computer graphics approach is to use environment maps – originally developed in rendering to enable a surrounding environment to be reflected in a shiny object (see Chapter 8). In image-based rendering we simply replace the shiny object with a virtual viewer. Consider a user positioned at a point from which a six-view (cubic) environment map has been constructed (either photographically or synthetically). If we make the approximation that the user's eyes are always positioned exactly at the environment map's view point then we can compose any view direction-dependent projection demanded by the user changing his direction of gaze by sampling the appropriate environment maps. This idea is shown schematically in Figure 16.18. Thus we have, for a stationary viewer, coincidentally positioned at the environment map view point, achieved our goal of a view-independent solution. We have decoupled the viewing direction from the rendering pipeline. Composing a new view now consists of sampling environment maps and the scene complexity problem has been bound by the resolution of the pre-computed or photographed maps.

The highest demand on an image generator used in immersive virtual reality comes from head movements (we need to compute at 60 frames per second to avoid the head latency effect) and if we can devise a method where the rendering cost is almost independent of head movement this would be a great step forward. However, the environment map suggestion only works for a stationary viewer. We would need a set of maps for each position that the viewer could be in. Can we extend the environment map approach to cope with complete walkthroughs? Using the constraint that in a walkthrough the eyes of the user are always at a constant height, we could construct a number of environment maps

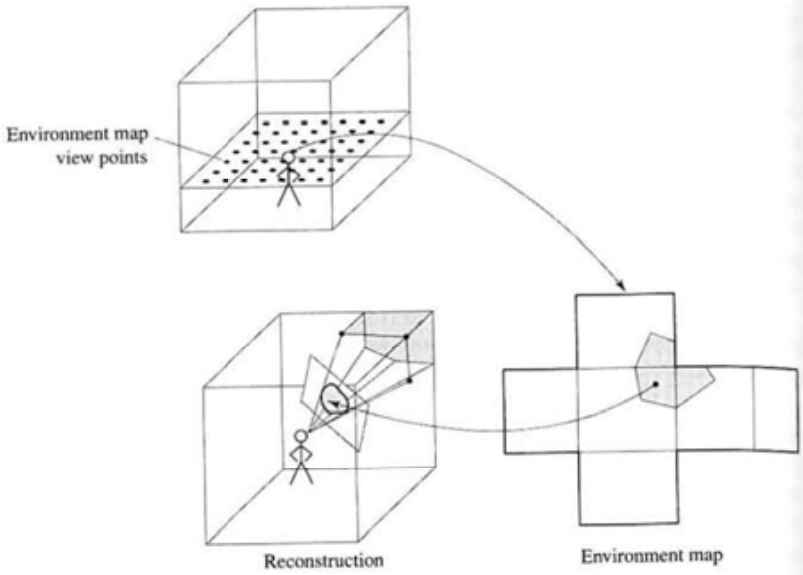


Figure 16.18
Compositing a user projection from an environment map.

whose view points were situated at the lattice points of a coarse grid in a plane, parallel to the ground plane and positioned at eye height. For any user position could we compose, to some degree of accuracy, a user projection by using information from the environment maps at the four adjacent lattice points? The quality of the final projections are going to depend on the resolution of the maps and the number of maps taken in a room – the resolution of the eye plane lattice. The map resolution will determine the detailed quality of the projection and the number of maps its geometric accuracy.

To be able to emulate the flexibility of using a traditional graphics pipeline approach, by using photographs (or pre-rendered environment maps), we either have to use a brute-force approach and collect sufficient views compatible with the required 'resolution' of our walkthrough, or we have to try to obtain new views from the existing ones.

Currently, viewing from cylindrical panoramas is being established as a popular facility on PC-based equipment (see Section 16.5.1). This involves collecting the component images by moving a camera in a semi-constrained manner – rotating it in a horizontal plane. The computer is used merely to 'stitch' the component images into a continuous panorama – no attempt is made to recover depth information.

This system can be seen as the beginning of development that may eventually result in being able to capture all the information in a scene by walking around with a video camera resulting in a three-dimensional photograph of the scene. We could see such a development as merging the separate stages of modelling and rendering, there is now no distinction between them. The virtual

viewer can then be immersed in a photographic-quality environment and have the freedom to move around in it without having his movement restricted to the excursions of the camera.

16.6.1 Image-based rendering using photographic panoramas

Developed in 1994, Apple's QuickTime® VR is a classic example of using a photographic panorama as a pre-stored virtual environment. A cylindrical panorama is chosen for this system because it does not require any special equipment beyond a standard camera and a tripod with some accessories. As for re-projection – a cylindrical map has the advantage that it only curves in one direction thus making the necessary warping to produce the desired planar projection fast. The basic disadvantage of the cylindrical map – the restricted vertical field of view – can be overcome by using an alternative cubic or spherical map but both of these involve a more difficult photographic collection process and the sphere is more difficult to warp. The inherent viewing disadvantage of the cylinder depends on the application. For example, in architectural visualization it may be a serious drawback.

Figure 16.19 (Colour Plate) is an illustration of the system. A user takes a series of normal photographs, using a camera rotating on a tripod, which are then 'stitched' together to form a cylindrical panoramic image. A viewer positions himself at the view point and looks at a portion of the cylindrical surface. The re-projection of selected part of the cylinder onto a (planar) view surface involves a simple image warping operation which, in conjunction with other speed-up strategies, operates in real time on a standard PC. A viewer can continuously pan in the horizontal direction and the vertical direction to within the vertical field of view limit.

Currently restricted to monocular imagery, it is interesting to note that one of the most lauded aspects of virtual reality – three-dimensionality and immersion – has been for the moment ignored. It may be that in the immediate future monocular non-immersive imagery, which does not require expensive stereo viewing facilities and which concentrates on reproducing a visually complex environment, will predominate in the popularization of virtual reality facilities.

16.6.2 Compositing panoramas

Compositing environment maps with synthetic imagery is straightforward. For example, to construct a cylindrical panorama we map view space coordinates (x, y, z) onto a cylindrical viewing surface (θ, h) as:

$$\theta = \tan^{-1}(x/z) \qquad h = y/(x^2 + z^2)^{1/2}$$

Constructing a cylindrical panorama from photographs involves a number of practical points. Instead of having three-dimensional coordinates we now have

photographs. The above equations can still be used substituting the focal length of the lens for z and calculating x and y from the coordinates in the photograph plane and the lens parameters. This is equivalent to considering the scene as a picture of itself – all objects in the scene are considered to be at the same depth.

Another inherent advantage of a cylindrical panorama is that after the overlapping planar photographs are mapped into cylindrical coordinates (just as if we had a cylindrical film plane in the camera) the construction of the complete panorama can be achieved by translation only – implying that it is straightforward to automate the process. The separate images are moved over one another until a match is achieved – a process sometimes called 'stitching'. As well as translating the component images, the photographs may have to be processed to correct for exposure differences that would otherwise leave a visible vertical boundary in the panorama.

The overall process can now be seen as a warping of the scene onto a cylindrical viewing surface followed by the inverse warping to re-obtain a planar projection from the panorama. From the user's point of view the cylinder enables both an easy image collection model and a natural model for viewing in the sense that we normally view an environment from a fixed height – eye level – look around and up and down.

16.6.3 Photo-modelling for image-based rendering

In one of the first comprehensive studies of photo-modelling for image-based rendering, Debevec *et al.* (1996) describe an approach with a number of interesting and potentially important features. Their basic approach is to derive sufficient information from sparse views of a scene to facilitate image-based rendering (although the derived model can also be used in a conventional rendering system). The emphasis of their work is architectural scenes and is based on three innovations:

(1) **Photogrammetric modelling** in which they recover a three-dimensional geometric model of a building based on simple volumetric primitives together with the camera view points from a set of sparse views.

(2) **View-dependent texture mapping** which is used to render the recovered model.

(3) **Model-based stereo** which is used to solve the correspondence problem (and thus enable view interpolation) and the recovery of detail not modelled in (1).

Debevec *et al.* (1996) state that their approach is successful because:

> it splits the task of modelling from images into tasks that are easily accomplished by a person (but not by a computer algorithm), and tasks which are easily performed by a computer algorithm (but not by a person).

The photogrammetric modelling process involves the user viewing a set of photographs of a building and associating a set of volumetric primitives with the photographic views to define an approximate geometric model. This is done by invoking a component of the model, such as a rectangular solid and interactively associating edges in the model with edges in the scene. In this way a box, say, can be fitted semi-automatically to a view or views that contain a box as a structural element. This manual intervention enables a complete geometric model to be derived from the photographs even though only parts of the model may be visible in the scene. The accuracy of the geometric model – that is the difference between the model and the reality – depends on how much detail the user invokes, the nature of the volumetric primitives and the nature of the scene. The idea is to obtain a geometric model that reflects the structure of the building and which can be used in subsequent processing to derive camera positions and facilitate a correspondence algorithm. Thus a modern tower block may be represented by a single box, and depth variations, which occur over a face due to windows that are contained in a plane parallel to the wall plane, are at this stage of the process ignored.

Once a complete geometric model has been defined, a reconstruction algorithm is invoked, for each photographic view. The purpose of this process is to recover the camera view points, which is necessary for view interpolation, together with the world coordinates of the model, which are necessary if the model is going to be used in a conventional rendering system. This is done by projecting the geometric model, using a hypothesized view point, onto the photographic views and comparing the position of the image edges with the position of the projected model edges. The algorithm works by minimizing an objective function which operates on the error between observed image edges and the projected model edges. Correspondence between model edges and image edges having already been established, the algorithm has to proceed towards the solution without getting stuck at a local minimum.

These two processes – photogrammetric modelling and reconstruction – extract sufficient information to enable a conventional rendering process that Debevec calls 'view-dependent texture mapping'. Here a new view of a building is generated by projecting the geometric model from the required view point, treating the reference views as texture maps and reprojecting these from the new view point onto the geometric model. The implication here is that the building is 'oversampled' and any one point will appear in two or more photographic views. Thus when a new or virtual view is generated there will, for each pixel in the new view, be a choice of texture maps with (perhaps) different values for the same point on the building due to specularities and unmodelled geometric detail. This problem is approached by mixing the contributions in inverse proportion to the angles that the new view makes with the reference view directions as shown in Figure 16.20. Hence the term 'view-dependent texture mapping' – the contributions are selected and mixed according to the position of the virtual view point with respect to the reference views.

The accuracy of this rendering is limited to the detail captured by the geometric model and there is a difference between the real geometry and that of the

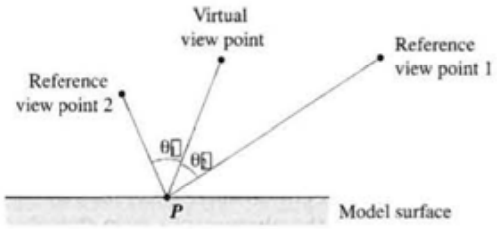


Figure 16.20
The pixel that corresponds to point ***P*** in the virtual view receives a weighted average of the corresponding pixels in the reference images. The weights are inversely proportional to θ_1 and θ_2.

model. The extent of this difference depends on the labour that the user has put into the interactive modelling phase and the assumption is that the geometric model will be missing such detail as window recesses and so on. For example, a facade modelled as a plane may receive a texture that contains such depth information as shading differences and this can lead to images that do not look correct. The extent of this depends on the difference between the required viewing angle and the angle of the view from which the texture map was selected. Debevec *et al.* (1996) go on to extend their method by using the geometric model to facilitate a correspondence algorithm that enables a depth map to be calculated and the geometric detail missing from the original model to be extracted. Establishing correspondence also enables view interpolation.

This process is called 'model-based' stereo and it uses the geometric model as *a priori* information which enables the algorithm to cope with views that have been taken from relatively far apart – one of the practical motivations of the work is that it operates with a sparse set of views. (The main problem with traditional stereo correspondence algorithms is that they try to operate without prior knowledge of scene structure. Here the extent of the correspondence problem predominantly depends on how close the two views are to each other.)

17 Computer animation

Computer animation is a huge subject and deserves a complete textbook in its own right. This chapter concentrates on foundation topics that have become established in the field and serves as an introduction to the subject, rather than comprehensive coverage. The aim of the material is to give the reader a good grounding in the concepts on which most modern systems are based.

Introduction

Leaving aside some toys of the nineteenth century, it is interesting to consider that we have only had the ability to create and disseminate moving imagery for a very short period – since the advent of film. In this time it seems that animation has not developed as a mainstream art form. Outside the world of Disney and his imitators there is little film animation that reaches the eyes of the common man. It is curious that we do not seem to be interested in art that represents movement and mostly consign animation to the world of children's entertainment. Perhaps we can thank Disney for raising film animation to an art form and at the same time condemning it to a strange world of cute animals who are imbued with a set of human emotions.

With the advent of computer animation, will this situation change? Perhaps it is too early to say. Computer animation has for many years been locked into its own 'artistic' domain and its most common manifestations are for TV title sequences and TV commercials. These productions, known derisively as 'flying logos' move rigid bodies around in three-dimensional space and have been the mainstream form of computer animation for around two decades. Their novelty having palled a long time ago, the productions exhibit a strange ambivalence: the characters have to retain their traditional function and at the same time have an 'animation personality'.

Computer animation is becoming increasingly used in the cinema, more, however, as a special effects tool than as a medium in its own right. (And indeed one of the most ubiquitous tools used by recent productions – morphing – is strictly not computer animation at all, but the two-dimensional pixel-by-pixel post-processing of filmed imagery.) The late 1990s, have seen the emergence of full-length computer animation productions, but it is still too early to judge whether this medium will develop and endure.

At first there was much optimism for computer animation. In a 1971 edition of the classic *The Techniques of Film Animation* (Hallas and Manvell 1971) the authors commenting on early scientific computer animation state:

> The position at present is that the scientist and the animator can now create drawings that move in three or four dimensions, drawings that can rotate in space, and drawings involving great mathematical precision representing a complex mathematical factor or scientific principle. The process takes a fraction of the time for a production of a conventional cartoon, a condition every animator has wished for ever since the invention of cinematography. What may now be needed is an artist of Klee's talent who could invent a new convention for creating shapes and forms. The tools are there and the next ten years will surely lead to the development of exciting visual discoveries.

In fact, the next 20 years saw little development of computer animation beyond its utilitarian aspects, but perhaps in the 1990s we are beginning to see evidence of this early prediction.

What can computer animation offer to an animation artist? Two major tools certainly. First, the substantial shortening of routine workload over conventional cel animation. Second, the ability to make three-dimensional animation which means that we can 'film' the movement and interaction of three-dimensional objects. Film animation has been firmly locked into two-dimensional space with most effort being spent on movement and characterization with only a nod here and there to three-dimensional considerations such as shading and shadows. It would seem that animators still want to use manual techniques in the main, and indeed some of the most popular commercial productions in the 1990s have used stop-motion animation of characters made from modelling clay.

Leaving aside the issue of art, the main practical problem that is central to all computer animation is motion specification or control. Beyond the obvious labour involved in building complex models of objects or characters that are going to make up a computer animation (which are the same problems faced by static rendering), there is the scripting or control of realistic movement, which

is after all the basis of the art of animation. This becomes more and more difficult as models become more and more complex. Animating a single rigid body that possesses a single reference point is reasonably straightforward; animating a complex object such as an animal which may have many parts moving, albeit in a constrained manner, relative to each other, is extremely difficult. Certainly at the moment the most complex computer animations are being produced in Hollywood and to highlight the difficult problem of movement control we will start the chapter by examining a contemporary example.

Steven Speilberg's film *Jurassic Park* is reckoned to be the most life-like computer animation accomplished to date. It has an interesting history, and recognizing that it is a pinnacle of achievement in realistic animation we will look briefly at the techniques that were used to produce it (in Section 17.3). The role of computer animation in this case was to bring to life creatures that could not be filmed and the goal was 'realism'. This, however, is not the only way in which computer animation is being used in films. In the Disney production *The Lion King* (1994), computer animation is used to imitate Disney-type animation, to give the same look and feel as the traditional animation so that it can mix seamlessly with traditional cel animation. In this production a stampede sequence was produced using techniques similar to those described in Sections 17.7 and 17.8. The sequence was perhaps more complex, in terms of the number of animal characters used and their interaction, than could have been produced manually; and this was the motivation for using computer techniques.

For *Jurassic Park*, Speilberg originally hired a stop-motion (puppet or model) animation expert to bring the creatures to life using this highly developed art form. The only computer involvement was to be the post-processing of the stop-motion animation (with motion blur) to make the sequences smoother and more realistic. This task was to be undertaken by Industrial Light and Magic (ILM) – a company already very experienced both in the use of 'traditional' special effects and the use of digital techniques such as morphing. However, at the same time ILM developed a Tyrannosaurus Rex test sequence using just computer animation techniques and when Speilberg was shown this sequence, so the story goes, he immediately decided that all the animation should be produced by ILM's computers. *Jurassic Park* is viewed as a turning point in the film industry and many people see this film as finally establishing computer graphics as the preferred tool in the special effects industry and as a technique (given the commercial success of *Jurassic Park*) that Hollywood will make much of in the years to come.

The advance in realism that emerged from this animation was the convincing movements of the characters. Although great attention was paid to modelling and detail such as the skin texture, it is in the end the motion that impresses. The realism of the motion was almost certainly due to the unique system for scripting the movements of the model. Although the computer techniques gave much freedom over stop-motion puppet animation, where the global movements of the model are restricted by the mechanical fact that it is attached to a support rig, it is the marriage of effective scripting with the visual

realism of the model that produced a film that will, perhaps, be perceived in the future as *King Kong* is now.

17.1 A categorization and description of computer animation techniques

Computer animation techniques can be categorized by a somewhat unhappy mix of the type and nature of the objects that are going to be animated and the programming technique used to achieve the animation. We have chosen to describe the following types of computer animation:

- Rigid body animation.
- Articulated structure animation.
- Dynamic simulation.
- Particle animation.
- Behavioural animation.

These categories are not meant to be a complete set of computer animation techniques; for example, we have excluded the much studied area of soft body or deformable object animation. Techniques that have been used are as wide and varied as the animation productions – we have chosen these particular five because they seem to have become reasonably well established over the relatively short history of computer animation. Some animation may, of course, be produced using a mixture of the above techniques.

Rigid body animation is self-explanatory and is the easiest and most ubiquitous form. In its simplest form it means using a standard renderer and moving objects and/or the view point around.

Articulated structures are computer graphics models that simulate quadrupeds and bipeds. Such models can range from simple stick figures up to attempts that simulate animals and human beings complete with a skin and/or clothes surface representation. The difficulty of scripting the motion of articulated structures is a function of the complexity of the object and the complexity of the required movements. Usually we are interested in very complex articulated structures, humans or animals, and this implies, as we shall see, that motion control is difficult.

Dynamic simulation means using physical laws to simulate the motion. The motivation here is that these laws should produce more realistic motion than that which can be achieved manually. The disadvantage of dynamic simulation is that it tends to remove artistic control from the animator.

Particle animation means individually animating large populations of particles to simulate some phenomenon viewed as the overall movement of the particle 'cloud' such as a fireworks display. Particles, as the name implies, are small bodies each of which normally has its own animation script.

Behavioural animation means modelling the behaviour of objects. What we mean by 'behaviour' is something more complex than basic movement, and may depend on certain behavioural rules which are a function of object attributes and the evolving spatial relationship of an object to neighbouring objects. Behavioural animation is like particle animation with the important extension that particle scripts are not independent. A collection of entities in behavioural animation evolve according to the behaviour of neighbours in the population. The stampeding animals in *The Lion King*, for example, moved individually and also according to their position in the stampeding herd. Another example is the way in which birds move in a flock and fishes move in a shoal. Each individual entity has both autocratic movement and also movement influenced by its continually changing spatial relationship with other entities in the scene. The goal of the behavioural rules in this context is to have a convincing depiction of the herd as an entity.

17.2 Rigid body animation

Rigid body animation is the oldest and most familiar form of computer animation. Its most common manifestation is the ubiquitous 'flying logo' on our TV screens and it appears to have established itself as a mandatory technique for titles at the beginning of TV programmes. Rigid body animation could be described as the fundamental animation requirement and is likely to be used in some form by all of the other categories. It is the simplest form of computer animation to implement and is the most widely used. It is mainly used by people who do not have a formal computer or programming background, consequently the interface issue is critically important. This type of animation was an obvious extension of programs that could render three-dimensional scenes. We can produce animated sequences by rendering a scene with an object in different positions, or by moving the view point (the virtual camera) around, recording the resulting single frames on video tape or film.

The problem is: how do we specify and control the movement of objects in a scene. Either the objects can move, or the virtual camera can move or we can make both move at the same time. We will describe how to move a single object but the technique extends in an obvious way to the other cases.

There are two established approaches to 'routine' rigid body animation – keyframing or interpolation systems and explicit scripting systems.

17.2.1 Interpolation or keyframing

Keyframing systems are based on a well-known production technique in film or cel animation. To cope with the prodigious workload in developing an animation sequence of any length, animation companies developed a hierarchical system wherein talented animators specify a sequence by drawing keyframes at

certain intervals. These are passed to 'inbetweeners' who draw the intermediate frames which are then coloured by 'inkers'. (This hierarchy was reflected in the rewards received by the members of the team. In Disney's *Snow White and the Seven Dwarfs* the four chief animators were paid \$100 a week, the inbetweeners \$35 and the inkers \$20.)

It was natural that this process be extended to three-dimensional computer animation – the spatial juxtaposition of objects in a scene can be defined by keyframes and the computer can interpolate the inbetween frames. However, many problems arise and these are mainly due to the fact that simple interpolation strategies cannot replace the intelligence of a human inbetweeners. In general, we need to specify more keyframes in a computer system than would be required in traditional animation.

Consider the simple problem of a bouncing ball. If we use three key frames – the start position, the end position and the zenith together with linear interpolation, then the resulting trajectory will be unrealistic (Figure 17.1). Linear interpolation is generally inadequate in most contexts.

We can improve on this by allowing the animator to specify more information about the motion characteristics between the key frames. For example, a curved path could be defined. This, however, would say nothing about how the velocity varied along the path. A ball moving with uniform velocity along, say, a parabola would again look unrealistic. Thus to control motion correctly when we are moving objects around we must explicitly define both the positional variation as a function of time and the dynamic behaviour along the specified path.

We can give such information in a number of ways. We could, for example, work with a set of points – key frame points – defining where an object is to be at certain points in time and fit a cubic, say, through these points.

If we use B-splines for the interpolation – as described in Section 3.6.3 – then the key positions become knot points. Generally, we require a curve that is C^2 continuous to simulate the motion of a rigid body. If we restrict ourselves to considering position then we emplace an object as a function of time in the scene using a 4×4 modelling transformation $\boldsymbol{M}$ of the form:

$$\boldsymbol{M}(t) = \begin{bmatrix} 0 & 0 & 0 & t_x(t) \\ 0 & 0 & 0 & t_y(t) \\ 0 & 0 & 0 & t_z(t) \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

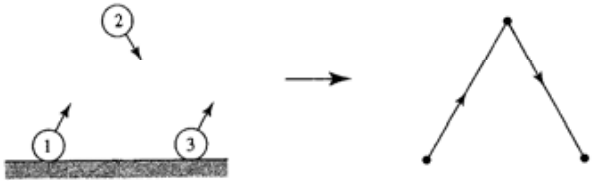


Figure 17.1
Linear interpolation will produce an unrealistic trajectory for a bouncing ball specified at three key positions.

Newton's law gives us that:

$$\frac{\boldsymbol{F}}{m} = \boldsymbol{a} = \frac{\mathrm{d}^2\boldsymbol{M}(t)}{\mathrm{d}t^2}$$

Thus, to imitate the effect of a moving object in space – to make the motion look 'natural' – the elements of the transformation matrix must have continuous second derivatives. So it seems that interpolation from keys is straightforward, and indeed it is for simple cases. However, there are a number of problems. Usually we want to animate an object that exhibits a some rotation as it translates along a path. We cannot apply the same interpolation scheme to:

$$\boldsymbol{M}(t) = \begin{bmatrix} a_{11}(t) & a_{12}(t) & a_{13}(t) & t_x(t) \\ a_{21}(t) & a_{22}(t) & a_{23}(t) & t_y(t) \\ a_{31}(t) & a_{32}(t) & a_{33}(t) & t_z(t) \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

because the matrix elements $a_{11}, \ldots, a_{33}$ are not independent. We do not want the body to change shape and so the sub-matrix $\boldsymbol{A}$ must remain orthonormal at all times – the column vectors must be unit vectors and form a perpendicular triple. Thus, positional elements can be interpolated independently but rotational elements cannot. If we attempt to linearly interpolate between nine pairs of elements $a_{11}, \ldots, a_{33}$ then the in-between matrices $\boldsymbol{A}_i$ will not be orthonormal and the object will change shape. (The subject of interpolation of rotation is dealt with separately in Section 17.2.3.)

Another problem arises from the fact that the kinematics of the motion (the velocity and acceleration) of the body and the geometry of the path are specified by the same entity – the transformation matrix $\boldsymbol{M}(t)$. In general an animator will require control so that that the kinematics of the body along the path can be modified.

Yet another problem emerges from the specifics of the interpolation scheme. It may be that the nature of the path between keys is not what the animator requires, in particular depending on the number of keys specified, unwanted excursion may occur. Also, there is the problem of the locality of influence of the keys which are the knot points in the B-spline curve. It may be that the animator requires to change the path in a way that is not possible by changing the position of a single key and requires the insertion of new keys. These disadvantages suggest an alternative approach where the animator explicitly specifies the curves for path and motion along a path rather than presenting a set of keys to an interpolation scheme whose behaviour is 'mysterious'.

17.2.2 Explicit scripting

We are thus led the idea of an explicit script and some kind of interface that enables a person to write the script. The best approach is to use a graphical interface. This will suffer from the usual problem of trying to perceive the three-

dimesionality of a scene or scene representation from a two-dimensional projection, but if we can produce the sequences, or a wireframe version of the finished sequence, in real time then this difficulty is ameliorated.

An obvious idea is to use cubic parametric curves as a script form (Chapter 3). Such a curve can be used as a path over which the reference point or origin of the object is to move. These can be easily edited and stored for possible future use. The best approach, called the double interpolant method, is to use two curves, one for the path of the object through space and one for its motion characteristic along the path. Then a developer can alter one characteristic independently of the other.

An interface possibility is shown in Figure 17.2. The path characteristic is visualized and altered in three windows that are the projections of the curve in the *xy*, *yz* and *xz* planes. The path itself can be shown embedded in the scene with three-dimensional interpretative clues coming from the position of other objects in the scene and vertical lines drawn from the curve to the *xy* plane. The animator sets up the path curve $\boldsymbol{Q}(u)$, applies a velocity curve $\boldsymbol{V}(u)$ and views the resulting animation, editing either or both characteristics if necessary.

Generating the animation from these characteristics means deriving the position of the object at equal intervals in time along the path characteristic. This is shown in principle in Figure 17.3. The steps are:

(1) For a frame at time t find the distance s corresponding to the frame time t from $\boldsymbol{V}(u)$.

(2) Measure s units along the path characteristic $\boldsymbol{Q}(u)$ to find the corresponding value for u.

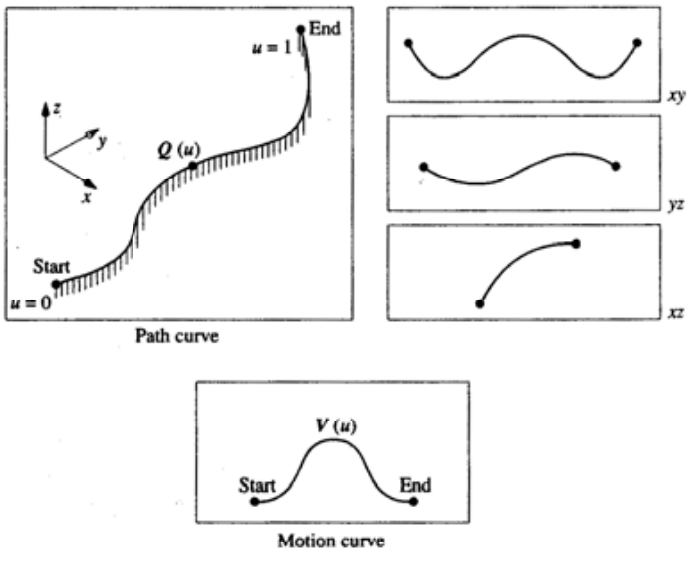


Figure 17.2
Motion specification for rigid body animation – an interface specification.

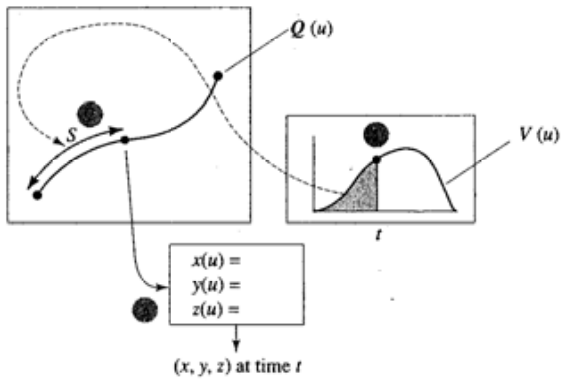


Figure 17.3
Finding the object position (x, y, z) at time t.

(3) Substitute this value of u into the equations for $\boldsymbol{Q}(u)$ to find the position of the object (x, y, z).

(4) Render the object in this position.

This simple process hides a subsidiary problem called reparametrization. $\boldsymbol{V}$ is parametrized in terms of u, that is:

$$\boldsymbol{V}(u) = (t, s)$$

where $t = T(u)$ and $s = S(u)$

Given the frame time t_f we have to find the value of u such that $t_f = T(u)$. We then substitute this value of u into $s = S(u)$ and 'plot' this distance on the path characteristic $\boldsymbol{Q}(u)$. Here we have exactly the same problem. The path characteristic is parametrized in terms of u not s. The significance of this problem is demonstrated graphically in Figure 17.4 which compares equal (arclength) intervals with equal intervals in the curve parameter.

The general problem of reparametrization in both cases involves inverting the two equations:

$$u = T^{-1}(t) \quad \text{and} \quad u = \boldsymbol{Q}^{-1}(s)$$

An approximate method that given t or s finds a close value of u, is accumulated chord length. Shown in principle in Figure 17.5 the algorithm is:

(1) Construct a table of accumulated chord lengths by taking some small interval in u and calculating the distances $l_1, l_2, l_3, \ldots$ and inserting in the table $l_1, (l_1+l_2), (l_1+l_2+l_3), \ldots$ the accumulated lengths.

(2) To find the value of u corresponding to s, say, to within the accuracy of this method, we take the nearest entry in the table to s.

This simple approach does not address many of the requirements of a practical system, but it is a good basic method from which context-dependent enhancements can be grown. In particular it can form the basis for both a scripting system and an interactive interface. We briefly describe some of the more important omissions.

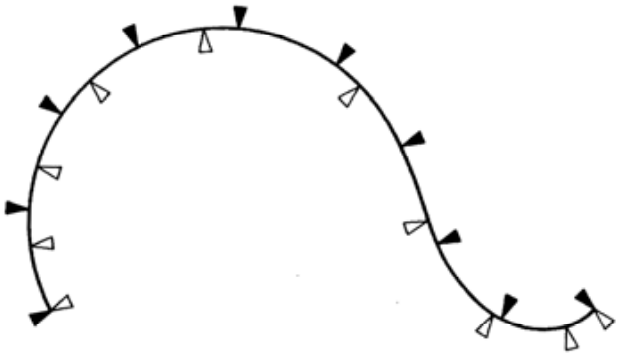


Figure 17.4
Intervals of equal parametric length (outline arrowheads) do not correspond to equal intervals of arclength (black arrowheads)

The first is that if we freely change $\boldsymbol{V}(u)$, then the total time taken for the object to travel along the curve will in general change. We may, for example, make the object accelerate more quickly from rest shortening the time taken to travel along the path. Many animations, however, have to fit into an exact time slot and a more normal situation would involve changing $\boldsymbol{V}(u)$ under the constraint that the travel time remains fixed.

Another more obvious problem is: what attitude does the object take as it moves along a path? The method as it stands is only suitable for single particles or, equivalently, a single reference point in an object which would just translate 'upright' along the path. Usually we want the object to rotate as it translates. Simplistically we can introduce another three script curves to represent the attitude of the object as it moves along the path. The easiest way to do this is to parametrize the rotation by using three angles specifying the rotation about each of three coordinate axes rigidly attached to the object. These are known as Euler angles and are called roll, pitch and yaw.

If we are producing an animation with many objects moving in the scene and if these objects are animated, one at a time, independently, then what do we do about collisions? If we use a standard rendering pipeline (with a Z-buffer) then colliding objects will simply move through each other, unless we explicitly detect this event and signal it through the design interface. Collision detection is a distinctly non-trivial problem. Objects that we normally want to deal with in computer animation can be extremely complex – their spatial extent specified by a geometric description that in most cases will not be amenable to collision

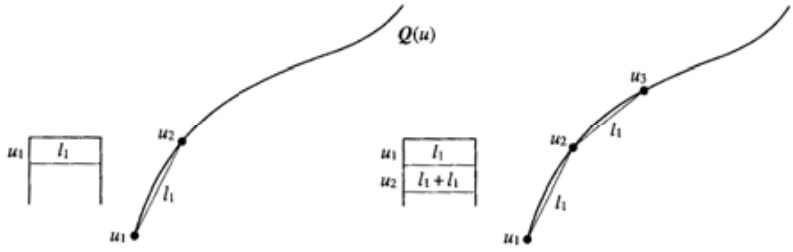


Figure 17.5
Accumulated chord-length approximation.

detection. Consider two polygon mesh objects that each contain a large number of polygons. It is not obvious how to detect the situation that a vertex from one object has moved inside the space of another. A straightforward approach would involve a comparison between each vertex in one object against every polygon in the other – an extremely time consuming problem. And the detection of a collision is only part of the problem; how do we model the reaction of objects, their deformation and movement after a collision? (Collision detection is described in more detail in Section 17.5.)

17.2.3 Interpolation of rotation

In rigid body animation we usually want to be able to deal with both translation and rotation. An object moves through space and changes its orientation in the process. To do this we need to parametrize rotation. (We distinguish between rotation and orientation as: orientation is specified by a normal vector embedded in an object; rotation is specified by an axis and an angle.) The traditional method is to use Euler angles where rotation is represented by using angles with respect to three mutually perpendicular axes. In many engineering applications – aeronautics, for example – these are known as roll, pitch and yaw angles. We can thus write down a rotation as:

$$R(\theta_1, \theta_2, \theta_3)$$

Euler angles are implemented by using a transformation matrix – one matrix for each Euler angle – as introduced in Chapter 1. A general rotation is thus effected by the product of the three matrices. As we saw in Chapter 1, to effect a rotation we specify three rotation matrices noting that rotation matrices are not commutative and the nature of the rotation depends on the order in which they are applied. However, leaving that problem aside we will now see that more significant difficulties for an animator occur if rotation is parametrized in this way.

We now consider a simple example. Figure 17.6 shows a letter ***R*** moving from an initial to a final position. In both cases the start and final positions are the same but the path between these is substantially different. In the first case a single rotation about the x axis of 180° has been applied. In the second case two rotations of 180°, about the y and z axes are applied simultaneously. The single rotation results in the character moving in a two-dimensional plane without 'twisting' while in the latter case the character follows a completely different path through space twisting about an axis through the character as it translates. What are we to conclude from this? There are two important implications. If an animator requires a certain path from one position and orientation to another then, in general, all three Euler angles must be controlled in a manner that will give the desired effect. Return to Figure 17.6. The examples here were generated by the following two sequences:

$$R(0, 0, 0), \ldots, R(\pi t, 0, 0), \ldots, R(\pi, 0, 0) \qquad t \in [0, 1]$$

for the first route, and

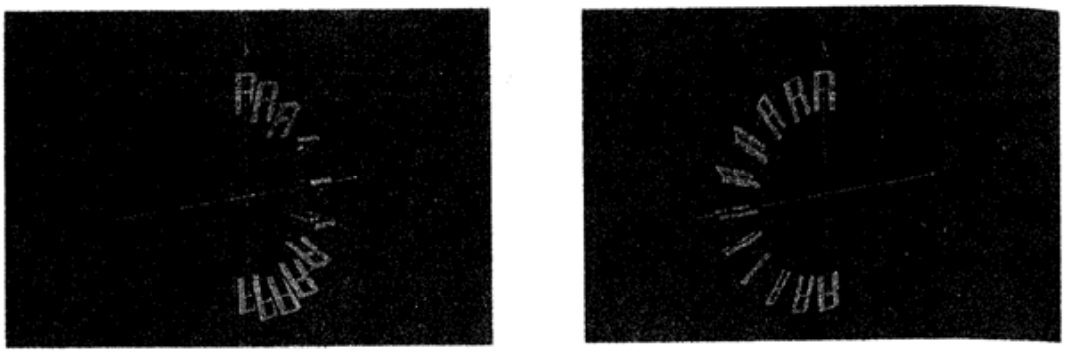


Figure 17.6
Euler angle parametrization.
(a) A single x-roll of π.
(b) A y-roll of π followed by a z-roll of π.

$$R(0, 0, 0), \ldots, R(0, \pi t, \pi t), \ldots, R(0, \pi, \pi)$$

for the second route. Examining in particular the second case we could conclude that it would be practically unworkable to expect an animator to translate an idea involving an object twisting through space into a particular movement specified by Euler angles.

The same consideration applies to interpolation: if an animator specifies keys, how is the interpolation to proceed? In fact there exists an infinity of ways of getting from one key to another in the parameter space of Euler angles. Clearly there is a need for an understood rotation from one key to the other. This single rotation may not be what the animator desires, but it is better than the alternative situation where no unique rotation is available.

Euler's theorem tells us that it is possible to get from one orientation to another by a single steady rotation. In particular it states that for two orientations O and O' there exists an axis l and an angle θ such that O undergoes rotation to O' when rotated θ about l. And we can interpret Figure 17.6 in the light of this – the first example being the single-axis rotation that takes us from the start to the stop position. But that was a special case and easy to visualize; in general for two orientations O and O' how do we find or specify this motion? This problem is solved by using quaternions.

There is another potentially important consideration in the above interpolation scheme of Euler angles. We separated motion and path in the explicitly scripted animation method because we considered that an animator would, in general, require control of the motion an object exhibited along a path separate to the specification of the path in space. The same consideration is likely to apply in specifying rotation – it may be that the motion (angular velocity) that results from linearly interpolating Euler angles is not what the animator requires.

17.2.4 Using quaternions to represent rotation

A useful introductory notion concerning quaternions is to consider them as an operator, like a matrix, that changes one vector into another, but where the

Figure 1
Angular
of r

infinite choice of matrix elements is removed. Instead of specifying the nine elements of a rotation matrix we define four real numbers. We begin by looking at angular displacement of a vector, rotating a vector by θ about an axis $\boldsymbol{n}$.

We define rotation as an angular displacement given by $(\theta, \boldsymbol{n})$ of an amount θ about an axis $\boldsymbol{n}$. That is, instead of specifying rotation as $R(\theta_1, \theta_2, \theta_3)$ we write $R(\theta, \boldsymbol{n})$. Consider the angular displacement acting on a vector $\boldsymbol{r}$ taking it to position $R\boldsymbol{r}$ as shown in Figure 17.7.

The problem can be decomposed by resolving $\boldsymbol{r}$ into components parallel to $\boldsymbol{n}$, $\boldsymbol{r}_{\parallel}$, which by definition remains unchanged after rotation, and perpendicular to $\boldsymbol{n}$, $\boldsymbol{r}_{\perp}$ in the plane passing through $\boldsymbol{r}$ and $R\boldsymbol{r}$.

$$\boldsymbol{r}_{\parallel} = (\boldsymbol{n}\cdot\boldsymbol{r})\boldsymbol{n}$$

$$\boldsymbol{r}_{\perp} = \boldsymbol{r} - (\boldsymbol{n}\cdot\boldsymbol{r})\boldsymbol{n}$$

$\boldsymbol{r}_{\perp}$ is rotated into position $R\boldsymbol{r}_{\perp}$. We construct a vector perpendicular to $\boldsymbol{r}_{\perp}$ and lying in the plane orthogonal to $\boldsymbol{n}$. In order to evaluate this rotation, we write:

$$\boldsymbol{V} = \boldsymbol{n} \times \boldsymbol{r}_{\perp} = \boldsymbol{n} \times \boldsymbol{r}$$

where × specifies the cross-product. So:

$$R\boldsymbol{r}_{\perp} = (\cos\theta)\,\boldsymbol{r}_{\perp} + (\sin\theta)\boldsymbol{V}$$

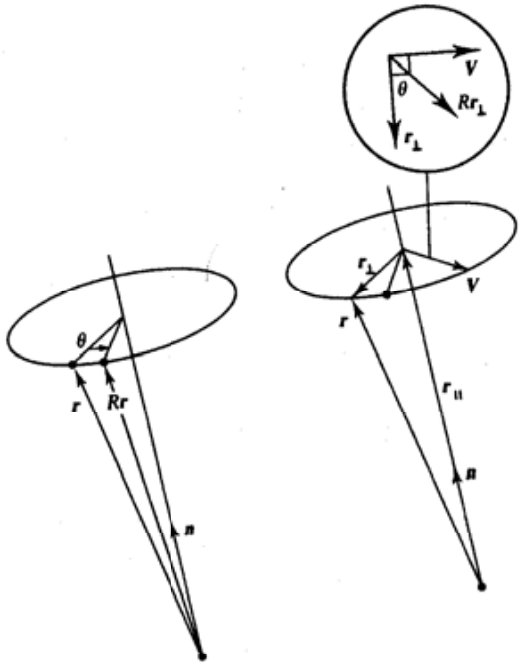


Figure 17.7
Angular displacement $(\theta, \boldsymbol{n})$ of $\boldsymbol{r}$.

hence:

$$\begin{aligned} R\boldsymbol{r} &= R\boldsymbol{r}_{\parallel} + R\boldsymbol{r}_{\perp} \\ &= R\boldsymbol{r}_{\parallel} + (\cos\theta)\,\boldsymbol{r}_{\perp} + (\sin\theta)\boldsymbol{V} \\ &= (\boldsymbol{n}\cdot\boldsymbol{r})\boldsymbol{n} + \cos\theta(\boldsymbol{r} - (\boldsymbol{n}\cdot\boldsymbol{r})\boldsymbol{n}) + (\sin\theta)\boldsymbol{n}\times\boldsymbol{r} \\ &= (\cos\theta)\boldsymbol{r} + (1-\cos\theta)\boldsymbol{n}(\boldsymbol{n}\cdot\boldsymbol{r}) + (\sin\theta)\,\boldsymbol{n}\times\boldsymbol{r} \end{aligned} \qquad [17.1]$$

We will now show that rotating the vector $\boldsymbol{r}$ by the angular displacement can be achieved by a quaternion transformation. That is, we apply a quaternion like a matrix to change a vector.

We begin by noting that to effect such an operation we only need four real numbers (this compares with the nine elements in a matrix). We require:

- The change of length of the vector.
- The plane of the rotation (which can be defined by two angles from two axes).
- The angle of the rotation.

In other words, we need a representation that only possesses the four degrees of freedom required according to Euler's theorem. For this we will use unit quaternions. As the name implies, quaternions are 'four-vectors' and can be considered as a generalization of complex numbers with s as the real or scalar part and x, y, z as the imaginary part:

$$\begin{aligned} q &= s + x\boldsymbol{i} + y\boldsymbol{j} + z\boldsymbol{k} \\ &= (s, \boldsymbol{v}) \end{aligned}$$

Here we can note their similarity to a two-dimensional complex number that can be used to specify a point or vector in two-dimensional space. A quaternion can specify a point in four-dimensional space and, if $s = 0$, a point or vector in three-dimensional space. In this context they are used to represent a vector plus rotation. $\boldsymbol{i}$, $\boldsymbol{j}$, and $\boldsymbol{k}$ are unit quaternions and are equivalent to unit vectors in a vector system; however, they obey different combination rules:

$$\boldsymbol{i}^2 = \boldsymbol{j}^2 = \boldsymbol{k}^2 = \boldsymbol{ijk} = -1, \quad \boldsymbol{ij} = \boldsymbol{k}, \quad \boldsymbol{ji} = -\boldsymbol{k}$$

Using these we can derive addition and multiplication rules each of which yields a quaternion:

$$\begin{aligned} &\text{Addition} && q + q' = (s + s',\ \boldsymbol{v} + \boldsymbol{v}') \\ &\text{Multiplication} && qq' = (ss' - \boldsymbol{v}\cdot\boldsymbol{v}',\ \boldsymbol{v}\times\boldsymbol{v}' + s\boldsymbol{v}' + s'\boldsymbol{v}) \end{aligned}$$

The conjugate of the quaternion:

$$q = (s, \boldsymbol{v})$$

is:

$$\bar{q} = (s, -\boldsymbol{v})$$

and the product of the quaternion with its conjugate defines its magnitude:

$$q\bar{q} = s^2 + |\boldsymbol{v}^2| = q^2$$

If:

$$|q| = 1$$

then q is called a unit quaternion. The set of all unit quaternions forms a unit sphere in four-dimensional space and unit quaternions play an important part in specifying general rotations.

It can be shown that if:

$$q = (s, \boldsymbol{v})$$

then there exists a $\boldsymbol{v}'$ and a $\theta \in [-\pi, \pi]$ such that:

$$q = (\cos\theta, \boldsymbol{v}' \sin\theta)$$

and if q is a unit quaternion then:

$$q = (\cos\theta, \sin\theta\ \boldsymbol{n}) \qquad \text{[Proposition 17.1]}$$

where $|\boldsymbol{n}| = 1$.

We now consider operating on a vector $\boldsymbol{r}$ in Figure 17.7 by using quaternions. $\boldsymbol{r}$ is defined as the quaternion $p = (0, \boldsymbol{r})$ and we define the operation as:

$$R_q(p) = qpq^{-1}$$

That is, it is proposed to rotate the vector $\boldsymbol{r}$ by expressing it as a quaternion multiplying it on the left by q and on the right by q^{-1}. This guarantees that the result will be a quaternion of the form $(0, \boldsymbol{v})$, in other words a vector. q is defined to be a unit quaternion $(s, \boldsymbol{v})$. It is easily shown that:

$$R_q(p) = (0, (s^2 - \boldsymbol{v}\cdot\boldsymbol{v})\boldsymbol{r} + 2\boldsymbol{v}(\boldsymbol{v}\cdot\boldsymbol{r}) + 2s(\boldsymbol{v} \times \boldsymbol{r}))$$

Using Proposition 17.1 and substituting gives:

$$\begin{aligned} Rq(p) &= (0, (\cos^2\theta - \sin^2\theta)\boldsymbol{r} + 2\sin^2\theta\ \boldsymbol{n}(\boldsymbol{n}\cdot\boldsymbol{r}) + 2\cos\theta\sin\theta\ (\boldsymbol{n} \times \boldsymbol{r})) \\ &= (0, \boldsymbol{r}\cos 2\theta + (1 - \cos 2\theta)\ \boldsymbol{n}(\boldsymbol{n}\cdot\boldsymbol{r}) + \sin 2\theta\ (\boldsymbol{n} \times \boldsymbol{r})) \end{aligned}$$

Now compare this with Equation 17.1. You will notice that aside from a factor of 2 appearing in the angle they are identical in form. What can we conclude from this? The act of rotating a vector $\boldsymbol{r}$ by an angular displacement $(\theta, \boldsymbol{n})$ is the same as taking this angular displacement, 'lifting' it into quaternion space, by representing it as the unit quaternion:

$$(\cos(\theta/2), \sin(\theta/2)\ \boldsymbol{n})$$

and performing the operation $q()q^{-1}$ on the quaternion $(0, \boldsymbol{r})$. We could therefore parametrize orientation in terms of the four parameters:

$$\cos(\theta/2), \quad \sin(\theta/2)\ \boldsymbol{n}_x, \quad \sin(\theta/2)\ \boldsymbol{n}_y, \quad \sin(\theta/2)\ \boldsymbol{n}_z$$

using quaternion algebra to manipulate the components.

Let us now return to our example of Figure 17.6 to see how this works in practice. The first single x-roll of π is represented by the quaternion:

$$(\cos(\pi/2), \sin(\pi/2)\ (1, 0, 0)) = (0, (1, 0, 0))$$

Similarly a y-roll of π and a z-roll of π are given by $(0, (0, 1, 0))$ and $(0, (0, 0, 1))$ respectively. Now the effect of a y-roll of π followed by a z-roll of π can be represented by the single quaternion formed by multiplying these two quaternions together:

$$\begin{aligned}(0, (0, 1, 0))\,(0, (0, 0, 1)) &= (0, (0, 1, 0) \times (0, 0, 1)) \\ &= (0, (1, 0, 0))\end{aligned}$$

which is identically the single x-roll of π.

We conclude this section by noting that quaternions are used exclusively to represent orientation – they can be used to represent translation but combining rotation and translation into a scheme analogous to homogeneous coordinates is not straightforward.

17.2.5 Interpolating quaternions

Given the superiority of quaternion parametrization over Euler angle parametrization, this section covers the issue of interpolating rotation in quaternion space. Consider an animator sitting at a workstation and interactively setting up a sequence of key orientations by whatever method is appropriate. This is usually done with the principal rotation operations, but now the restrictions that were placed on the animator when using Euler angles, namely using a fixed number of principal rotations in a fixed order for each key, can be removed. In general, each key will be represented as a single rotation matrix. This sequence of matrices will then be converted into a sequence of quaternions. Interpolation between key quaternions is performed and this produces a sequence of in-between quaternions, which are then converted back into rotation matrices. The matrices are then applied to the object. The fact that a quaternion interpolation is being used is transparent to the animator.

Moving in and out of quaternion space

The implementation of such a scheme requires us to move into and out of quaternion space, that is, to go from a general rotation matrix to a quaternion and vice versa. Now to rotate a vector $\boldsymbol{p}$ with the quaternion q we use the operation:

$$q(0, \boldsymbol{p})q^{-1}$$

where q is the quaternion:

$$(\cos(\theta/2), \sin(\theta/2)\boldsymbol{n}) = (s, (x, y, z))$$

It can be shown that this is exactly equivalent to applying the following rotation matrix to the vector:

$$\boldsymbol{M} = \begin{bmatrix} 1-2(y^2+z^2) & 2xy-2sz & 2sy+2xz & 0 \\ 2xy+2sz & 1-2(x^2+z^2) & -2sx+2yz & 0 \\ -2sy+2xz & 2sx+2yz & 1-2(x^2+y^2) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

By these means then, we can move from quaternion space to rotation matrices.

The inverse mapping, from a rotation matrix to a quaternion is as follows. All that is required is to convert a general rotation matrix:

$$\begin{bmatrix} M_{00} & M_{01} & M_{02} & M_{03} \\ M_{10} & M_{11} & M_{12} & M_{13} \\ M_{20} & M_{21} & M_{22} & M_{23} \\ M_{30} & M_{31} & M_{32} & M_{33} \end{bmatrix}$$

where $M_{03} = M_{13} = M_{23} = M_{30} = M_{31} = M_{32} = 0$ and $M_{33} = 1$, into the matrix format directly above. Given a general rotation matrix the first thing to do is to examine the sum of its diagonal components M_{ii} which is:

$$4 - 4(x^2 + y^2 + z^2)$$

Since the quaternion corresponding to the rotation matrix is of unit magnitude we have:

$$s^2 + x^2 + y^2 + z^2 = 1$$

and:

$$4 - 4(x^2 + y^2 + z^2) = 4 - 4(1 - s^2) = 4s^2$$

Thus, for a 4 × 4 homogeneous matrix we have:

$$s = \pm \frac{1}{2} \sqrt{M_{00} + M_{11} + M_{22} + M_{33}}$$

and:

$$x = \frac{M_{21} - M_{12}}{4s}$$

$$y = \frac{M_{02} - M_{20}}{4s}$$

$$z = \frac{M_{10} - M_{01}}{4s}$$

Spherical linear interpolation (slerp)

Having outlined our scheme we now discuss how to interpolate in quaternion space. Since a rotation maps onto a quaternion of unit magnitude, the entire group of rotations maps onto the surface of the four-dimensional unit hypersphere in quaternion space. Curves interpolating through key orientations should therefore lie on the surface of this sphere. Consider the simplest case of interpolating between just two key quaternions. A naive, straightforward linear interpolation

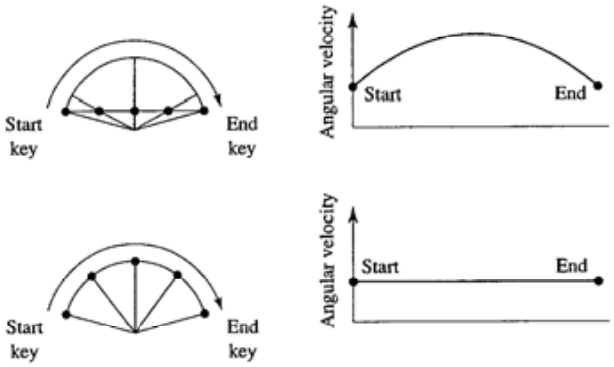


Figure 17.8
A two-dimensional analogy showing the difference between simple linear interpolation and simple spherical linear interpolation (slerp).

between the two keys results in a motion that speeds up in the middle. An analogy of this process in a two-dimensional plane is shown in Figure 17.8 which shows that the path on the surface of the sphere yielded by linear interpolation gives unequal angles and causes a speed-up in angular velocity.

This is because we are not moving along the surface of the hypersphere but cutting across it. In order to ensure a steady rotation we must employ spherical linear interpolation (or *slerp*), where we move along an arc of the geodesic that passes through the two keys.

The formula for spherical linear interpolation is easy to derive geometrically. Consider the two-dimensional case of two vectors $\boldsymbol{A}$ and $\boldsymbol{B}$ separated by angle Ω and vector $\boldsymbol{P}$ which makes an angle θ with $\boldsymbol{A}$ as shown in Figure 17.9. $\boldsymbol{P}$ is derived from spherical interpolation between $\boldsymbol{A}$ and $\boldsymbol{B}$ and we write:

$$\boldsymbol{P} = \alpha\boldsymbol{A} + \beta\boldsymbol{B}$$

Trivially we can solve for α and β given:

$$|\boldsymbol{P}| = 1$$
$$\boldsymbol{A}\cdot\boldsymbol{B} = \cos\Omega$$
$$\boldsymbol{A}\cdot\boldsymbol{P} = \cos\theta$$

to give:

$$\boldsymbol{P} = \boldsymbol{A}\,\frac{\sin(\Omega-\theta)}{\sin\Omega} + \boldsymbol{B}\,\frac{\sin\theta}{\sin\Omega}$$

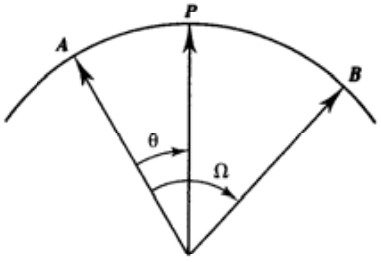


Figure 17.9
Spherical linear interpolation

Spherical linear interpolation between two unit quaternions q_1 and q_2, where:

$$q_1 \cdot q_2 = \cos \Omega$$

is obtained by generalizing the above to four dimensions and replacing θ by Ωu where $u \in [0, 1]$. We write:

$$\text{slerp}\,(q_1, q_2, u) = q_1 \frac{\sin\,(1-u)\Omega}{\sin\,\Omega} + q_2 \frac{\sin\,\Omega u}{\sin\,\Omega}$$

Now given any two key quaternions, p and q, there exist two possible arcs along which one can move, corresponding to alternative starting directions on the geodesic that connects them. One of them goes around the long way and this is the one that we wish to avoid. Naively one might assume that this reduces to either spherically interpolating between p and q by the angle Ω, where:

$$p \cdot q = \cos \Omega$$

or interpolating in the opposite direction by the angle $2\pi - \Omega$. This, however, will not produce the desired effect. The reason for this is that the topology of the hypersphere of orientation is not just a straightforward extension of the three-dimensional Euclidean sphere. To appreciate this, it is sufficient to consider the fact that every rotation has two representations in quaternion space, namely q and $-q$, that is, the effect of q and $-q$ is the same. That this is so is due to the fact that algebraically the operator $q()q^{-1}$ has exactly the same effect as $(-q)()(-q)^{-1}$. Thus, points diametrically opposed represent the same rotation. Because of this topological oddity care must be taken when determining the shortest arc. A strategy that works is to choose interpolating between either the quaternion pair p and q or the pair p and $-q$. Given two key orientations p and q find the magnitude of their difference, that is $(p-q)\cdot(p-q)$, and compare this to the magnitude of the difference when the second key is negated, that is $(p+q)\cdot(p+q)$. If the former is smaller then we are already moving along the smallest arc and nothing needs to be done. If, however, the second is smallest, then we replace q by $-q$ and proceed. These considerations are shown schematically in Figure 17.10.

So far we have described the spherical equivalent of linear interpolation between two key orientations, and, just as was the case for linear interpolation, spherical linear interpolation between more than two key orientations will produce jerky, sharply changing motion across the keys. The situation is summarized in Figure 17.11 as a three-dimensional analogy which shows that the curve

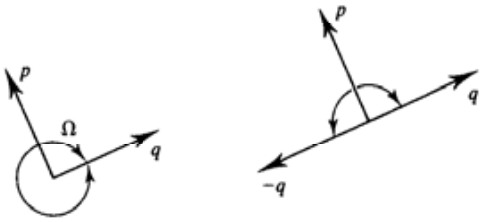


Figure 17.10
Shortest arc determination on quaternion hypersphere.

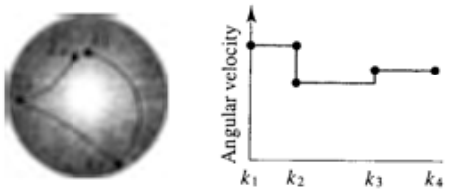


Figure 17.11
A three-dimensional analogy of using slerp to interpolate between four keys.

on the surface of the sphere is not continuous through the keys. Also shown in this figure is the angular velocity which is not constant and which is discontinuous at the keys. The angular velocity can be made constant across all frames by assigning to each interval between keys a number of frames proportional to the magnitude of the interval. That is, we calculate the magnitude of the angle θ between a pair of keys q_i and q_{i+1} as:

$$\cos\theta = q_i \cdot q_{i+1}$$

where the inner product of two quaternions $q = (s, \mathbf{v})$ and $q' = (s', \mathbf{v}')$ is defined as:

$$q \cdot q' = ss' + \mathbf{v} \cdot \mathbf{v}'$$

Curing the path continuity is more difficult. What is required for higher order continuity is the spherical equivalent of the cubic spline. Unfortunately because we are now working on the surface of a four-dimensional hypersphere, the problem is far more complex than constructing splines in three-dimensional Euclidean space. Duff (1986) and Shoemake (1985) have all tackled this problem.

Finally, we mention a potential difficulty when applying quaternions. Quaternion interpolation is indiscriminate in that it does not prefer any one direction to any other. Interpolating between two keys produces a move that depends on the orientations of the keys and nothing else. This is inconvenient when choreographing the virtual camera. Normally when moving a camera the film plane is always required to be upright – this is usually specified by an 'up' vector. By its very nature, the notion of a preferred direction cannot easily be built into the quaternion representation and if it is used in this context the camera-up vector may have to be reset or some other fix employed. (Roll of the camera is, of course, used in certain contexts.)

17.2.6 The camera as an animated object

Any or all of the external camera parameters can be animated but the most common type of camera animation is surely that employed in first person computer games and similar applications where a camera flies through a mostly static environment under user interface control – the so-called 'walkthrough' or 'flyby'. Here, the user is controlling the view point, and usually a (two degrees of freedom) viewing direction. Interpolation is usually required between consecu-

tive interface samples that form the keys and the most important constraint is to keep the camera-up vector up; normally no orientation about the view direction vector is tolerated. (The only time the camera rolls about the view direction in first person games is when you die.)

Another common application is where the view point is under user control but the camera always points to an object of interest which can be static or itself moving. A common example of this is the (confusingly called) third person computer games where the camera is tied to (say) the head of the character via an 'invisible' rigid link. Instead of seeing the environment through the eyes of the character the user sees over the character's shoulder. In this case the view direction vector is derived from the character. The view point effectively moves over a part of the surface of a sphere centred on the character. If quaternion interpolation is used in this application then the up vector has to be reset after an interpolation.

17.3 Linked structures and hierarchical motion

Scripting of movement of quadruped or biped models in computer animation has, for some time, been an energetically pursued research topic. The computer models are known as articulated or linked structures and most approaches for movement control in animation have attempted to extend techniques developed in the industrial robotics field. Just as interpolation was the first idea to be applied to rigid body animation, parametrizing the movement of links or limbs in an articulated structure using robotic methodology seemed the way to proceed. Although this is perhaps an obvious approach it has not proved very fruitful. One problem is that robot control is itself a research area – by no means have all the problems been solved in that field. Probably a more important reason is that the techniques required to control the precise mechanical movements of an industrial robot do not make a comfortable and creative environment in which an animator can script the freer, more complex and subtler movements of a human or an animal.

Yet another reason is that animal structures are not rigid and the links themselves deform as illustrated in Figure 17.12. In fact, the most successful articulated structure animation to date, *Jurassic Park*, used an ad hoc technique to represent or to derive the motion of the links in complicated (dinosaur) models. Let us look briefly at these techniques. This will give an appreciation of the difficulty of the problem faced by the animators in *Jurassic Park* and the efficacy of their solution.

First, what is an articulated structure? It is simply a set of rigid objects, or links, connected to each other by joints which enable the various parts of the structure to move, in some way, with respect to each other. For animal animation the links form a simplified skeleton, a stick figure, and only exist to facilitate control of the structure. They are an abstraction which is not rendered. Instead, the link is 'covered' with the external surface of the animal object and

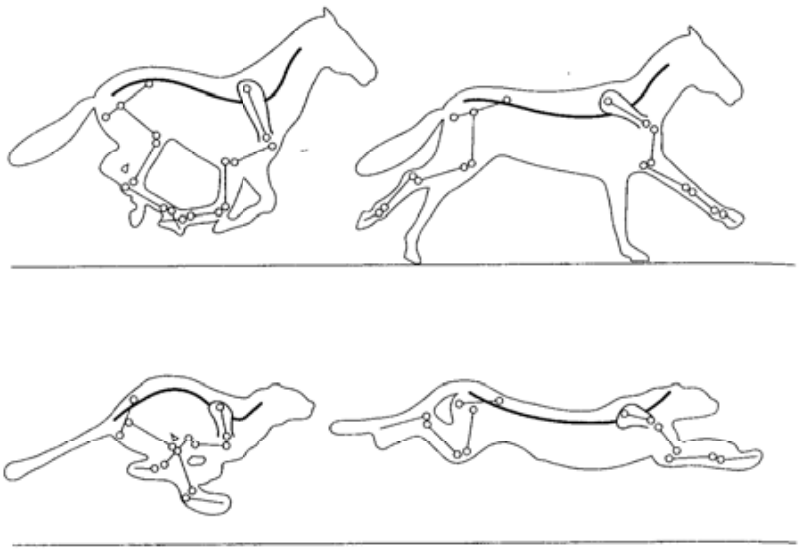


Figure 17.12
Spine flexion in a horse and a cheetah (after Gray (1968)).

this is rendered. (This is no different in principle from a rigid object represented by a polygon mesh. Here, we are effectively controlling the position and orientation of a vector representing the object.)

Consider a simple example – a single human leg. We might model this as represented in Figure 17.13(a) using two links connecting three joints – the hip joint, the knee joint and the ankle joint. Simplistically, we could constrain movement to the plane containing the joints and allow the link between the hip and the knee to rotate, between certain limits, about the hip joint and allow the link between the ankle and the knee to rotate about the knee joint (and, of course, we know that this link can only rotate in one direction). The rotation of the foot about the ankle joint is more complicated since the foot itself is an articulated structure. Given such a structure how do we begin to specify a script for, say, the way the leg structure is to behave to execute a walk action? It is fairly obvious that the motion of the structure is constrained by the overall connectivity – the structure comprises some chain of links and one link causes its neighbour to move, and constraints that the links themselves possess, like the rotational limits in human animal skeletal joints. The practical effect of this is that we cannot easily use a key frame system because these constraints must function across all interpolated positions. Thus, a system is adopted where the links have their relative motion specified. That is, the motion of link i is specified relative to that of link j to which it is connected. Such systems are thus animated by separately animating each link. They also, by definition, must possess a hierarchy – every link has one above it, unless it is the top link, and one below it, unless it is the bottom link or end effector. A link inherits the transformations of all links above it.

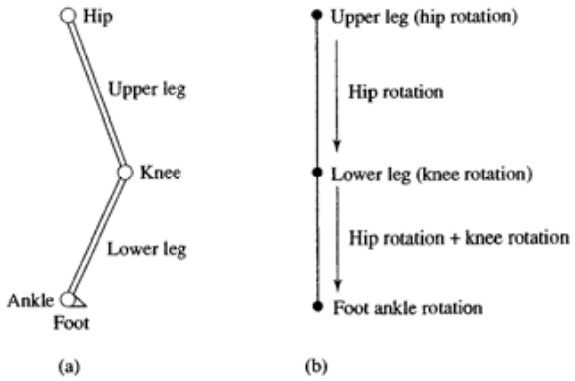


Figure 17.13
A simple articulated structure and its hierarchical representation.

There are two major approaches to this problem both of which come out of robotics – forward kinematics and inverse kinematics.

Forward kinematics is a somewhat tedious low-level approach where the animator has to specify explicitly all the motions of every part of the articulated structure. Like any low-level approach the amount of work that has to be done by the animator is a function of the complexity of the structure. The articulated structure is considered as a hierarchy of nodes (Figure 17.13(b)) with an associated transformation which moves the link connected to the node in some way. Each node represents a body part such as an upper or lower arm. We could animate such a structure by using explicit scripting curves to specify the transformation values as a function of time. Instead of having just a single path characteristic which moves a reference point for a rigid body, we may now have many characteristics each moving one part of the structure. Consider the inheritance in this structure. The hip rotation in the example causes the lower leg as well as the upper leg to rotate. The following considerations are apparent:

- **Hip joint** This is the 'top' joint in the structure and needs to be given global movement. In a simple walk this is just translation in a plane parallel to the ground plane. In a more realistic simulation we would have to take into account the fact that the hip rises and falls during the walk cycle due to the lifting action of the feet.
- **Hip–knee link rotation about the hip joint** We can specify the rotation as an angular function of time. If we leave everything below this link fixed then we have a stiff-legged walk (politely known as a compass gait but possibly more familiar as the 'goose step').
- **Knee–ankle link rotation about the knee joint** To relax the goose step into a natural walk we specify rotation about the knee joint.

And so on. To achieve the desired movement the animator starts at the top of the hierarchy and works downwards explicitly applying a script at every point. The evolution of a script is shown in Figure 17.14. Applying the top script would result in a goose step. The second script – knee rotation – allows the lower leg to

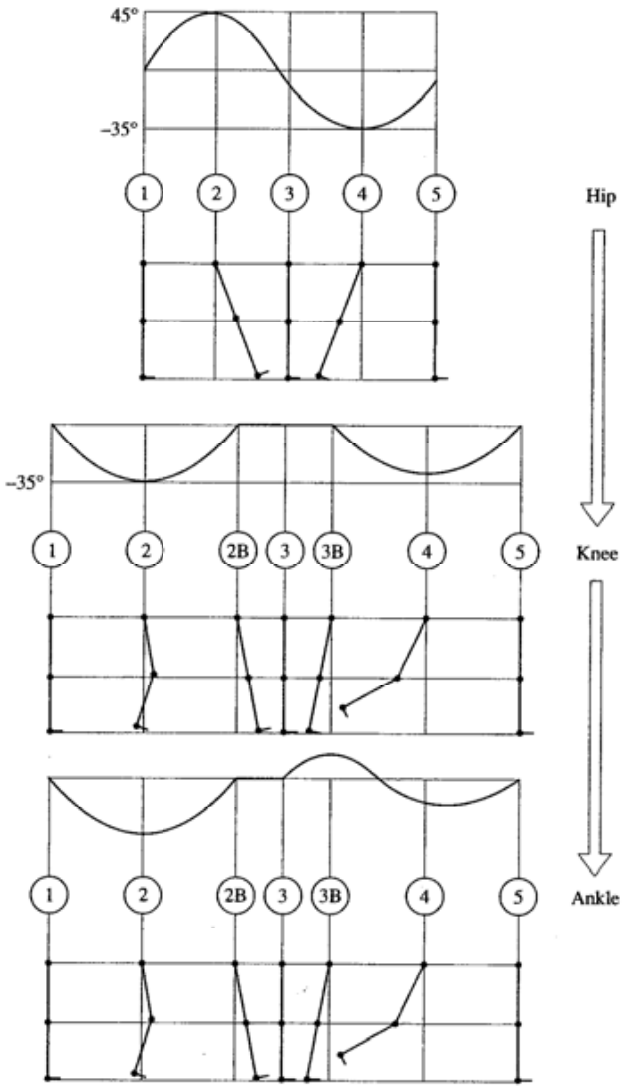


Figure 17.14
Evolution of a script for a leg.

bend. Applying both these scripts would result in a walk where the foot was always at right angles to the lower leg. This leads us into the ankle script.

Even in this simple example problems begin to accrue. It is not too difficult to see that when we come to script the foot, we cannot tolerate the hip joint moving in a straight line parallel to the floor. This would cause the foot to penetrate the floor. We have to apply some vertical displacement to the hip as a function of time and so on. And we are considering a very simple example – a walk action. How do we extend this technique for a complex articulated struc-

ture that has to execute a fight sequence rather than make a repetitive walk cycle?

Inverse kinematics, on the other hand, is a more high-level approach. Here an animator might specify something like: walk slowly from point A to point B. And the inverse kinematics technique works out a precise script for all the parts of the structure so that the whole body will perform the desired action. More precisely inverse kinematics means that only the required position of the end (or ends) of the structure are specified. The animator does not indicate how each separate part of the articulated structure is to move only that the ends of it move in the desired way. The idea comes from robotics where we mostly want the end effector of a robot arm to take up precise positions and to perform certain actions. The inverse kinematics then works out the attitude that all the other joints in the structure have to take up so that the end effector is positioned as required. However, herein lurks the problem. As the articulated structure becomes more and more complex the inverse kinematics solution becomes more and more difficult to work out. Also inverse kinematics does not, generally, leave much scope for the animator to inject 'character' into the movements, which is after all the art of animation. The inverse kinematics functions as a black box into which is input the desired movement of the ends of the structure and the detailed movement of the entire structure is controlled by the inverse kinematics method. An animator makes character with movement. Forward kinematics is more flexible in this respect, but if we are dealing with a complex model there is much expense. Figure 17.15 (also a Colour Plate) shows a simple character executing a somewhat flamboyant gait that was animated using forward kinematics.

Thus, we have two 'formal' approaches to scripting an articulated structure. Inverse kinematics enables us to specify a script by listing the consecutive positions of the end points of the hierarchy – the position of the hands or feet as a function of time. But the way in which the complete structure behaves is a function of the method used to solve the inverse kinematic equations and the animator has no control over the 'global' behaviour of the structure. Alternatively, if the structure is complex it may be impossible to implement an inverse kinematics solution anyway. On the other hand, forward kinematics enables the complete structure to be explicitly scripted but at the expense of inordinate labour, except for very simple structures. Any refinements have to be made by starting at the top of the hierarchy again and working downwards.

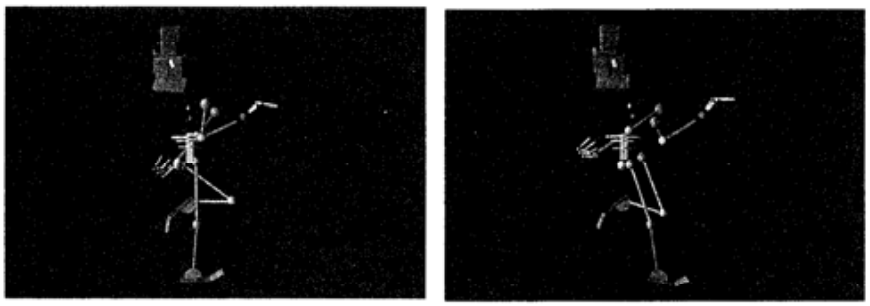


Figure 17.15
Simple characterization using an articulated structure – the flamboyant gait was animated using forward kinematics. (See also Colour Plate version.)

We now illustrate the distinction between forward and inverse kinematics more formally using as an example the simplest articulated structure possible – a two-link machine where one link is fixed and each link moves in the plane of the paper (Figure 17.16). In forward kinematics we explicitly specify the motion of all the joints. All the joints are linked and the motion of the end effector (hands or feet in the case of an animal figure) is determined by the accumulation of all transformations that lead to the end effector. We say that:

$$\boldsymbol{X} = f(\Theta)$$

where $\boldsymbol{X}$ is the motion of the end effector and Θ is a state vector specifying the position, orientation and rotation of all joints in the system. In the case of the simple two-link mechanism we have:

$$\boldsymbol{X} = (l_1 \cos\theta_1 + l_2 \cos(\theta_1 + \theta_2),\ l_1 \sin\theta_1 + l_2 \sin(\theta_1 + \theta_2)) \qquad [17.2]$$

but this expression is irrelevant in the sense that to control or animate such an arm using forward kinematics we would simply specify:

$$\Theta = (\theta_1, \theta_2)$$

and the model would have applied to it the two angles which would result in the movement $\boldsymbol{X}$.

In inverse kinematics we specify the position of the end effector and the algorithm has to evaluate the required Θ given $\boldsymbol{X}$. We have:

$$\Theta = f^{-1}(\boldsymbol{X})$$

and in our simple example we can obtain from trigonometry:

$$\theta_2 = \frac{\cos^{-1}(x^2 + y^2 - l_1^2 - l_2^2)}{2l_1l_2}$$

$$\theta_1 = \tan^{-1}\left(\frac{-(l_2 \sin\theta_2)x + (l_1 + l_2\cos\theta_2)y}{(l_2 \sin\theta_2)y + (l_1 + l_2\cos\theta_2)x}\right)$$

Now, as the complexity of the structure increases the inverse kinematics solution becomes more and more difficult. Quickly the situation develops where many configurations satisfy the required end effector movement. In the simple two-link mechanism, for example, it is easy to see that there are two link configurations possible for each position $\boldsymbol{X}$, one with the inter-link joint above

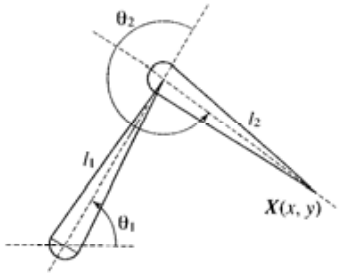


Figure 17.16
A two-link structure.

the end effector the other with it below. The attitude or state of this mechanism is specified by two angles (degrees of freedom) and we can easily foresee from this that as a structure becomes more complex it becomes increasingly difficult to derive an expression of the form $\Theta = f^{-1}(\boldsymbol{X})$. Thus, with forward kinematics the animator has to handle more and more transformations while in inverse kinematics a solution may not be possible except for reasonably simple mechanisms. A human body possesses more than 200 degrees of freedom. An inverse kinematics solution for this is practically impossible and a forward kinematics script is inordinately complicated. A way forward is to invest such models with pre-written forward kinematic scripts for common gestures such as walking, running, grasping etc. An animator then creates a script by putting together a sequence from pre-written parts.

In animating the dinosaurs in *Jurassic Park*, ILM used neither of these approaches, and in the time-honoured tradition of efficacious innovations, came up with a much simpler solution than those offered by the literature of articulated computer graphics animation. Their approach was to drive the models with a low-level forward kinematics script but they by-passed the script complexity problem by creating a script semi-automatically. They effectively enabled stop-motion animators to input their expertise directly into the computer. The stop-motion animators moved their models in the normal way and the computer sampled the motion producing a script for the computer models. ILM describe their technique in the following way:

> The system is precise, fast, compact, and easy to use. It lets traditional stop-motion animators produce animation on a computer without requiring them to learn complex software. The working environment is very similar to the traditional environment but without the nuisances of lights, a camera and delicate foam-latex skin. The resulting animation lacks the artefacts of stop-motion animation, the stops and jerkiness, and yet retains the intentional subtleties and hard stops that computer animation often lacks.

The general idea is not original. For many years it has been possible to train industrial robots by having a human operator hold the robot's hand, taking it through the actions that the robot is eventually going to perform in the stead of the human operator. Spot welding and paint spraying in the car industry is a good example of the application of this technique. Movements of all the joints in the robot's articulated structure are then read from sensors and from these a script to control the robot is produced. Future invocations of the motion sequence involved in a task can then be endlessly and perfectly repeated – indeed the robot will go on reproducing the sequence perfectly even if something else has gone wrong and the car is not present.

In *Jurassic Park* robots were already available because the stop-motion animators had already built 'animatronic' models in anticipation of the film being produced by stop-motion techniques. These were then used, in reverse as it were, by the stop-motion animators, to produce a script for the computer models. Figure 17.17 shows a stop-motion animator working out the movements for the dinosaur wrestling with the car scene. The models now, instead of being clothed

Figure 17.17
A stop-motion animator using a (real) model fitted with transducers from which a script is derived for a (virtual) computer model. (*Source*: Magid, R. 'After Jurassic Park', *American Cinematographer*, December 1993.)

and filmed one frame at a time, are turned into an input device from which a script is derived.

A similar approach, known as 'motion capture', is to use human actors from which to derive a motion script for a computer model. This involves fixing motion tracking devices to the appropriate positions of the actor's body and deriving a kinematic script in this way from the real movements of the actor. This approach is particularly popular in the video games industry which in recent years has made a transition from two-dimensional to three-dimensional animation. In this type of interactive computer animation the pre-recorded motion sequences are replayed in response to user interaction events. It is natural and economic to use motion capture in this context to record the original motion scripts for the computer models, although implicit in the approach is the limitation that the animation seen by the user can only be combinations of pre-calculated sequences.

Thus, we see from these examples that we are only at the beginning of this difficult problem of specifying motion for complex articulated structures and that many solutions to the problem have involved going outside the computer and deriving a script from the real world (reminiscent somewhat of early photographs of Disney animators who were to be seen building up facial animations by using their own image in a mirror as a guide).

17.3.1 Solving the inverse kinematics problem

In this section we look at an important notion that forms the basis for inverse kinematics algorithms. We will deal with the topic enough to give an

appreciation of the difficulties involved. A full treatment of an inverse kinematics engine is given in Watt and Watt (1992). Most approaches to this problem involve iteration towards a desired goal. That is we compute a small change $\partial\Theta$ in the joint angles that will cause the end effector to move towards the goal. This is given by:

$$d\boldsymbol{x} / d\Theta = J(\Theta)$$

J is the so-called Jacobian – a multi-dimensional extension to differentiation of a single variable. In this case it relates differential changes in Θ to those in $\boldsymbol{x}$, the position of the end effector. Note that J is a function of the current state of the structure Θ. We recall that the general problem encountered in inverse kinematics systems stems from the fact that in:

$$\Theta = f^{-1}(\boldsymbol{x})$$

the function $f()$ is non-linear and becomes more and more complex as the number of links increases. The inversion of this function soon becomes impossible analytically. The problem can be made linear by inverting the Jacobian and localizing the behaviour of the structure to small movements about the current operating point:

$$d\Theta = J^{-1}(\Theta)\ (d\boldsymbol{x})$$

The goal is known and so the iteration consists of calculating a $d\boldsymbol{x}$ by subtracting the current position and the goal and substituting into the above equation to get $d\Theta$ and proceeds as:

repeat

$d\boldsymbol{x}$:= *small movement in the direction of* $\boldsymbol{x}$

$d\Theta := J^{-1}(\Theta)\ (d\boldsymbol{x})$

$\boldsymbol{x} := f(\Theta + d\Theta)$

$J := d\boldsymbol{x}/d\Theta$

invert J

$\boldsymbol{x} := \boldsymbol{x} + d\boldsymbol{x}$

until *goal is reached*

An iteration for the three-link arm is shown in Figure 17.18.

Using the chain rule to differentiate Equation 17.2, the Jacobian for the two-link arm is given as:

$$J = \begin{bmatrix} -l_1\sin\theta_1 - l_2\sin(\theta_1 + \theta_2) & -l_2\sin(\theta_1 + \theta_2) \\ l_1\cos\theta_1 + l_2\cos(\theta_1 + \theta_2) & l_2\cos(\theta_1 + \theta_2) \end{bmatrix}$$

We are now in a position to discuss the problems engendered by this approach. First, the complexity of the expression for $\boldsymbol{x}$ makes differentiation extremely difficult to perform and a geometric approach to determining the Jacobian is desirable (Watt and Watt 1992). Second, the Jacobian is not invertible unless it is a

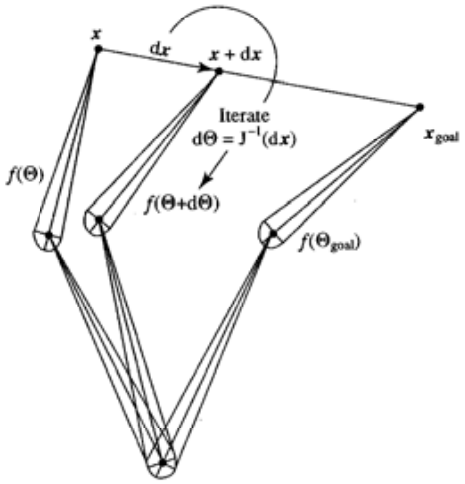


Figure 17.18
One iteration step towards the goal.

square matrix and in most (skeletal) structures that we would want to use in computer graphics applications this is not the case. Approximate solutions to this problem introduce difficulties in the iteration. In particular, tracking errors where the desired change in $\boldsymbol{x}$ is different from the actual change. Tracking error is given by:

$$\|\mathbf{J}(d\Theta) - d\mathbf{x}\|$$

A more intelligent iteration than the one given by the pseudo-code above must be employed. This involves starting with a $d\boldsymbol{x}$ evaluating the tracking error and subdividing $d\boldsymbol{x}$ until the error falls below a threshold. Yet another problem arises from singularities that exist in any system. In the two-link arm when both links line up ($\theta_2 = 0$) changes in either θ_1 or θ_2 produce motion in the end effector in exactly the same direction – perpendicular to the (common) link axis. There is now no motion possible towards the base – one degree of freedom has been lost. Another singularity occurs in this case when $\theta_2 = \pi$ when the outer link folds back to line up with the inner one. These singularities are called workspace boundary singularities in robotics because that is where they occur. The work space – the region that can be reached by the end effector – of the two-link arm is a hollow disc (providing $l_1 \neq l_2$) and the circumference of the inner and outer boundaries form the loci of all points in the two-dimensional space at which singularities occur.

Of course, we have only discussed a very simple mechanical structure. Animal skeletons are as we know far more complicated. In particular, they contain branching links. Figure 17.19 shows a simple structure used in typical human animation. In this case the root of the structure is the joint located between the hips (which has six degrees of freedom). Also shown is a categorization of the nodes from the perspective of an inverse kinematics solution. There is a single root node – the remainder of the nodes being children of the root. The base

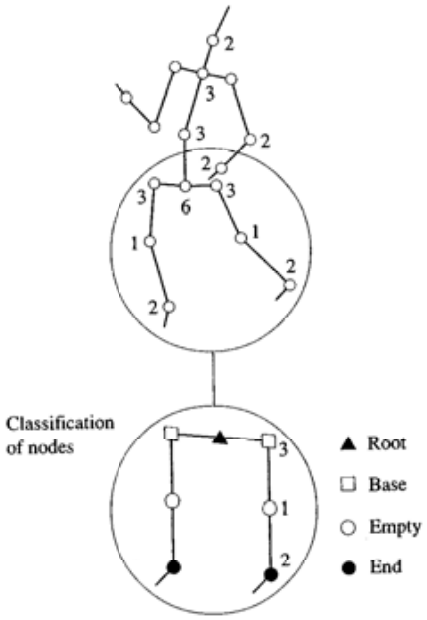


Figure 17.19
A 'minimum' stick figure for a human-type articulated structure.

nodes and the end nodes define a chain for which we may invoke an inverse kinematics solution. We can apply inverse kinematics between any two nodes in the skeleton; the only rule to be observed is that the end node is lower down the chain than the base node. The inverse kinematics solution specifies the position and orientation of all nodes between the end and base nodes – called empty nodes.

Other arrangements are possible; Philips and Badler (1991), for example, positioned the root at one foot, making the other an end node, in order to animate such motions of a standing figure as shifting the weight from one foot to another and turning.

However, there are other major differences between robot systems and animals involving the constraints. In robotics, the predominant constraints are determined by the degrees of freedom and joint angle constraints which determine the workspace of the machine. A simple example is the links in a human finger which, because of the tendon that runs through the finger do not tend to move independently. Another consideration is whether energy constraints should be taken into account to influence an inverse kinematics approach into producing a visually convincing solution for the motion of the structure. This means that both the geometric constraints and the muscle constraints are satisfied – we could presume that animals effect a change in the geometric state of their structure by minimizing the energy needed for the change. Satisfying only the geometric constraints may produce an animation that does not 'look right'.

17.4 Dynamics in computer animation

The approaches to computer animation that we have described so far have been kinematic, that is they involve the specification of motion without consideration of the masses and forces involved in the physical environment that we are trying to simulate. In this section we will look at how we can in principle write programs that simulate the forces in a scene and through Newtonian mechanics produce the desired motion 'automatically'. Such an approach is known as dynamic simulation or physically based animation.

Luxo Jr. (Figure 17.20 Colour Plate), an animated short produced by John Lasseter of Pixar in 1987, was possibly the first computer graphics animation that was perceived to have motion and appeal comparable in quality to that of traditional animators. The skills of John Lasseter imbued a desk lamp with some of the anthropomorphic behaviour reminiscent of Disney-type cartoons. The motion in *Luxo Jr.* was produced by keyframing where the animator specifies the state of the articulated structure as a key and the global motion of the structure as a spline curve. Although it is not 100% explicit control, where the animator specifies the entire state of every frame, he has a high degree of control. And, in fact, the title of Lasseter's presentation to SIGGRAPH '87, 'Principles of Traditional Animation Applied to 3D Computer Animation', reinforces this observation.

In 1988, Witkin and Kass presented a paper (Witkin and Kass 1988) in which they demonstrated that a higher-level motion control technique, based on dynamic simulation, could be used to animate *Luxo Jr.* and commented on their motivation as follows:

> Although *Luxo Jr.* showed us that the team of animator, keyframe system, and renderer can be a powerful one, the responsibility defining the motion remains almost entirely with the animator. Some aspects of animation – personality and appeal, for example – will surely be left to the animator's artistry and skill for a long time to come. However, many of the principles of animation are concerned with making the character's motion look *real* at a basic mechanical level that ought to admit to formal physical treatment . . .. Moreover, simple changes to the goals of the motion or to the physical model give rise to interesting variations on the basic motion. For example, doubling (or quadrupling) the mass of *Luxo Jr.* creates amusingly exaggerated motion in which the base *looks* heavy.

In other words, they are saying that this type of computer animation can benefit by higher-level motion control. Beyond performing rudimentary interpolation for in-between frames, a program can be set up to interpret scripts such as 'jump from A to B'. The dynamic simulation will then produce motion that is accurate and therefore realistic. Thus, we see that the motivation for using dynamics in computer animation is that in certain contexts it is easier to write the differential equations that control the motion than it is to specify the motion directly or by using keyframing. We also assume that if the physical simulation is set up correctly the subsequent motion will be more 'natural' than that produced by a kinematics system. Set against these apparent

advantages the disadvantages of using dynamics is that the environments that are easy to set up – particle systems – are too simple for most animation environments of interest and complex interacting environments are far more difficult to specify. Another problem is that the solution for such systems is computationally intensive.

Dynamic simulation may not provide a complete solution to many animation applications – there is still the problem of overall artistic control. In comparing dynamic simulation with computer methods that imitate traditional animation techniques, Cohen (1992) puts it this way:

> Traditional animation methods provide great *control* to the artist, but do not provide any tools for *automatically* creating realistic motion. Dynamic simulations on the other hand, generate physically correct motion (within limits) but it does not provide sufficient control for an artist or scientist to create desired motion.

17.4.1 Basic theory for a rigid body – particles

The basic familiar law of motion – Newton's Second Law is:

$$\boldsymbol{F} = m\,\boldsymbol{a}$$

and this is easiest to consider in the context of a particle or a point mass. $\boldsymbol{F}$ is a three-dimensional vector as is $\boldsymbol{a}$, the acceleration that the point undergoes. A point mass is a simple abstraction that can be used to model simple behaviour – we can assume that a rigid body that has extent behaves like a particle because we consider its mass concentrated at a single point – the centre of mass. A point mass can only undergo translation under the application of a force.

Newton's Second Law can also be written as:

$$\boldsymbol{F} = m\,\frac{\mathrm{d}\boldsymbol{v}}{\mathrm{d}t} = m\,\frac{\mathrm{d}^2\boldsymbol{x}}{\mathrm{d}t^2}$$

where $\boldsymbol{v}$ is the velocity and $\boldsymbol{x}$ the position of the particle. This leads to a method that finds, by integration, the position of the particle at time $t+\mathrm{d}t$ giving its position at time t as:

$$\boldsymbol{v}(t+\mathrm{d}t) = \boldsymbol{v}(t) + \frac{\boldsymbol{F}}{m}\,\mathrm{d}t$$

$$\boldsymbol{x}(t+\mathrm{d}t) = \boldsymbol{x}(t) + \boldsymbol{v}(t)\mathrm{d}t + \frac{1}{2}\,\frac{\boldsymbol{F}}{m}\,\mathrm{d}t^2$$

$\boldsymbol{F}$ can itself be a function of time and we may have more than one force acting on the body and in that case we simply calculate the net force using vector addition. If the mass of the body changes as it travels, the case of a vehicle burning fuel, for example, then the Second Law is expressed as:

$$\boldsymbol{F} = \frac{\mathrm{d}(m\cdot\boldsymbol{v})}{\mathrm{d}t}$$

As a simple example, consider a cannonball being fired from the mouth of a cannon. This could be modelled using the above equations. The cannonball is acted on by two forces – the constant acceleration due to gravity and an air resistance force that acts opposite to the velocity and is a function (quadratic) of the velocity and the square of the cross-sectional area. A simulation would be provided with the initial (muzzle) velocity and the inclination of the barrel and Newton's Second Law used to compute the arc of the missile. What we have achieved here is a simulation where at each time step the program computes continuous behaviour as a function of time.

This basic theory can only be applied directly to initial value problems where the course of the simulation is completely determined by the start conditions. We may fire a cannonball out of a cannon and its parabolic track is then completely determined by the muzzle velocity, its mass and gravity. However, an animator may rather require a system where he specifies that a cannon situated at point A is to eject a missile which is to hit the castle wall at point B.

In simulations of the initial value type the animator has no control once the start conditions have been specified. In other words we need to supply constraints to the problem. It is through those constraints that the animator is able to design a desired motion. Any potentially useful system has to be both a valid physical model, that can provide realistic motion, and at the same time admit constraints that enable the animator to achieve the desired overall motion. These have been called space–time constraints and, along with such other problems as collision response, comprise a much more difficult aspect of dynamic simulation than the application of the physical laws. We shall return to these problems later.

17.4.2 The nature of forces

Only in very simple cases can we proceed by considering an object as a point mass or equivalently as a lumped mass undergoing acceleration upon application of a force. The way in which an object moves in the modelled environment depends on the model itself, its constraints and the nature of the force. Common examples of the different types of forces used in physically based animation are:

- Acceleration due to gravity (which we have already discussed): is a constant downwards force on a body proportional to its mass and acting on the centre of mass.
- A damping force: this is opposite and proportional to the body's velocity and resists its motion. Damping forces remove energy from the body dissipating it as heat. A viscous damping force is linearly proportional to velocity and a quadratic force is proportional to the square of speed. Air resistance is approximately quadratic if we ignore effects due to the disturbance of the air.
- Elastic springs: these can connect two bodies with a force proportional to the displacement of the string from its rest length (Hooke's law).

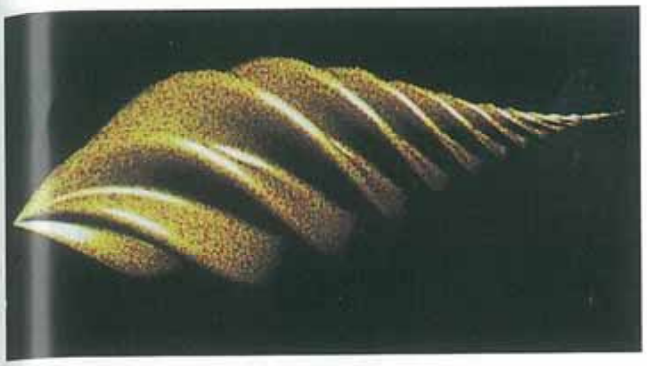
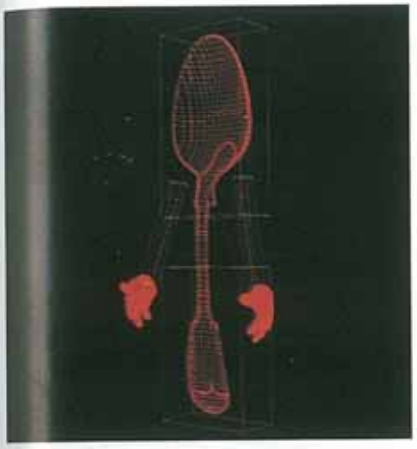
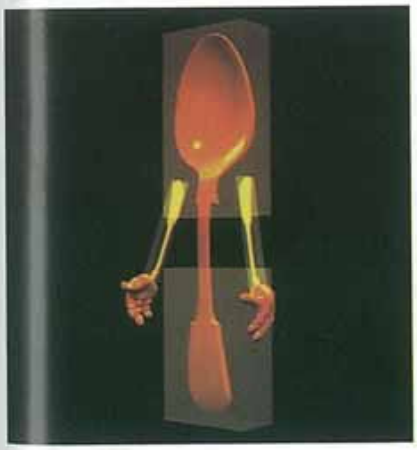


Figure 1.5
Global transformations on a polygon mesh model – a corrugated cylinder, twisted and tapered. (Courtesy of Steve Maddock.)



(a)



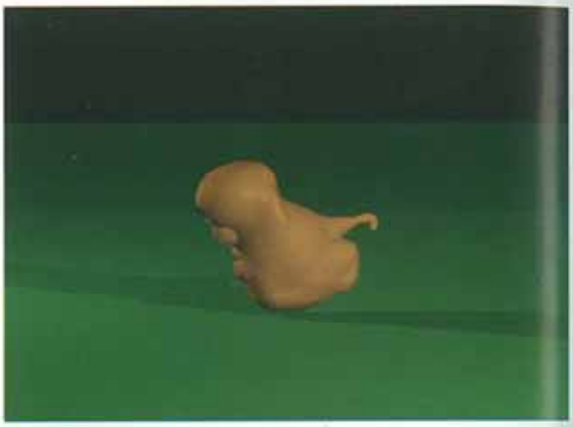
(b)

(c)

Figure 3.42
FFD applied to a polygon mesh object. (a) Wireframe of the object. (b) The object rendered with the trivariate patch grid shown as semi-transparent grey boxes. (c) Moving the control points in the patch causes the object model to deform in an appropriate manner.

An object (based on a famous Salvador Dali painting)

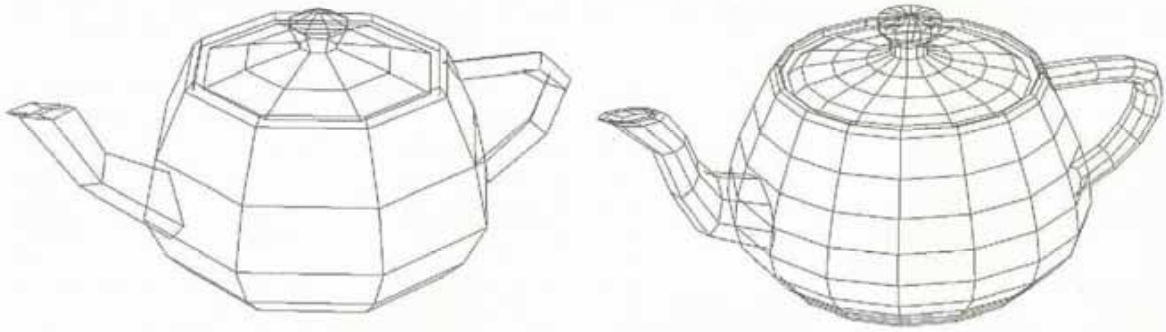
Point generators – the radius of each sphere is the influence of each generator



Unwanted blending as the generators are moved

Unwanted separation as the generators are moved

Figure 2.20
An example of an implicit function modelling system. (Courtesy of Agata Opalach.)



Polygons for the 128 and 512 rendered images.

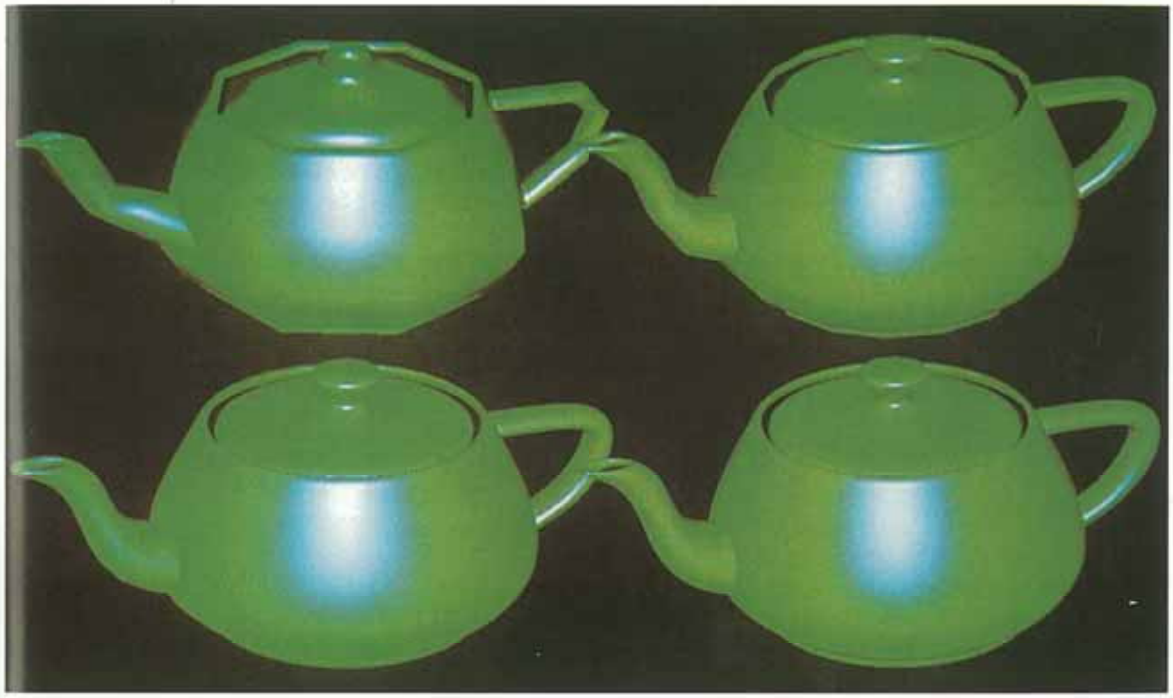


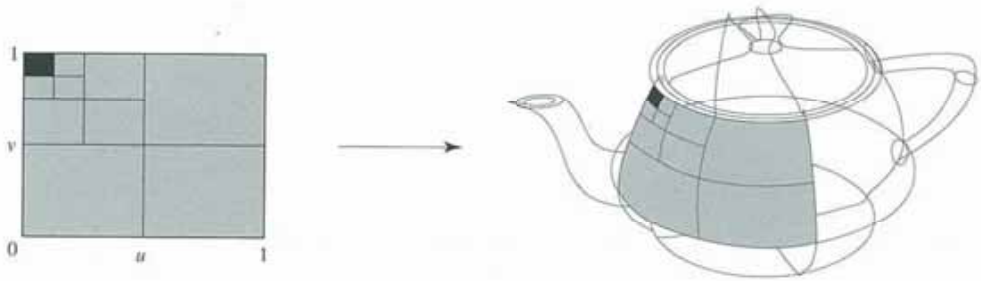
Figure 4.9
Parametric patch rendering at different levels of uniform subdivision (128, 512, 2048 and 8192 polygons). (Courtesy of Steve Maddock.)

Figure 6.12
The 'traditional' way of illustrating Phong shading. k_s and k_d are constant throughout. k_s is increasing from left to right and the exponent is increasing from top to bottom. The model attempts to increase 'shininess' by increasing the exponent. This makes the extent of the specular highlight smaller which could also be interpreted as the reflection of a light source of varying size. (The light is a point source.)

Figure 7.8
A selection of materials simulated using the model described in Section 7.6. The differences between some of the materials (for example, polished brass and gold) would be difficult to obtain by fine-tuning the parameters in Phong shading. In these images the reflection model was used as the local component in a ray tracer.

(a) (b) (c)

Figure 8.7
Examples of two-part texture mapping with a solid of revolution. The intermediate surfaces are: (a) a plane (or no surface); (b) a cylinder; and (c) a sphere.



Texture map

1 Bézier patch

Figure 8.8 (Left) Texture map. (Right) One Bézier patch on the object. (Below) Recursive teapot. (Courtesy of Steve Maddock.)

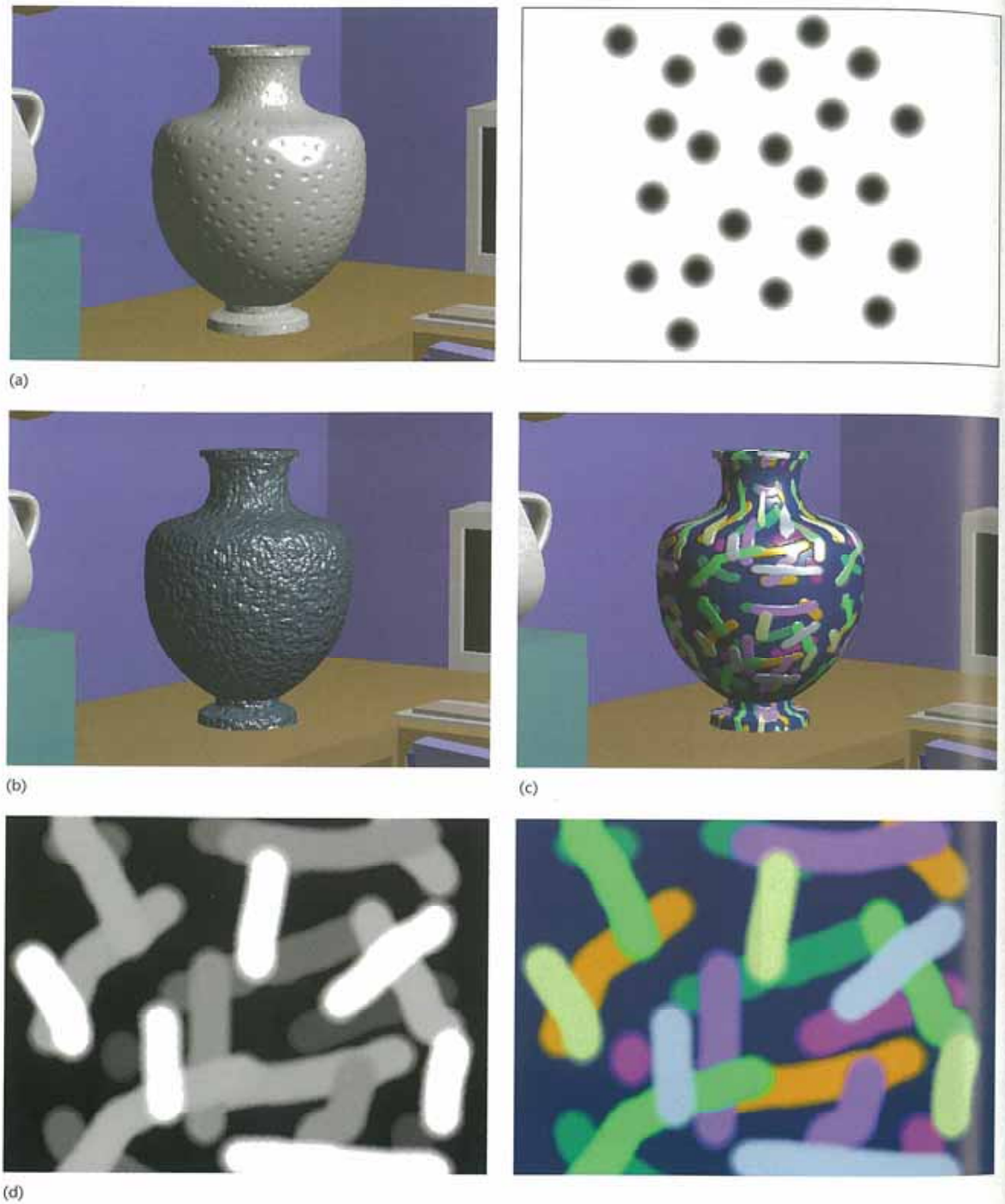
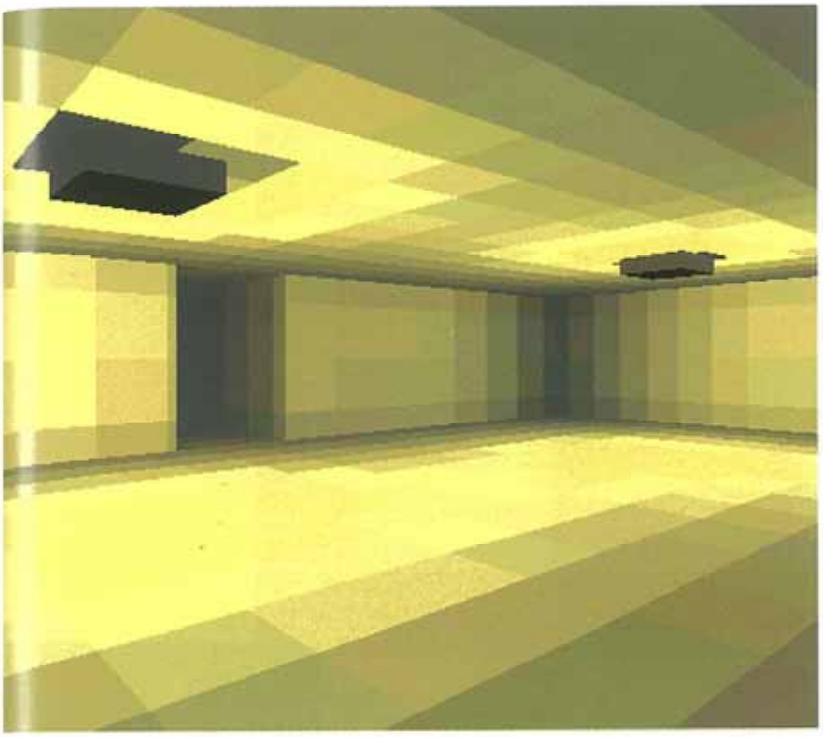


Figure 8.10
Bump mapping. (a) A bump mapped object together with the bump map. (b) A bump mapped object from a procedurally generated height field. (c) Combining bump and colour mapping. (d) The bump and colour map for (c).



(a)

(b)

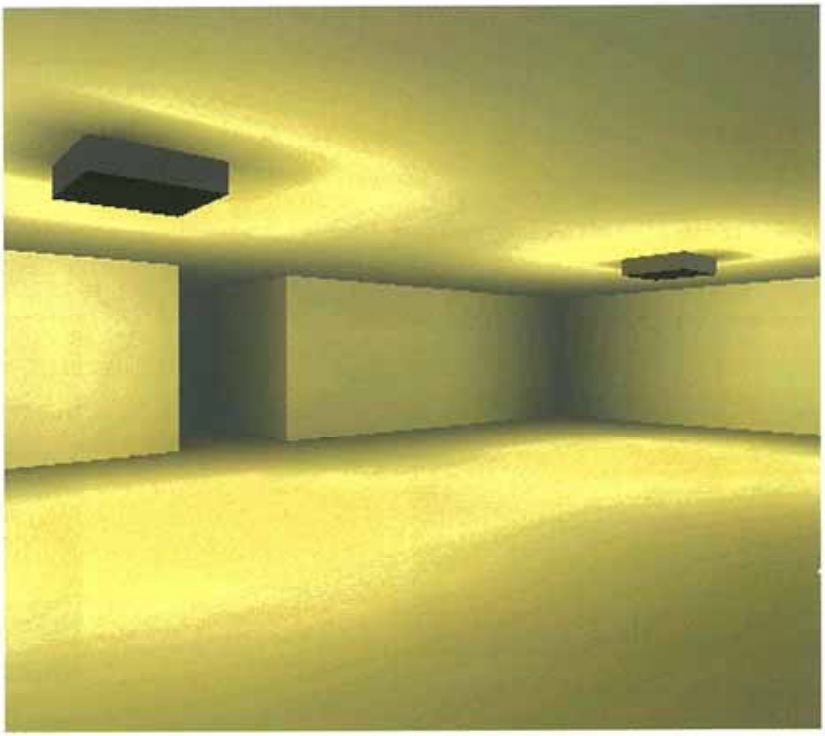


Figure 8.14
A simple scene lit using light maps. (a) In this image the size of the lumels in the scene is shown. (b) In this image a bilinear interpolation technique – known as texture interpolation – has been used to diminish the visibility of the lumels in (a).

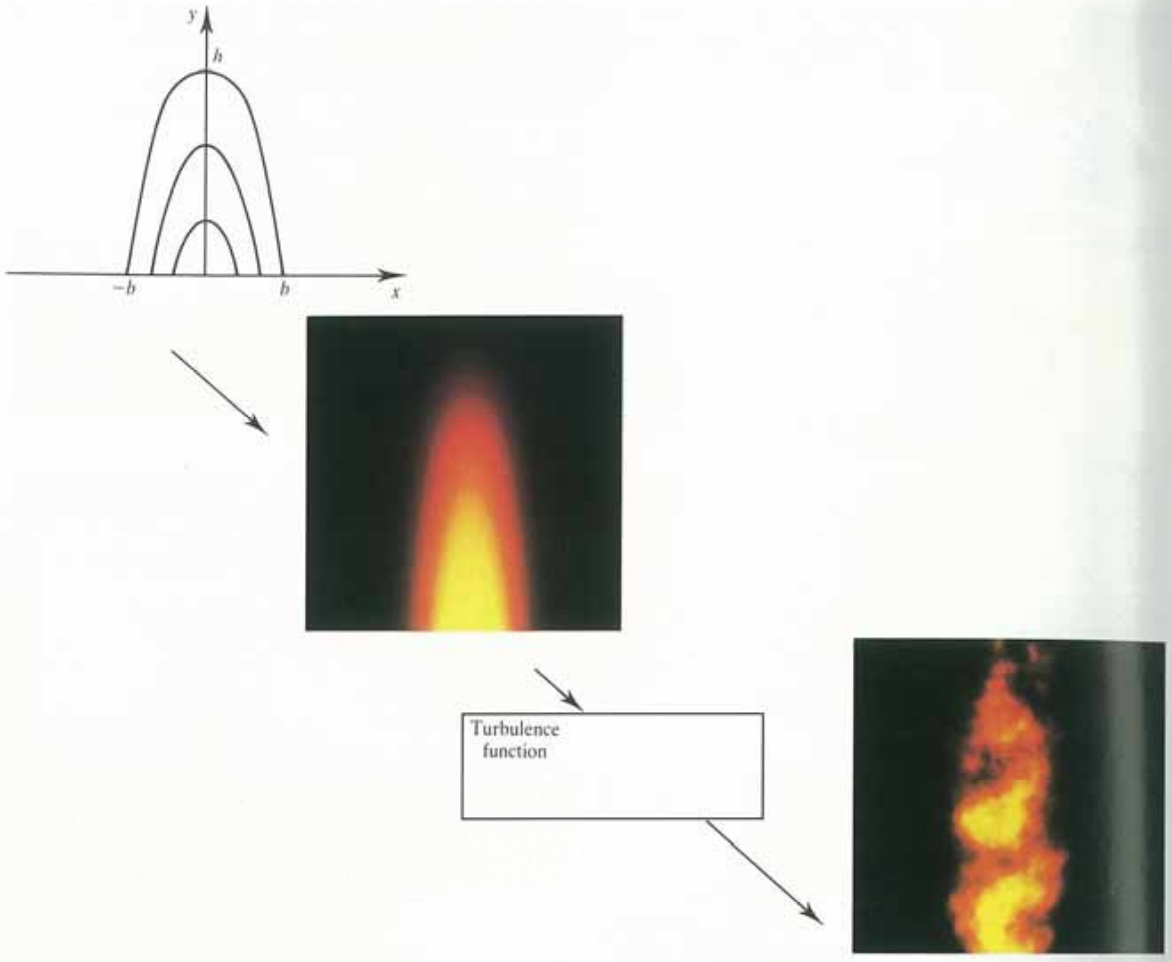


Figure 8.22 Modelling and simulating flame using a turbulence function. (Above) Unturbulated flame. (Right) Turbulated flame.

Figure 8.21 Imitating marble – the classic example of three-dimensional procedural texture.

Figure 8.26 Mip-map used in Figure 8.8. (Courtesy of Steve Maddock.)

Figure 10.4 (a)
An image generated using RADIANCE.

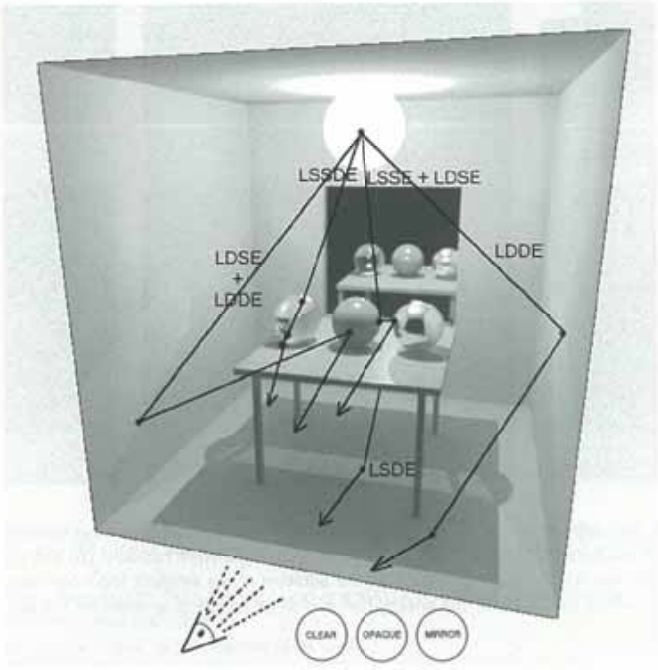


Figure 10.4 (b)
A selection of global illuminations paths in (a).

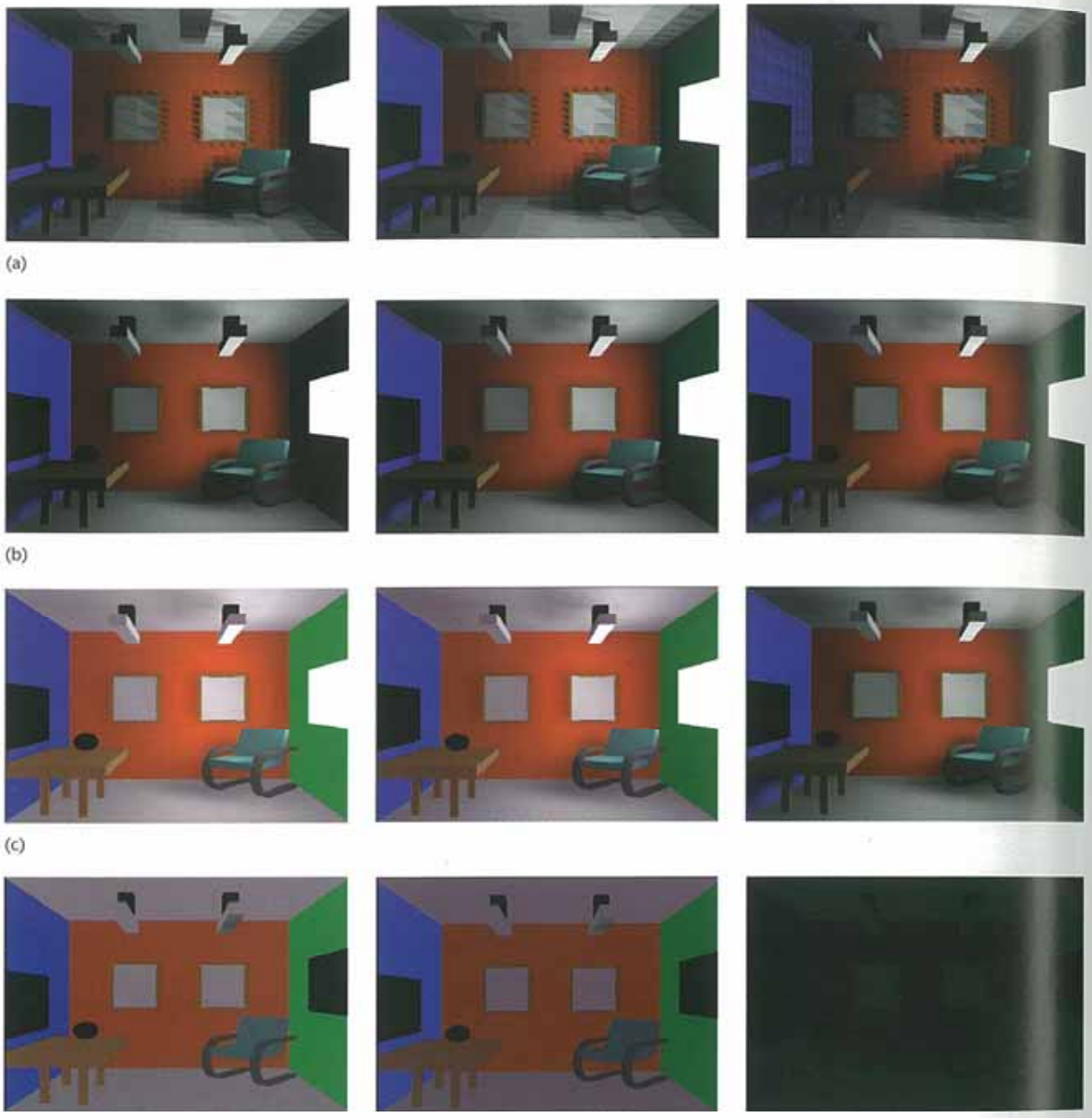


Figure 10.7
A radiosity image after 20, 250, and 5000 iterations of the progressive refinement method. From top to bottom for each column: (a) The radiosity solution as output from the iteration process. Each patch is allocated a constant radiosity. (b) The previous solution after it has been subjected to the interpolation process. (c) The same solution with the addition of the ambient term. (d) The difference between the previous two images. This gives a visual indication of the energy that had to be added to account for the unshot radiosity.

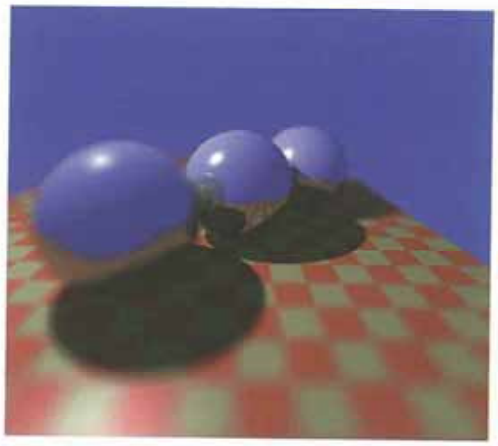


Figure 10.14
Depth of field effect rendered using a distributed ray tracer.

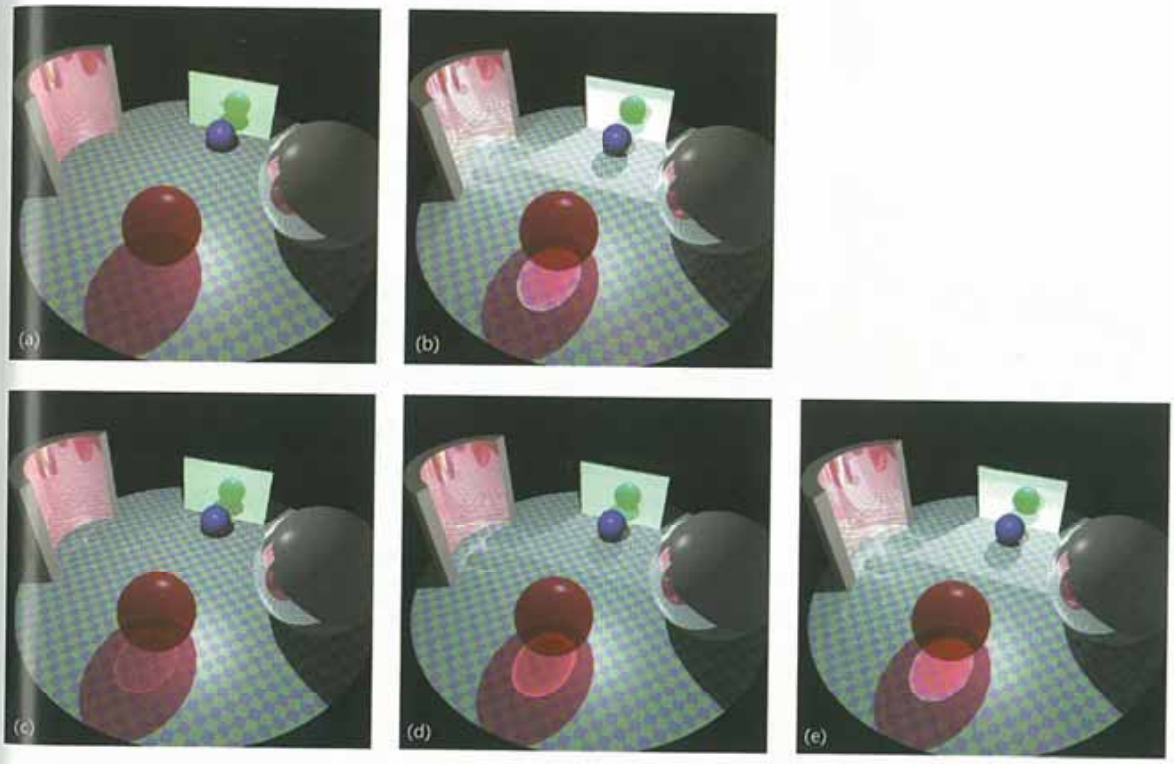
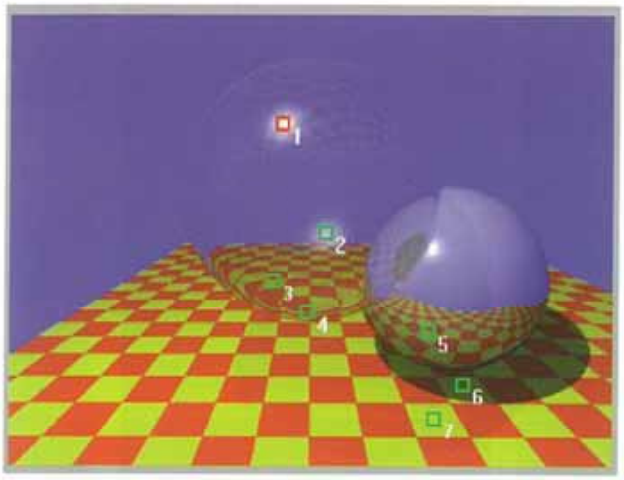


Figure 10.20
Two pass ray tracing example. (a) and (b) show a scene rendered using both a Whitted and two-pass ray tracer. In this scene there are three LSD paths:

- two caustics from the red sphere – one directly from the light and one from the light reflected from the curved mirror
- one (cusp) reflected caustic from the cylindrical mirror
- secondary illumination from the planar mirror (a non-caustic LSDE path).

(c) to (e) were produced by shooting an increasing number of light rays and show the effect of the light sprinkled on the diffuse surface. As the number of rays in the light pass increases, the rays can eventually be merged to form well defined LSD paths in the image. The number of rays shot in the light pass was 200, 400, and 800 respectively.

Figure 11.5
Shows the state of a hemicube placed on the window after all other patches in the scene have been projected onto it. A colour identifies each patch in the scene (and every partial patch) that can be seen by this hemicube. The algorithm then simply sums all the hemicube element form factors associated with each patch. (The scene for this figure is shown in Figure 10.7.)



The Whitted scene simple recursive ray tracing.

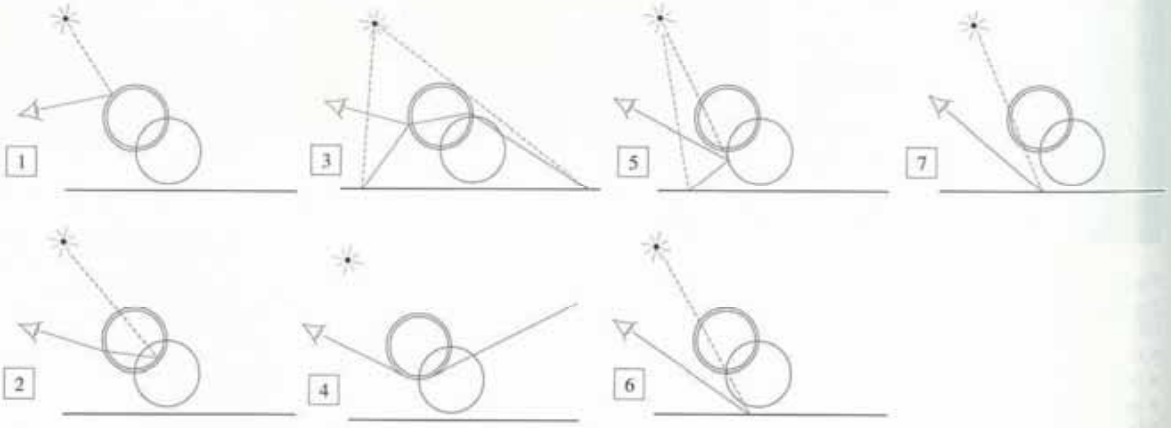


Figure 12.4
The Whitted scene.

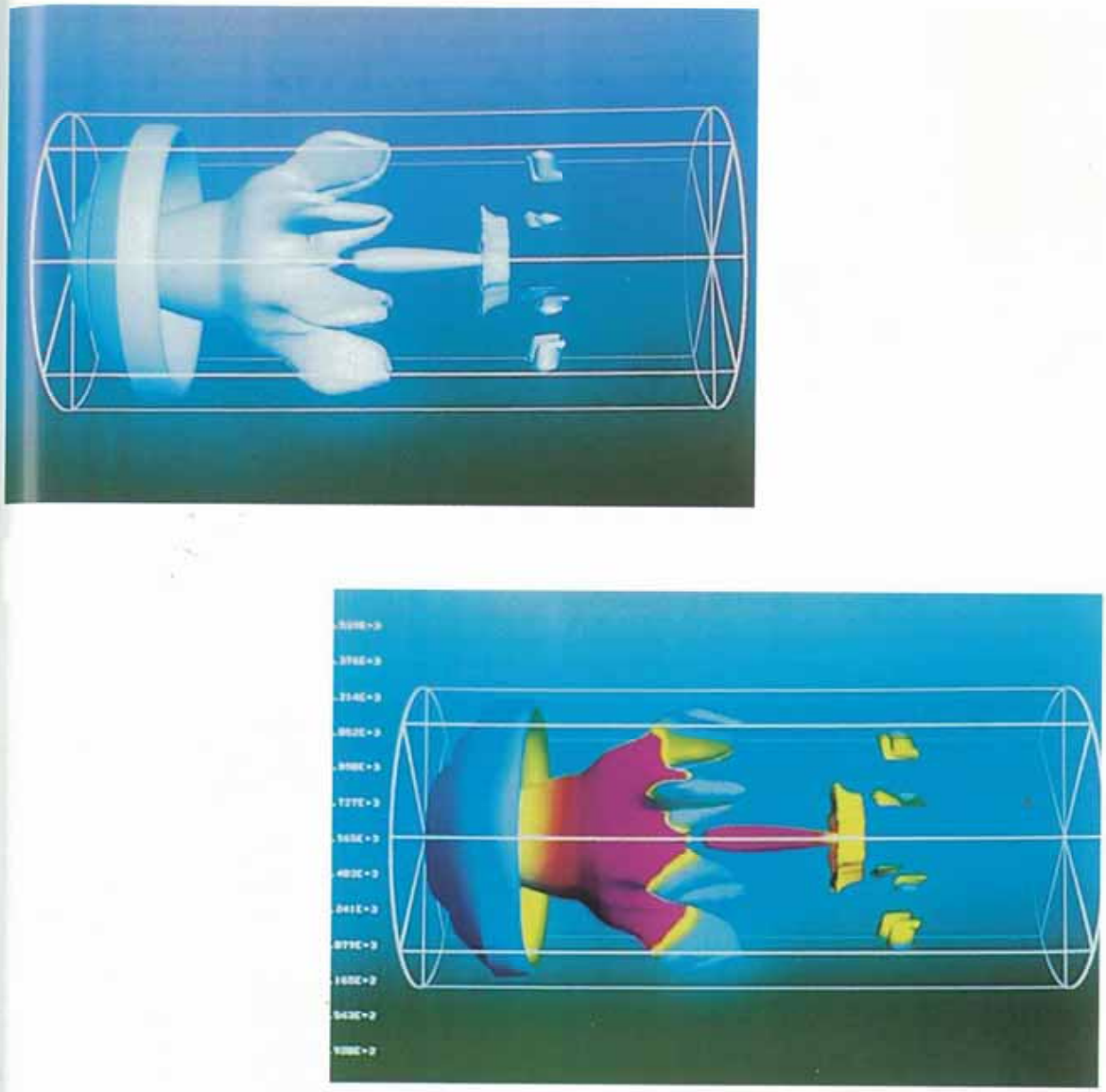


Figure 13.1
Marching cubes and CFD data. (top) A Navier–Stokes CFD simulation of a reverse flow pipe combuster. Flow occurs from left to right and from right to left. The interface between these flows defines a zero velocity isosurface. The marching cubes algorithm is used to extract this surface which is then conventionally rendered. (bottom) A texture-mapped zero velocity surface. A pseudo-colour scale that represents field temperature is combined with the colour used for shading in the illustration above. (Courtesy of Mark Fuller.)

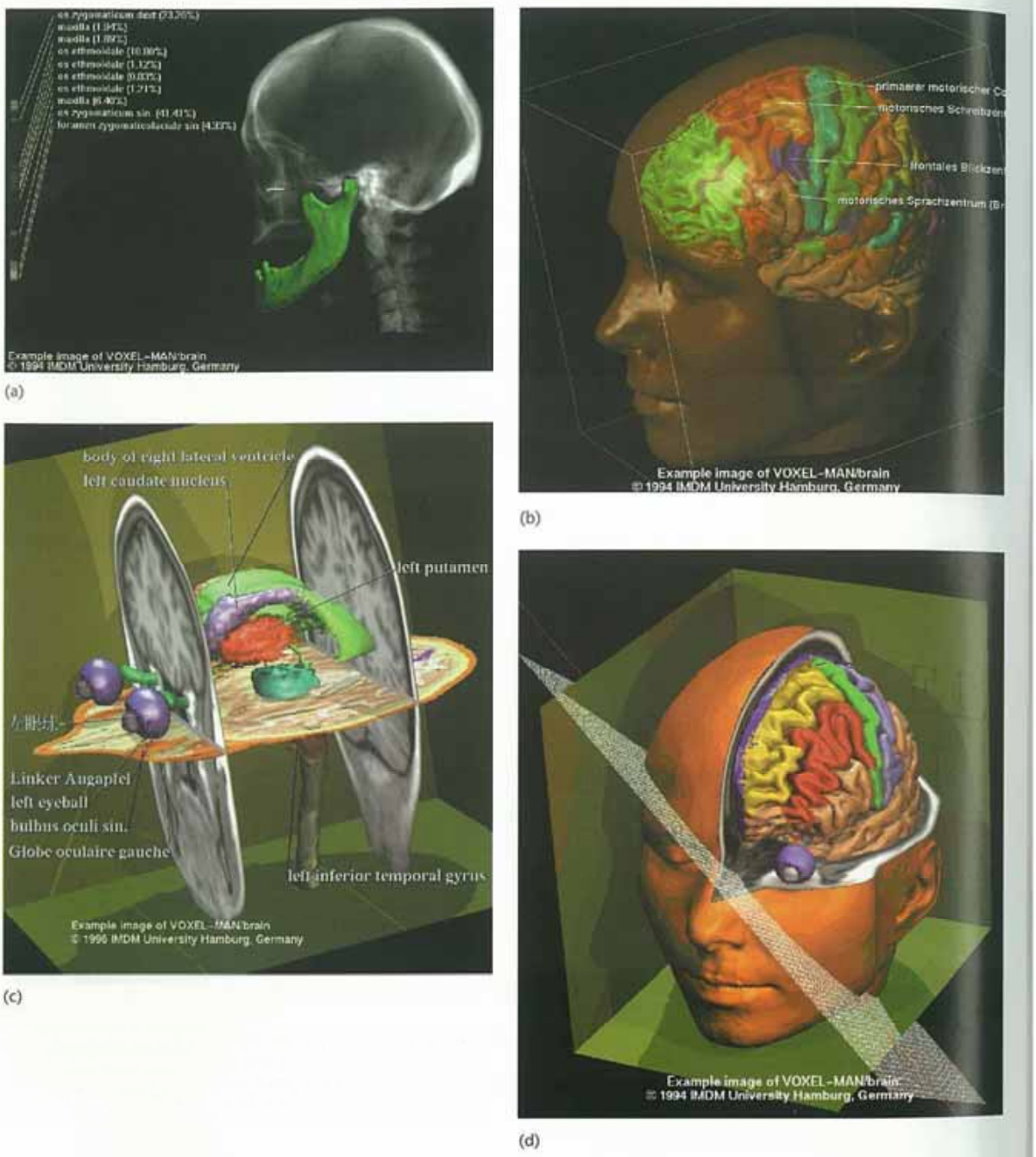


Figure 13.3
(a) and (b) show extracted objects embedded in a transparent surround of the skull. The extracted structures have been turned into computer graphics objects and rendered normally. They are then effectively re-embedded in the three-dimensional data volume which is displayed with the surrounding voxels set to some semi-transparent value. (c) and (d) are examples of cutting away a rendered version of the skin to show internal organs as a cross-section positioned within a three-dimensional model. Here the organs are assigned an appropriate pseudo-colour simply to highlight their shape. (Courtesy IMDM University, Hamburg.)

Figure 13.10
The marching cubes algorithm applied to X-ray CT data. (Courtesy of Klaus de Geuss.)

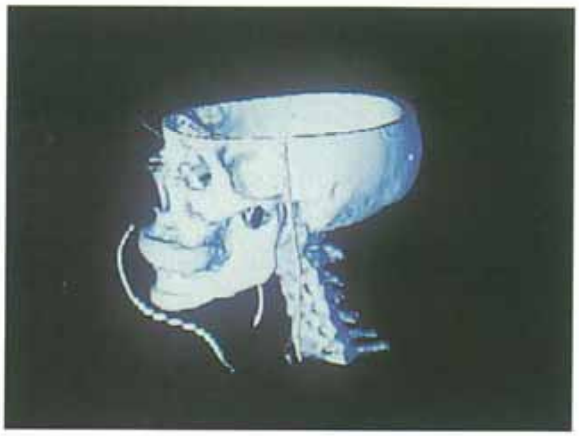
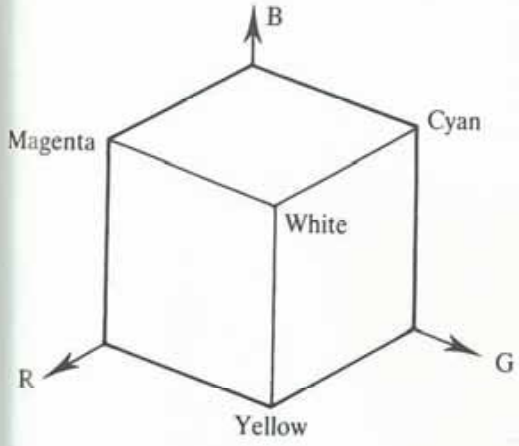
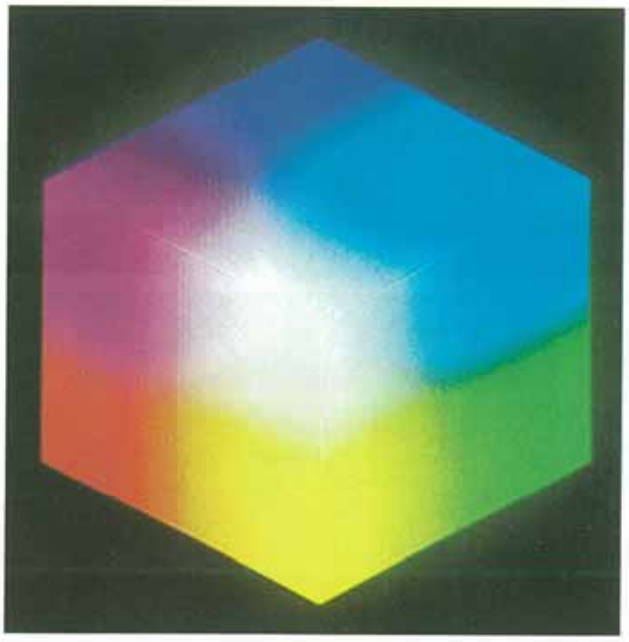


Figure 13.11
The same data using volume rendering with the bone voxels set to unity opacity and others set to zero. (Courtesy of Klaus de Geuss.)

Figure 15.3
The RGB cube.





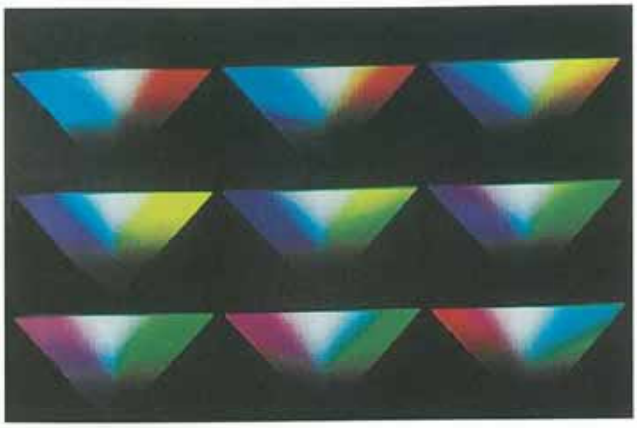


Figure 15.5
HSV colour model: slices through the value axis at 20° intervals.

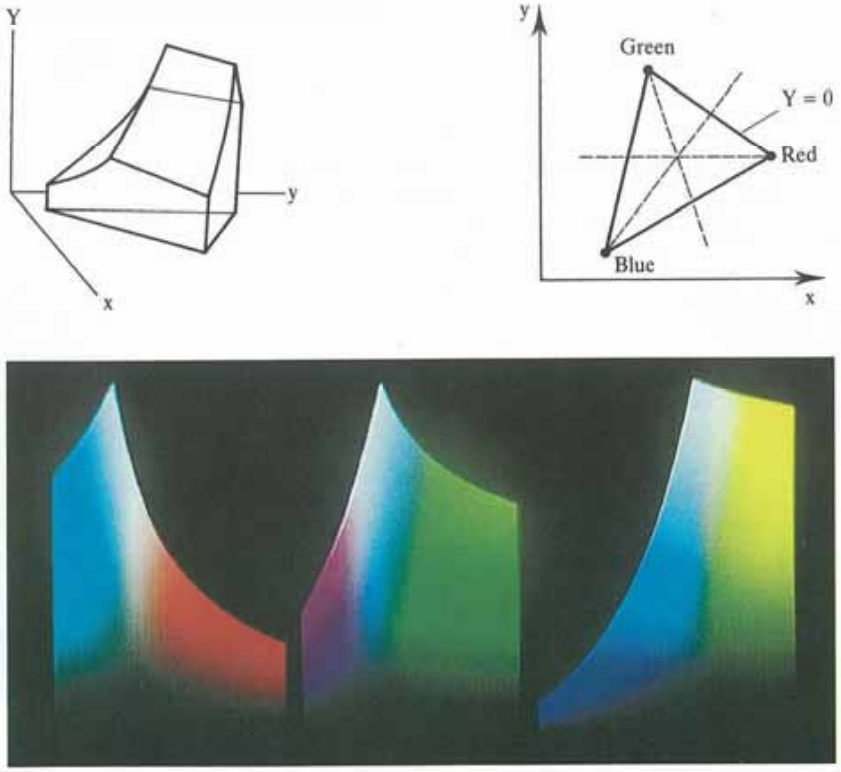


Figure 15.10
(Top left) Monitor gamut solid in CIE xyY space; (above) three cross-sections through the solid CIE xyY space; (top right) the position of the cross-sections on the plane Y=0.

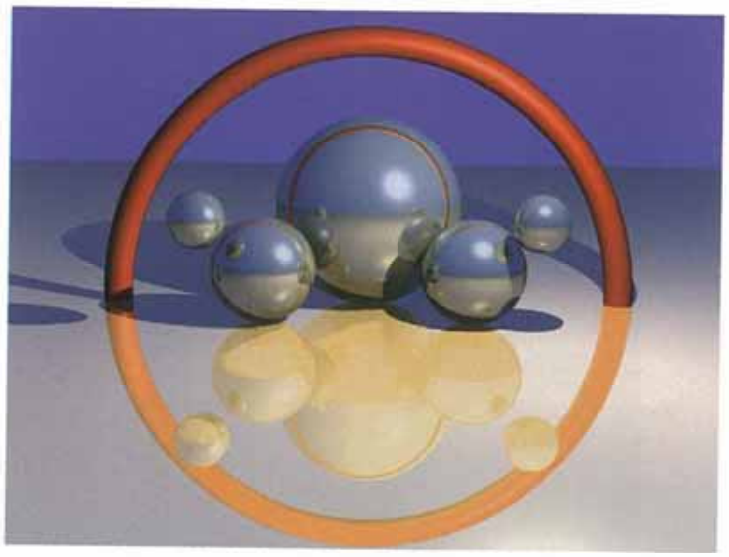


Figure 15.12
Rendering in spectral space compared with RGB space for a ray traced image.

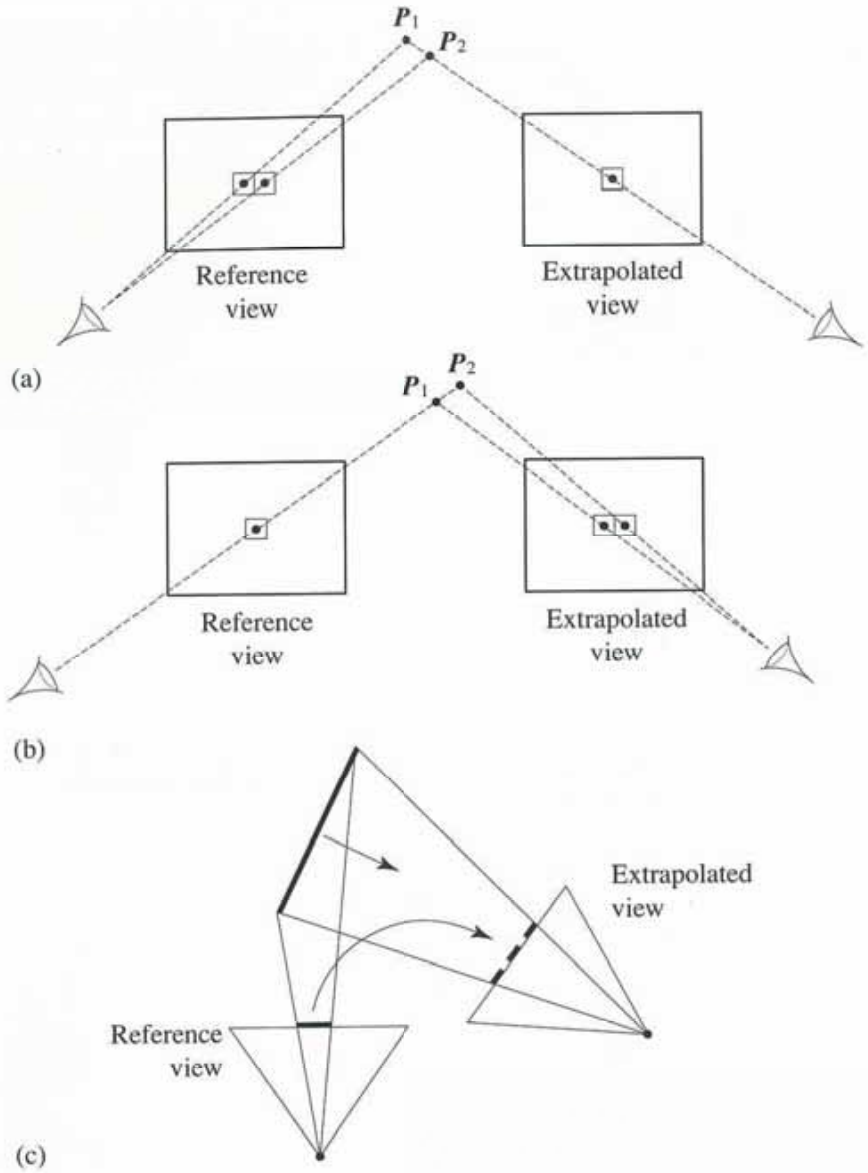
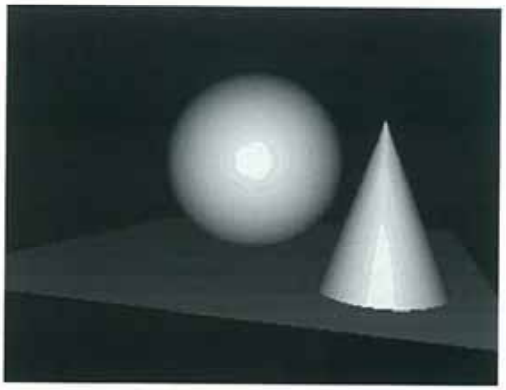
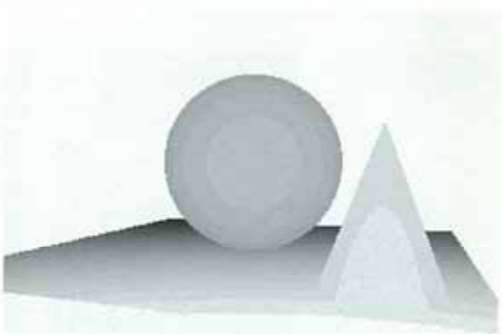


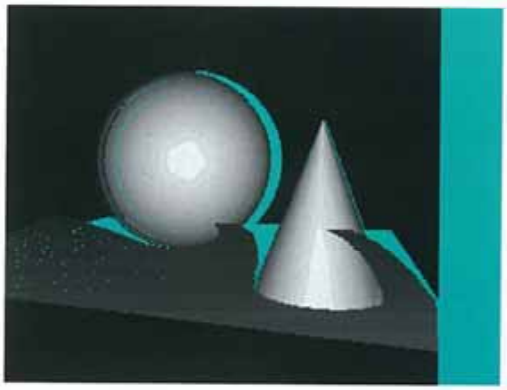
Figure 16.8
Problems in image warping. (a) Image folding: more than one pixel in the reference view maps into a single pixel in the extrapolated view. (b) Holes: information occluded in the reference view is required in the extrapolated view. (c) Holes: the projected area of a surface increases in the extrapolated view because its normal rotates towards the viewing direction. (d) See opposite page.



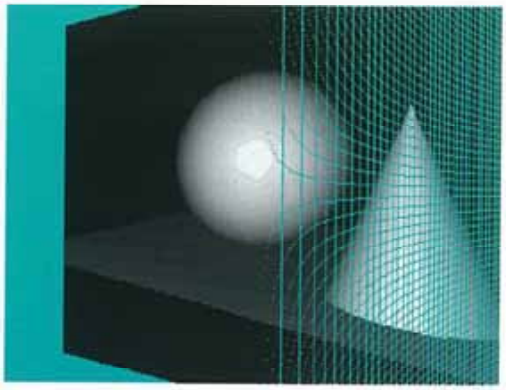
(d1)



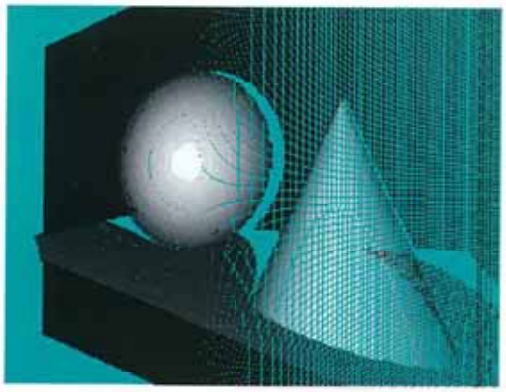
(d2)



(d3)



(d4)



(d5)

Figure 16.8 (d1 and d2)
A simple scene and the corresponding Z-buffer image.

Figure 16.8 (d3)
Artefacts due to translation (only) in this case are holes (cyan) caused by missing information and image folding.

Figure 16.8 (d4)
Artefacts due to rotation (only) are holes caused by increasing the projected area of surfaces. Note how these form coherent patterns.

Figure 16.8 (d5)
Artefacts caused by both rotation and translation.

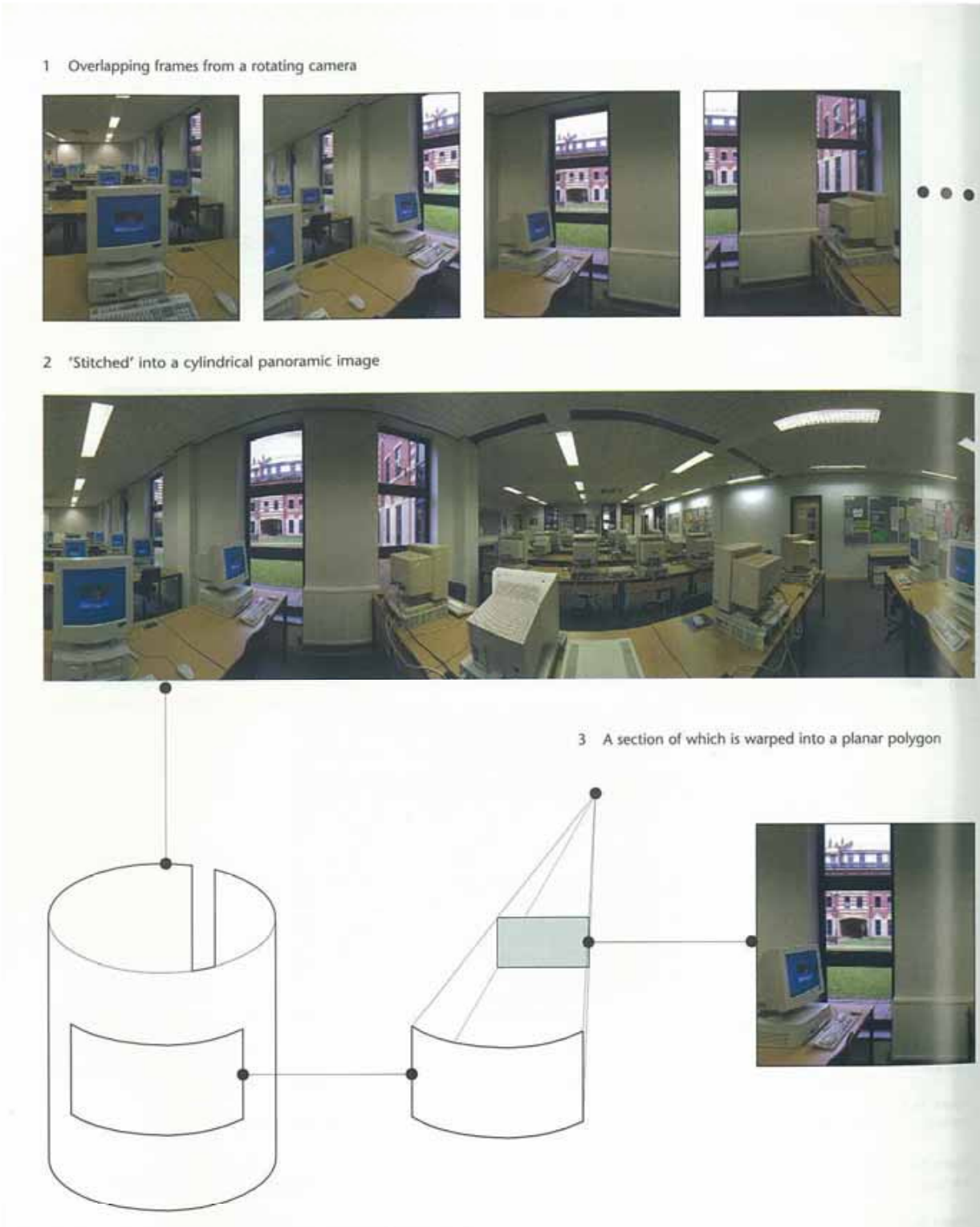


Figure 16.19 QuickTime® VR system. (Courtesy of Guy Brown.)

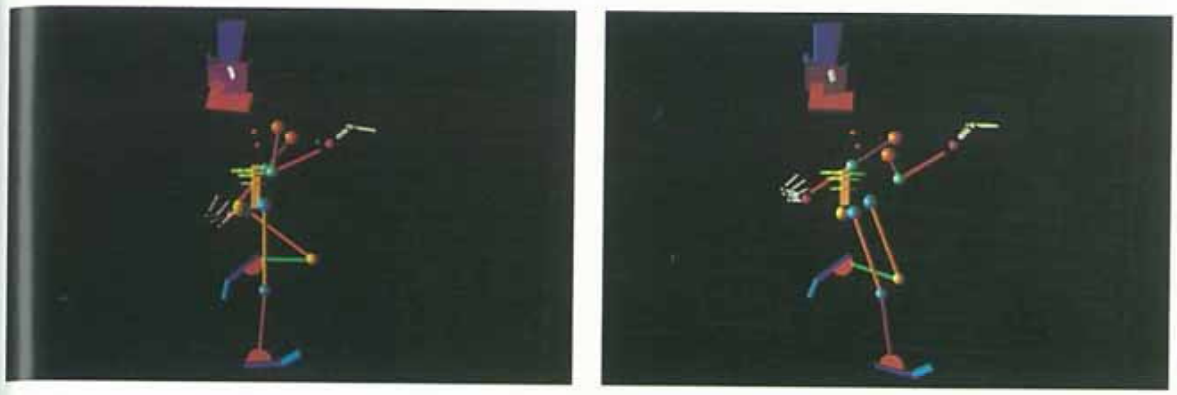


Figure 17.15

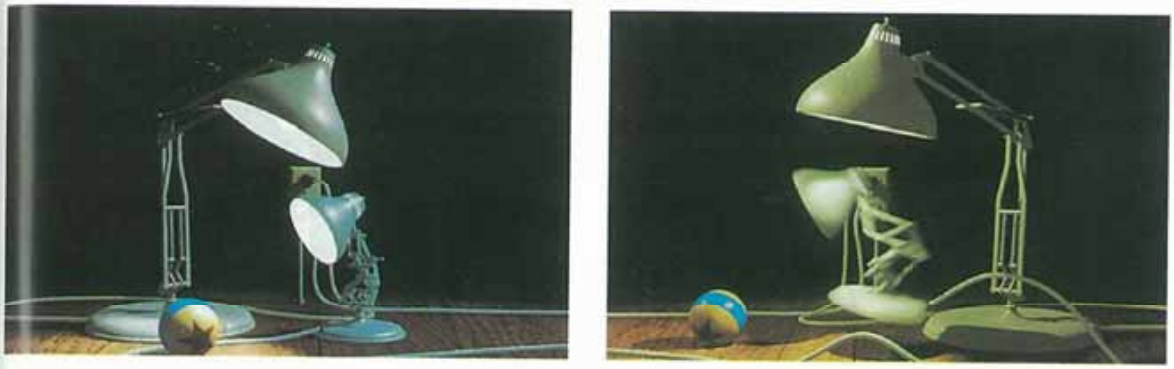
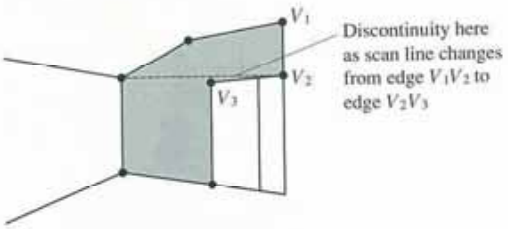


Figure 17.20
(Left) A frame from *Luxo Jr.* produced by John Lasseter, Bill Reeves, Eben Ostby and Sam Leffler; 1986 Pixar; Luxo is a trademark of Jak Jacobson Industries. The film was animated by a keyframe animation system with procedural animation assistance, and frames were rendered with multiple light sources and procedural texturing techniques. (Right) This frame from *Luxo Jr.* exhibits motion blur as described in Chapter 10.

Figure 18.1
An office scene, together with a wireframe visualization, that has been shaded using the constant ambient term only.

Figure 18.2
The same scene using flat shading. Flat shading shows the polygonal nature of the surfaces due to discontinuities in intensity.

(a)



(b)

(c)

Figure 18.3
Main defects in Gouraud interpolation. (a) Colour image. The two defects in this image (described in detail in the text) are: Mach banding (may not be visible in the reproduction) and the interpolation artefact on the back wall. (b). Dotted line shows the position of the discontinuity. (c) New wireframe triangulation necessary to eliminate the interpolation artefact.

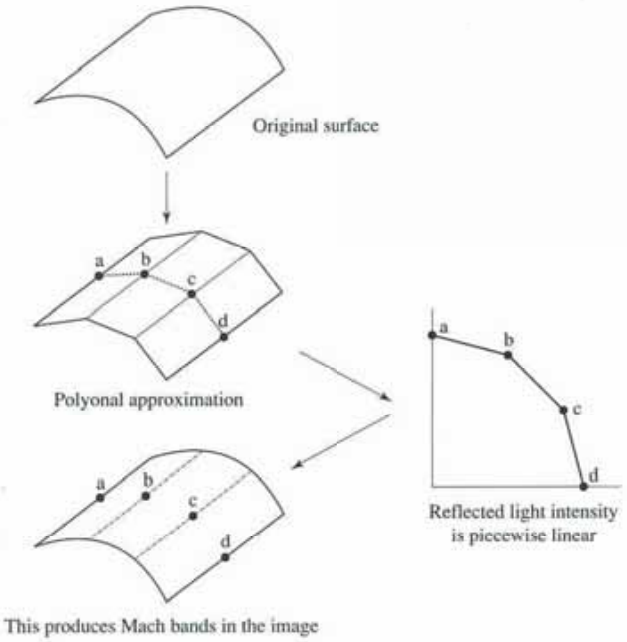


Figure 18.4
Mach bands in Gouraud shading.

Figure **18.5**
The same scene using Phong shading. A glaring defect in Phong interpolation is demonstrated in this figure. Here the reflected light from the wall light and the image of the light have become separated due to the nature of the interpolation.