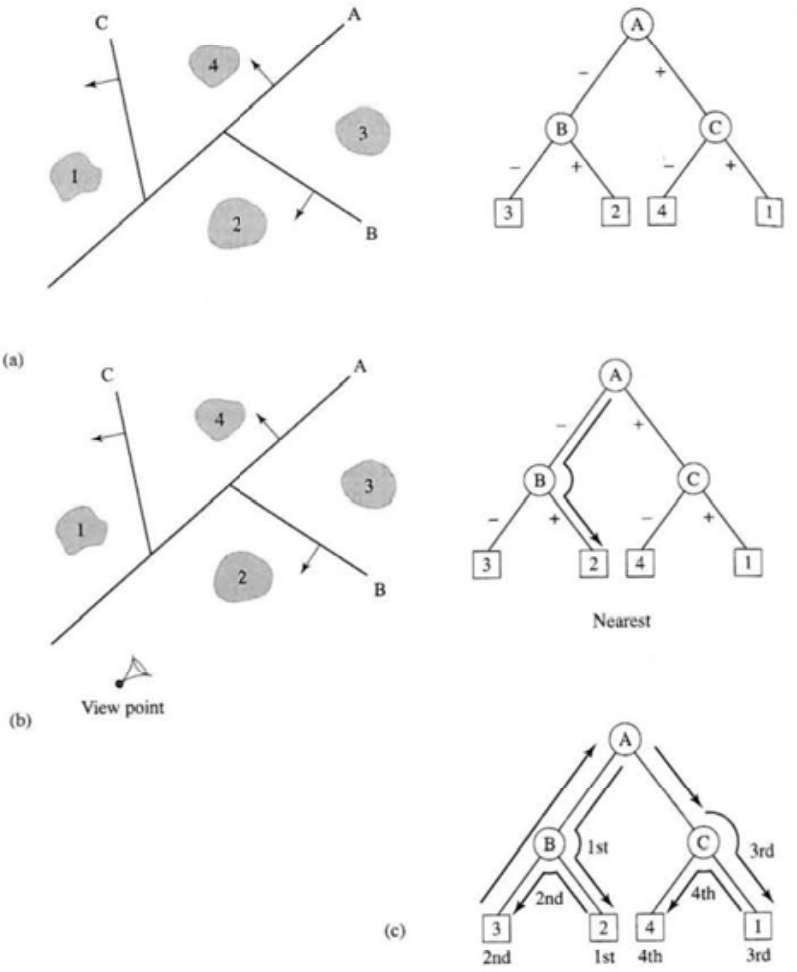


Figure 6.23
BSP operations for a four object scene. (a) Constructing a BSP tree. (b) Descending the tree with the view point coordinates gives the nearest object. (c) Evaluating a visibility order for all objects.

(number of polygons per object) is much greater than scene complexity (number of objects per scene) and for the approach to be useful we have to deal with polygons within objects rather than entire objects. Also there is the problem of positioning the planes – itself a non-trivial problem. If the number of objects is small then we can have a separating plane for every pair of objects – a total of n^2 for an n object scene.

For polygon visibility ordering we can choose planes that contain the face polygons. A polygon is selected and used as a root node. All other polygons are tested against the plane containing this polygon and placed on the appropriate descendant branch. Any polygon which crosses the plane of the root polygon is

split into two constituents. The process continues recursively until all polygons are contained by a plane. Obviously the procedure creates more polygons than were originally in the scene but practice has shown that this is usually less than a factor of two.

The process is shown for a simple example in Figure 6.24. The first plane chosen, plane A, containing a polygon from object 1, splits object 3 into two parts. The tree builds up as before and we now use the convention IN/OUT to say which side of a partition an entity lies since this now has meaning with respect to the polygonal objects.

Far to near ordering was the original scheme used with BSP trees. Rendering polygons into the frame buffer in this order results in the so-called painter's algorithm – near polygons are written 'on top of' farther ones. Near to far ordering can also be used but in this case we have to mark in some way the fact that a pixel has already been visited. Near to far ordering can be advantageous in extremely complicated scenes if some strategy is adopted to avoid rendering completely occluded surfaces, for example, by comparing their image plane extents with the (already rendered) projections of nearer surfaces.

Thus to generate a visibility order for a scene we:

- Descend the tree with view point coordinates.
- At each node, we determine whether the view point is in front of or behind the node plane.
- Descend the far side subtree first and output polygons.
- Descend the near side subtree and output polygons.

This results in a back to front ordering for the polygons with respect to the current view point position and these are rendered into the frame buffer in this order. If this procedure is used then the algorithm suffers from the same efficiency disadvantage as the Z-buffer – rendered polygons may be subsequently obscured. However, one of the disadvantages of the Z-buffer is immediately overcome. Polygon ordering allows the unlimited use of transparency with no additional effort. Transparent polygons are simply composited according to their transparency value.

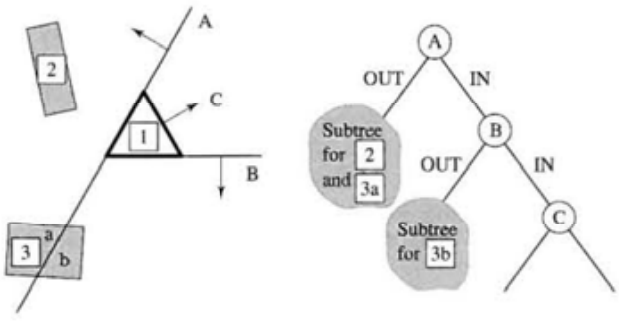


Figure 6.24
A BSP tree for polygons.

6.7 Multi-pass rendering and accumulation buffers

The rendering strategies that we have outlined have all been single pass approaches where a rendered image is composed by one pass through a graphics pipeline. In Section 6.6.3 we looked at a facility that enabled certain operations on separately rendered images with αZ components to be combined. In this section we will look at multi-pass rendering which means composing a single image of a scene from a combination of images of that scene rendered by passing it through the pipeline with different values for the rendering parameters. This approach is possible due to the continuing expansion of hardware and memory dedicated to rendering, manifested in texture mapping hardware which has substantially increased the visual complexity of real time imagery generated on a PC, and the availability of multiple screen resolution buffers such as a stencil buffer and an accumulation buffer (as well as the frame buffer and Z-buffer). The accumulation buffer is a simplified version of the A-buffer and the availability of such a facility has led to an expansion of the algorithms that employ a multi-pass technique.

As the name implies, an accumulation buffer accumulates rendered images and the standard operations are addition and multiplication combined into an 'add with weight' operation. In practice an accumulation buffer may have higher precision than a screen buffer to diminish the effect of rounding errors. The use of an accumulator buffer enables the effect of particular single pass algorithms to be obtained by a number of passes. After the passes are complete the final result in the accumulation buffer is transferred into the screen buffer.

The easiest example is the common anti-aliasing algorithm (see Section 14.7 for full details of this approach) which is to generate a virtual image, at $n \times$ the resolution of the final image, then reduce this to the final image by using a filter. The same effect can be obtained by jittering the view port and generating n images and accumulating these with the appropriate weighting value which is a function of the jitter value. In Figure 6.25, to generate the four images that are required to sample each pixel four times we displace the view window through a $^1/_2$ pixel distance horizontally and vertically. To find this displacement we only have to calculate the size of the view port in pixel units. (Note that this cannot be implemented using the simple viewing system given in Chapter 5, which assumes that the view window is always centred on the line through the view point.)

In this case we only save on memory. However, in many instances an algorithm implemented as a multi-pass rendering is of lower complexity than the single pass equivalent. Additional examples of motion blur, soft shadows and depth of field are given in Haeberli and Akeley (1990). These effects can be achieved by distributed ray tracing as described in Chapter 10 and the marked difference between the complexity of the two approaches is obvious.

To create a motion blurred image it is only necessary to accumulate a series of images rendered while the moving objects in the scene change their position over

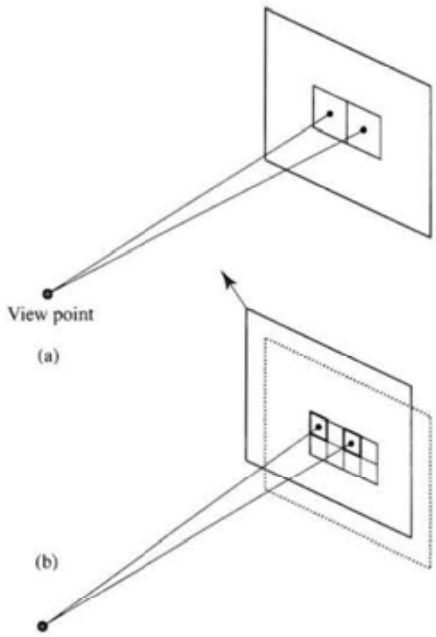


Figure 6.25
Multi-pass super-sampling. (a) Aliased image (1 sample/pixel). (b) A one component/pass of the anti-aliased image (four samples/pixel or four passes). For this pass the view point is moved up and to the left by $^1/_2$ pixel dimension).

time. Exactly analogous to the anti-aliasing example, we are now anti-aliasing in the time domain. There are two approaches to motion blur. We can display a single image by averaging n images built up in the accumulation buffer. Alternatively we can display an image for every calculated image by averaging over a window of n frames moving in time. To do this we accumulate n images initially. At the next step the frame that was accumulated n–1 frames ago is re-rendered and subtracted from the accumulation buffer. Then the contents of the accumulation buffer are displayed. Thus, after the initial sequence is generated, each time a frame is displayed two frames have to be rendered – the $(n - 1)$th and the current one.

Simulating depth of field is achieved (approximately) by jittering both the view window as was done for anti-aliasing and the view point. Depth of field is the effect seen in a photograph where, depending on the lens and aperture setting, objects a certain distance from the camera are in focus where others nearer and farther away are out of focus and blurred. Jittering the view window makes all objects out of focus and jittering the view point at the same time ensures objects in the equivalent of the focal plane remain in focus. The idea is shown in Figure 6.26. A plane of perfect focus is decided on. View port jitter values and view point perturbations are chosen so that a common rectangle is maintained in the plane of perfect focus. The overall transformation applied to the view frustum is a shear and translation. Again this facility cannot be implemented using the simple view frustum in Section 5.2 which does not admit shear projections.

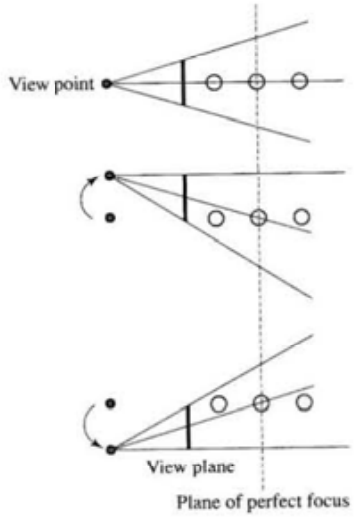


Figure 6.26
Simulating depth of field by shearing the view frustum and translating the view point.

Soft shadows are easily created by accumulating n passes and changing the position of a point light source between passes to simulate sampling of an area source. Clearly this approach will also enable shadows from separate light sources to be rendered.

7 Simulating light–object interaction: local reflection models

Introduction

Local reflection models, and in particular the Phong model (introduced in Chapter 5), have been part of mainstream rendering since the mid-1970s. Combined with interpolative shading of polygons, local reflection models are incorporated in almost every conventional renderer. The obvious constraint of locality is the strongest disadvantage of such models but despite the availability of ray tracers and radiosity renderers the mainstream rendering approach is still some variation of the strategy described earlier – in other words a local reflection model is at the heart of the process. However, nowadays it would be difficult to find a renderer that did not have ad hoc additions such as texture mapping and shadow calculation (see Chapters 8 and 9). Texture mapping adds interest and variety, and geometrical shadow calculations overcome the most significant drawback of local models.

Despite the understandable emphasis on the development of global models, there has been some considerable research effort into improving local reflection models. However, not too much attention has been paid to these, and most

renderers still use the Phong model: in one sense a tribute to the efficacy and simplicity of this technique, in another, an unfortunate ignoring of the real advances that have been made in this area.

An important point concerning local models is that they are used in certain global solutions. As will be discovered in Chapter 12, most simple ray tracers are hybrid models that combine a local reflection model with a global ray traced model. A local model is used at every point to evaluate a contribution that is due to any direct illumination that can be seen from that point. To this is added a (ray traced) component that accounts for indirect illumination. (In fact, this is inconsistent because different parameters are used for the local and global contribution, but it is a practice that is widely adopted.)

In this chapter we will look at a representative selection of local models, delving into such questions as: how do we simulate the different light reflection behaviour between, say, shiny plastic and metal that is the same colour? We can usually perceive such subtle differences in real objects and it is appropriate that we should be able to simulate them in computer graphics.

The foundation of most local reflection models involves an empirical or imitative approach in which we devise an easily evaluated function to imitate reflection of light from a surface or the theory of reflection from a perfect surface together with the simulation of an imperfect surface.

7.1 Reflection from a perfect surface

We begin by examining the behaviour of light incident on an optically smooth surface – a perfect mirror. This is determined by the Fresnel formulae – themselves derived from Maxwell's wave equations. This is the source of the ray tracing formulae given in Section 1.4.6. The formula is a coefficient that relates the ratio of reflected and transmitted energy as a function of incident direction, polarization and the properties of the material. Assuming for simplicity that the light is unpolarized (the approach usually taken in computer graphics) and travelling through air (approximated as a vacuum) and assuming that a factor known as the extinction coefficient (see Section 7.6.4) is zero we have:

$$F = \frac{1}{2}\left\{\frac{\sin^2(\phi-\theta)}{\sin^2(\phi+\theta)} + \frac{\tan^2(\phi-\theta)}{\tan^2(\phi+\theta)}\right\} \qquad [7.1]$$

where:

ϕ is the angle of incidence
θ is the angle of refraction
$\sin\theta = \sin\phi/\mu$ (where μ is the refractive index of the material)

These angles are shown in Figure 7.1. F is minimum, that is most light is absorbed when $\phi = 0$ or normal incidence. No light is absorbed by the surface and F is equal to unity for $\phi = \pi/2$. The wavelength dependent property of F comes from the fact that μ is a function of wavelength.

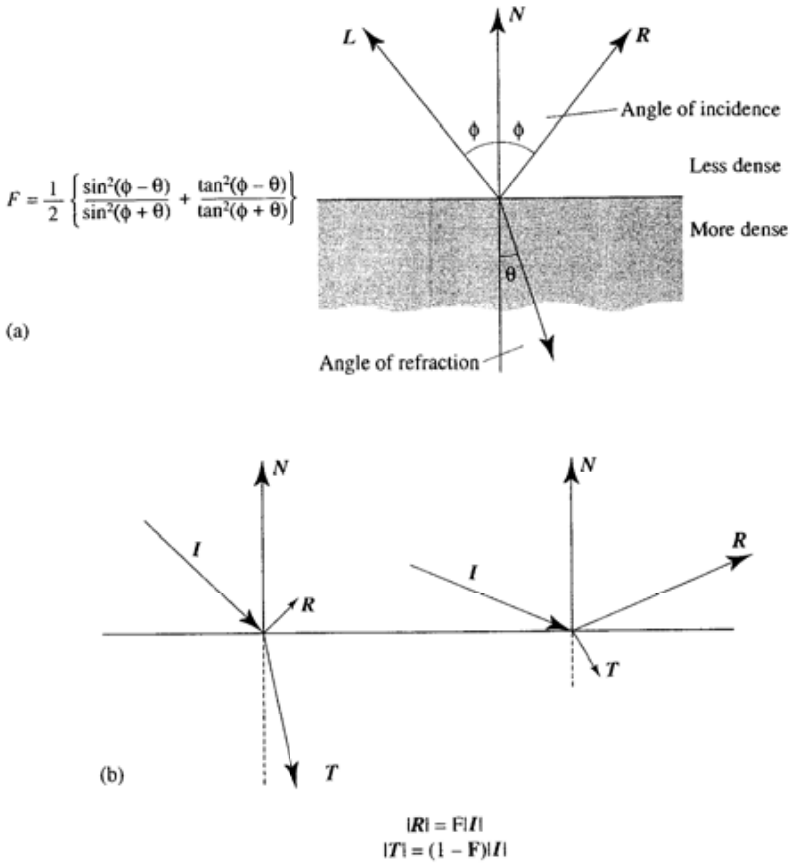


Figure 7.1
The Fresnel equation. (a) Angles in the Fresnel equation. (b) Two examples showing the behaviours of the equation.

7.2 Reflection from an imperfect surface

In practice, surfaces are not optically perfect. With the exception of glass or still water, surfaces exhibit a microgeometry. We can, however, still use the Fresnel equation if we incorporate it in a model that simulates the microgeometry. Figure 7.2 shows one way of doing this – we consider the surface to be a collection of microfacets which are for simplicity considered as symmetric V-shaped pits. Over a small region we can describe the reflection of light incident on a representative region of such grooves to form a lobe which we can parametrize.

Of course, surface microgeometry is not the only imperfection that occurs in reality. For example, shiny metal surfaces age and acquire a film of dirt as well as large imperfections like scratches. This kind of 'real' surface is much more difficult to model and it has to be emphasized that a surface whose microgeometry is modelled in the way described is still assumed to be perfectly clean – an unlikely practical event.

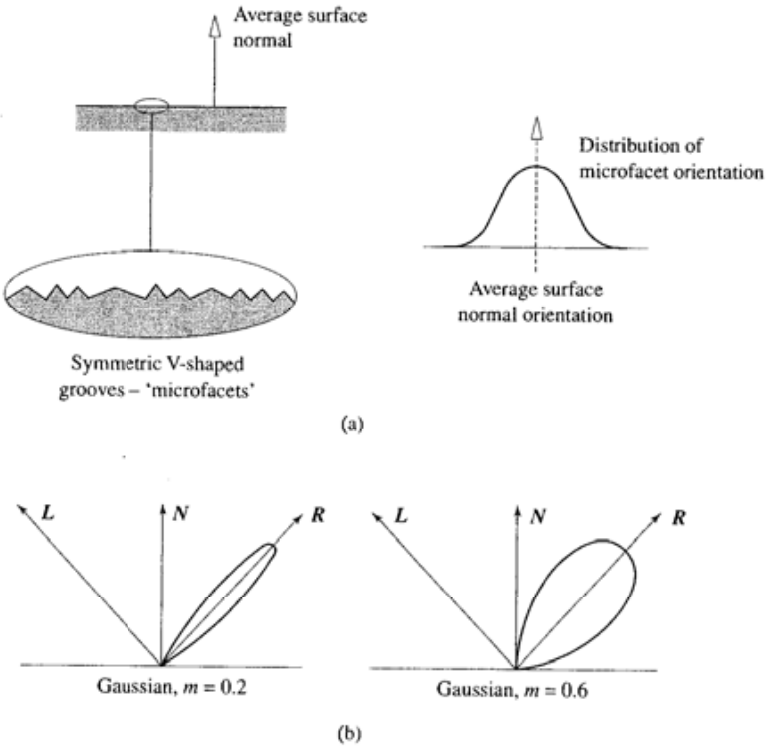


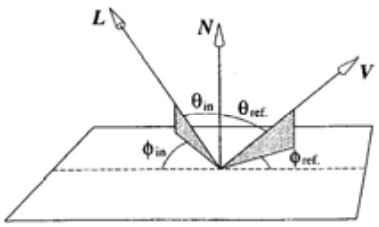
Figure 7.2
Simulating a rough surface with a collection of microfacets each considered as a perfect mirror. (a) Modelling a surface as a collection of V-shaped grooves. (b) Reflection lobes for different values of m in a Gaussian distribution.

7.3 The bi-directional reflectance distribution function

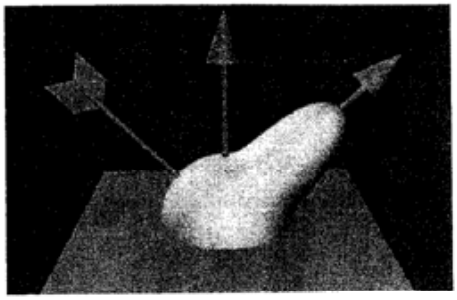
In general, light reflected from a point on the surface of an object is categorized by a bi-directional reflection distribution function, or BRDF. This term emphasizes that the light reflected in any particular direction (in computer graphics we are mostly interested in light reflected along the viewing direction $\boldsymbol{V}$) is a function not only of this direction but also of the direction of the incoming light. A BRDF can be written as:

$$\text{BRDF} = f(\theta_{in}, \phi_{in}, \theta_{ref}, \phi_{ref}) = f(\boldsymbol{L}, \boldsymbol{V})$$

and many models used in computer graphics differ amongst themselves according to which of these dependencies are simulated. Figure 7.3 shows these angles together with a BRDF computed for a particular set of angles. The rendered BRDF shows the magnitude of the reflected light (in any outgoing direction) for an infinitely thin beam of light incident in the direction shown. In practice, light may be incident on a surface point from more than one direction and the total reflected light would be obtained by considering a separate BRDF for each incoming light beam and integrating.



(a)



(b)

Figure 7.3
Bi-directional reflectivity function. (a) A BRDF relates light incident in direction ***L*** to light reflected along direction ***V*** as a function of the angles θ_{in}, ϕ_{in}, θ_{ref}, ϕ_{ref}. (b) An example of a BRDF.

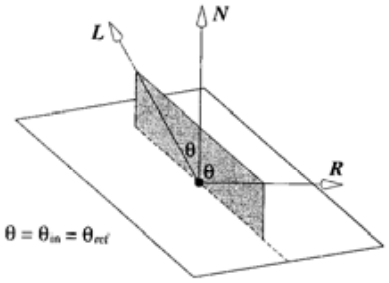
For many years computer graphics has worked with simple, highly constrained BRDFs such as that shown in Figure 7.3. Figure 7.4 gives an idea of the difference between such computer graphics models and what actually happens in practice. The illustrations are cross-sections of the BRDF in the plane containing ***L*** and ***R***, the mirror direction for different angles of θ, the angle of incidence (and reflection). In particular, note the great variation in the shape of the reflection lobes as a function of the wavelength of the incident light, the angle of incidence and the material. In the case of aluminium we see that it can behave either like a mirror surface or a directional diffuse surface depending on the wavelength of the incident light. When we also take into account that, in practice, incident light is never monochromatic (and we thus need a separate BRDF for each wavelength of light that we are considering) we see that the behaviour of reflected light is a far more complex phenomenon than we can model by using simple approximations like the Phong model at three wavelengths.

An important distinction that has to be made is between isotropic and anisotropic surfaces. An isotropic surface exhibits a BRDF whose shape is independent of the incoming azimuth angle ϕ_i (Figure 7.3). An anisotropic surface is, for example, brushed aluminium or a surface that retains coherent patterns from a milling machine. In the case of a brushed surface the magnitude of the specular lobe depends on whether the incoming light is aligned with the grain of the surface or not.

Another complication that occurs in reality is the nature of the atmosphere. Most BRDFs used in local reflection models are constrained to apply to light reflected from opaque materials in a vacuum. We mostly do not consider any scattering of reflected light in an atmosphere (in the same way that we do not consider light scattered by an atmosphere before it reaches the object). The reason for this is, of course, simplicity and the subsequent reduction of light intensity calculations to simple comparisons between vectors categorizing surface shapes, light directions and viewing direction.

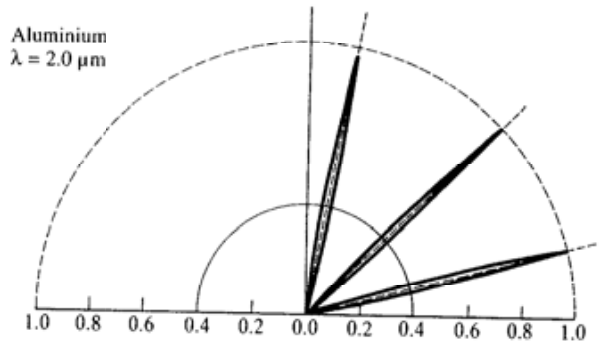
We might imagine that if we have a BRDF for a material that the light–object interaction is solved. However, a number of problems remain to this day despite a quarter of a century of research. Some of these are:

Figure 7.4
BRDF cross-section for different materials and wavelengths (after an illustration by He *et al.* (1991)).

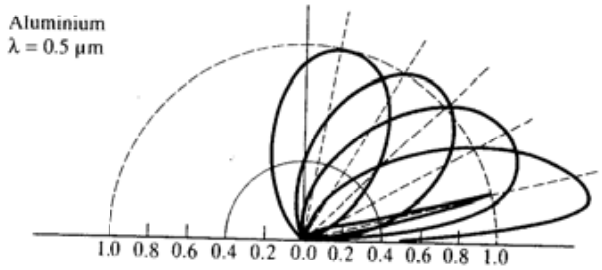


Plane containing L and R the mirror direction

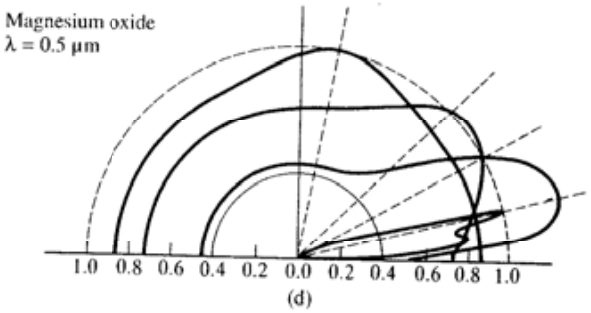
(a)



(b)



(c)



(d)

- Where do we get the BRDF from (particularly for real as opposed to perfect materials)? Some data are available for some materials in the metallurgical literature but this is by no means complete.
- At what scale do we attempt to represent a BRDF? What is the area of the region that we should consider receives incident light? Should this be large enough so that the statistical model is consistent over the surface? Should it be large enough to include surface imperfections such as scratches? This is very much an unaddressed problem.
- How do we represent the BRDF? This final point accounts for most variation amongst models. In particular, the distinction between empirical and physically based models is often made. An empirical model is one that imitates light–object interaction. For example, in the Phong model a simple mathematical function is used to represent the specular lobe. In the Cook and Torrance model a statistical ditribution represents the surface geometry and this is termed a physically based model (Cook and Torrance 1982). It is interesting to note that there is no general agreement on the visual efficacy of empirical versus physically based models. Often a better result can be obtained by carefully tuning the parameters of an empirical model than by using a physically based model.

What follows is a review of the early local reflection models and a short selection of more recent advances. In particular we start by looking at the defects inherent in the Phong model and how these can be overcome. The material is by no means a comprehensive review, but is intended as a representation of these departures from the Phong model that have been simulated to provide ever more subtle variations on the way in which light 'paints' an object.

7.4 Diffuse and specular components

Local reflection models used in computer graphics are normally considered as a combination of a diffuse and a specular component. This works well for many cases but it is a simplification. The simpler models of specular reflection consider some imperfect behaviour to be a modification of perfect specular reflection. Perfect specular reflection occurs when light strikes a perfect mirror surface and a thin beam of light incident on such a surface reflects according to the well-known law: the outgoing angle is equal to the angle of incidence. Perfect diffuse reflection occurs when incident light is scattered equally in all directions from a perfect matte surface, which could in practice be a very fine layer of powder. Combining separately calculated specular and diffuse components imitates the behaviour of real surfaces and is an enabling assumption in many computer graphics models. Imitating the subtle visual differences between real surfaces has mostly been achieved by incorporating various effects into the specular component as we shall now examine by looking at a selection of such models. These are:

(1) The Phong model – perfect diffuse reflection combined with empirically spread specular reflection (Phong 1975).

(2) A physically based specular refection model developed by Blinn (1977) and Cook and Torrance (1982).

(3) Pre-calculating BRDFs to be indexed during a rendering process Cabral *et al.* (1987).

(4) A physically based diffuse model developed by Hanrahan and Kreuger (1993).

This selection is both an historical sample and an illustration of the diverse approaches of researchers to local reflection models.

7.5 Perfect diffuse – empirically spread specular reflection (Phong 1975)

This is, in fact, the Phong reflection model. We have already discussed the practicalities of this, in particular, how it is integrated into a rendering system or strategy. Here we will look at it from a more theoretical point of view that enables a comparison with other direct reflection models.

The Phong reflection model accounts for diffuse reflection by Lambert's Cosine Law, where the intensity of the reflected light is a function of the cosine between the surface normal and the incoming light direction.

Phong used an empirically spread specular term. Here the idea is that a practical surface, say, shiny metal, reflects light in a lobe around the perfect mirror direction because it can be considered to be made up of tiny mirrors all oriented in slightly different directions, instead of being made up of a perfectly smooth mirror that takes the shape of the object. Thus the coarseness or roughness of the (shiny) surface can be simulated by the index n – the higher n is, the tighter the lobe and the smoother the surface. All surfaces simulated by this model have a plastic-like appearance.

Geometrically, in three-dimensional space the model produces a cone of rays centred on $\boldsymbol{R}$ whose intensity drops off exponentially as the angle between the ray and $\boldsymbol{R}$ increases.

A more subtle aspect of real behaviour, and one that accounts for the difference in the look of plastic and shiny metal, is missing entirely from this model. This is that the amount of light that is specularly reflected depends on the elevation angle θ_i (Figure 7.3) of the incoming light. Drive a car into the setting sun and you experience a blinding glare from the road surface – a dull surface at midday with little or no specular component. It was to account for this behaviour, which for any object accounts for subtle changes in the shape of a highlight as a function of the incoming light direction, that an early local reflection model, based on a physical microfacet simulation of the surface, emerged.

We can say that, although the direction of the specular bump in the Phong model depends on the incident direction – the specular bump is symmetrically disposed about the mirror direction – its magnitude does not vary and the Phong model implements a BRDF 'reduced' to:

$$\text{BRDF} = f(\theta_{ref}, \phi_{ref})$$

The BRDF shown in Figure 7.3 was calculated using the Phong reflection model.

7.6 Physically based specular reflection (Blinn 1977; Cook and Torrance 1982)

Two years after the appearance of Phong's work in 1975, J. Blinn (1977) published a paper describing how a physically based specular component could be used in computer graphics. In 1982, Cook and Torrance extended this model to account for the spectral composition of highlights – their dependency on material type and the angle of incidence of the light. These advances have a subtle effect on the size and colour of a highlight compared with that obtained from the Phong model. The model still retains the separation of the reflected light into a diffuse and specular component, and the new work concentrates entirely on the specular component, the diffuse component being calculated in the same way as before. The model is most successful in rendering shiny metallic-like surfaces, and through the colour variation in the specular highlight being able to, for example, render similarly coloured metals differently.

The problem of highlight shape is quite subtle. A highlight is just the image of a light source or sources reflected in the object. Unless the object surface is planar, this image is distorted by the object, and as the direction of the incoming light changes, it falls on a different part of the object and its shape changes. Therefore we have a highlight image whose overall shape depends on the curvature of the object surface over the area struck by the incident light and the viewing direction, which determines how much of the highlight is visible from the viewing direction. These are the primary factors that determine the shape of the patches of bright light that we see on the surface of an object and are easily calculated by using the Phong model.

The secondary factors which determine the highlight image are the dependence of its intensity and colour on the angle of incoming light with respect to a tangent plane at the point on the surface under consideration. This identifies the nature of the material to us and enables us to distinguish between metallic and non-metallic objects.

Curiously, despite producing more accurate highlights, these models were not taken up by the graphics community and the cheaper and simpler Phong model remained the more popular, as indeed it does to this day. The possible reason for this is that the differences produced by the more elaborate models are subtle. Objects rendered by the Phong model, although inaccurate and incorrect in highlight rendering, produce objects that look real. In most graphics applications, then and now, this is all that is required. Photo-realism, the much stated goal, of three-dimensional computer graphics, depends on very many factors other than local reflection models. To make objects look more real, only in this manifestly narrow sense, was perhaps not deemed to be worth the cost.

What is meant by a physical simulation in the context of light reflection is that we attempt to model the micro-geometry of the surface that causes the light to reflect, rather than simply imitating the behaviour, as we do in the Phong model, with an empirical term.

This early simulation of specular highlights has four components, and is based on a physical microfacet model consisting of symmetric V-shaped grooves occurring around an average surface (Figure 7.2). We now describe each of these components in turn.

7.6.1 Modelling the micro-geometry of the surface

A statistical distribution is set up for the orientation of the microfacets and this gives a term D for the light emerging in a particular (viewing) direction. A simple Gaussian can be used:

$$D = k \exp[-(\alpha/m)^2)]$$

where α is the angle of the microfacet with respect to the normal of the (mean) surface, that is the angle between $\boldsymbol{N}$ and $\boldsymbol{H}$, and m is the standard deviation of the distribution. Evaluating the distribution at this angle simply returns the number of microfacets with this orientation, that is the number of microfacets that can contribute to light emerging in the viewing direction. Two reflection lobes for $m = 0.2$ and 0.6 are shown in Figure 7.2(b).

Using microfacets to simulate the dependence of light reflection on surface roughness makes two enabling assumptions:

(1) It is assumed that the microfacets, although physically small, are large with respect to the wavelength of light.

(2) The diameter of the incident beam is large enough to intersect a number of microfacets that is sufficient to result in representative behaviour of the reflected light.

In BRDF terms this factor controls the extent to which the specular role bulges.

7.6.2 Shadowing and masking effects

Where the viewing vector, or the light orientation vector begins to approach the mean surface, interference effects occur. These are called shadowing and masking. Masking occurs when some reflected light is trapped and shadowing when incident light is intercepted, as can be seen from Figure 7.5.

The degree of masking and shadowing is dependent on the ratio l_1/l_2 (Figure 7.5(b)) which describes the proportionate amount of the facets contributing to reflected light that is given by:

$$G = 1 - l_1/l_2$$

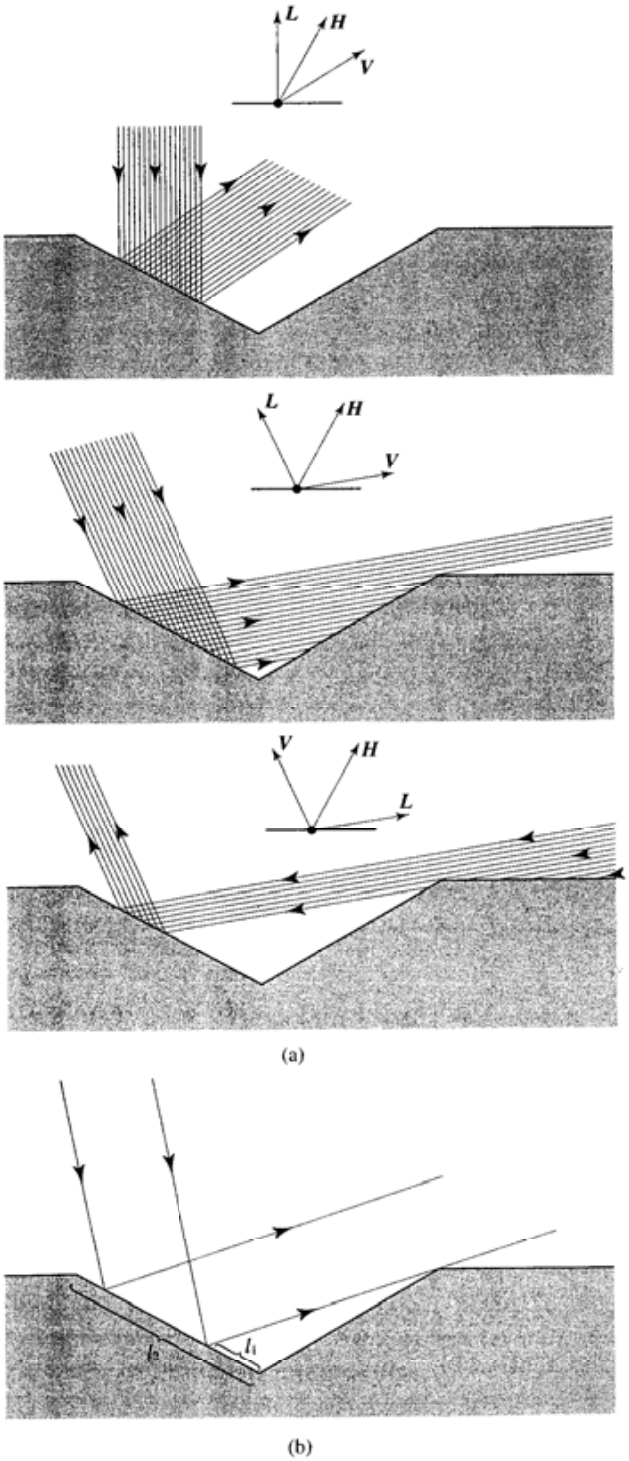


Figure 7.5
The interaction of light with a microfacet reflecting surface. (a) Shadowing and masking. (b) Amount of light which escapes depends on $1 - h_1/l_2$.

In the case where l_1 reduces to zero then all the reflected light escapes and:

$$G = 1$$

A detailed derivation of the dependence of l_1/l_2 on $\boldsymbol{L}$, $\boldsymbol{V}$ and $\boldsymbol{H}$ was given by Blinn (1977). For masking:

$$G_m = 2(\boldsymbol{N}\cdot\boldsymbol{H})(\boldsymbol{N}\cdot\boldsymbol{V})/\boldsymbol{V}\cdot\boldsymbol{H}$$

For shadowing the situation is geometrically identical with the role of the vectors $\boldsymbol{L}$ and $\boldsymbol{V}$ interchanged. For masking we have:

$$G_s = 2(\boldsymbol{N}\cdot\boldsymbol{H})(\boldsymbol{N}\cdot\boldsymbol{L})/\boldsymbol{V}\cdot\boldsymbol{H}$$

The value of G that must be used is the minimum of G_s and G_m. Thus:

$$G = \min\{1, G_s, G_m\}$$

7.6.3 Viewing geometry

Another pure geometric term is implemented to account for the glare effect mentioned in Section 7.5. As the angle between the view vector and the mean surface normal is increased towards 90°, an observer sees more and more microfacets and this is accounted for by a term:

$$1/\boldsymbol{N}\cdot\boldsymbol{V}$$

that is, the increase in area of the microfacets seen by a viewer is inversely proportional to the angle between the viewing direction and the surface normal. If there is incident light at a low angle then more of this light is reflected towards the viewer than if the viewer was intercepting light from an angle of incidence close to normal. This effect is countered by the shadowing effect which comes into play also as the viewing orientation approaches the mean surface orientation.

7.6.4 The Fresnel term

The next term to consider is the Fresnel term, F (see Section 7.1). This term concerns the amount of light that is reflected as opposed to being absorbed – a factor that depends on the material type considered as a perfect mirror surface – which our individual microfacets are. In other words we now consider behaviour for a perfect planar surface having previously modelled the entire surface as a set of such microfacets which individually behave as perfect mirrors. This factor determines the strength of the reflected lobe as a function of incidence angle and wavelength. The wavelength dependence accounts for subtle colour effects in the specular highlight.

The coefficients required to calculate F for any angle of incidence are not usually known and Cook and Torrance (1982) suggest a practical compromise which

is to use known (measured) values of F_0 – the value of F at normal incidence – to calculate μ then to use Equation 7.1 to evaluate F for any angle of incidence. At normal incidence, Equation 7.1 reduces to:

$$F_0 = \frac{(\mu - 1)^2}{(\mu + 1)^2}$$

(μ is, in fact, complex and contains an imaginary term – the extinction coefficient. This is zero for dielectrics – plastics, for example – and it is also zero for conductors, for metals at normal incidence; and it can thus be ignored for both categories of materials at normal incidence.)

Another way of calculating F for any incidence angle from F_0 is due to Schlick (1993) and is the formula:

$$F_\phi = F_0 + (1 - \cos\phi)^5(1 - F_0)$$

The practical effect of this term is to account for subtle changes in colour of the specular highlight as a function of angle of incidence. For any material, when the light is incident at an angle nearly parallel to the surface then the colour of the highlight approaches that of the light source. For other angles the colour depends on both the angle of incidence and the material. An example of this dependency is shown for polished copper in Figure 7.6.

The effect of this term is to cause the reflected intensity to increase as the angle of incidence increases (just as did the previous term $1/\boldsymbol{N}\cdot\boldsymbol{V}$) – less light is absorbed by the material and more is reflected. (A more subtle effect is that the peak of the specular lobe shifts away from the perfect mirror direction as the angle of incidence increases – see Figure 7.7.)

Thus putting these together the specular term now becomes:

$$\text{specular component} = DGF/(\boldsymbol{N}\cdot\boldsymbol{V})$$

where:

D is the micro-geometry term
G is the shadowing/masking term
F is the Fresnel term
$(\boldsymbol{N}\cdot\boldsymbol{V})$ is the glare effect term

Summarizing we have:

(1) A factor that models reflected light intensity as a function of the physical nature of the surface to within the approximations of the geometric simulation.

(2) Two interacting factors that simulate the behaviour of the 'glare' effect which occurs when light is incoming at a high angle (with respect to $\boldsymbol{N}$, the surface normal) of incidence.

(3) A factor that relates the reflected light intensity at each (perfect mirror) microfacet to the electro-optical properties of the material. This is a function of the direction of the incoming light and controls subtle second order

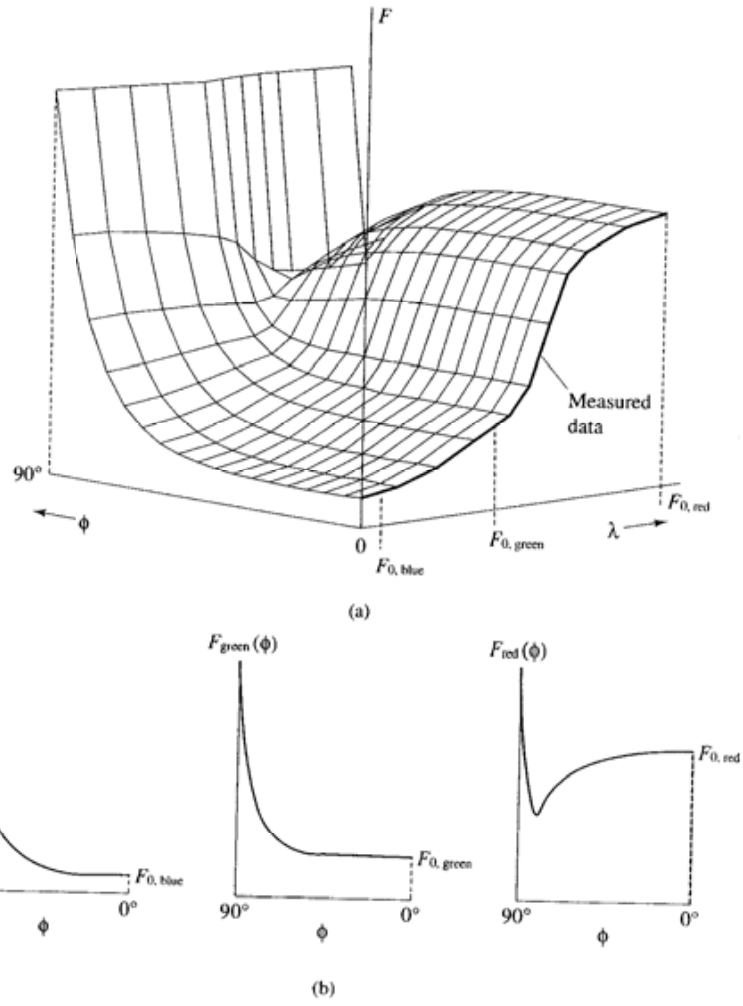


Figure 7.6
Fresnel equation and polished copper. (a) Reflectance F as a function of wavelength (λ) and angle of incidence (polished copper). (b) The dependence of F on ϕ for red, green and blue wavelengths.

effects concerning the shape and the colour of the highlight. This effect is important when trying to simulate the difference between, say, shiny plastic and metals. Gold, for example, exhibits yellow highlights when illuminated with white light and the highlight only tends to white when the light grazes the surface.

The specular term is separately calculated and combined with a uniform diffuse term:

$$\text{BRDF} = sR_s + dR_d \quad (\text{where } s + d = 1)$$

For example, metals are simulated, usually with $d = 0$ and $s = 1$ and shiny plastics with $d = 0.9$ and $s = 0.1$. Note that if d is set to zero for metals the specular

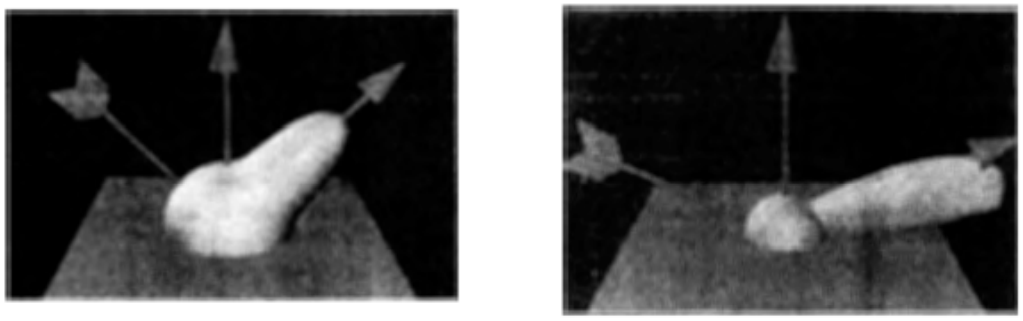


Figure 7.7
BRDFs for different angles of incidence in the Blinn reflection model.

term controls the colour of the object over its entire surface. Compare this with the Phong reflection model where the colour of the object is always controlled by the diffuse component. The Phong model, because of this, is incapable of producing metallic looking surfaces and all surfaces rendered using Phong have a distinct plastic look.

In this model the reflected light intensity depends on the elevation angle of the incoming light but the model is independent of the azimuth angle of the incident light. Whatever the azimuthal direction of the incoming light, it encounters the same statistical distribution of long, parallel, symmetric V-shaped grooves (a somewhat impossible situation in practice). Thus:

$$\text{BRDF} = f(\theta_{in}, \theta_{ref}, \phi_{ref})$$

A pair of BRDFs for a low and high angle of incidence are shown in Figure 7.7. This shows a specular lobe increasing in value (and also moving away from the mirror reflection direction) as the angle of incidence is increased towards the grazing angle. Figure 7.8 (Colour Plate) gives an idea of the variety of object appearances that can be achieved using this model.

7.7 Pre-computing BRDFs

One of the main inadequacies of the previous approach is that it cannot be used to model anisotropic surfaces. Many surfaces exhibit anisotropy reflection characteristics. Cloth and 'brushed' metal used in 'decorative' engineering applications – like car wheels – are two examples. Consider cloth, for example: this exhibits anisotropic reflection because it is made up of parallel threads with circular cross-sections. Each thread scatters light narrowly when the incident light is in a plane parallel to the direction of the thread, and more widely when the incident plane is parallel to the circular cross-section of the thread. The two popular approaches to including anisotropic behaviour in BRDFs have been to set up special surface models – usually based on cylinders – and pre-calculation.

In 1987 a model (Cabral *et al.* 1987) was reported that could deal with the dependence of the azimuth angle of the incoming light. The model pre-

calculates a BRDF for each $\boldsymbol{L}$ represented by a hemisphere divided into bins indexed by $\boldsymbol{V}$. The BRDF is calculated by ray tracing for each incoming direction a bundle of parallel, randomly positioned rays as they strike the surface and reflect to hit the surrounding hemisphere. The dependency of the BRDF on angles is then:

$$\text{BRDF} = f(\theta_{in}, \phi_{in}, \theta_{ref}, \phi_{ref})$$

The BRDF is generated by firing rays or beams onto a surface element that encompasses a sufficiently large area of the microsurface. The surface element is modelled by an array or grid of triangular microfacets. The rays that hit the element without being shadowed and emerge without being masked make a contribution to the BRDF, and the complete function is the sum of all such contributions. This information is built up by dividing a hemisphere into a number of cells or bins. A representation of this process is shown in Figure 7.9. The surface microfacets are perturbed out of the mean plane by a bump map as the figure suggests. Note that an advantage of this approach is that there is no restriction on the small-scale geometry – the microfacets do not need to form a Gaussian distribution, for example.

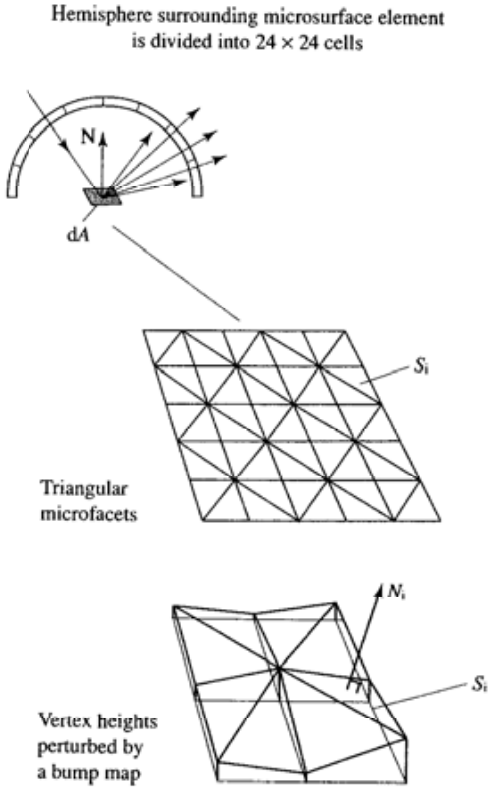


Figure 7.9
Modelling a surface with height-field perturbed triangular microfacets.

The BRDF is then a coarsely sampled version of a continuous BRDF. The pre-calculation is built up by considering each cell to be a source of incoming light and calculating the resulting reflection into the hemisphere. When a surface is being rendered the hemisphere closest to the angle of incident light is selected. The north pole of this distribution then has to be aligned with the surface normal at the point and the pre-calculated BRDF then gives the reflected intensity in the viewing direction.

7.8 Physically based diffuse component (Hanrahan and Kreuger 1993)

Until fairly recently local reflection models in computer graphics have concentrated almost exclusively on the specular component of reflected light and, as we have seen, these have been based on physical microsurface modelling.

Diffuse light is usually modelled on Lambert's Cosine Law which assumes that reflected light is isotropic and proportional in intensity to the cosine of the angle of incidence. Surface simulations of diffuse light are not possible because diffuse reflection originates from light that actually enters the material. This component is absorbed and scattered within the reflecting material. The wavelength-dependent absorption accounts for the colour of the material – incident white light is, in effect, filtered by the material. It is the multiple scattering within the material that causes the emerging light to be (approximately) isotropic. Thus a physical simulation of diffuse reflection would have to be based on subsurface scattering.

We could ask the question: what is wrong with sticking with Lambert's Law? The answer to this would be the same as the motivation for the development of physically based specular models – there are subtle effects produced by diffusely reflecting light that are responsible for the distinctive look of certain materials. Recent work by Hanrahan and Kreuger (1993) develops a physically based model for diffuse reflection that the authors claim is particularly appropriate for layered materials appearing in nature, such as biological tissues (skin, leaves and so on) and inorganic substances like snow and sand. The outcome of the model is, of course, anisotropy – reflecting the fact that very few real materials exhibit isotropic diffuse behaviour.

Hanrahan and Kreuger specify the reflected light from a point on the surface as

$$L_r = L_{rs} + L_{rv}$$

where L_{rs} is the reflected light due to surface scattering – imperfect specular reflection – and L_{rv} is the reflected light that is due to subsurface scattering. The algorithm that determines the subsurface scattering is based on a 1D transport model solved using a Monte Carlo approach. The details are outside the scope of this text; more important for our purposes is a conceptual understanding of the advances made by these researchers and their visual ramifications.

The combination of those two components produces anisotropic behaviour because of a number of factors that we will now describe. First, consider the angle

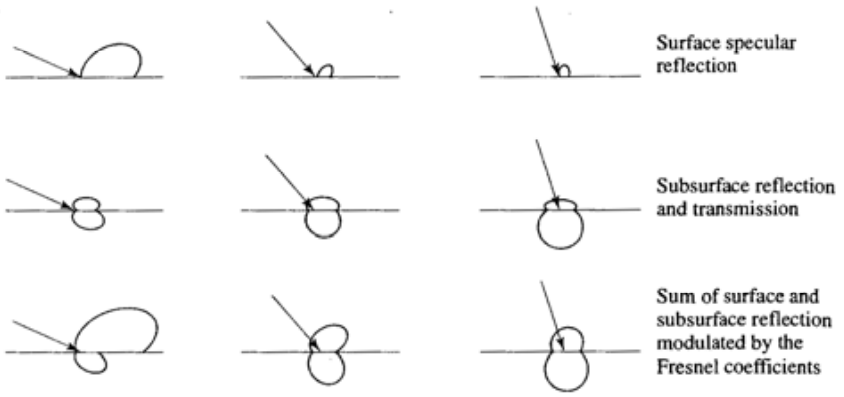


Figure 7.10
Reflection behaviour due to Hanrahan and Kreuger's model (after Hanrahan and Kreuger (1993)).

of incidence of the light. For a plane surface the amount of light entering the surface depends on Fresnel's law – the more light that enters the surface, the higher will be the contribution or influence from subsurface events to the total reflected light L_r. So the influence of L_{rv} depends on the angle of incidence. Subsurface scattering depends on the physical properties of the material. A material is modelled by a suspension of scattering sites or particles and parametrized by absorption and scattering cross-sections. These express the probability of occurrence per unit path length of scattering or absorption. The relative size of these parameters determines whether the scattering is forward, backward or isotropic.

The effect of these two factors is shown, for a simple case, in Figure 7.10. The first row shows high/low specular reflection as a function of angle of incidence. The behaviour of reflected light is dominated by surface scattering or specular reflection when the angle of incidence is high and by subsurface scattering when the angle of incidence is low. As we have seen, this behaviour is modelled, to a certain extent, by the Cook and Torrance model of Section 7.6. The second row shows reflection lobes due to subsurface scattering and it can be seen that materials can exhibit backward, isotropic or forward scattering behaviour. (The bottom lobes do not, of course, contribute to L_r but are nevertheless important when considering materials that are made up of multiple layers and thin translucent materials that are backlit.) The third row shows that the combination of L_{rs} and L_{rv} will generally result in non-isotropic behaviour which exhibits the following general attributes:

- Reflection increases as material layer thickness increases due to increased subsurface scattering.
- Subsurface scattering can be backward, isotropic or forward.
- Reflection from subsurface scattering tends to produce functions that are flattened on top of the lobe compared with the (idealized) hemisphere of Lambert's law.

Such factors result in the subtle differences between the model and Lambert's law.

8 Mapping techniques

Introduction

In this chapter we will look at techniques which store information (usually) in a two-dimensional domain which is used during rendering to simulate textures. The mainstream application is texture mapping but many other applications are described such as reflection mapping to simulate ray tracing. With the advent of texture mapping hardware the use of such facilities to implement real time rendering has seen the development of light maps. These use the texture facilities to enable the pre-calculation of (view independent) lighting which then 'reduces' rendering to a texture mapping operation.

Texture mapping became a highly developed tool in the 1980s and was the technique used to enhance Phong shaded scenes so that they were more visually interesting, looked more realistic or esoteric. Objects that are rendered using only Phong shading look plastic-like and texture mapping is the obvious way to add interest without much expense.

Texture mapping developed in parallel with research into global illumination algorithms – ray tracing and radiosity (see Chapters 10, 11 and 12). It was a device that could be used to enhance the visual interest of a scene, rather than

its photo-realism and its main attraction was cheapness – it could be grafted onto a standard rendering method without adding too much to the processing cost. This contrasted to the global illumination methods which used completely different algorithms and were much more expensive than direct reflection models.

Another use of texture mapping that became ubiquitous in the 1980s was to add pseudo-realism to shiny animated objects by causing their surrounding environment to be reflected in them. Thus tumbling logos and titles became chromium and the texture reflected on them moved as the objects moved. This technique – known as environment mapping – can also be used with a real photographed environment and can help to merge a computer animated object with a real environment. Environment mapping does not accomplish anything that could not be achieved by ray tracing – but it is much more efficient. A more recent use of environment mapping techniques is in image-based rendering which is discussed in Chapter 16.

As used in computer graphics, 'texture' is a somewhat confusing term and generally does not mean controlling the small-scale geometry of the surface of a computer graphics object – the normal meaning of the word. It is easy to modulate the colour of a Phong shaded object by controlling the value of the three diffuse coefficients and this became the most common object parameter to be controlled by texture mapping. (Colour variations in the physical world are not, of course, generally regarded as texture.) Thus as the rendering proceeds at pixel-by-pixel level, we pick up values for the Phong diffuse reflection coefficients and the diffuse component (the colour) of the shading changes as a function of the texture map(s). A better term is colour mapping and this appears to be coming into common usage.

This simple pixel-level operation conceals many difficulties and the geometry of texture mapping is not straightforward. As usual we make simplifications that lead to a visually acceptable solution. There are three origins to the difficulties:

(1) We mostly want to use texture mapping with the most popular representation in computer graphics – the polygon mesh representation. This, as we know, is a geometric representation where the object surface is approximated, and this approximation is only defined at the vertices. In a sense we have no surface – only an approximation to one – so how can we physically derive a texture value at a surface point if the surface does not exist?

(2) We want to use, in the main, two-dimensional texture maps because we have an almost endless source of textures that we can derive by frame-grabbing the real world, by using two-dimensional paint software or by generating textures procedurally. Thus the mainstream demand is to map a two-dimensional texture onto a surface that is approximated by a polygon mesh. This situation has become consolidated with the advent of cheap texture mapping hardware facilities.

(3) Aliasing problems in texture mapping are usually highly visible. By definition textures usually manifest some kind of coherence or periodicity.

Aliasing breaks this up and the resulting mess is usually high visible. This effect occurs as the periodicity in the texture approaches the pixel resolution.

We now list the possible ways in which certain properties of a computer graphics model can be modulated with variations under control of a texture map. We have listed these in approximate order of their popularity (which also tends to relate to their ease of use or implementation). These are:

(1) **Colour** As we have already pointed out, this is by far the most common object property that is controlled by a texture map. We simply modulate the diffuse reflection coefficients in the Phong reflection model with the corresponding colour from the texture map. (We could also change the specular coefficients across the surface of an object so that it appears shiny and matte as a function of the texture map. But this is less common, as being able to perceive this effect on the rendered object depends on producing specular highlights on the shiny parts if we are using the basic Phong reflection model.)

(2) **Specular 'colour'** This technique – known as environment mapping or chrome mapping – is a special case of ray tracing where we use texture map techniques to avoid the expense of full ray tracing. The map is designed so that it looks as if the (specular) object is reflecting the environment or background in which it is placed.

(3) **Normal vector perturbation** This elegant technique applies a perturbation to the surface normal according to the corresponding value in the map. The technique is known as bump mapping and was developed by a famous pioneer of three-dimensional computer graphic techniques – J. Blinn. The device works because the intensity that is returned by a Phong shading equation reduces, if the appropriate simplifications are made, to a function of the surface normal at the point currently being shaded. If the surface normal is perturbed then the shading changes and the surface that is rendered looks as if it is textured. We can therefore use a global or general definition for the texture of a surface which is represented in the database as a polygon mesh structure.

(4) **Displacement mapping** Related to the previous technique, this mapping method uses a height field to perturb a surface point along the direction of its surface normal. It is not a convenient technique to implement since the map must perturb the geometry of the model rather than modulate parameters in the shading equation.

(5) **Transparency** A map is used to control the opacity of a transparent object. A good example is etched glass where a shiny surface is roughened (to cause opacity) with some decorative pattern.

There are many ways to perform texture mapping. The choice of a particular method depends mainly on time constraints and the quality of the image required. To start with we will restrict the discussion to two-dimensional texture

maps – the most popular and common form – used in conjunction with polygon mesh objects. (Many of the insights detailed in this section are based on descriptions in Heckbert's (1986) defining work in this area.)

Mapping a two-dimensional texture map onto the surface of an object then projecting the object into screen space is a two-dimensional to two-dimensional transformation and can thus be viewed as an image warping operation. The most common way to do this is to inverse map – for each pixel we find its corresponding 'pre-image' in texture space (Figure 8.1(b)). However, for reasons that will shortly become clear, specifying this overall transformation is not straightforward and we consider initially that texture mapping is a two-stage process that takes us from the two-dimensional texture space into the three-dimensional space of the object and then via the projective transform into two-dimensional screen space (Figure 8.1(a)). The first transformation is known as parametrization and the second stage is the normal computer graphics projective transformation. The parametrization associates all points in texture space with points on the object surface.

The use of an anti-aliasing method is mandatory with texture mapping. This is easily seen by considering an object retreating away from a viewer so that its projection in screen space covers fewer and fewer pixels. As the object size decreases, the pre-image of a pixel in texture space will increase covering a larger area. If we simply point sample at the centre of the pixel and take the value of $T(u, v)$ at the corresponding point in texture space, then grossly incorrect results will follow (Figure 8.2(a), (b) and (c)). An example of this effect is shown in Figure 8.3. Here, as the chequerboard pattern recedes into the distance, it begins to break up in a disturbing manner. These problems are highly visible and move when animated. Consider Figure 8.2(b) and (c). Say, for example, that an object projects onto a single pixel and moves in such a way that the pre-image translates across the $T(u, v)$. As the object moves it would switch colour from black to white.

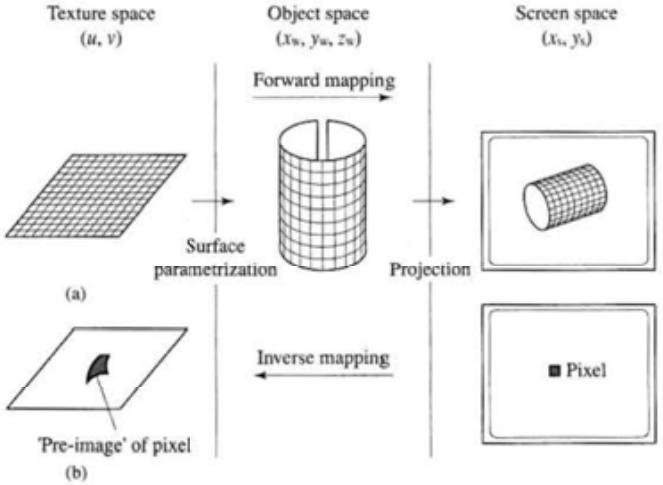


Figure 8.1
Two ways of viewing the process of two-dimensional texture mapping.
(a) Forward mapping.
(b) Inverse mapping.

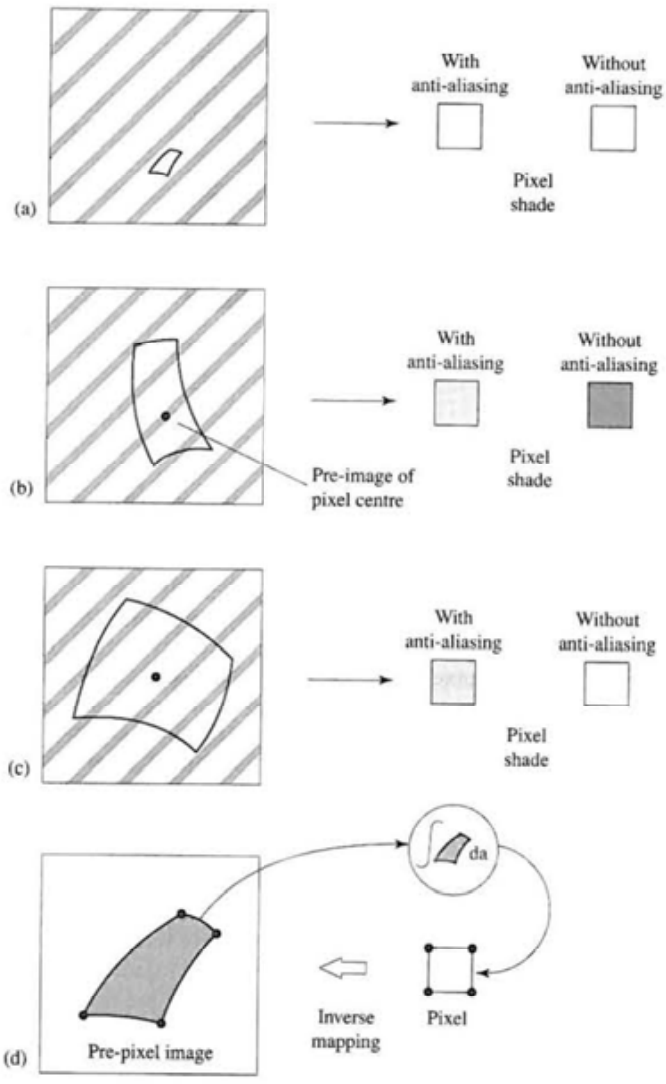
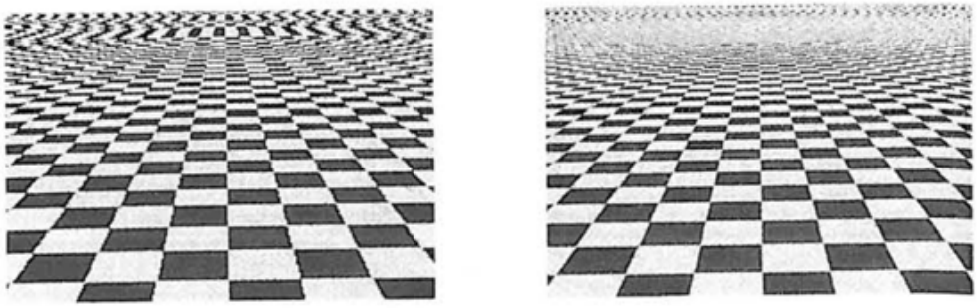


Figure 8.2
Pixels and pre-images in $T(u,v)$ space.

Anti-aliasing in this context then means integrating the information over the pixel pre-image and using this value in the shading calculation for the current pixel (Figure 8.2(d)). At best we can only approximate this integral because we have no knowledge of the shape of the quadrilateral, only its four corner points.



(a) (b)

Figure 8.3
Aliasing in texture mapping. The pattern in (b) is a super-sampled (anti-aliased) version of that in (a). Aliases still occur but appear at a higher spatial frequency.

8.1 Two-dimensional texture maps to polygon mesh objects

The most popular practical strategy for texture mapping is to associate, during the modelling phase, texture space coordinates (u, v) with polygon vertices. The task of the rendering engine then is to find the appropriate (u, v) coordinate for pixels internal to each polygon. The main problem comes about because the geometry of a polygon mesh is only defined at the vertices – in other words there is no analytical parametrization possible. (If the object has an analytical definition – a cylinder, for example – then we have a parametrization and the mapping of the texture onto the object surface is trivial.)

There are two main algorithm structures possible in texture mapping, inverse mapping (the more common) and forward mapping. (Heckbert refers to these as screen order and texture order algorithms respectively.) Inverse mapping (Figure 8.1(b)) is where the algorithm is driven from screen space and for every pixel we find by an inverse mapping its 'pre-image' in texture space. For each pixel we find its corresponding (u, v) coordinates. A filtering operation integrates the information contained in the pre-image and assigns the resulting colour to the pixel. This algorithm is advantageous if the texture mapping is to be incorporated into a Z-buffer algorithm where the polygon is rasterized and depth and lighting interpolated on a scan line basis. The square pixel produces a curvilinear quadrilateral as a pre-image.

In forward mapping the algorithm is driven from texture space. This time a square texel in texture space produces a curvilinear quadrilateral in screen space and there is a potential problem due to holes and overlaps in the texture image when it is mapped into screen space. Forward mapping is like considering the texture map as a rubber sheet – stretching it in a way (determined by the parametrization) so that it sticks on the object surface thereby performing the normal object space to screen space transform.

8.1.1

Inverse mapping by bilinear interpolation

Although forward mapping is easy to understand, in practical algorithms inverse mapping is preferred and from now on we will only consider this strategy for polygon mesh objects. For inverse mapping it is convenient to consider a single (compound) transformation from two-dimensional screen space (x, y) to two-dimensional texture space (u, v). This is just an image warping operation and it can be modelled as a rational linear projective transform:

$$x = \frac{au + bv + c}{gu + hu + i} \qquad y = \frac{du + ev + f}{gu + hv + i} \qquad [8.1]$$

This is, of course, a non-linear transformation as we would expect. Alternatively, we can write this in homogeneous coordinates as:

$$\begin{bmatrix} x' \\ y' \\ w \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} u' \\ v' \\ q \end{bmatrix}$$

where:

$$(x, y) = (x'/w, y'/w) \quad \text{and} \quad (u, v) = (u'/q, v'/q)$$

This is known as a rational linear transformation. The inverse transform – the one of interest to us in practice – is given by:

$$\begin{bmatrix} u' \\ v' \\ q \end{bmatrix} = \begin{bmatrix} A & B & C \\ D & E & F \\ G & H & I \end{bmatrix} \begin{bmatrix} x' \\ y' \\ w \end{bmatrix}$$

$$= \begin{bmatrix} ei - fh & ch - bi & bf - ce \\ fg - di & ai - cg & cd - af \\ dh - eg & bg - ah & ae - bd \end{bmatrix} \begin{bmatrix} x' \\ y' \\ w \end{bmatrix}$$

Now recall that in most practical texture mapping applications we set up, during the modelling phase, an association between polygon mesh vertices and texture map coordinates. So, for example if we have the association for the four vertices of a quadrilateral we can find the nine coefficients $(a, b, c, d, e, f, g, h, i)$. We thus have the required inverse transform for any point within the polygon. This is done as follows. Return to the first half of Equation 8.1, the equation for x. Note that we can multiply top and bottom by an arbitrary non-zero scalar constant without changing the value of y, in effect we only have five degrees of freedom – not six – and because of this we can, without loss of generality set $i = 1$. Thus, in the overall transformation we only have 8 coefficients to determine and our quadrilateral-to-quadrilateral association will give a set of 8 equations in 8 unknowns which can be solved by any standard algorithm for linear equations – Gaussian elimination, for example. Full details of this procedure are given in Heckbert (1986).

A better practical alternative is to achieve the same effect by bilinear interpolation in screen space. So we interpolate the texture coordinates at the same time as interpolating lighting and depth. However, we note from the above that it is the homogeneous coordinates (u', v', q) that we have to interpolate, because the u and v do not change linearly with x and y.

Assuming vertex coordinate/texture coordinate for all polygons we consider each vertex to have homogeneous texture coordinates:

$$(u', v', q)$$

where:

$$u = u'/q$$
$$v = v'/q$$
$$q = 1/z$$

We interpolate using the normal bilinear interpolation scheme within the polygon (see Section 1.5) using these homogeneous coordinates as vertices to give (u', v', q') for each pixel then the required texture coordinates are given by:

$$(u, v) = u'/q, v'/q$$

Note that this costs two divides per pixel. For the standard incremental implementation of this interpolation process we need three gradients down each edge (in the current edge-pair) and three gradients for the current scan line.

8.1.2 Inverse mapping by using an intermediate surface

The previous method for mapping two-dimensional textures is now undoubtedly the most popular approach. The method we now describe can be used in applications where there is no texture coordinate–vertex coordinate correspondence. Alternatively it can be used as a pre-process to determine this correspondence and the first method then used during rendering.

Two-part texture mapping is a technique that overcomes the surface parametrization problem in polygon mesh objects by using an 'easy' intermediate surface onto which the texture is initially projected. Introduced by Bier and Sloan (1986), the method can also be used to implement environment mapping and is thus a method that unifies texture mapping and environment mapping.

The process is known as two-part mapping because the texture is mapped onto an intermediate surface before being mapped onto the object. The intermediate surface is, in general, non-planar but it possesses an analytic mapping function and the two-dimensional texture is mapped onto this surface without difficulty. Finding the correspondence between the object point and the texture point then becomes a three-dimensional to three-dimensional mapping.

The basis of the method is most easily described as a two-stage forward mapping process (Figure 8.4):

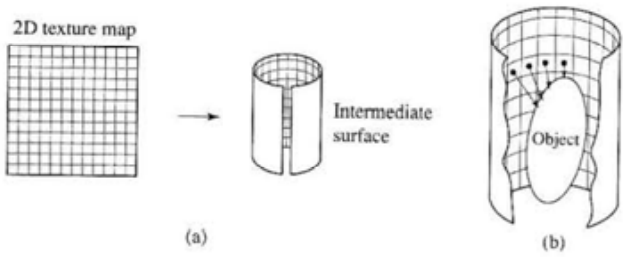


Figure 8.4
Two-stage mapping as a foward process. (a) *S* mapping. (b) *O* mapping.

(1) The first stage is a mapping from two-dimensional texture space to a simple three-dimensional intermediate surface such as a cylinder.

$$T(u, v) \rightarrow T'(x_i, y_i, z_i)$$

This is known as the *S* mapping.

(2) A second stage maps the three-dimensional texture pattern onto the object surface.

$$T'(x_i, y_i, z_i) \rightarrow O(x_w, y_w, z_w)$$

This is referred to as the *O* mapping.

These combined operations can distort the texture pattern onto the object in a 'natural' way, for example, one variation of the method is a 'shrinkwrap' mapping, where the planar texture pattern shrinks onto the object in the manner suggested by the eponym.

For the *S* mapping, Bier describes four intermediate surfaces: a plane at any orientation, the curved surface of a cylinder, the faces of a cube and the surface of a sphere. Although it makes no difference mathematically, it is useful to consider that $T(u, v)$ is mapped onto the interior surfaces of these objects. For example, consider the cylinder. Given a parametric definition of the curved surface of a cylinder as a set of points (θ, h), we transform the point (u, v) onto the cylinder as follows. We have:

$$S_{\text{cylinder}}: (\theta, h) \rightarrow (u, v)$$

$$= \left(\frac{r}{c} (\theta - \theta_0), \frac{1}{d} (h - h_0) \right)$$

where c and d are scaling factors and θ_0 and h_0 position the texture on the cylinder of radius r.

Various possibilities occur for the *O* mapping where the texture values for $O(x_w, y_w, z_w)$ are obtained from $T'(x_i, y_i, z_i)$, and these are best considered from a ray tracing point of view. The four *O* mappings are shown in Figure 8.5 and are:

(1) The intersection of the reflected view ray with the intermediate surface, T'. (This is, in fact, identical to environment mapping described in Section 8.6. The only difference between the general process of using this *O* mapping and environment mapping is that the texture pattern that is mapped onto the intermediate surface is a surrounding environment like a room interior.)

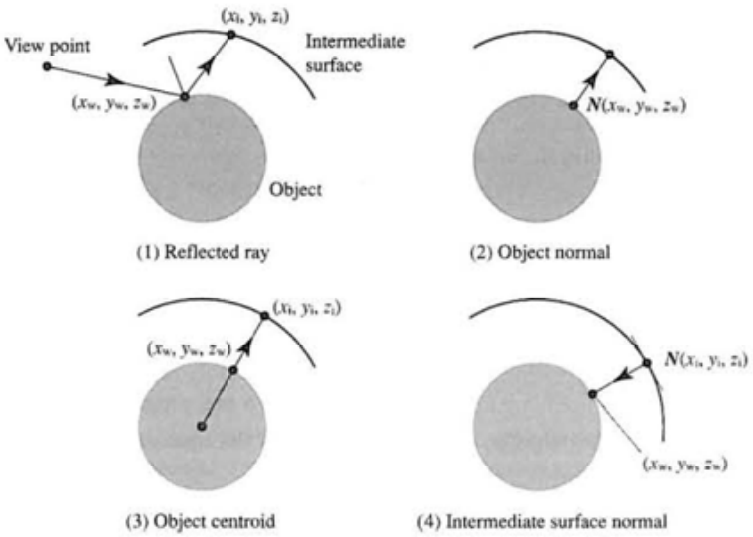


Figure 8.5
The four possible O mappings that map the intermediate surface texture T' onto the object

(2) The intersection of the surface normal at (x_w, y_w, z_w) with T'.

(3) The intersection of a line through (x_w, y_w, z_w) and the object centroid with T'.

(4) The intersection of the line from (x_w, y_w, z_w) to T' whose orientation is given by the surface normal at (x_i, y_i, z_i). If the intermediate surface is simply a plane then this is equivalent to considering the texture map to be a slide in a slide projector. A bundle of parallel rays of light from the slide projector impinges on the object surface. Alternatively it is also equivalent to three-dimensional texture mapping (see Section 8.7) where the field is defined by 'extruding' the two-dimensional texture map along an axis normal to the plane of the pattern.

Let us now consider this procedure as an inverse mapping process for the shrinkwrap case. We break the process into three stages (Figure 8.6).

(1) Inverse map four pixel points to four points (x_w, y_w, z_w) on the surface of the object.

(2) Apply the O mapping to find the point (θ, h) on the surface of the cylinder. In the shrinkwrap case we simply join the object point to the centre of the cylinder and the intersection of this line with the surface of the cylinder gives us (x_i, y_i, z_i).

$$x_w, y_w, z_w \rightarrow (\theta, h)$$

$$= (\tan^{-1}(y_w/x_w), z_w)$$

(3) Apply the S mapping to find the point (u, v) corresponding to (θ, h).

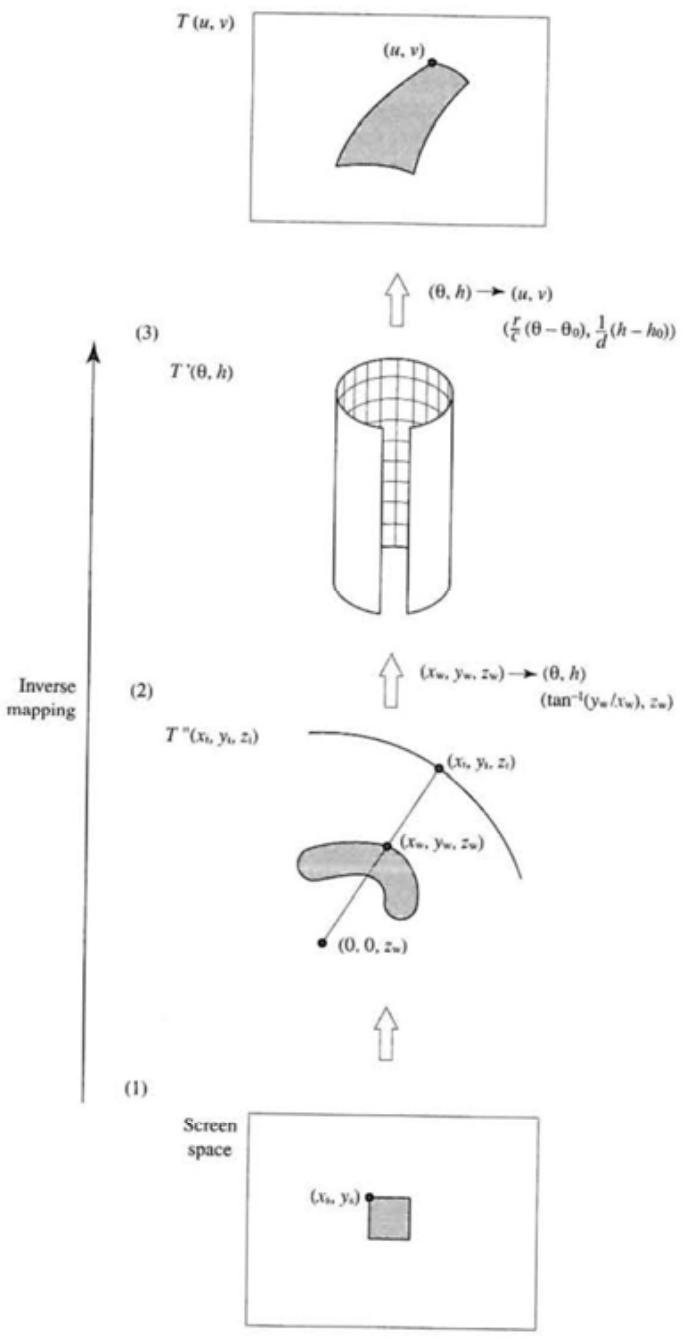


Figure 8.6
Inverse mapping using the shrinkwrap method.

Figure 8.7 (Colour Plate) shows examples of mapping the same texture onto an object using different intermediate surfaces. The intermediate objects are a plane (equivalently no intermediate surface – the texture map is a plane), a cylinder and a sphere. The simple shape of the vase was chosen to illustrate the different distortions that each intermediate object produces. There are two points that can be made from these illustrations. First, you can choose an intermediate mapping that is appropriate to the shape of the object. A solid of revolution may be best suited, for example, to a cylinder. Second, although the method does not place any constraints on the shape of the object, the final visual effect may be deemed unsatisfactory. Usually what we mean by texture does not involve the texture pattern being subject to large geometric distortions. It is for this reason that many practical methods are interactive and involve some strategy like pre-distorting the texture map in two-dimensional space until its produces a good result when it is stuck onto the object.

8.2 Two-dimensional texture domain to bi-cubic parametric patch objects

If an object is a quadric or a cubic then surface parametrization is straightforward. In the previous section we used quadrics as intermediate surfaces exactly for this reason. If the object is a bi-cubic parametric patch, texture mapping is trivial since a parametric patch by definition already possesses (u, v) values everywhere on its surface.

The first use of texture in computer graphics was a method developed by Catmull. This technique applied to bi-cubic parametric patch models; the algorithm subdivides a surface patch in object space, and at the same time executes a corresponding subdivision in texture space. The idea is that the patch subdivision proceeds until it covers a single pixel (a standard patch rendering approach described in detail in Chapter 4). When the patch subdivision process terminates the required texture value(s) for the pixel is obtained from the area enclosed by the current level of subdivision in the texture domain. This is a straightforward technique that is easily implemented as an extension to a bi-cubic patch renderer. A variation of this method was used by Cook where object surfaces are subdivided into 'micro-polygons' and flat shaded with values from a corresponding subdivision in texture space.

An example of this technique is shown in Figure 8.8 (Colour Plate). Here each patch on the teapot causes subdivision of a single texture map, which is itself a rendered version of the teapot. For each patch, the u, v values from the parameter space subdivision are used to index the texture map whose u, v values also vary between 0 and 1. This scheme is easily altered to, say, map four patches into the entire texture domain by using a scale factor of two in the u, v mapping.

8.3 Billboards

Billboard is the name given to a technique where a texture map is considered as a three-dimensional entity and placed in the scene, rather than as a map that controls the colour over the surface of an object. It is a simple technique that utilizes a two-dimensional image in a three-dimensional scene by rotating the plane of the image so that it is normal to its viewing direction (the line from the view point to its position). The idea is illustrated in Figure 8.9. Probably the most common example of this is the image of a tree which is approximately cylindrically symmetric. Such objects are impossible to render in real time and the visual effect of this trick is quite convincing providing the view vector is close to the horizontal plane in scene space. The original two-dimensional nature of the object is hardly noticeable in the two-dimensional projection, presumably because we do not have an accurate internal notion of what the projection of the tree should look like anyway. The billboard is in effect a two-dimensional object which is rotated about its y axis (for examples like the tree) through an angle which makes it normal to the view direction and translated to the appropriate position in the scene. The background texels in the billboard are set to transparent.

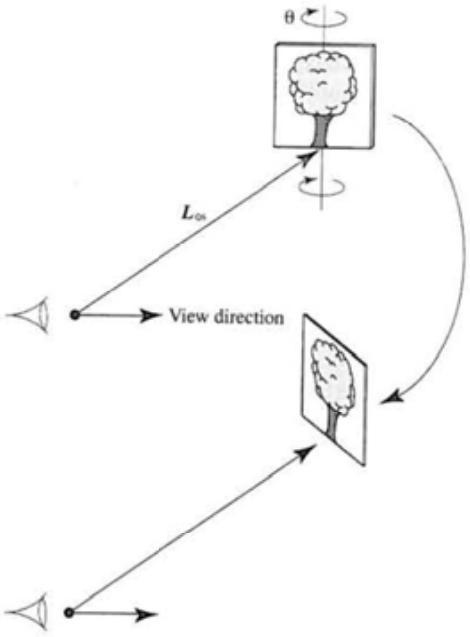


Figure 8.9
Providing the viewing direction is approximately parallel to the ground plane, objects like trees can be represented as a billboard and rotated about their y axis so that they are oriented normal to the L_{os} vector.

The modelling rotation for the billboard is given as:

$$\theta = \pi - \cos^{-1}(\boldsymbol{L}_{os} \cdot \boldsymbol{B}_n)$$

where:

$\boldsymbol{B}_n$ is the normal vector of the billboard, say (0,0,1)

$\boldsymbol{L}_{os}$ is the viewing direction vector from the view point to the required position of the billboard in world coordinates

Given θ and the required translation we can then construct a modelling transformation for the geometry of the billboard and transform it. Of course, this simple example will only work if the viewing direction is parallel or approximately parallel to the ground plane. When this is not true the two-dimensional nature of the billboard will be apparent.

Billboards are a special case of impostors or sprites which are essentially precomputed texture maps used to by-pass normal rendering when the view point is only changing slightly. These are described in detail in Chapter 14.

8.4 Bump mapping

Bump mapping, a technique developed by Blinn (1978), is an elegant device that enables a surface to appear as if it were wrinkled or dimpled without the need to model these depressions geometrically. Instead, the surface normal is angularly perturbed according to information given in a two-dimensional bump map and this 'tricks' a local reflection model, wherein intensity is a function mainly of the surface normal, into producing (apparent) local geometric variations on a smooth surface. The only problem with bump mapping is that because the pits or depressions do not exist in the model, a silhouette edge that appears to pass through a depression will not produce the expected cross-section. In other words the silhouette edge will follow the original geometry of the model.

It is an important technique because it appears to texture a surface in the normal sense of the word rather than modulating the colour of a flat surface. Figure 8.10 (Colour Plate) shows examples of this technique.

Texturing the surface in the rendering phase, without perturbing the geometry, by-passes serious modelling problems that would otherwise occur. If the object is polygonal the mesh would have to be fine enough to receive the perturbations from the texture map – a serious imposition on the original modelling phase, particularly if the texture is to be an option. Thus the technique converts a two-dimensional height field $B(u, v)$, called the bump map, and which represents some desired surface displacement, into appropriate perturbations of the local surface normal. When this surface normal is used in the shading equation the reflected light calculations vary as if the surface had been displaced.

Consider a point $\boldsymbol{P}(u, v)$ on a (parameterized) surface corresponding to $B(u, v)$. We define the surface normal at the point to be:

$$\boldsymbol{N} = \frac{\partial \boldsymbol{P}}{\partial u} \times \frac{\partial \boldsymbol{P}}{\partial v}$$

$$= \boldsymbol{P}_u \times \boldsymbol{P}_v$$

where $\boldsymbol{P}_u$ and $\boldsymbol{P}_v$ are the partial derivatives lying in the tangent plane to the surface at point $\boldsymbol{P}$. What we want to do is to have the same effect as displacing the point $\boldsymbol{P}$ in the direction of the surface normal at that point by an amount $B(u, v)$ – a one-dimensional analogue is shown in Figure 8.11. That is:

$$\boldsymbol{P}'(u, v) = \boldsymbol{P}(u, v) + B(u, v)\boldsymbol{N}$$

Locally the surface would not now be as smooth as it was before because of this displacement and the normal vector $\boldsymbol{N}'$ to the 'new' surface is given by differentiating this equation:

$$\boldsymbol{N}' = \boldsymbol{P}'_u + \boldsymbol{P}'_v$$

$$\boldsymbol{P}'_u = \boldsymbol{P}_u + B_u\boldsymbol{N} + B(u, v)\boldsymbol{N}_u$$

$$\boldsymbol{P}'_v = \boldsymbol{P}_v + B_v\boldsymbol{N} + B(u, v)\boldsymbol{N}_v$$

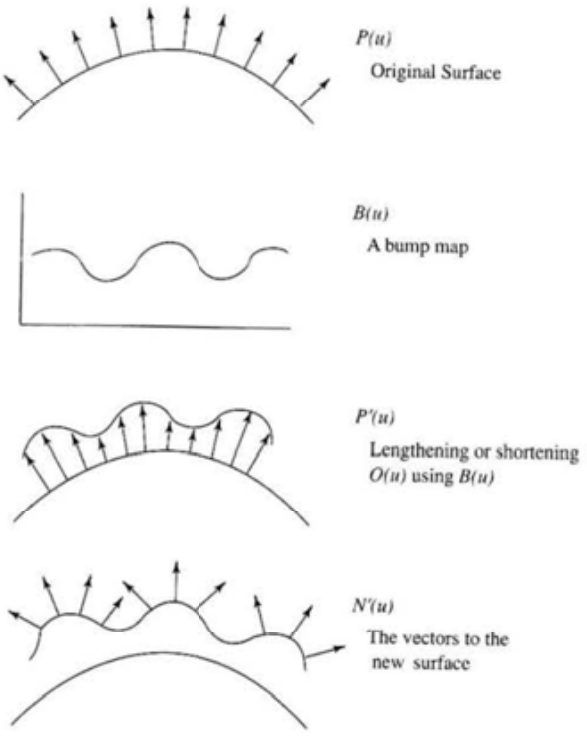


Figure 8.11
A one-dimensional example of the stages involved in bump mapping (after Blinn (1978)).

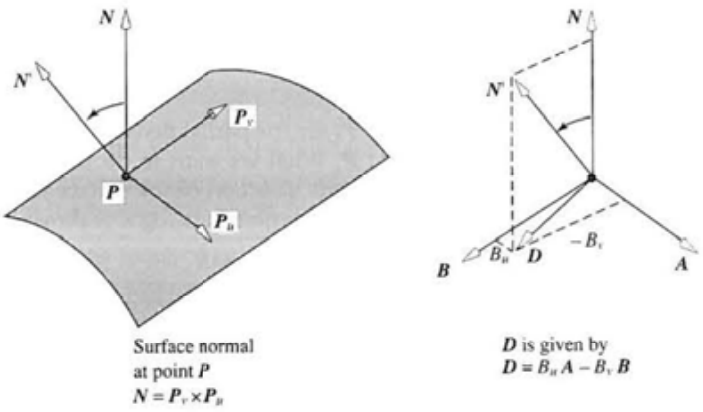


Figure 8.12
Geometric interpretation of bump mapping.

If B is small we can ignore the final term in each equation and we have:

$$\boldsymbol{N}' = \boldsymbol{N} + B_u \boldsymbol{N} \times \boldsymbol{P}_v + B_v \boldsymbol{P}_u \times \boldsymbol{N}$$

or

$$\begin{aligned}\boldsymbol{N}' &= \boldsymbol{N} + B_u \boldsymbol{N} \times \boldsymbol{P}_v - B_v \boldsymbol{N} \times \boldsymbol{P}_u \\ &= \boldsymbol{N} + (B_u \boldsymbol{A} - B_v \boldsymbol{B}) \\ &= \boldsymbol{N} + \boldsymbol{D}\end{aligned}$$

Then $\boldsymbol{D}$ is a vector lying in the tangent plane that 'pulls' $\boldsymbol{N}$ into the desired orientation and is calculated from the partial derivatives of the bump map and the two vectors in the tangent plane (Figure 8.12).

8.4.1 A multi-pass technique for bump mapping

For polygon mesh objects McReynolds and Blythe (1997) define a multi-pass technique that can exploit standard texture mapping hardware facilities. To do this they split the calculation into two components as follows. The final intensity value is proportional to $\boldsymbol{N}' \cdot \boldsymbol{L}$ where:

$$\boldsymbol{N}' \cdot \boldsymbol{L} = \boldsymbol{N} \cdot \boldsymbol{L} + \boldsymbol{D} \cdot \boldsymbol{L}$$

The first component is the normal Gouraud component and the second component is found from the differential coefficient of two image projections formed by rendering the surface with the height field as a normal texture map. To do this it is necessary to transform the light vector into tangent space at each vertex of the polygon. This space is defined by $\boldsymbol{N}$, $\boldsymbol{B}$ and $\boldsymbol{T}$, where:

$\boldsymbol{N}$ is the vertex normal
$\boldsymbol{T}$ is the direction of increasing u (or v) in the object space coordinate system
$\boldsymbol{B} = \boldsymbol{N} \times \boldsymbol{T}$

The normalized components of these vectors defines the matrix that transforms points into tangent space:

$$\boldsymbol{L}_{TS} = \begin{bmatrix} T_X & T_Y & T_Z & 0 \\ B_X & B_Y & B_Z & 0 \\ N_X & N_Y & N_Z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \boldsymbol{L}$$

The algorithm is as follows:

(1) The object is rendered using a normal renderer with texture mapping facilities. The texture map used is the bump map or height field.

(2) $\boldsymbol{T}$ and $\boldsymbol{B}$ are found at each vertex and the light vector transformed into tangent space.

(3) A second image is created in the same way but now the texture/vertex correspondence is shifted by small amounts in the direction of the X, Y components of $\boldsymbol{L}_{TS}$. We now have two image projections where the height field or the bump map has been mapped onto the object and shifted with respect to the surface. If we subtract these two images we get the differential coefficient which is the required term $\boldsymbol{D} \cdot \boldsymbol{L}$. (Finding the differential coefficient of an image by subtraction is a standard image processing technique – see, for example, Watt and Policarpo (1998)).

(4) The object is rendered in the normal manner *without* any texture and this component is added to the subtrahend calculated in step (3) to give the final bump-mapped image.

Thus we replace the explicit bump mapping calculations with two texture mapped rendering passes, an image subtract, a Gouraud shading pass then an image added to get the final result.

8.4.2 A pre-calculation technique for bump mapping

Tangent space can also be used to facilitate a pre-calculation technique as proposed by Peercy *et al.* (1997). This depends on the fact that the perturbed normal $\boldsymbol{N}'_{TS}$ in tangent space is a function only of the surface itself and the bump map. Peercy *et al.* define this normal at each vertex in terms of three pre-calculated coefficients.

It can be shown (Peercy *et al.* 1997) that the perturbed normal vector on tangent space is given by:

$$\boldsymbol{N}'_{TS} = \frac{a, b, c}{(a^2 + b^2 + c^2)^{1/2}}$$

where:

$$a = -B_u(\boldsymbol{B}.\boldsymbol{P}_v)$$

$$b = -(B_v|\boldsymbol{P}_u| - B_u(\boldsymbol{T}.\boldsymbol{P}_v))$$

$$c = |\boldsymbol{P}_u \times \boldsymbol{P}_v|$$

For each point in the bump map these points can be pre-computed and a map of perturbed normals is stored for use during rendering instead of the bump map.

8.5 Light maps

Light maps are an obvious extension to texture maps that enable lighting to be pre-calculated and stored as a two-dimensional texture map. We sample the reflected light over a surface and store this in a two-dimensional map. Thus shading reduces to indexing into a light map or a light modulated texture map. An advantage of the technique is that there is no restriction on the complexity of the rendering method used in the pre-calculation – we could, for example, use radiosity or any view-independent global illumination method to generate the light maps.

In principle light maps are similar to environment maps (see Section 8.6). In environment mapping we cache, in a two-dimensional map, all the illumination incident at a single point in the scene. With light maps we cache the reflected light from every surface in the scene in a set of two-dimensional maps.

If an accurate pre-calculation method is used then we would expect the technique to produce better quality shading and be faster than Gouraud interpolation. This means that we can incorporate shadows in the final rendering. The obvious disadvantage of the technique is that for moving objects we can only invoke a very simple lighting environment (diffuse shading with the light source at infinity). A compromise is to use dynamic shading for moving objects and assume that they do not interact, as far as shading is concerned, with static objects shaded with a light map.

Light maps can either be stored separately from texture maps, or the object's texture map can be pre-modulated by the light map. If the light map is kept as a separate entity then it can be stored at a lower resolution than the texture map because view-independent lighting, except at shadow edges, changes more slowly than texture detail. It can also be high-pass filtered which will ameliorate effects such as banding in the final image and also has the benefit of blurring shadow edges (in the event that a hard-edged shadow generation procedure has been used).

If an object is to receive a texture then we can modulate the brightness of the texture during the modelling phase so that it has the same effect as if the (unmodulated) texture colours were injected into, say, a Phong shading

equation. This is called surface caching because it stores the final value required for the pixel onto which a surface point projects and because texture caching hardware is used to implement it. If this strategy is employed then the texture mapping transform and the transform that maps light samples on the surface of the object into a light map should be the same.

Light maps were first used in two-pass ray tracing (see Section 10.7) and are also used in Ward's (1994) RADIANCE renderer. Their motivation in these applications was to cache diffuse illumination and to enable the implementation of a global illumination model that would work in a reasonable time. Their more recent use in games engines has, of course, been to facilitate shading in real time.

The first problem with light maps is how do we sample and store, in a two-dimensional array, the calculated reflected light across the face of a polygon in three-dimensional space. In effect this is the reverse of texture mapping where we need a mapping from two-dimensional space into three-dimensional object space. Another problem concerns economy. For scenes of any complexity it would clearly be uneconomical to construct a light map for each polygon – rather we require many polygons to share a single light map.

Zhukov *et al.* (1998) approach the three-dimensional sampling problem by organizing polygons into structures called 'polypacks'. Polygons are projected into the world coordinate planes and collected into polypacks if their angle with a coordinate plane does not exceed some threshold (so that the maximal projection plane is selected for a polygon) and if their extent does not overlap in the projection. The world space coordinate planes are subdivided into square cells (the texels or 'lumels') and back projected onto the polygon. The image of a square cell on a polygon is a parallelogram (whose larger angle ≤ 102°). These are called patches and are the subdivided polygon elements for which the reflected light is calculated. This scheme thus samples every polygon with almost square elements storing the result in the light map (Figure 8.13).

These patches form a subdivision of the scene sufficient for the purpose of generating light maps and a single light intensity for each patch can be calculated using whatever algorithm the application demands (for example Phong shading or radiosity). After this phase is complete there exists a set of (parallelogram-shaped) samples for each polygon. These then have to be 'stuffed'

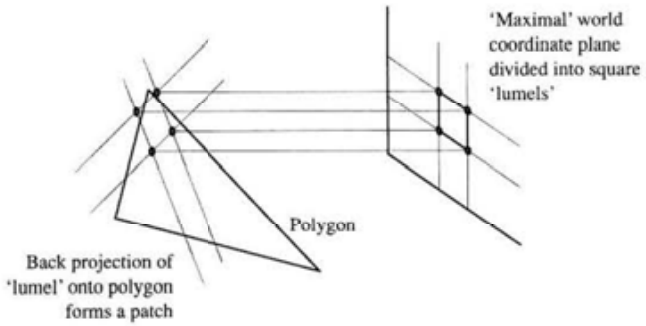


Figure 8.13
Forming a light map in the 'maximal' world coordinate plane.

into the minimum number of two-dimensional light maps in each coordinate plane. Zhukov *et al.* (1998) do this by using a 'first hit decreasing' algorithm which first sorts each polygon group by the number of texels.

Another problem addressed by the authors is that the groups of samples corresponding to a polygon have to be surrounded by 'sand' texels. These are supplementary texels that do not belong to a face group but are used when bilinear interpolation is used in conjunction with a light map and prevent visual lighting artefacts appearing at the edges of polygons. Thus each texture map consists of a mixture of light texels, sand texels and unoccupied texels. Zhukov *et al.* (1998) report that for a scene consisting of 24 000 triangles (700 patches) 14 light maps of 256 × 256 texels were produced which exhibited a breakdown of texels as: 75% light texels, 15% sand texels and 10% unoccupied texels.

A direct scheme for computing light maps for scenes made up of triangles and for which we already have vertex/texture coordinate association is to use this correspondence to derive an affine transformation between texture space and object space and then use this transformation to sample the light across the face of a triangle. The algorithm is then driven from the texture map space (by scan-converting the polygon projection in texture space) and for each texel finding its corresponding point or projection on the object surface from:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix}$$

where (x, y, z) is the point on the object corresponding to the texel (u, v). This transformation can be seen as a linear transformation in three-dimensional space with the texture map embedded in the $z = 1$ plane. The coefficients are found from the vertex/texture coordinate correspondence by inverting the U matrix in:

$$\begin{bmatrix} x_0 & x_1 & x_2 \\ y_0 & y_1 & y_2 \\ z_0 & z_1 & z_2 \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} u_0 & u_1 & u_2 \\ v_0 & v_1 & v_2 \\ 1 & 1 & 1 \end{bmatrix}$$

writing this as:

$$X = AU$$

we have:

$$A = XU^{-1}$$

The inverse U^{-1} is guaranteed to exist providing the three points are non-collinear. Note that in terms of our treatment in Section 8.1 this is a forward mapping from texture space to object space. Examples of a scene lit using this technique are shown in Figure 8.14 (Colour Plate).

8.6 Environment or reflection mapping

Originally called reflection mapping and first suggested by Blinn and Newell (1976), environment mapping was consolidated into mainline rendering techniques in an important paper by Greene (1986). Environment maps are a shortcut to rendering shiny objects that reflect the environment in which they are placed. They can approximate the quality of a ray tracer for specular reflections and do this by reducing the problem of following a reflected view vector to indexing into a two-dimensional map which is no different from a conventional texture map. Thus processing costs that would be incurred in ray tracing programs are regulated to the (off-line) construction of the map(s). In this sense it is a classic partial off-line or pre-calculation technique like pre-sorting for hidden surface removal. An example of a scene and its corresponding (cubic) environment map is shown in Figure 18.8.

The disadvantages of environment mapping are:

- It is (geometrically) correct only when the object becomes small with respect to the environment that contains it. This effect is usually not noticeable in the sense that we are not disturbed by 'wrong' reflections in the curved surface of a shiny object. The extent of the problem is shown in Figure 18.9 which shows the same object ray traced and environment mapped.
- An object can only reflect the environment – not itself – and so the technique is 'wrong' for concave objects. Again this can be seen in Figure 18.9 where the reflection of the spout is apparent in the ray traced image.
- A separate map is required for each object in the scene that is to be environment mapped.
- In one common form of environment mapping (sphere mapping) a new map is required whenever the view point changes.

In this section we will examine three methods of environment mapping which are classified according to the way in which the three-dimensional environment information is mapped into two-dimensions. These are cubic, latitude–longitude and sphere mapping. (Latitude–longitude is also a spherical mapping but the term sphere mapping is now applied to the more recent form.) The general principles are shown in Figure 8.15. Figure 8.15(a) shows the conventional ray tracing paradigm which we replace with the scheme shown in Figure 8.15(b). This involves mapping the reflected view vector into a two-dimensional environment map. We calculate the reflected view vector as (Section 1.3.5):

$$\boldsymbol{R}_v = 2(\boldsymbol{N}\cdot\boldsymbol{V})\boldsymbol{N} - \boldsymbol{V} \qquad [8.2]$$

Figure 8.15(c) shows that, in practice, for a single pixel we should consider the reflection beam, rather than a single vector, and the area subtended by the beam in the map is then filtered for the pixel value. A reflection beam originates either

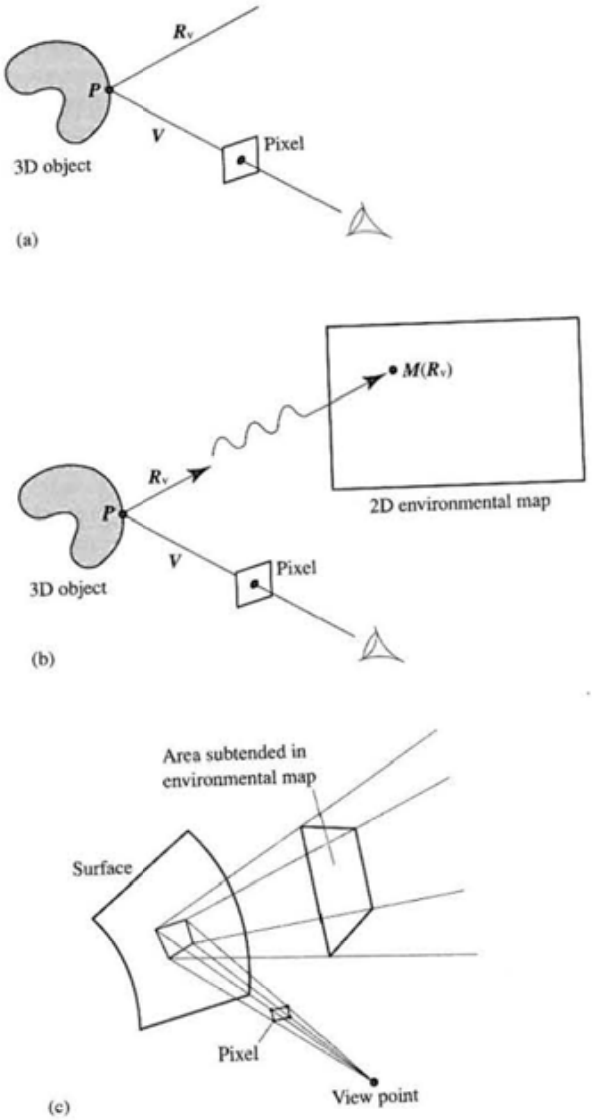


Figure 8.15
Environment mapping
(a) The ray tracing model – that part of the environment reflected at point ***P*** is determined by reflecting the view ray $\boldsymbol{R}_v$. (b) We try to achieve the same effect as in (a) by using a function of $\boldsymbol{R}_v$ to index into a two-dimensional map. (c) A pixel subtends a reflection beam.

from four pixel corners if we are indexing the map for each pixel, or from polygon vertices if we are using a fast (approximate) scheme. An important point to note here is that the area intersected in the environment map is a function of the curvature of the projected pixel area on the object surface. However, because we are now using texture mapping techniques we can employ pre-filtering anti-aliasing methods (see Section 8.8).

In real time polygon mesh rendering, we can calculate reflected view vectors only at the vertices and use linear interpolation as we do in conventional texture mapping. Because we expect to see fine detail in the resulting image, the quality of this approach depends strongly on the polygon size.

In effect an environment map caches the incident illumination from all directions at a single point in the environment with the object that is to receive the mapping removed from the scene. Reflected illumination at the surface of an object is calculated from this incident illumination by employing the aforementioned geometric approximation – that the size of the object itself can be considered to approach the point and a simple BRDF which is a perfect specular term – the reflected view vector. It is thus a view-independent pre-calculation technique.

8.6.1 Cubic mapping

As we have already implied, environment mapping is a two-stage process that involves – as a pre-process – the construction of the map. Cubic mapping is popular because the maps can easily be constructed using a conventional rendering system. The environment map is in practice six maps that form the surfaces of a cube (Figure 8.16). An example of an environment map is shown in Figure 18.8. The view point is fixed at the centre of the object to receive the environment map, and six views are rendered. Consider a view point fixed at the centre of a room. If we consider the room to be empty then these views would contain the

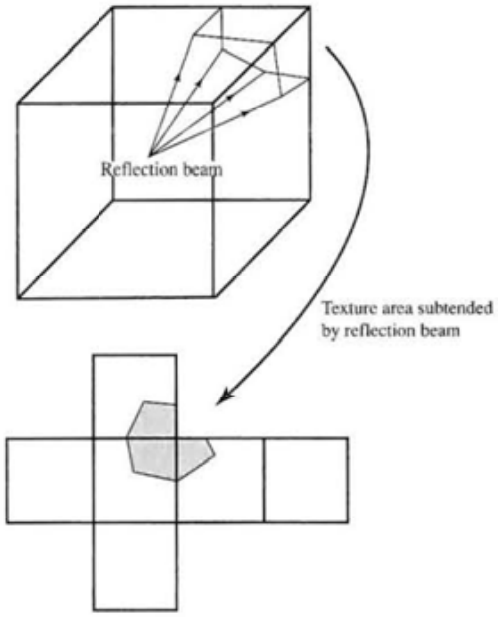


Figure 8.16
Cubic environment mapping: the reflection beam can range over more than one map.

four walls and the floor and ceiling. One of the problems of a cubic map is that if we are considering a reflection beam formed by pixel corners, or equivalently by the reflected view vectors at a polygon vertex, the beam can index into more than one map (Figure 18.16). In that case the polygon can be subdivided so that each piece is constrained to a single map.

With cubic maps we need an algorithm to determine the mapping from the three-dimensional view vector into one or more two-dimensional maps. (With the techniques described in the next section this mapping algorithm is replaced by a simple calculation.) If we consider that the reflected view vector is in the same coordinate frame as the environment map cube (the case if the view were constructed by pointing the (virtual or real) camera along the world axes in both directions), then the mapping is as follows.

For a single reflection vector:

(1) Find the face it intersects – the map number. This involves a simple comparison of the components of the normalized reflected view vector against the (unit) cube extent which is centred on the origin.

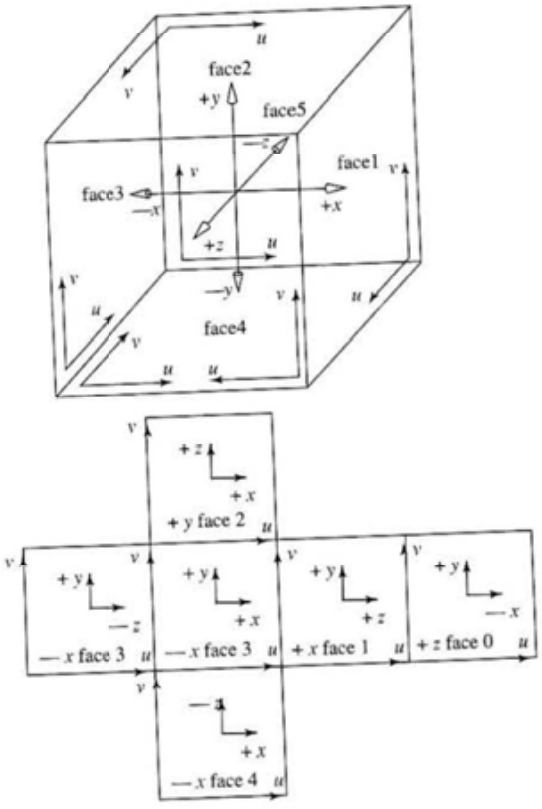


Figure 8.17
Cubic environment map convention.

(2) Map the components into (u, v) coordinates. For example, a point (x, y, z) intersecting the face normal to the negative z axis is given by:

$$u = x + 0.5$$
$$v = -z + 0.5$$

for the convention scheme shown in Figure 8.17.

One of the applications of cubic environment maps (or indeed any environment map method) that became popular in the 1980s is to 'matte' an animated computer graphics object into a real environment. In that case the environment map is constructed from photographs of a real environment and the (specular) computer graphics object can be matted into the scene, and appear to be part of it as it moves and reflects its surroundings.

8.6.2 Sphere mapping

The first use of environment mapping was by Blinn and Newell (1976) wherein a sphere rather than a cube was the basis of the method used. The environment map consisted of a latitude–longitude projection and the reflected view vector, R_v, was mapped into (u, v) coordinates as:

$$u = \frac{1}{2}\left(1 + \frac{1}{\pi}\tan^{-1}\left(\frac{R_{vy}}{R_{vx}}\right)\right) \quad -\pi < \tan^{-1} < \pi$$

$$v = \frac{R_{vz} + 1}{2}$$

The main problem with this simple technique is the singularities at the poles. In the polar area small changes in the direction of the reflection vector produce large changes in (u, v) coordinates. As $R_{vz} \rightarrow \pm 1$, both R_{vx} and $R_{vy} \rightarrow 0$ and R_{vy}/R_{vx} becomes ill-defined. Equivalently, as $v \rightarrow 1$ or 0 the behaviour of u starts to break down causing visual disturbances on the surface. This can be ameliorated by modulating the horizontal resolution of the map with $\sin\theta$ (where θ is the elevation angle in polar coordinates).

An alternative sphere mapping form (Haeberli and Segal 1993; Miller *et al.* 1998) consists of a circular map which is the orthographic projection of the reflection of the environment as seen in the surface of a perfect mirror sphere (Figure 8.18). Clearly such a map can be generated by ray tracing from the view plane. (Alternatively a photograph can be taken of a shiny sphere.) Although the map caches the incident illumination at the reference point by using an orthographic projection it can be used to generate, to within the accuracy of the process, a normal perspective projection.

To generate the map we proceed as follows. We trace a parallel ray bundle – one ray for each texel (u, v) and reflect each ray from the sphere. The point on the sphere at the point hit by the ray from (u, v) is $\boldsymbol{P}$, where:

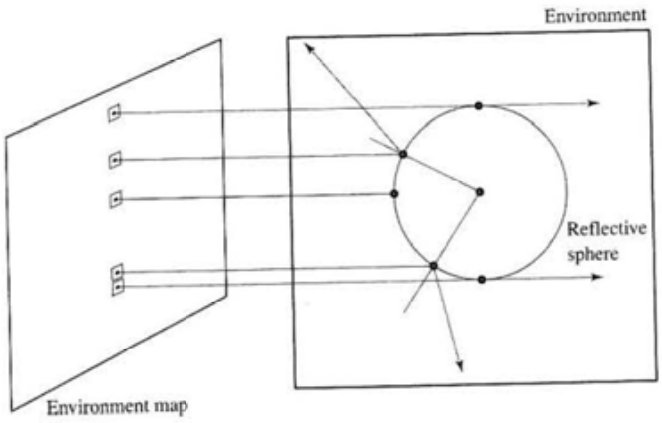


Figure 8.18
Constructing a spherical map by ray tracing from the map texels onto a reflective sphere.

$$\boldsymbol{P}_x = u \qquad \boldsymbol{P}_y = v$$
$$\boldsymbol{P}_z = (1.0 - \boldsymbol{P}_x^2 - \boldsymbol{P}_y^2)^{1/2}$$

This is also the normal to the sphere at the hit point and we can compute the reflected vector using Equation 8.2.

To index into the map we reflect the view vector from the object (either for each pixel or for each polygon vertex) and calculate the map coordinates as:

$$u = \frac{R_x}{m} + \frac{1}{2}$$

$$v = \frac{R_y}{m} + \frac{1}{2}$$

where:

$$m = 2(R_x^2 + R_y^2 + (R_z + 1)^2)^{1/2}$$

8.6.3 Environment mapping: comparative points

Sphere mapping overcomes the main limitation of cubic maps which require, in general, access to a number of the face maps, and is to be preferred when speed is important. However, both types of sphere mapping suffer more from non-uniform sampling than cubic mapping. Refer to Figure 8.19 which attempts to demonstrate this point. In all three cases we consider that the environment map is sampling incoming illumination incident on the surface of the unit sphere. The illustration shows the difference between the areas on the surface of the sphere sampled by a texel in the environment map. Sampling only approaches uniformity when the viewing direction during the rendering phase aligns with the viewing direction from which the map was computed. For this reason this type of spherical mapping is considered to be view dependent and a new map has to be computed when the view direction changes.

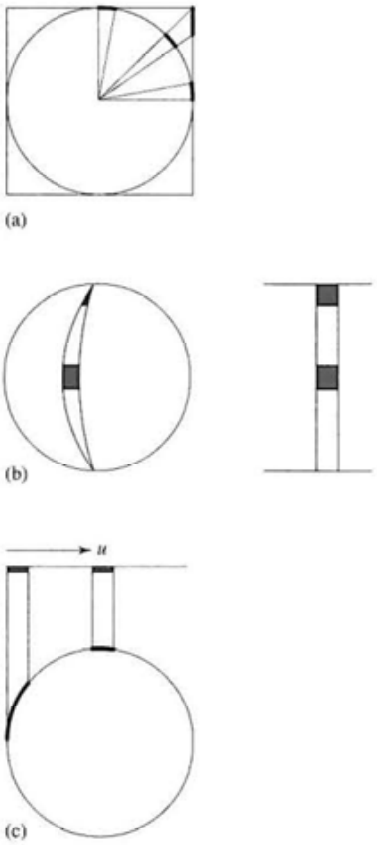


Figure 8.19
Sampling the surface of a sphere. (a) Cubic perspective: under-sampling at the centre of the map (equator and meridian) compared to the corners. (b) Mercator or latitude–longitude: severe over-sampling at edges of the map in the *v* direction (poles). (c) Orthographic: severe under-sampling at the edges of the map in the *u* direction (equator).

8.6.4 Surface properties and environment mapping

So far we have restricted the discussion to geometry and assumed that the object which is environment mapped possesses a perfect mirror surface and the map is indexed by a single reflected view ray. What if we want to use objects with reflecting properties other than that of a perfect mirror. Using the normal Phong local reflection model, we can consider two components – a diffuse component plus a specular component – and construct two maps. The diffuse map is indexed by the surface normal at the point of interest and the specular map is indexed by the reflected view vector. The relative contribution from each map is determined by diffuse and specular reflection coefficients just as in standard Phong shading. This enables us to render objects as if they were Phong shaded but with the addition of reflected environment detail which can be blurred to

simulate a non-smooth surface. (Note that this approximates an effect that would otherwise have to be rendered using distributed ray tracing.)

This technique was first reported by Miller and Hoffman (1984). (This reference is to SIGGRAPH course notes. These, particularly the older ones, are generally unavailable and we only refer to them if the material does not, as far as we know, appear in any other publication) and it is their convention that we follow here. Both the diffuse and specular maps are generated by processing the environment map. Thus we can view the procedure as a two-step process where the first step – the environment map – encodes the illumination at a point due to the scene with the object removed and the second step filters the map to encode information about the surface of the object.

Miller and Hoffman (1984) generate the diffuse map from the following definition:

$$D(\boldsymbol{N}) = \frac{\sum_{\boldsymbol{L}} I(\boldsymbol{L}) \times \text{Area}(\boldsymbol{L}) \times f_d\,(\boldsymbol{N}\cdot\boldsymbol{L})}{4\pi}$$

where:

$\boldsymbol{N}$ is the surface normal at the point of interest

$I(\boldsymbol{L})$ is the environment map as a function of $\boldsymbol{L}$ the incident direction to which the entry I in the map corresponds

Area is the area on the surface of the unit sphere associated with $\boldsymbol{L}$

f_d is the diffuse convolution function:

$f_d(x) = k_d\,x$ for $x > 0$ and $f_d(x) = 0$ for $x \leq 0$

k_d is the diffuse reflection coefficient that weights the contribution of $D(\boldsymbol{N})$ in summing the diffuse and specular contributions

Thus for each value of $\boldsymbol{N}$ we sum over all values of $\boldsymbol{L}$ the area-weighted dot product or Lambertian term.

The specular map is defined as:

$$S(\boldsymbol{R}) = \frac{\sum_{\boldsymbol{L}} I(\boldsymbol{L}) \times \text{Area}(\boldsymbol{L}) \times f_s\,(\boldsymbol{R}\cdot\boldsymbol{L})}{4\pi}$$

where:

$\boldsymbol{R}$ is the reflected view vector

f_s is the specular convolution function;

$f_s(x) = k_s\,x^n$ for $x > 0$ and $f_s(x) = 0$ for $x \leq 0$

k_s is the specular reflection coefficient

(Note that if f_s is set to unity the surface is a perfect mirror and the environment map is unaltered.)

The reflected intensity at a surface point is thus:

$D(\boldsymbol{N}) + S(\boldsymbol{R})$

8.7 Three-dimensional texture domain techniques

We have seen in preceding sections that there are many difficulties associated with mapping a two-dimensional texture onto the surface of a three-dimensional object. The reasons for this are:

(1) Two-dimensional texture mapping based on a surface coordinate system can produce large variations in the compression of the texture that reflect a corresponding variation in the curvature of the surface.

(2) Attempting to continuously texture map the surface of an object possessing a non-trivial topology can quickly become very awkward. Textural continuity across surface elements that can be of a different type and can connect together in any ad hoc manner is problematic to maintain.

Three-dimensional texture mapping neatly circumvents these problems since the only information required to assign a point a texture value is its position in space. Assigning an object a texture just involves evaluating a three-dimensional texture function at the surface points of the object. A fairly obvious requirement of this technique is that the three-dimensional texture field is procedurally generated. Otherwise the memory requirements, particularly if three-dimensional mip-mapping is used, become exorbitant. Also, it is inherently inefficient to construct an entire cubic field of texture when we only require these values at the surface of the object.

Given a point (x, y, z) on the surface of an object, the colour is defined as $T(x, y, z)$, where T is the value of texture field. That is, we simply use the identity mapping (possibly in conjunction with a scaling):

$$u = x \quad v = y \quad w = z$$

where:

(u, v, w) is a coordinate in the texture field

This can be considered analogous to actually sculpting or carving an object out of a block of material. The colour of the object is determined by the intersection of its surface with the texture field. The method was reported simultaneously by Perlin (1985) and Peachey (1985) wherein the term 'solid texture' was coined.

The disadvantage of the technique is that although it eliminates mapping problems, the texture patterns themselves are limited to whatever definition that you can think up. This contrasts with a two-dimensional texture map; here any texture can be set up by using, say, a frame-grabbed image from a television camera.

8.7.1 Three-dimensional noise

A popular class of procedural texturing techniques all have in common the fact that they use a three-dimensional, or spatial, noise function as a basic modelling

primitive. These techniques, the most notable of which is the simulation of turbulence, can produce a surprising variety of realistic, natural-looking texture effects. In this section we will concern ourselves with the issues involved in the algorithmic generation of the basic primitive – solid noise.

Perlin (1985) was the first to suggest this application of noise, defining a function *noise*() that takes a three-dimensional position as its input and returns a single scalar value. This is called model-directed synthesis – we evaluate the noise function only at the point of interest. Ideally the function should possess the following three properties:

(1) Statistical invariance under rotation.

(2) Statistical invariance under translation.

(3) A narrow bandpass limit in frequency.

The first two conditions ensure that the noise function is controllable – that is, no matter how we move or orientate the noise function in space, its general appearance is guaranteed to stay the same. The third condition enables us to sample the noise function without aliasing. Whilst an insufficiently sampled noise function may not produce noticeable defects in static images, if used in animation applications, incorrectly sampled noise will produce a shimmering or bubbling effect.

Perlin's method of generating noise is to define an integer lattice, or a set of points in space, situated at locations (i, j, k) where i, j and k are all integers. Each point of the lattice has a random number associated with it. This can be done either by using a simple look-up table or, as Perlin (1985) suggests, via a hashing function to save space. The value of the noise function, at a point in space coincident with a lattice point, is just this random number. For points in space not on the lattice – in general (u, v, w) – the noise value can be obtained by linear interpolation from the nearby lattice points. If, using this method, we generate a solid noise function $T(u, v, w)$ then it will tend to exhibit directional (axis aligned) coherences. These can be ameliorated by using cubic interpolation but this is far more expensive and the coherences still tend to be visible. Alternative noise generation methods that eliminate this problem are to be found in Lewis (1989); however, it is worth bearing in mind that the entire solid noise function is sampled by the surface and usually undergoes a transformation (it is modulated, for example, to simulate turbulence) and this in itself may be enough to eliminate the coherences.

8.7.2 Simulating turbulence

A single piece of noise can be put to use to simulate a remarkable number of effects. By far the most versatile of its applications is the use of the so-called turbulence function, as defined by Perlin, which takes a position x and returns a turbulent scalar value. It is written in terms of the progression, a one-dimensional version of which would be defined as:

$$\text{turbulence}(x) = \sum_{i=0}^{k} \text{abs}\left(\frac{\text{noise}\,(2^i x)}{2^i}\right)$$

The summation is truncated at k which is the smallest integer satisfying:

$$\frac{1}{2^{k+1}} < \text{the size of a pixel}$$

The truncation band limits the function ensuring proper anti-aliasing. Consider the difference between the first two terms in the progression, noise (x) and noise ($2x$)/2. The noise function in the latter term will vary twice as fast as the first – it has twice the frequency – and will contain features that are half the size of the first. Moreover, its contribution to the final value for the turbulence is also scaled by one-half. At each scale of detail the amount of noise added into the series is proportional to the scale of detail of the noise and inversely proportional to the frequency of the noise. This is self-similarity and is analogous to the self-similarity obtained through fractal subdivision, except that this time the subdivision drives not displacement, but octaves of noise, producing a function that exhibits the same noisy behaviour over a range of scales. That this function should prove so useful is best seen from the point of view of signal analysis, which tells us that the power spectrum of *turbulence*() obeys a $1/f$ power law, thereby loosely approximating the $1/f^2$ power law of Brownian motion.

The turbulence function in isolation only represents half the story, however. Rendering the turbulence function directly results in a homogeneous pattern that could not be described as naturalistic. This is due to the fact that most textures which occur naturally, contain some non-homogeneous structural features and so cannot be simulated by turbulence alone. Take marble, for example, which has easily distinguished veins of colour running through it that were made turbulent before the marble solidified during an earlier geological era. In the light of this fact we can identify two distinct stages in the process of simulating turbulence, namely:

(1) Representation of the basic, first order, structural features of a texture through some basic functional form. Typically the function is continuous and contains significant variations in its first derivatives.

(2) Addition of second and higher order detail by using turbulence to perturb the parameters of the function.

The classic example, as first described by Perlin, is the turbulation of a sine wave to give the appearance of marble. Unperturbed, the colour veins running through the marble are given by a sine wave passing through a colour map. For a sine wave running along the x axis we write:

$$\text{marble}(x) = \text{marble\_colour}\,(\sin(x))$$

The colour map *marble_colour*() maps a scalar input to an intensity. Visualizing this expression, Figure 8.20(a) is a two-dimensional slice of marble rendered with the colour spline given in Figure 8.20(b). Next we add turbulence:

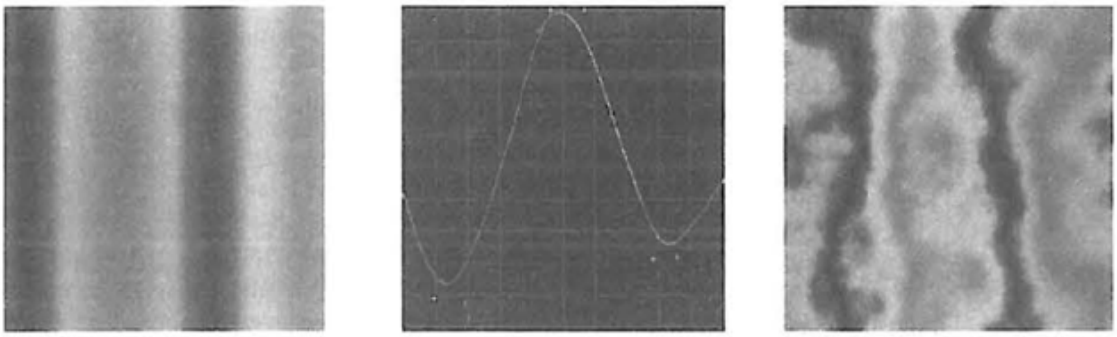


Figure 8.20
Simulating marble.
(a) Unturbulated slice obtained by using the spline shown in (b). (b) Colour spline used to produce (a). (c) Marble section obtained by turbulating the slice shown in (a).

$$\text{marble}(x) = \text{marble\_colour}\ (\sin(x + \text{turbulence}(x))$$

to give us Figure 8.20(c), a convincing simulation of marble texture. Figure 8.21 (Colour Plate) shows the effect in three dimensions.

Of course, use of the turbulence function need not be restricted to modulate just the colour of an object. Any parameter that affects the appearance of an object can be turbulated. Oppenheimer (1986) turbulates a sawtooth function to bump map the ridges of bark on a tree. Turbulence can drive the transparency of objects such as clouds. Clouds can be modelled by texturing an opacity map onto a sphere that is concentric with the earth. The opacity map can be created with a paint program; clouds are represented as white blobs with soft edges that fade into complete transparency. These edges become turbulent after perturbation of the texture coordinates. In an extension to his earlier work, Perlin (1989) uses turbulence to volumetrically render regions of space rather than just evaluating texture at the surface of an object. Solid texture is used to modulate the geometry of an object as well as its appearance. Density modulation functions that specify the soft regions of objects are turbulated and rendered using a ray marching algorithm. A variety of applications are described, including erosion, fire and fur.

8.7.3 Three-dimensional texture and animation

The turbulence function can be defined over time as well as space simply by adding an extra dimension representing time, to the noise integer lattice. So the lattice points will now be specified by the indices (i, j, k, l) enabling us to extend the parameter list to noise (x, t) and similarly for turbulence (x, t). Internal to these procedures the time axis is not treated any differently from the three spatial axes.

For example, if we want to simulate fire, the first thing that we do is to try to represent its basic form functionally, that is, a 'flame shape'. The completely ad hoc nature of this functional sculpting is apparent here. The final form decided

on was simply that which after experimentation gave the best results. We shall work in two space due to the expense of the three-dimensional volumetric approach referred to at the end of the last section.

A flame region is defined in the xy plane by the rectangle with minimax co-ordinates $(-b, 0)$, (b, h). Within this region the flame's colour is given by:

$$\text{flame}(x) = (1 - y/h)\ \text{flame\_colour}(\text{abs}(x/b))$$

This is shown schematically in Figure 8.22 (Colour Plate). *Flame_colour* (x) consists of three separate colour splines that map a scalar value x to a colour vector. Each of the R, G, B splines have a maximum intensity at $x = 0$ which corresponds to the centre of the flame and a fade-off to zero intensity at $x = 1$. The green and blue splines go to zero faster than the red. The colour returned by *flame_colour*() is weighted according to its height from the base of the flame to get an appropriate variation along y. The flame is rendered by applying *flame*() to colour a rectangular polygon that covers the region of the flames definition. The opacity of the polygon is also textured by using a similar functional construction. Figure 8.22 also shows the turbulated counterpart obtained by introducing the turbulence function thus:

$$\text{flame}(x, t) = (1 - y/h)\ \text{flame\_colour}(\text{abs}(x/b) + \text{turbulence}(x, t)$$

To animate the flame we simply render successive slices of noise which are perpendicular to the time axis and equispaced by an amount corresponding to the frame interval. It is as if we are translating the polygon along the time axis. However, mere translation in time is not enough, recognizable detail in the flame, though changing shape with time, remained curiously static in space. This is because there is a general sense of direction associated with a flame, convection sends detail upwards. This was simulated, and immediately gave better results, by moving the polygon down in y as well as through time, as shown in Figure 8.23. The final construction is thus:

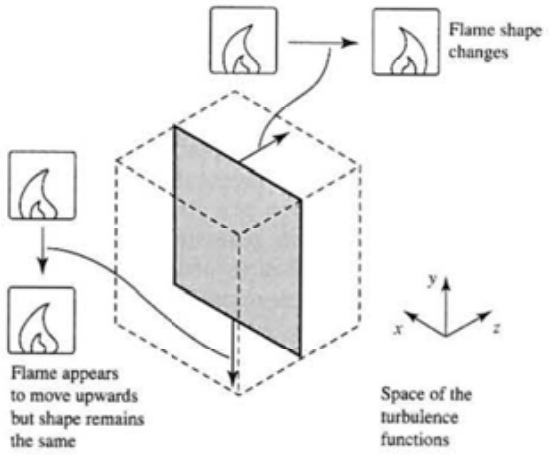


Figure 8.23
Animating turbulence for a two-dimensional object.

$$\text{flame}(x, t) = (1 - y/h)\,\text{flame\_colour}(\text{abs}(x/b) + \text{turbulence}(x + (0, t\Delta y, 0), t))$$

where Δy is the distance moved in y by the polygon relative to the noise per unit time.

8.7.4 Three-dimensional light maps

In principle there is no reason why we cannot have three-dimensional light maps – the practical restriction is the vast memory resources that would be required. In the event that it is possible we have a method of caching the reflected light at every point in the scene. We use any view-independent rendering method and assign the calculated light intensity at point (x, y, z) in object space to $T(x, y, z)$. It is interesting to now compare our pre-calculation mapping methods.

With environment mapping we cache all the incoming illumination at a *single* point in object space in a two-dimensional map which is labelled by the direction of the incoming light at the point. A reflected view vector is then used to retrieve the reflected light directed towards the user. These are normally used for perfect specular surfaces and give us fast view-dependent effects.

With two-dimensional light maps we cache the reflected light for each surface in the scene in a set of two-dimensional maps. Indexing into these maps during the rendering phase depends on the method that was used to sample three-dimensional object space. We use these to cache view-independent non-dynamic lighting.

With three-dimensional light maps we store reflected light at a point in a three-dimensional structure that represents object space. Three-dimensional light maps are a subset of light fields (see Chapter 16).

8.8 Anti-aliasing and texture mapping

As we have discussed in the introduction to this chapter, artefacts are extremely problematic in texture mapping and most textures produce visible artefacts unless the method is integrated with an anti-aliasing procedure. Defects are highly noticeable, particularly in texture that exhibits coherence or periodicity, as soon as the predominant spatial frequency in the texture pattern approaches the dimension of a pixel. (The classic example of this effect is shown in Figure 8.3.) Artefacts generated by texture mapping are not well handled by the common anti-aliasing method – such as supersampling – and because of this standard two-dimensional texture mapping procedures usually incorporate a specific anti-aliasing technique.

Anti-aliasing in texture mapping is difficult because, to do it properly, we need to find the pre-image of a pixel and sum weighted values of $T(u, v)$ that fall within the extent of the pre-image to get a single texture intensity for the pixel. Unfortunately the shape of the pre-image changes from pixel to pixel and this

filtering process consequently becomes expensive. Refer again to Figure 8.2. This shows that when we are considering a pixel its pre-image in texture space is, in general, a curvilinear quadrilateral, because the net effect of the texture mapping and perspective mapping is of a non-linear transformation. The figure also shows, for the diagonal band, texture for which, unless this operation is performed or approximated, erroneous results will occur. In particular, if the texture map is merely sampled at the inverse mapping of the pixel centre then the sampled intensity may be correct if the inverse image size of the pixel is sufficiently small, but in general it will be wrong.

In the context of Figure 8.24(a), anti-aliasing means approximating the integration shown in the figure. An approximate, but visually successful, method ignores the shape but not the size or extent of the pre-image and pre-calculates all the required filtering operations. This is mip-mapping invented by Williams (1983) and probably the most common anti-aliasing method developed specifically for texture mapping. His method is based on pre-calculation and an assumption that the inverse pixel image is reasonably close to a square. Figure 8.24(b) shows the pixel pre-image approximated by a square. It is this approximation that enables the anti-aliasing or filtering operation to be pre-calculated. In fact there are two problems. The first is more common and is known as compression or minification. This occurs when an object becomes small in screen space and consequently a pixel has a large pre-image in texture space. Figure 8.24(c) shows this situation. Many texture elements (sometimes called 'texels') need to be mapped into a single pixel. The other problem is called magnification. Here an object becomes very close to the viewer and only part of the object may occupy the whole of screen space, resulting in pixel pre-images that have less area than one texel (Figure 8.24(d)). Mip-mapping deals with compression and some elaboration to mip-mapping is usually required for the magnification problem.

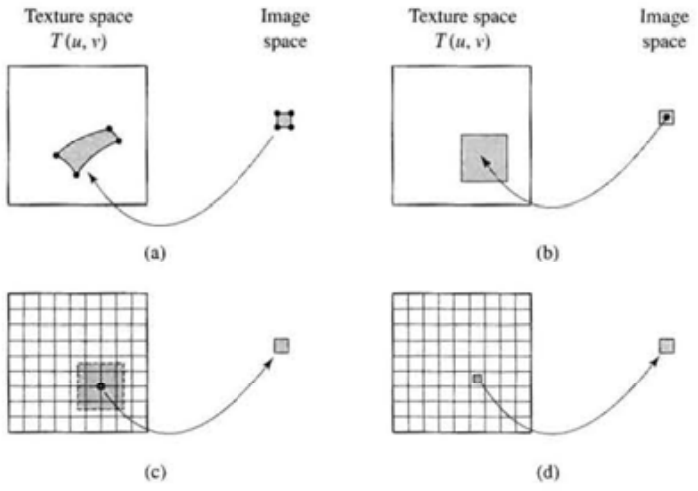


Figure 8.24
Mip-mapping approximations. (a) The pre-image of a pixel is a curvilinear quadrilateral in texture space. (b) A pre-image can be approximated by a square. (c) Compression is required when a pixel maps onto many texels. (d) Magnification is required when a pixel maps onto less than one texel.

In mip-mapping, instead of a texture domain comprising a single image, Williams uses many images, all derived by averaging down the original image to successively lower resolutions. In other word they form a set of pre-filtered texture maps. Each image in the sequence is exactly half the resolution of the previous. Figure 8.25 shows an approximation to the idea. An object near to the viewer, and large in screen space, selects a single texel from a high-resolution map. The same object further away from the viewer and smaller in screen space selects a single texel from a low-resolution map. An appropriate map is selected by a parameter D. Figure 8.26 (Colour Plate) shows the mip-map used in Figure 8.8.

In a low-resolution version of the image each texel represents the average of a number of texels from the previous map. By a suitable choice of D, an image at appropriate resolution is selected and the filtering cost remains constant – the many texels to one pixel cost problem being avoided. The centre of the pixel is mapped into that map determined by D and this single value is used. In this way the original texture is filtered and, to avoid discontinuities between the images at varying resolutions, different levels are also blended. Blending between levels occurs when D is selected. The images are discontinuous in resolution but D is a continuous parameter. Linear interpolation is carried out from the two nearest levels.

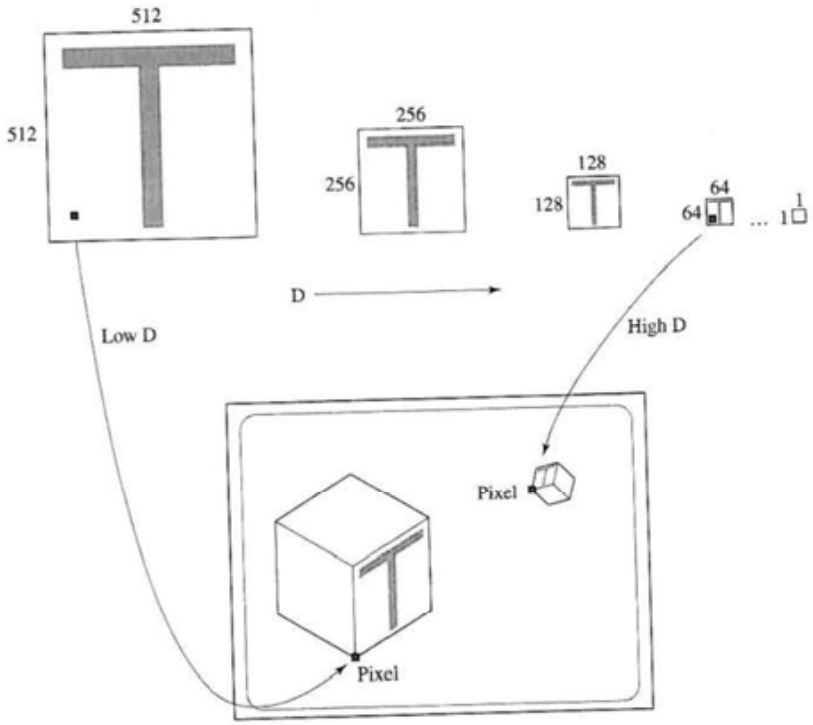


Figure 8.25
Showing the principle of mip-mapping.

Williams selects D from:

$$D = \text{max\_of}\left(\left(\left(\frac{\partial u}{\partial x}\right)^2 + \left(\frac{\partial v}{\partial x}\right)^2\right)^{1/2}, \left(\left(\frac{\partial u}{\partial y}\right)^2 + \left(\frac{\partial v}{\partial y}\right)^2\right)^{1/2}\right)$$

where ∂u and ∂v are the original dimensions of the pre-image in texture space and $\partial x = \partial y = 1$ for a square pixel.

A 'correct' or accurate estimation of D is important. If D is too large then the image will look blurred, too small and aliasing artefacts will still be visible. Detailed practical methods for determining depending on the mapping context are given in Watt and Watt (1992).

In a theoretical sense the magnification problem does not exist. Ideally we would like mip-maps that can be used at any level of detail, but in practice, storage limitations restrict the highest resolution mask to, say, 512 × 512 texels. This problem does not seem to have been addressed in the literature and the following two approaches are supplied by Silicon Graphics for their workstation family. Silicon Graphics suggest two solutions. First, to simply extrapolate beyond the highest resolution mip-map, and a more elaborate procedure that extracts separate texture information into low and high frequency components.

Extrapolation is defined as:

$$\text{LOD}(+1) = \text{LOD}(0) + (\text{LOD}(0) - \text{LOD}(-1))$$

where LOD (level of detail) represents mip-maps as follows:

LOD(+1) is the extrapolated mip-map
LOD(0) is the highest resolution stored mip-map
LOD(–1) is the next highest resolution stored mip-map

This operation derives an extrapolated mip-map of blocks of 4 × 4 pixels over which there is no variation. However, the magnification process preserves edges – hence the name.

Extrapolation works best when high frequency information is correlated with low frequency structural information, that is when the high frequency information represents edges in the texture. For example, consider that texture pattern is made up of block letters. Extrapolation will blur/magnify the interior of the letters, while keeping the edges sharp.

When high frequency information is not correlated with low frequency information, extrapolation causes blurring. This occurs with texture that tends to vary uniformly throughout, for example wood grain. Silicon Graphics suggest separating the low and high frequency information and converting a high resolution (unstorable at, say, 2K × 2K) into a 512 × 512 map that stores low frequency or structural information and a 256 × 256 map that stores high frequency detail. This separation can be achieved accurately using classical filtering techniques. Alternatively a space domain procedure is as follows:

(1) Make a 512 × 512 low frequency map by simply re-sampling the original 2K × 2K map.

(2) Make the 256 × 256 detail mask as follows:

(i) Select a 256 × 256 window from the original map that contains representative high frequency texture.

(ii) Re-sample this to 64 × 64 and re-scale to 256 × 256 resulting in a blurred version of the original 256 × 256 map.

(iii) Subtract the blurred map from the original, adding a bias to make the subtrahend image unsigned. This results in a 256 × 256 high frequency.

Now when magnification is required a mix of the 512 × 512 low resolution texture with the high resolution detail is used.

8.9 Interactive techniques in texture mapping

One of the main problems in designing a conventional two-dimensional texture map is the visualization of the result on the rendered object. Say an artist or a designer is creating a texture map by painting directly in the two-dimensional *uv* space of the map. We know that the distortion of the map, when it is 'stuck' on the object is both a function of the shape of the object and the mapping method that is used. To design a texture interactively the artist needs to see the final rendered object and have some intuition of the mapping mechanism so that he can predict the effect of changes made to the texture map.

We will now describe two interactive techniques. In the first the designer paints in *uv* or texture space. The second attempts to make the designer think that he is painting directly on the object in 3D world space.

The first technique is extremely simple and was evolved to texture animals/objects that exhibit a plane of symmetry. It is simply an interactive version of two-part texture mapping with a plane as the intermediate object (see Section 8.1.2). The overall idea is shown in Figure 8.27. The animal model is enclosed in a bounding box. The texture map $T(u, v)$ is then 'stuck' on the two faces of the box using the 'minimax' coordinates of the box and points in $T(u, v)$ are projected onto the object using a parallel projection, with projectors normal to the plane of symmetry.

The second technique is to allow the artist to interact directly with the rendered version on the screen. The artist applies the texture using an interactive device simulating a brush and the effect on the screen is as if the painter was applying paint directly to the 3D object. It is easy to see the advantages of such a method by looking first at how it differs from a normal 2D paint program which basically enables a user to colour selected pixels on the screen.

Say we have a sphere (circle in screen space). With a normal paint program, if we selected, say, the colour green and painted the sphere, then unless we explicitly altered the colour, the sphere's projection would be filled with the selected uniform green colour. However, the idea of using a paint interaction in object space is that as you apply the green paint its colour changes according to the application of the Phong shading equation, and if the paint were gloss a specular highlight would appear. Extending the idea to texture mapping means

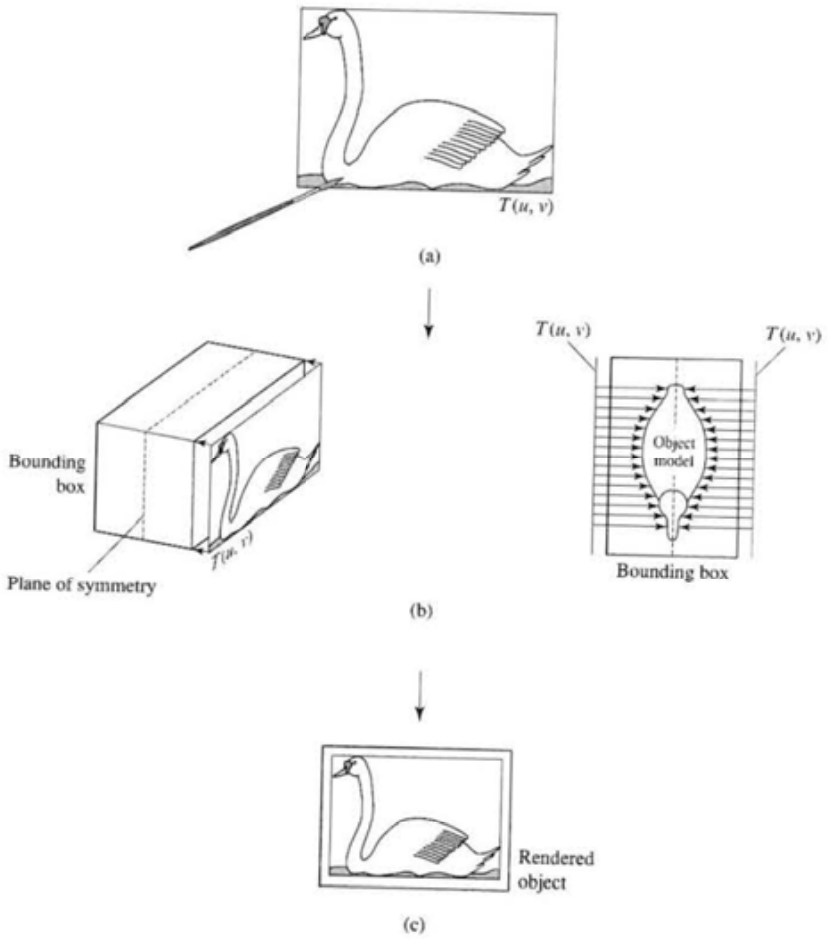


Figure 8.27
Interactive texture mapping – painting in $T(u,v)$ space. (a) Texture is painted using an interactive paint program. (b) Using the object's bounding box, the texture map points are projected onto the object. All projectors are parallel to each other and normal to the bounding box face. (c) The object is rendered, the 'distortion' visualized and the artist repeats the cycle if necessary.

that the artist can paint the texture on the object directly and the program, reversing the normal texture mapping procedure, can derive the texture map from the object. Once the process is complete, new views of the object can be rendered and texture mapped in the normal way.

This approach requires a technique that identifies, from the screen pixel that is being pointed to, the corresponding point on the object surface. In the method described by Hanrahan and Haeberli (1990) an auxiliary frame buffer, known as an item buffer, is used. Accessing this buffer with the coordinates of the screen cursor gives a pointer to the position on the object surface and the corresponding (u, v) coordinate values for the texture map. Clearly we need an object representation where the surface is everywhere parametrized and Hanrahan and Haeberli (1990) divide the object surface into a large number of micropolygons. The overall idea is illustrated in Figure 8.28.

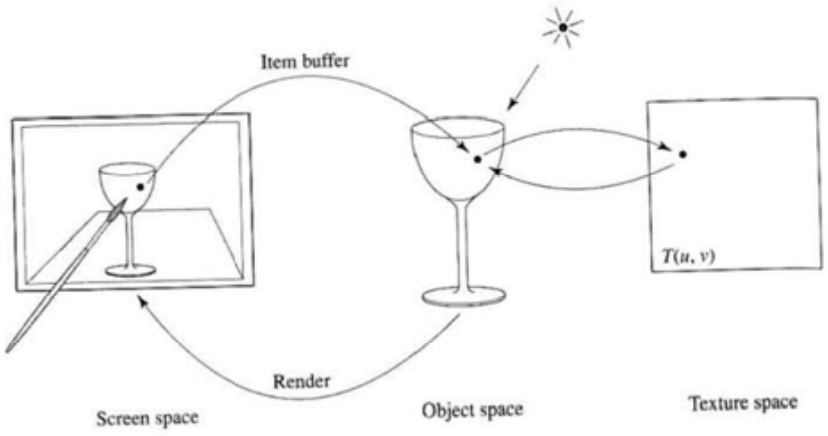


Figure 8.28
Iterative texture mapping – painting in object space.

9 Geometric shadows

Introduction

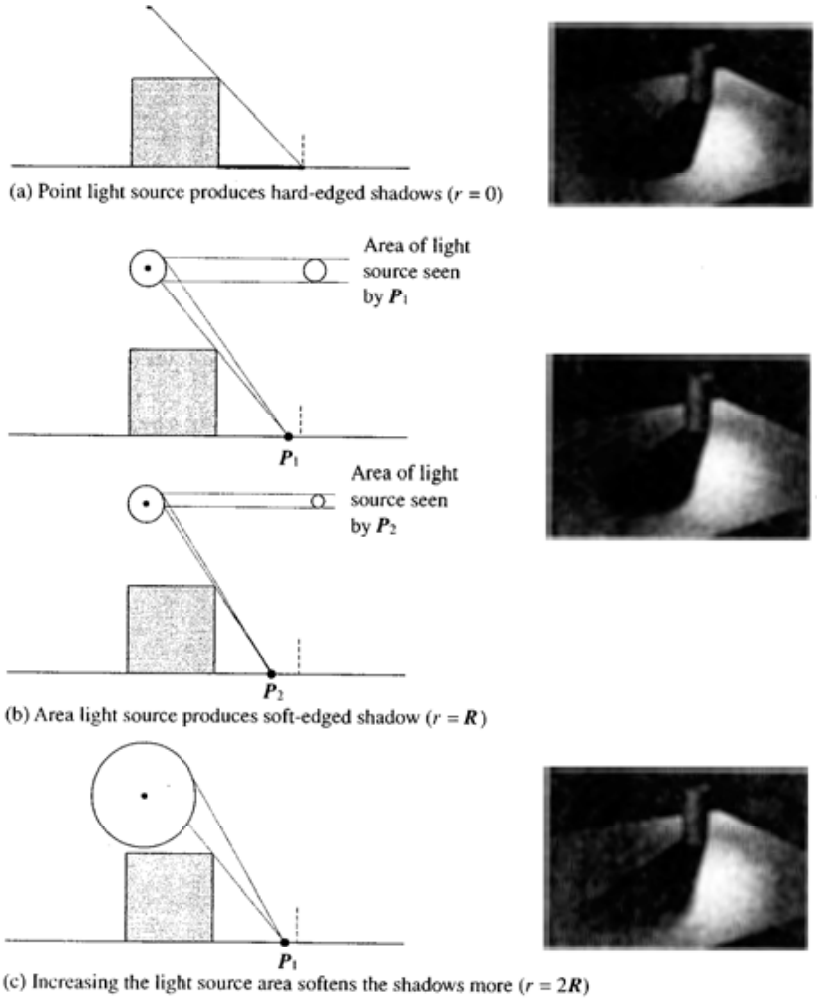
This chapter deals with the topic of 'geometric' shadows or algorithms that calculate the shape of an area in shadow but only guess at its reflected light intensity. This restriction has long been tolerated in mainstream rendering; the rationale presumably being that it is better to have a shadow with a guessed intensity than to have no shadow at all.

Shadows like texture mapping are commonly handled by using an empirical add-on algorithm. They are pasted into the scene like texture maps. The other parallel with texture maps is that the easiest algorithm to use computes a map for each light source in the scene, known as a shadow map. The map is accessed during rendering just as a texture map is referenced to find out if a pixel is in shadow or not. Like the Z-buffer algorithm in hidden surface removal, this algorithm is easy to implement and has become a pseudo-standard. Also like the Z-buffer algorithm it trades simplicity against high memory cost.

Shadows are important in scenes. A scene without shadows looks artificial. They give clues concerning the scene, consolidate spatial relationships between objects and give information on the position of the light source. To compute shadows completely we need knowledge both of their shape and the light intensity inside them. An area of the scene in shadow is not completely bereft of light. It is simply not subject to direct illumination, but receives indirect illumination from another nearby object. Thus shadow intensity can only be calculated taking this into account and this means using a global illumination model such as radiosity. In this algorithm (see Chapter 11) shadow areas are treated no differently from any other area in the scene and the shadow intensity is a light intensity, reflected from a surface, like any other.

Shadows are a function of the lighting environment. They can be hard edged or soft edged and contain both an umbra and a penumbra area. The relative size

of the umbra/penumbra is a function of the size and the shape of the light source and its distance from the object (Figure 9.1). The umbra is that part of a shadow that is completely cut off from the light source, whereas the penumbra is an area that receives some light from the source. A penumbra surrounds an umbra and there is always a gradual change in intensity from a penumbra to an umbra. In computer graphics, if we are not modelling illumination sources, then we usually consider point light sources at large distances, and assume in the simplest case that objects produce umbrae with sharp edges. This is still only an approximation. Even although light from a large distance produces almost parallel rays, there is still light behind the object due to diffraction and the shadow grades off. This effect also varies over the distance a shadow is thrown. These effects, that determine the quality of a shadow, enable us to infer information



(a) Point light source produces hard-edged shadows ($r = 0$)

(b) Area light source produces soft-edged shadow ($r = R$)

(c) Increasing the light source area softens the shadows more ($r = 2R$)

Figure 9.1
Shadows cast by spherical light sources.

concerning the nature of the light source and they are clearly important to us as human beings perceiving a three-dimensional environment. For example, the shadows that we see outdoors depend on the time of day and whether the sky is overcast or not.

9.1 Properties of shadows used in computer graphics

A number of aspects of shadows are exploited in the computer generation of the phenomenon. These are:

- A shadow from polygon *A* that falls on polygon *B* due to a point light source can be calculated by projecting polygon *A* onto the plane that contains polygon *B*. The position of the point light source is used as the centre of projection.
- No shadows are seen if the view point is coincident with the (single) light source. An equivalent form of this statement is that shadows can be considered to be areas hidden from the light source, implying that modified hidden surface algorithms can be used to solve the shadow problem.
- If the light source, or sources, are point sources then there is no penumbra to calculate and the shadow has a hard edge.
- For static scenes, shadows are fixed and do not change as the view point changes. If the relative position of objects and light sources change, the shadows have to be re-calculated. This places a high overhead on three-dimensional animation where shadows are important for depth and movement perception.

Because of the high computational overheads, shadows have been regarded in much the same way as texture mapping – as a quality add-on. They have not been viewed as a necessity and compared with shading algorithms there has been little consideration of the quality of shadows. Most shadow generation algorithms produce hard edge point light source shadows and most algorithms deal only with polygon mesh models.

9.2 Simple shadows on a ground plane

An extremely simple method of generating shadows is reported by Blinn (1988). It suffices for single object scenes throwing shadows on a flat ground plane. The method simply involves drawing the projection of the object on the ground plane. It is thus restricted to single object scenes, or multi-object scenes where objects are sufficiently isolated so as not to cast shadows on each other. The ground plane projection is easily obtained from a linear transformation and the projected polygon can be scanned into a Z-buffer as part of an initialization procedure at an appropriate (dark) intensity.

If the usual illumination approximation is made – single point source at an infinite distance – then we have parallel light rays in a direction $\boldsymbol{L} = (x_l, y_l, z_l)$ as shown in Figure 9.2. Any point on the object $\boldsymbol{P} = (x_p, y_p, z_p)$ will cast a shadow at $\boldsymbol{S} = (x_{sw}, y_{sw}, 0)$. Considering the geometry in the figure, we have:

$$\boldsymbol{S} = \boldsymbol{P} - \alpha \boldsymbol{L}$$

and given that $z_{sw} = 0$, we have:

$$0 = z_p - \alpha z_l$$

$$\alpha = z_p / z_l$$

and:

$$x_{sw} = x_p - (z_p/z_l)\, x_l$$

$$y_{sw} = y_p - (z_p/z_l)\, y_l$$

As a homogeneous transformation this is

$$\begin{bmatrix} x_{sw} \\ y_{sw} \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & -x_l/z_l & 0 \\ 0 & 1 & -y_l/z_l & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_p \\ y_p \\ z_p \\ 1 \end{bmatrix}$$

Note from this that it is just as easy to generate shadows on a vertical back or side plane. Blinn also shows how to extend this idea to handle light sources that are at a finite distance from the object.

This type of approximate shadow (on a flat ground plane) is beloved by traditional animators and its use certainly enhances movement in three-dimensional computer animation.

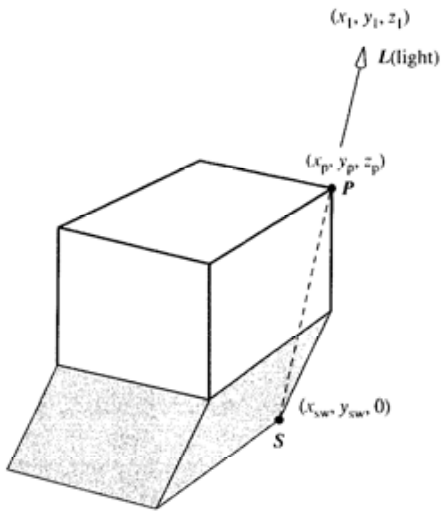


Figure 9.2
Ground plane shadows for single objects.

9.3 Shadow algorithms

Unlike hidden surface removal algorithms, where one or two algorithms now predominate and other methods are only used in special cases, no popular candidate has emerged as the top shadow algorithm. In fact, shadow computation is a rather neglected area of computer graphics. What follows, therefore, is a brief description of four major approaches. Shadow generation in ray tracing is separately described in Chapter 12.

9.3.1 Shadow algorithms: projecting polygons/scan line

This approach was developed by Appel (1968) and Bouknight and Kelley (1970). Adding shadows to a scan line algorithm requires a pre-processing stage that builds up a secondary data structure which links all polygons that may shadow a given polygon. Shadow pairs – a polygon together with the polygon that it can possibly shadow – are detected by projecting all polygons onto a sphere centred at the light source. Polygon pairs that cannot interact are detected and discarded. This is an important step because for a scene containing n polygons the number of possible projected shadows is $n(n-1)$.

The algorithm processes the secondary data structure simultaneously with a normal scan conversion process to determine if any shadows fall on the polygon that generated the visible scan line segment under consideration. If no shadow polygon(s) exists then the scan line algorithm proceeds as normal. For a current polygon: if a shadow polygon exists then using the light source as a centre of projection, the shadow is generated by projecting onto the plane that contains the current polygon. Normal scan conversion then proceeds simultaneously with a process that determines whether a current pixel is in shadow or not. Three possibilities now occur:

(1) The shadow polygon does not cover the generated scan line segment and the situation is identical to an algorithm without shadows.

(2) Shadow polygons completely cover the visible scan line segment and the scan conversion process proceeds but the pixel intensity is modulated by an amount that depends on the number of shadows that are covering the segment. For a single light source the segment is either in shadow or is not.

(3) A shadow polygon partially covers the visible scan line segment. In this case the segment is subdivided and the process is applied recursively until a solution is obtained.

A representation of these possibilities is shown in Figure 9.3. These are, in order along the scan line:

(a) Polygon A is visible, therefore it is rendered.

(b) Polygon B is visible and is rendered.

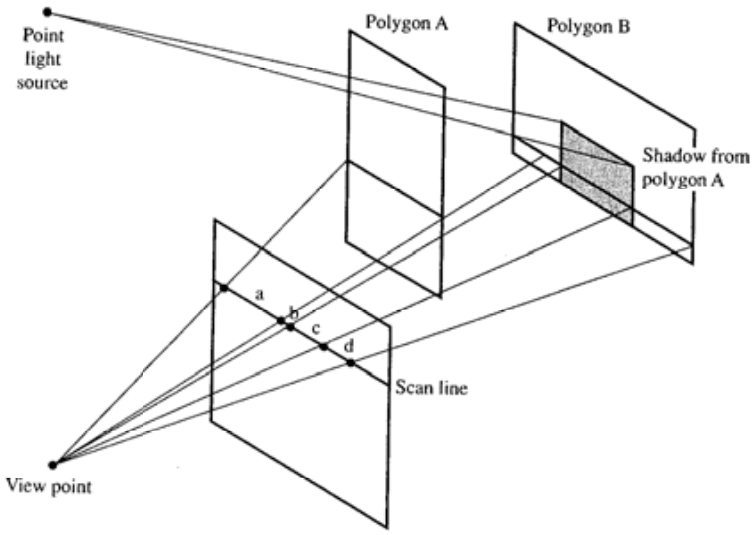


Figure 9.3
Polygons that receive a shadow from another polygon are linked in a secondary data structure. Scan line segments are now delineated by both view point projection boundaries and shadow boundaries.

(c) Polygon *B* is shadowed by polygon A and is rendered at an appropriately reduced intensity.

(d) Polygon *B* is visible and is rendered.

9.3.2 Shadow algorithms: shadow volumes

The shadow volume approach was originally developed by Crow (1977) and subsequently extended by others. In particular Brotman and Badler (1984) used the idea as a basis for generating 'soft' shadows – that is, shadows produced by a distributed light source.

A shadow volume is the invisible volume of space swept out by the shadow of an object. It is the infinite volume defined by lines emanating from a point light source through vertices in the object. Figure 9.4 conveys the idea of a shadow volume. A finite shadow volume is obtained by considering the intersection of the infinite volume with the view volume. The shadow volume is computed by first evaluating the contour or silhouette edge of the object, as seen from the light source. The contour edge of a simple object is shown in Figure 9.4(a). A contour edge of an object is the edge made up of one or more connected edges of polygons belonging to the object. A contour edge separates those polygons that can receive light from the light source from those that cannot.

Polygons defined by the light source and the contour edges define the bounding surface of the shadow volume as shown in Figure 9.4(b). Thus each object, considered in conjunction with a point light source, generates a shadow volume object that is made up of a set of shadow polygons. Note that these shadow polygons are 'invisible' and should not be confused with the visible shadow polygons described in the next section. These shadow polygons are themselves used to determine shadows – they are not rendered.

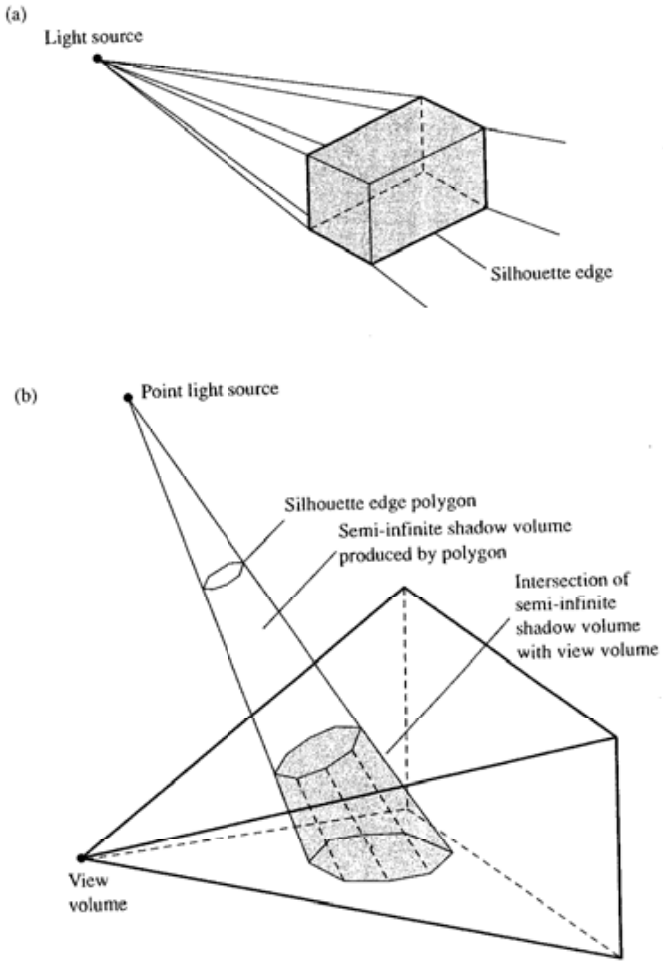


Figure 9.4
Illustrating the formation of a shadow volume. (a) Silhouette edge of an object. (b) Finite shadow volume defined by a silhouette edge polygon, a point light source and a view volume.

This scheme can be integrated into a number of hidden surface removal algorithms and the polygons that define the shadow volume are processed along with the object polygons except that they are considered invisible. A distinction is made between 'front-facing' polygons and 'back-facing' polygons and the relationship between shadow polygons labelled in this way and object polygons is examined. A point on an object is deemed to be in shadow if it is behind a front-facing shadow polygon and in front of a back-facing polygon. That is, if it is contained within a shadow volume. Thus a front-facing shadow polygon puts anything behind it in shadow and a back-facing shadow polygon cancels the effect of a front-facing one.

As it stands, the algorithm is most easily integrated with a depth priority hidden surface removal algorithm. Consider the operation of the algorithm for a particular pixel. We consider a vector or ray from the view point through the

pixel and look at the relationship between real polygons and shadow polygons along this vector. For a pixel a counter is maintained. This is initialized to 1 if the view point is already in shadow, 0 otherwise. As we descend the depth sorted list of polygons, the counter is incremented when a front-facing polygon is passed and decremented when a back-facing polygon is passed. The value of this counter tells us, when we encounter a real polygon, whether we are inside a shadow volume. This is shown schematically in Figure 9.5.

Brotman and Badler (1984) use an enhanced Z-buffer algorithm and this approach has two significant advantages:

(1) The benefits of the Z-buffer rendering approach are retained.

(2) Their method is able to compute soft shadows or umbra/penumbra effects.

The price to be paid for using a shadow volume approach in conjunction with a Z-buffer is memory cost. The Z-buffer has to be extended such that each pixel location is a record of five fields. As shadow polygons are 'rendered' they modify counters in a pixel record and a decision can be made as to whether a point is in shadow or not.

Soft shadows are computed by modelling distributed light sources as arrays of point sources and linearly combining computations due to each point source.

The original shadow volume approach places heavy constraints on the database environment; the most serious restriction is that objects must be convex polyhedrons. Bergeron (1986) developed a general version of Crow's algorithm that overcomes these restrictions and allows concave objects and penetrating polygons.

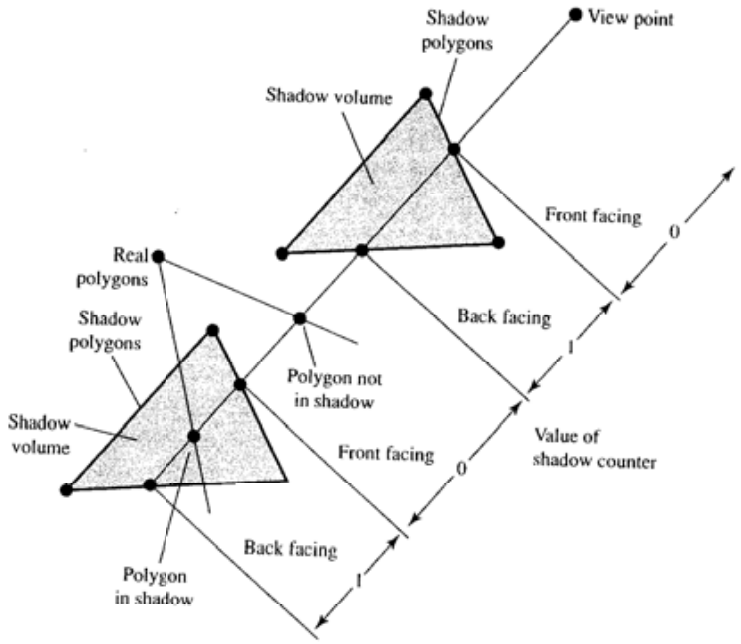


Figure 9.5
Front-facing and back-facing shadow polygons and the shadow counter value.

9.3.3 Shadow algorithms: derivation of shadow polygons from light source transformations

This approach was developed by Atherton *et al.* (1978) and relies on the fact that applying hidden surface removal to a view from the light source produces polygons or parts of polygons that are in shadow. It also relies on the object space polygon clipping algorithm (to produce shadow polygons that are parts of existing polygons) by Weiler and Atherton (1977).

A claimed advantage of this approach is that it operates in object space. This means that it is possible to extract numerical information on shadows from the algorithm. This finds applications, for example, in architectural CAD.

The algorithm enhances the object data structure with shadow polygons to produce a 'complete shadow data file'. This can then be used to produce any view of the object with shadows. It is thus a good approach in generating animated sequences where the virtual camera changes position but the relative position of the object and the light source remain unchanged. The working of the algorithm is shown for a simple example in Figure 9.6. A single shadow polygon is shown for clarity. Referring to Figure 9.6, the first step in the algorithm is to apply a transformation such that the object or scene is viewed from the light source position. Hidden surface removal then produces visible polygons, that is polygons that are visible to the light source and are therefore not in shadow. These are either complete or clipped as the illustration implies. This polygon set can then be combined with the original object polygons, provided both data sets are in the same coordinate system. The process of combining these sets results in a complete shadow data file – the original polygon set enhanced by shadow polygons for a particular light source. Transforming the database to the required view point and applying hidden surface removal will then result in an image with shadows. This algorithm exploits the fact that shadow polygons are view point independent. Essentially the scene is processed twice for hidden surface removal. Once using the light source as a view point, which produces the shadow polygons, and once using normal hidden surface removal (from any view point).

9.3.4 Shadow algorithms: shadow Z-buffer

Possibly the simplest approach to the shadow computation, and one that is easily integrated into a Z-buffer-based renderer is the shadow Z-buffer developed by Williams (1978). This technique requires a separate shadow Z-buffer for each light source and in its basic form is only suitable for a scene illuminated by a single light source. Alternatively a single shadow Z-buffer could be used for many light sources and the algorithm executed for each light source, but this would be somewhat inefficient and slow.

The algorithm is a two-step process. A scene is 'rendered' and depth information stored in the shadow Z-buffer using the light source as a view point. No

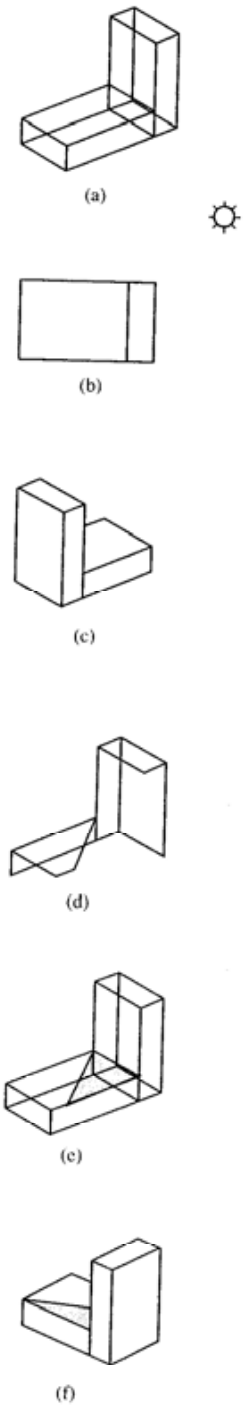


Figure 9.6
Derivation of shadow polygons from transformations. (a) Simple polygonal object in modelling coordinate system. (b) Plan view showing the position of the light source. (c) Hidden surface removal from the light source as a view point. (d) Visible polygons from (c) transformed back into modelling coordinate system. (e) Parts (a) and (d) merged to produce a database that contains shadow polygons. (f) Part (e) can produce any view of the object with shadows.

intensities are calculated. This computes a 'depth image' from the light source, of these polygons that are visible to the light source.

The second step is to render the scene using a Z-buffer algorithm. This process is enhanced as follows: if a point is visible, a coordinate transformation is used to map (x, y, z), the coordinates of the point in three-dimensional screen space (from the view point) to (x', y', z'), the coordinates of the point in screen space from the light point as a coordinate origin. The (x', y') are used to index the shadow Z-buffer and the corresponding depth value is compared with z'. If z' is greater than the value stored in the shadow Z-buffer for that point, then a surface is nearer to the light source than the point under consideration and the point is in shadow, thus a shadow 'intensity' is used, otherwise the point is rendered as normal. An example of shadow maps is shown in Figure 18.8. Note that in this particular example we have generated six shadow maps. This enables us to render a view of the room from a view point situated anywhere within the scene.

Apart from extending the high memory requirements of the Z-buffer hidden surface removal algorithm, the algorithm also extends its inefficiency. Shadow calculations are performed for surfaces that may subsequently be 'overwritten' – just as shading calculations are.

Anti-aliasing and the shadow Z-buffer

In common with the Z-buffer algorithm, the shadow Z-buffer is susceptible to aliasing artefacts due to point sampling. Two aliasing opportunities occur. First, straightforward point sampling in the creation phase of the shadow Z-buffer produces artefacts. These will be visible along shadow edges – we are considering a hard-edged shadow cast by a point light source. The second aliasing problem is created when accessing the shadow Z-buffer. It is somewhat analogous to the sampling problem created in texture mapping. This problem arises because we are effectively projecting a pixel extent onto the shadow Z-buffer map. This is shown schematically in Figure 9.7. If we consider the so-called pre-image of a square pixel in the shadow Z-buffer map then this will, in general, be a quadrilateral that encloses a number of shadow Z-buffer pixels. It is this many map pixels to one screen pixel problem that we have to deal with. It means that a pixel may be partly in shadow and partly not and if we make a binary decision then aliasing will occur. We thus consider the fraction of the pixel that is in shadow by computing this from the shadow Z-buffer. This fraction can be evaluated by the z' comparisons over the set of shadow Z-buffer pixels that the screen pixel projects onto. The fraction is then used to give an appropriate shadow intensity. The process in summary is:

(1) For each pixel calculate four values of (x', y') corresponding to the four corner points. This defines a quadrilateral in shadow Z-buffer space.

(2) Integrate the information over this quadrilateral by comparing the z value for the screen pixel with each z' value in the shadow Z-buffer quadrilateral. This gives a fraction that reflects the area of the pixel in shadow.

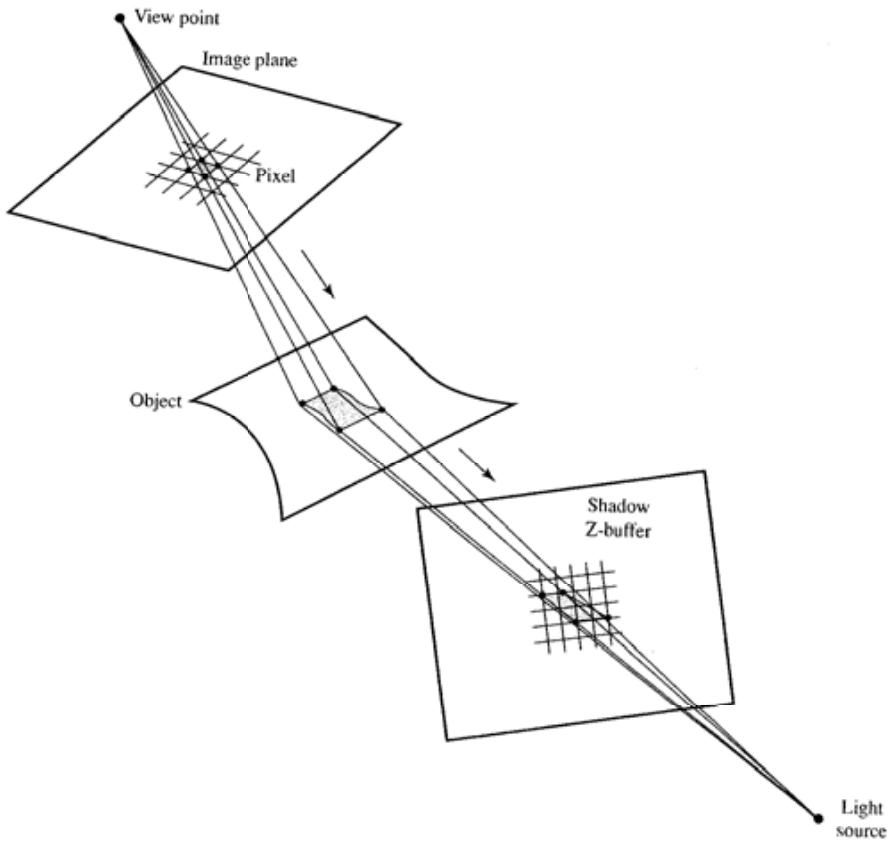


Figure 9.7
'Pre-image' of a pixel in the shadow Z-buffer.

(3) We use this fraction to give an appropriate attenuated intensity. The visual effect of this is that the hard edge of the shadow will be softened for those pixels that straddle a shadow boundary.

Full details of this approach are given in Reeves *et al.* (1987). The price paid for this anti-aliasing is a considerable increase in processing time. Pre-filtering techniques (see Chapter 14) cannot be used and a stochastic sampling scheme for integrating within the pixel pre-image in the shadow Z-buffer map is suggested in Reeves *et al.* (1987).

10 Global illumination

Introduction

In computer graphics, global illumination is the term given to models which render a view of a scene by evaluating the light reflected from a point x taking into account all illumination that arrives at a point. That is we consider not only the light arriving at the point directly from light sources but all indirect illumination that may have originated from a light source via other objects.

It is probably the case that in the general pursuit of photo-realism, most research effort has gone into solving the global illumination problem. Although, as we have seen in Chapter 7, considerable parallel work has been carried out with local reflection models, workers have been attracted to the difficult problem of simulating the interaction of light with an entire environment. Light has to be tracked through the environment from emitter(s) to sensor(s), rather than just from an emitter to a surface then directly to the sensor or eye. Such an approach does not then require add-on algorithms for shadows which are simply areas in which the illumination level is reduced due to the proximity of a

nearby object. Other global illumination effects such as reflection of objects in each other and transparency effects can also be correctly modelled.

It is not clear how important global illumination is to photo-realism. Certainly it is the case that we are accustomed to 'closed' man-made environments, where there is much global interaction, but the extent to which this interaction has to be simulated, to achieve a degree of realism acceptable for most computer graphics applications, is still an open question. Rather, the problem has been vigorously pursued as a pure research problem in its own right on the assumption that improvements in the accuracy of global interaction will be valuable.

Two established (partial) global algorithms have now emerged. These are ray tracing and radiosity and, for reasons that will soon become clear, they both, in their most commonly implemented forms, simulate only a subset of global interaction: ray tracing attending to (perfect) specular interaction and radiosity to (perfect) diffuse interaction. In other words, current practical solutions to the problem deal with its inherent intractability by concentrating on particular global interactions, ignoring the remainder and by considering interactions to be perfect. In the case of specular interaction 'perfect' means that an infinitesimally thin beam hitting a surface reflects without spreading – the surface is assumed perfect. In the case of perfect diffuse interaction we assume that an incoming beam of light reflects equally in all directions into the hemisphere centred at the point of reflection.

Ignoring finite computing resources, a solution to the global interaction problem is simply stated. We start at the light source(s) and follow every light path (or ray of light) as it travels through the environment stopping when the light hits the eye point, has its energy reduced below some minimum due to absorption in the objects that it has encountered, or travels out of the environment into space. To see the relevance of global illumination algorithms we need ways of describing the problem – models that capture the essence of the behaviour of light in an environment. In this chapter we will introduce two models of global illumination and give an overview of the many and varied approaches to global illumination. We devote separate chapters to the implementation details of the two well-established methods of ray tracing and radiosity.

We should note that it is difficult to categorize global illumination algorithms because most use a combination of techniques. Is two-pass ray tracing, for example, to be considered as a global illumination method or as an extension to ray tracing? Thus the breakdown by technique that appears in this chapter inevitably contains algorithms that straddle more than one category and the sorting is simply the author's preference.

10.1 Global illumination models

We start by introducing two 'models' of the global illumination problem. The first is a mathematical formulation and the second is a classification in terms of the nature of the type of interaction that can occur when light travels from one

surface to the other. The value of such models is that they enable a comparison between the multitude of global illumination algorithms most of which evaluate a less than complete solution. By their nature the algorithms consist of a wealth of heuristic detail and the global illumination models facilitate a comparison in terms of which aspects are evaluated and which are not.

10.1.1

The rendering equation

The first model that we will look at was introduced into the computer graphics literature in 1986 by Kajiya (Kajiya 1986) and is known as the rendering equation. It encapsulates global illumination by describing what happens at a point x on a surface. It is a completely general mathematical statement of the problem and global illumination algorithms can be categorized in terms of this equation. In fact, Kajiya states that its purpose:

> is to provide a unified context for viewing them [rendering algorithms] as more or less accurate approximations to the solution for a single equation.

The integral in Kajiya's original notation is given by:

$$I(x, x') = g(x, x')[\varepsilon(x, x') + \int_s \rho(x, x', x'')\, I(x', x'')dx'']$$

where:

$I(x, x')$ is the transport intensity or the intensity of light passing from point x' to point x. Kajiya terms this the unoccluded two point transport intensity.

$g(x, x')$ is the visibility function between x and x'. If x and x' cannot 'see' each other then this is zero. If they are visible then g varies as the inverse square of the distance between them.

$\varepsilon(x, x')$ is the transfer emittance from x' to x and is related to the intesity of any light self-emitted by point x' in the direction of x.

$\rho(x, x', x'')$ is the scattering term with respect to direction x' and x''. It is the intensity of the energy scattered towards x by a surface point located at x' arriving from point or direction x''. Kajiya calls this the unoccluded three-point transport reflectance. It is related to the BRDF (see Chapter 7) by:

$$\rho(x, x', x'') = \rho(\theta'_{in}, \phi'_{in}, \theta'_{ref}, \phi'_{ref}) \cos\theta \cos\theta'_{ref}$$

where θ' and ϕ' are the azimuth and elevation angles related to point x' (see Section 7.3) and θ is the angle between the surface normal at point x and the line $x'x$.

The integral is over s, all points on all surfaces in the scene, or equivalently over all points on the hemisphere situated at point x'. The equation states that the transport intensity from point x' to point x is equal to (any) light emitted from x' towards x plus the light scattered from x' towards x from all other surfaces in the scene – that is, that originate from direction x''.

Expressed in the above terms the rendering equation implies that we must have:

- A model of the light emitted by a surface ε().
- A representation of the BRDF ρ() for each surface.
- A method for evaluating the visibility function.

We have already met all these factors; here the formulation gathers them into a single equation. The important general points that come out of considering the rendering equation are:

(1) The complexity of the integral means that it cannot be evaluated analytically and most practical algorithms reduce the complexity in some way. The direct evaluation of the equation can be undertaken by using Monte Carlo methods and many algorithms follow this approach.

(2) It is a view-independent statement of the problem. The point x' is every point in the scene. Global illumination algorithms are either view independent – the common example is the radiosity algorithm – or view dependent where only those points x' visible from the viewing position are evaluated. View dependence can be seen as a way in which the inherent complexity of the rendering equation is reduced. (See Section 10.8 for a more detailed discussion on view dependence/independence.)

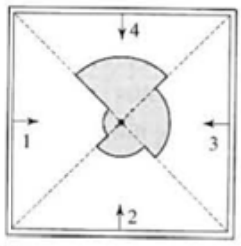
(3) It is a recursive equation – to evaluate $I(x, x')$ we need to evaluate $I(x', x'')$ which itself will use the same equation. This gives rise to one of the most popular practical methods for solving the problem which is to trace light from the image plane, in the reverse direction of light propagation, following a path that reflects from object to object. Algorithms that adopt this approach are: path tracing, ray tracing and distributed ray tracing, all of which will be described later.

10.1.2 Radiance, irradiance and the radiance equation

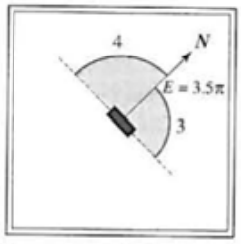
The original form of the rendering equation is not particularly useful in global illumination methods and in this section we will introduce definitions that enable us to write it in a different form called the radiance equation.

Radiance L is the fundamental radiometric quantity and for a point in three-dimensional space it is the light energy density measured in W/(sr–m^2). The radiance at a point is a function of direction and we can define a radiance distribution function for a point. This will generally be discontinuous as the two-dimensional example in Figure 10.1 demonstrates. Such a distribution function exists at all points in three-dimensional space and radiance is therefore a five-dimensional quantity. Irradiance is the integration of incoming radiance over all directions:

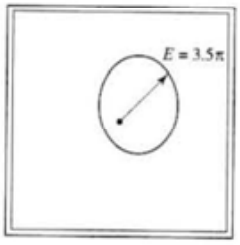
$$E = \int_{\Omega} L_{in} \cos\theta \, d\omega$$



(a)



(b)



(c)

Figure 10.1
Radiance, irradiance and irradiance distribution function (after Greger *et al.* (1998)).

(a) A two-dimensional radiance distribution for a point in the centre of a room where each wall exhibits a different radiance.

(b) The field radiance for a point on a surface element. Irradiance E is the cosine weighted average of the radiance – in this case 3.5 π.

(c) If the surface element is rotated an irradiance distribution function is defined.

where:

L_{in} is the incoming or field radiance from direction ω
θ is the angle between the surface normal and ω

If L_{in} is constant, we have for a diffuse surface:

$$L_{diffuse} = \rho E/\pi$$

The distinction between these two quantities is important in global illumination algorithms as the form of the algorithm can be classified as 'shooting' or 'gathering'. Shooting means distributing radiance from a surface and gathering means integrating the irradiance or accumulating light flux at the surface. (Radiosity B is closely related to irradiance having units W/m^2.)

An important practical point concerning radiance and irradiance distribution functions is that while the former is generally discontinuous the latter is generally continuous, except for shadow boundaries. This is demonstrated in Figure 10.1 which shows that in this simple example the irradiance distribution function will be continuous because of the averaging effect of the integration.

The rendering equation can be recast as the radiance equation which in its simplest form is:

$$L_{ref} = \int \rho L_{in}$$

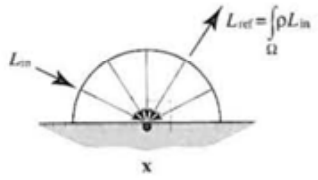
Including the directional dependence, we then write:

$$L_{ref}(\mathbf{x}, \omega_{ref}) = L_e(\mathbf{x}, \omega_{ref}) + \int_{\Omega} \rho(\mathbf{x}, \omega_{in} \rightarrow \omega_{out}) L_{in}(\mathbf{x}, \omega_{in}) \cos\theta_{in}\, d\omega_{in}$$

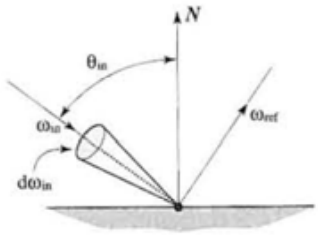
where the symbols are defined in Figure 10.2(b). This can be modified so that the integration is performed over all surfaces – usually more convenient in practical algorithms – rather than all incoming angles and this gives the rendering equation in terms of radiance:

$$L_{ref}(\mathbf{x}, \omega_{ref}) = L_e(\mathbf{x}, \omega_{ref}) + \int_{S} \rho(\mathbf{x}, \omega_{in} \rightarrow \omega_{out}) L_{in}(\mathbf{x}', \omega_{in}) g(\mathbf{x}, \mathbf{x}') \cos\theta_{in} \frac{\cos\theta_0\, dA}{\|\mathbf{x} - \mathbf{x}'\|^2}$$

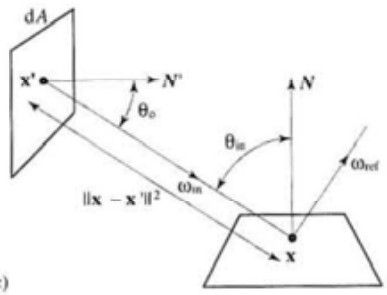
which now includes the visibility function. This comes about by expressing the solid angle $d\omega_{in}$ in terms of the projected area of the differential surface region visible in the direction of ω_{in} (Figure 10.2(c)):



(a)



(b)



(c)

Figure 10.2
The radiance equation.

(a) The domain of integration is the hemisphere of all incoming directions.

(b) Symbols used to define the directional dependence.

(c) $\cos\theta_0\, dA/\|\mathbf{x} - \mathbf{x}'\|^2$ is the projected area of dA visible in the direction ω_{in}.

$$d\omega_{in} = \frac{\cos\theta_0 \, dA}{\|\mathbf{x} - \mathbf{x}'\|^2}$$

10.1.3 Path notation

Another way of categorizing the behaviour of global illumination algorithms is to detail which surface-to-surface interactions that they implement or simulate. This is a much simpler non-mathematical categorization and it enables an easy comparison and classification of the common algorithms. We consider which interactions between pairs of interacting surfaces are implemented as light travels from source to sensor. Thus at a point, incoming light may be scattered or reflected diffusely or specularly and may itself have originated from a specular or diffuse reflection at the previous surface in the path. We can then say that for pairs of consecutive surfaces along a light path we have (Figure 10.3):

- Diffuse to diffuse transfer.
- Specular to diffuse transfer.
- Diffuse to specular transfer.
- Specular to specular transfer.

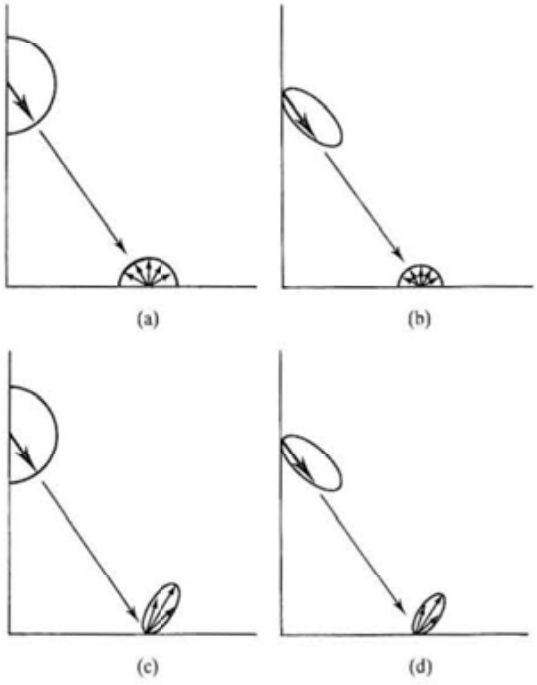


Figure 10.3
The four 'mechanisms' of light transport: (a) diffuse to diffuse; (b) specular to diffuse; (c) diffuse to specular; (d) specular to specular (after Wallace *et al.* (1987)).

In an environment where only diffuse surfaces exist only diffuse–diffuse interaction is possible and such scenes are solved using the radiosity method. Similarly an environment containing only specular surfaces can only exhibit specular interaction and (Whitted) ray tracing deals with these. Basic radiosity does not admit any other transfer mechanism except diffuse–diffuse and it excludes the important specular–specular transfer. Ray tracing, on the other hand can only deal with specular–specular interaction. More recent algorithms, such as 'backwards' ray tracing and enhancements of radiosity for specular interaction require a categorization of all the interactions in a light journey from source to sensor, and this led to Heckbert's string notation (Heckbert 1990) for listing all the interactions that occur along a path of a light ray as it travels from source (L) to the eye (E). Here a light path from the light source to the first hit is termed L, subsequent paths involving transfer mechanisms at a surface point are categorized as DD, SD, DS or SS. Figure 10.4 (also a Colour Plate) shows an example of a simple scene and various paths. The path that finally terminates in the eye is called E. The paths in the example are:

(1) LDDE For this path the viewer sees the shadow cast by the table. The light reflects diffusely from the right-hand wall onto the floor. Note that any light

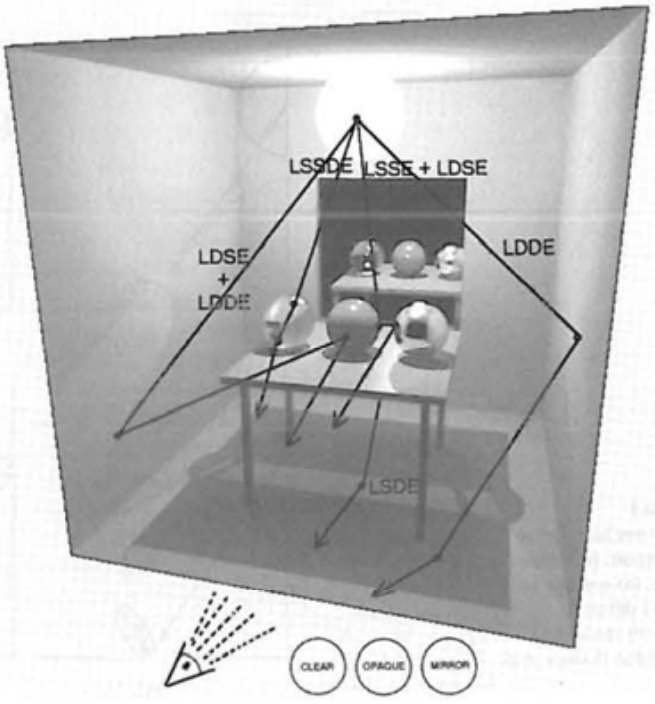


Figure 10.4
A selection of global illuminations paths in a simple environment. See also the Colour Plate version of this figure.

reflected from a shadow area must have a minimum of two interactions between L and E.

(2) LDSE + LDDE Here the user sees the dark side of the sphere which is not receiving any direct light. The light is modelled as a point source, so any area below the 'equator' of the sphere will be in shadow. The diffuse illumination reflected diffusely from the wall is directed towards the eye and because the sphere is shiny the reflection to the eye is both specular and diffuse.

(3) LSSE + LDSE Light is reflected from the perfect mirror surface to the eye and the viewer sees a reflection of the opaque or coloured ball in the mirror surface.

(4) LSDE Here the viewer sees a shadow area that is lighter than the main table shadow. This is due to the extra light reflected from the mirror and directed underneath the table.

(5) LSSDE This path has three interactions between L and E and the user sees a caustic on the table top which is a diffuse surface. The first specular interaction takes place at the top surface of the sphere and light from the point source is refracted through the sphere. There is a second specular interaction when the light emerges from the sphere and hits the diffuse table surface. The effect of the reflection is to concentrate light rays travelling through the sphere into a smaller area on the table top than they would occupy if the transparent sphere was not present. Thus the user sees a bright area on the diffuse surface.

A complete global illumination algorithm would have to include any light path which can be written as L(D|S)*E, where | means 'or' and * indicates repetition. The application of a local reflection model implies paths of type LD|S (the intensity of each being calculated separately then combined as in the Phong reflection model) and the addition of a hidden surface removal algorithm implies simulation of types LD|SE. Thus local reflection models only simulate strings of length unity (between L and E) and viewing a point in shadow implies a string which is at least of length 2.

10.2 The evolution of global illumination algorithms

We will now look at the development of popular or established global illumination algorithms using as a basis for our discussion the preceding concepts. The order in which the algorithms are discussed is somewhat arbitrary; but goes from incomplete solutions (ray tracing and radiosity) to general solutions. The idea of this section is to give a view of the algorithms in terms of global interaction.

Return to consideration of the brute force solution to the problem. There we considered the notion of starting at a light source and following every ray of light that was emitted through the scene and stated that this was a computationally intractable problem. Approximations to a solution come from constraining the light-object interaction in some way and/or only considering a

small subset of the rays that start at the light and bounce around the scene. The main approximations which led to ray tracing and radiosity constrained the scene to contain only specular reflectors or only (perfect) diffuse reflectors respectively.

In what follows we give a review of ray tracing and radiosity sufficient for comparison with the other methods we describe, leaving the implementation details of these important methods for separate chapters.

10.3 Established algorithms – ray tracing and radiosity

10.3.1 Whitted ray tracing

Whitted ray tracing (visibility tracing, eye tracing) traces light rays in the reverse direction of propagation from the eye back into the scene towards the light source. To generate a two-dimensional image plane projection of a scene using ray tracing we are only interested in these light rays that end at the sensor or eye point and therefore it makes sense to start at the eye and trace rays out into the scene. It is thus a view-dependent algorithm. A simple representation of the algorithm is shown in Figure 10.5. The process is often visualized as a tree where each node is a surface hit point. At each node we spawn a light ray and a reflected ray or a transmitted (refracted) ray or both.

Whitted ray tracing is a hybrid – a global illumination model onto which is added a local model. Consider the global interaction. The classic algorithm only includes perfect specular interaction. Rays are shot into the scene and when they hit a surface a reflected (and transmitted) ray is spawned at the point of intersection and they themselves are then followed recursively. The process stops when

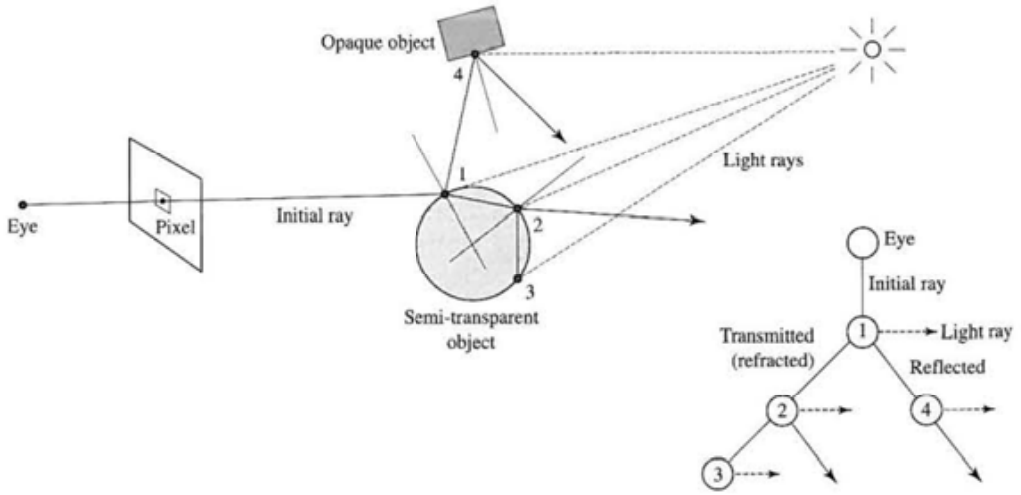


Figure 10.5
Whitted ray tracing.

the energy of a ray drops below a predetermined minimum or if it leaves the scene and travels out into empty space or if a ray hits a surface that is perfectly diffuse. Thus the global part of ray tracing only accounts for pure specular–specular interaction. Theoretically there is nothing to stop us calculating diffuse global interaction, it is just that at every hit point an incoming ray would have to spawn reflected rays in every direction into a hemispherical surface centred on the point.

To the global specular component is added a direct contribution calculated by shooting a ray from the point to the light source which is always a point source in this model. The visibility of the point from the light source and its direction can be used to calculate a local or direct diffuse component – the ray is just L in a local reflection model. Thus (direct) diffuse reflection (but not diffuse–diffuse) interaction is considered. This is sometimes called the shadow ray or shadow feeler because if it hits any object between the point under consideration and the light source then we know that the point is shadow. However, a better term is light ray to emphasize that it is used to calculate a direct contribution (using a local reflection model) which is then passed up the tree. The main problem with Whitted ray tracing is its restriction to specular interaction – most practical scenes consist of predominantly diffuse surfaces.

Consider the LSSE + LDSE path in Figure 10.4, reproduced in Figure 10.6 together with the ray tree. The initial ray from the eye hits the perfect mirror

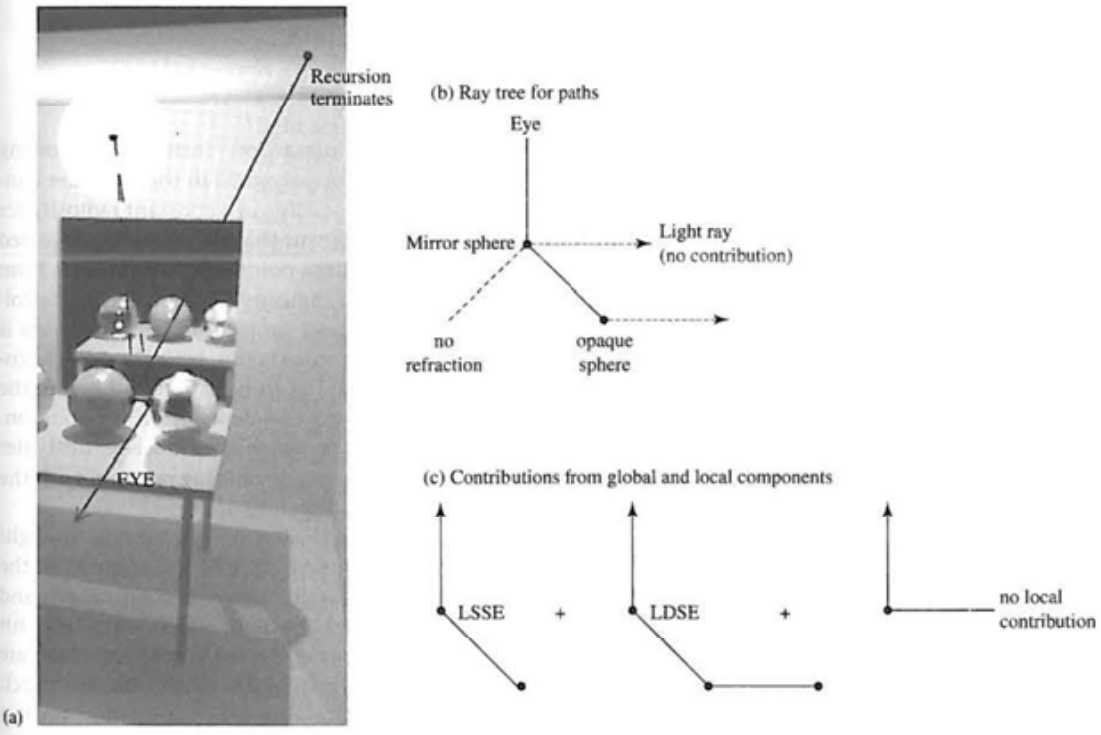


Figure 10.6
Whitted ray tracing: the relationship between light paths and local and global contributions for one of the cases shown in Figure 10.4.

sphere. For this sphere there is no contribution from a local diffuse model. At the next intersection we hit the opaque sphere and trace a global specular component which hits the ceiling, a perfect diffuse surface, and the recursion is terminated. Also at that point we have a contribution from the local diffuse model for the sphere and the viewer sees in the pixel associated with that ray the colour of the reflected image of the opaque sphere in the mirror sphere.

A little thought will reveal that the paths which can be simulated by Whitted ray tracing are constrained to be LS*E and LDS*E. Ray traced images therefore exhibit reflections in the surfaces of shiny objects of nearby objects. If the objects are transparent any objects that the viewer can see behind the transparent object are refracted. Also, as will be seen in Chapter 12, shadows are calculated as part of the model – but only 'perfect' or hard-edged shadows.

Considering Whitted ray tracing in terms of the rendering equation the following holds. The scattering term ρ is reduced to the law for perfect reflection (and refraction). Thus the integral over all S – the entire scene – reduces to calculating (for reflection) a single outgoing ray plus the light ray which gives the diffuse component and adding these two contributions together. Thus the recursive structure of the rendering equation is reflected perfectly in the algorithm but the integral operation is reduced to a sum of three analytically calculated components – the contributions from the reflected, transmitted and light rays.

10.3.2 Radiosity

Classic radiosity implements diffuse–diffuse interaction. Instead of following individual rays 'interaction' between patches (or polygons) in the scene are considered. The solution is view independent and consists of a constant radiosity for every patch in the scene. View independence means that a solution is calculated for every point in the scene rather than just those points that can be seen from the eye (view dependent). This implies that a radiosity solution has to be followed by another process or pass that computes a projection, but most work is carried out in the radiosity pass. A problem or contradiction with classical radiosity is that the initial discretization of the scene has to be carried out before the process is started but the best way of performing this depends on the solution. In other words, we do not know the best way to divide up the scene until after we have a solution or a partial solution. This is an outstanding problem with the radiosity method and accounts for most of its difficulty of use.

A way of visualizing the radiosity process is to start by considering the light source as an (array of) emitting patches. We shoot light into the scene from the source(s) and consider the diffuse–diffuse interaction between a light patch and all the receiving patches that are visible from the light patch – the first hit patches. An amount of light is deposited or cached on these patches which are then ordered according to the amount of energy that has fallen onto the patch and has yet to be shot back into the scene. The one with the highest unshot

energy is selected and this is considered as the next shooting patch. The process continues iteratively until a (high) percentage of the initial light energy is distributed around the scene. At any stage in the process some of the distributed energy will arrive back on patches that have already been considered and this is why the process is iterative. The process will eventually converge because the reflectivity coefficient associated with each patch is, by definition, less than unity and at each phase in the iteration more and more of the initial light is absorbed. Figure 10.7 (Colour Plate) shows a solution in progress using this algorithm. The stage shown is the state of the solution after 20 iterations. The four illustrations are:

(1) The radiosity solution as output from the iteration process. Each patch is allocated a constant radiosity.

(2) The previous solution after it has been subject to an interpolation process.

(3) The same solution with the addition of an ambient term. The ambient 'lift' is distributed evenly amongst all patches in the scene, to give an early well lit solution (this enhancement is described in detail in Chapter 11).

(4) 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.

The transfer of light between any two patches – the diffuse–diffuse interaction – is calculated by considering the geometric relationship between the patches (expressed as the form factor). Compared to ray tracing we follow light from the light source through the scene as patch-to-patch diffuse interaction, but instead of following individual rays of light, the form factor between two patches averages the effect of the paths that join the patches together. This way of considering the radiosity method is, in fact, implemented as an algorithm structure. It is called the progressive refinement method.

This simple concept has to be modified by a visibility process (not to be confused by the subsequent calculation of a projection which includes, in the normal way, hidden surface removal) that takes into account the fact that in general a patch may be only partially visible to another because of some intervening patch. The end result is the assignment of a constant radiosity to each patch in the scene – a view-independent solution which is then injected into a Gouraud-style renderer to produce a projection. In terms of path classification, conventional radiosity is LD*E.

The obvious problem with radiosity is that although man-made scenes usually consist mostly of diffuse surfaces, specular objects are not unusual and these cannot be handled by a radiosity renderer. A more subtle problem is that the scene has to be discretized into patches or polygons before the radiosities are computed and difficulties occur if this polygonization is too coarse.

We now consider radiosity in terms of the rendering equation. Radiosity is the energy per unit time per unit area and since we are only considering diffuse illumination we can rewrite the rendering equation as:

$$B(x') = \varepsilon(x') + \rho(x') \int_s B(x)F(x, x')dx$$

where now the only directional dependence is incorporated in the form factor F. The equation now states that the radiosity of a surface element x is equal to the emittance term plus the radiosity radiated by all other elements in the scene onto x. The form factor F is a coefficient that is a function only of the spatial relationship between x and x' and this determines that fraction of $B(x')$ arriving at x. F also includes a visibility calculation.

10.4 Monte Carlo techniques in global illumination

In this section we will give an intuitive introduction to Monte Carlo techniques. The mathematical details are outside the intended scope of this text (see Glassner (1995) for a comprehensive treatment of Monte Carlo theory and its application to global illumination) and it is the case that the methods that use Monte Carlo techniques can be explained in algorithmic terms. However, some intuition concerning the underlying factors is necessary to appreciate the particular strategies employed by the examples which we will describe. Without this intuition it is, for example, difficult to appreciate the difference between Whitted ray tracing and Kajiya's Monte Carlo approach which he termed path tracing (Section 10.5).

Monte Carlo techniques are used to solve integrals like the rendering equation which have no analytical or numerical solution. They do this by computing the average of random samples of the integrand, adding these together and taking the average. The visual effect of this process in the final rendered image is noise. The attraction of Monte Carlo techniques is that they are easy to implement because they are conceptually simple. An equally important advantage is their generality. No *a priori* simplifications have to be made (like perfect reflectors in Whitted ray tracing and perfect diffusers in radiosity). This comes about because they point sample both the geometry of the scene and the optical properties of the surface. The problem with Monte Carlo methods comes in devising techniques where an accurate or low variance estimate of the integral can be obtained quickly.

The underlying idea of Monte Carlo methods for estimating integrals can be demonstrated using a simple one-dimensional example. Consider estimating the integral:

$$I = \int_0^1 f(x)\, dx$$

I can be estimated by taking a random number $\xi \in [0,1]$ and evaluating $f(\xi)$. This is called a primary estimator. We can define the variance of the estimate as:

$$\sigma^2_{prim} = \int_0^1 f^2(x)dx - f^2(\xi)$$

which for a single sample we would expect to be high. In practice we would take N samples to give a so-called secondary estimate and it is easily shown that:

$$\sigma^2_{sec} = \frac{\sigma^2_{prim}}{N}$$

This observation, that the error in the estimate is inversely proportional to the square root of the number of samples, is extremely important in practice. To halve the error, for example, we must take four times as many samples. Equivalently we can say that each additional sample has less and less effect on the result and this has to be set against the fact that computer graphics implementations tend to involve an equal, and generally high cost, per sample. Thus the main goal in Monte Carlo methods is to get the best result possible with a given number of samples N. This means strategies that result in variance reduction. The two common strategies for selecting samples are stratified sampling and importance sampling.

The simplest form of stratified sampling divides the domain of the integration into equal strata and estimates each partial integral by one or more random samples (Figure 10.8). In this way each sub-domain is allocated the same number of samples. Thus:

$$I = \int_0^1 f(x)\,\mathrm{d}x$$

$$= \sum_{i=1}^{N} \int_{S_i} f(x)\mathrm{d}x$$

$$= \frac{1}{N} \sum_{i=1}^{N} f(\xi)$$

This estimate results in a variance that is guaranteed to be lower than that obtained by distributing random samples over the integration domain. The most familiar example of stratified sampling in computer graphics is jittering in pixel sampling. Here a pixel represents the domain of the integral which is subdivided into equal strata and a sample point generated by jittering the centre point of each stratum (Figure 10.9).

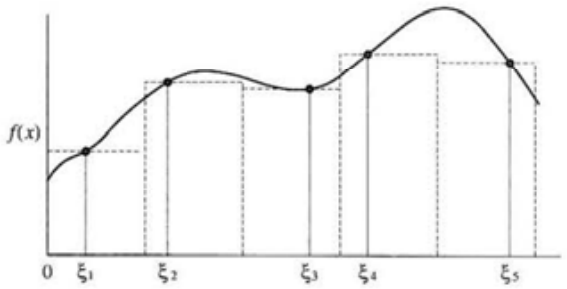


Figure 10.8
Stratified sampling of $f(x)$.

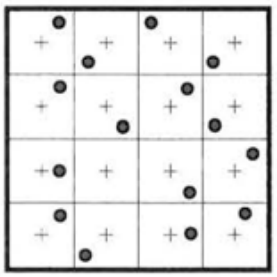


Figure 10.9
Stratified sampling in computer graphics: a pixel is divided into 16 sub-pixels and 16 sample points are generated by jittering the centre point of each sub-pixel or stratum.

As the name implies, importance sampling tends to select samples in important regions of the integrand. Importance sampling implies prior knowledge of the function that we are going to estimate an integral for, which at first sight appears to be a contradiction. However, most rendering problems involve an integrand which is the product of two functions, one of which is known *a priori* as in the rendering equation. For example, in a Monte Carlo approach to ray tracing a specular surface we would choose reflected rays which tended to cluster around the specular reflection direction thus sampling the (known) BRDF in regions where it is likely to return a high value. Thus, in general, we distribute the samples so that their density is highest in the regions where the function has a high value or where it varies significantly and quickly. Considering again our simple one-dimensional example we can write:

$$I = \int_0^1 p(x) \frac{f(x)}{p(x)} \, dx$$

where the first term $p(x)$ is an importance weighting function. This function $p(x)$ is then the probability density function (PDF) of the samples. That is the samples need to be chosen such that they conform to $p(x)$. To do this we define $P(x)$ to be the cumulative function of the PDF:

$$P(x) = \int_0^x p(t)dt$$

and choose a uniform random sample τ and evaluate $\xi = P^{-1}(\tau)$. Using this method the variance becomes:

$$\sigma^2_{imp} = \int_0^1 \left[\frac{f(x)}{p(x)} \right]^2 p(x)dx - I^2$$

$$= \int_0^1 \frac{f^2(x)}{p(x)} \, dx - I^2$$

The question is how do we choose $p(x)$. This can be a function that satisfies the following conditions:

$p(x) > 0$

$\int p(x)dx = 1$

$P^{-1}(x)$ is computable

For example, we could choose $p(x)$ to be the normalized absolute value of $f(x)$ or alternatively a smoothed or approximate version of $f(x)$ (Figure 10.10). Any function $f(x)$ that satisfies the above conditions will not necessarily suffice. If we choose an $f(x)$ that is too far from the ideal then the efficiency of this technique will simply drop below that of a naive method that uses random samples. Importance sampling is of critical importance in global illumination algorithms that utilize Monte Carlo approaches for the simple and obvious reason that although the rendering equation describes the global illumination at each and every point in the scene we do not require a solution that is equally accurate. We require, for example, a more accurate result for a brightly illuminated specular surface than for a dimly lit diffuse wall. Importance sampling enables us to build algorithms where the cost is distributed according to the final accuracy that we require as a function of light level and surface type.

An important practical implication of Monte Carlo methods in computer graphics is that they produce stochastic noise. For example, consider Whitted ray tracing and Monte Carlo approaches to ray tracing. In Whitted ray tracing the perfect specular direction is always chosen and in a sense the integration is reduced to a deterministic algorithm which produces a noiseless image. A crude Monte Carlo approach that imitated Whitted ray tracing would produce an image where the final pixels' estimates were, in general, slightly different from the Whitted solution. These differences manifest themselves as noticeable noise. Also note that in Whitted ray tracing if we ignore potential aliasing problems we need only initiate one ray per pixel. With a Monte Carlo approach we are using samples of the rendering equation to compute an estimate of intensity of a pixel and we need to fire many rays/pixels which bounce around the scene. In Kajiya's pioneering algorithm (Kajiya 1986), described in the next section, he used a total of 40 rays per pixel.

Global illumination algorithms that use a Monte Carlo approach are all based on these simple ideas. Their inherent complexity derives from the fact that the integration is now multi-dimensional.

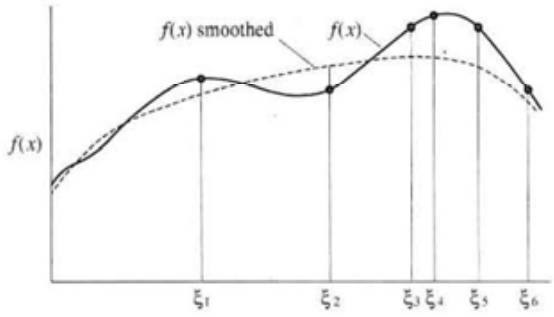


Figure 10.10
Illustrating the idea of importance sample.

10.5 Path tracing

In his classic paper that introduced the rendering equation, Kajiya (1986) was the first to recognize that Whitted ray tracing is a deterministic solution to the rendering equation. In the same paper he also suggested a non-deterministic variation of Whitted ray tracing – a Monte Carlo method that he called path tracing.

Kajiya gives a direct mathematical link between the rendering equation and the path tracing algorithm by rewriting the equation as:

$$I = g\varepsilon + gMI$$

where M is the linear operator given by the integral in the rendering equation. This can then be written as an infinite series known as a Neuman series as:

$$I = g\varepsilon + gMg\varepsilon + g(Mg)^2\varepsilon + g(Mg)^3\varepsilon + \ldots$$

where I is now the sum of a direct term, a once scattered term, a twice scattered term, etc. This leads directly to path tracing, which is theoretically known as a random walk. Light rays are traced backwards (as in Whitted ray tracing) from pixels and bounce around the scene from the first hit point, to the second, to the third, etc. The random walk has to terminate after a certain number of steps – equivalent to truncating the above series at some point when we can be sure that no further significant contributions will be encountered.

Like Whitted ray tracing, path tracing is a view-dependent solution. Previously we have said that there is no theoretical bar to extending ray tracing to handle all light–surface interactions including diffuse reflection and transmission from a hit point; just the impossibility of the computation. Path tracing implements diffuse interaction by initiating a large number of rays at each pixel (instead of, usually, one with Whitted ray tracing) and follows a single path for each ray through the scene rather than allowing a ray to spawn multiple reflected children at each hit point. The idea is shown in Figure 10.11 which can be compared with Figure 10.5. All surfaces, whether diffuse or specular can spawn a reflection/transmission ray and this contrasts with Whitted ray tracing where the encounter with a diffuse surface terminates the recursion. The other important difference is that a number of rays (40 in the original example) are initiated for each pixel enabling BRDFs to be sampled. Thus the method simulates full L(D|S)*E interaction.

A basic path tracing algorithm using a single path from source to termination will be expensive. If the random walks do not terminate on a light source then they return zero contribution to the final estimate and unless the light sources are large, paths will tend to terminate before they reach light sources. Kajiya addressed this problem by introducing a light or shadow ray that is shot towards a point on an (area) light source from each hit point in the random walk and accumulating this contribution at each point in the path (if the reflection ray from the same point directly hits the light source then the direct contribution is ignored).

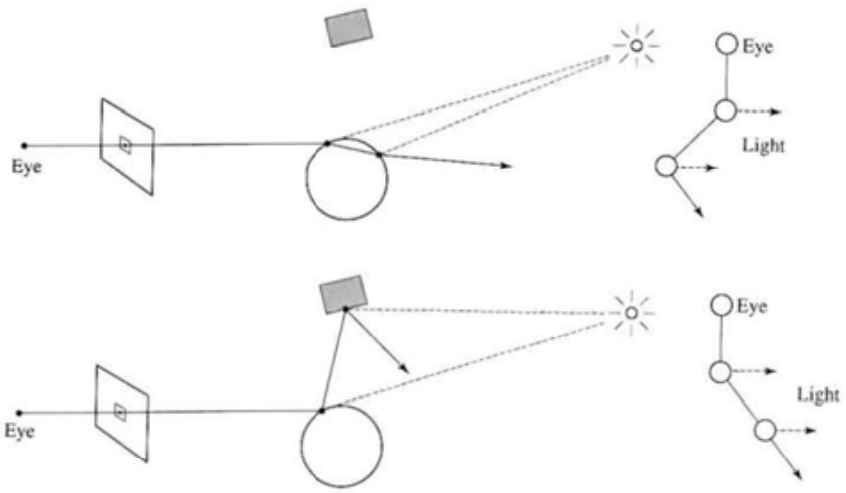


Figure 10.11
Two rays in path tracing (initiated at the same pixel).

Kajiya points out that Whitted ray tracing is wasteful in the sense that as the algorithm goes deeper into the tree it does more and more work. At the same time the contribution to the pixel intensity from events deep in the tree becomes less and less. In Kajiya's approach the tree has a branching ratio of one, and at each hit point a random variable, from a distribution based on the specular and diffuse BRDFs, is used to shoot a single ray. Kajiya points out that this process has to maintain the correct proportion of reflection, refraction and shadow rays for each pixel.

In terms of Monte Carlo theory the original algorithm reduces the variance for direct illumination but indirect illumination exhibits high variance. This is particularly true for LS*DS* E paths (see Section 10.7 for further consideration of this type of path) where a diffuse surface is receiving light from an emitter via a number of specular paths. Thus the algorithm takes a very long time to produce a good quality image. (Kajiya quotes a time of 20 hours for a 512×512 pixel image with 40 paths per pixel.)

Importance sampling can be introduced into path/ray tracing algorithms by basing it on the BRDF and ensuring that more rays are sent in directions that will return large contributions. However, this can only be done approximately because the associated PDF cannot be integrated and inverted. Another problem is that the BRDF is only one component of the integrand local to the current surface point – we have no knowledge of the light incident on this point from all directions over the hemispherical space – the field radiance (apart from the light due to direct illumination). In conventional path/ray tracing approaches all rays are traced independently of each other, accumulated and averaged into a pixel. No use is made of information gained while the process proceeds. This important observation has led to schemes that cache the information obtained during the ray trace. The most familiar of these is described in Section 10.9.

10.6 Distributed ray tracing

Like path tracing, distributed ray tracing can be seen as an extension of Whitted ray tracing or as a Monte Carlo strategy. Distributed ray tracing (distribution ray tracing, stochastic ray tracing), developed by Cook in 1986 (Cook 1986), was presumably motivated by the need to deal with the fact that Whitted ray tracing could only account for perfect specular interaction which would only occur in scenes made up of objects that consisted of perfect mirror surfaces or perfect transmitters. The effect that a Whitted ray tracer produces for (perfect) solid glass is particularly disconcerting or unrealistic. For example, consider a sphere of perfect glass. The viewer sees a circle inside of which perfectly sharp refraction has occurred (Figure 10.12). There is no sense of the sphere as an object as one would experience if scattering due to imperfections had occurred.

As far as light interaction is concerned, distributed ray tracing again only considers specular interaction but this time imperfect specular interaction is simulated by using the ray tracing approach and constructing at every hit point a reflection lobe. The shape of the lobe can depend on the surface properties of the material. Instead of spawning a single transmitted or reflected ray at an intersection a group of rays is spawned which samples the reflection lobe. This produces more realistic ray traced scenes. The images of objects reflected in the surfaces of nearby objects can appeared blurred, transparency effects are more realistic because scattering imperfections can be simulated. Area light sources can be included in the scene to produce shadows. Consider Figure 10.13: if, as would be the case in practice, the mirror surface of the sphere was not physically perfect, then we would expect to see a blurred reflection of the opaque sphere in the mirror sphere.

Thus the path classification scheme is again LDS*E or LS*E but this time all the paths are calculated (or more precisely an estimation of the effects of all the paths is calculated by judicious sampling). In Figure 10.13 three LDSE paths may be discovered by a single eye ray. The points on the wall hit by these rays are combined into a single ray (and eventually a single pixel).

As well as the above effects, Cook *et al.*'s (1984) method considered a finite aperture camera model which produced images that exhibit depth of field, motion blur

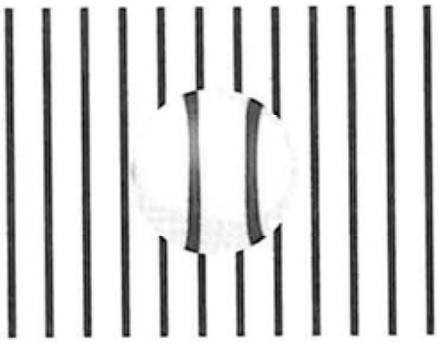


Figure 10.12
Perfect refraction through a solid glass sphere is indistinguishable from texture mapping.

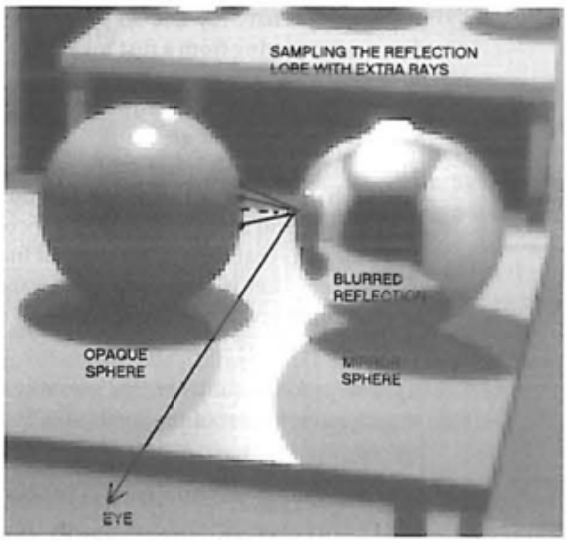


Figure 10.13
Distributed ray tracing for reflection (see Figure 10.4 for the complete geometry of this case).

due to moving objects and effective anti-aliasing (see Chapter 14 for the anti-aliasing implications of this algorithm). Figure 10.14 (Colour Plate) is an image rendered with a distributed ray tracer that demonstrates the depth of field phenomenon. The theoretical importance of this work is their realization that all these phenomena could be incorporated into a single multi-dimensional integral which was then evaluated using Monte Carlo techniques. A ray path in this algorithm is similar to a path in Kajiya's method with the addition of the camera lens. The algorithm uses a combination of stratified and importance sampling. A pixel is stratified into 16 sub-pixels and a ray is initiated from a point within a sub-pixel by using uncorrelated jittering. The lens is also stratified and one stratum on the pixel is associated with a single stratum on the lens (Figure 10.15). Reflection and transmission lobes are importance sampled and the sample point similarly jittered. Cook *et al.* (1984) pre-calculate these and store them in look-up tables associated with a surface type. Each ray derives an index as a function of its position in the

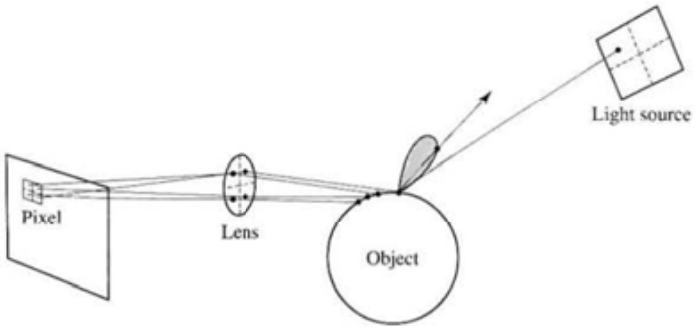


Figure 10.15
Distributed ray tracing: four rays per pixel. The pixel, lens and light source are stratified; the reflection lobe is importance sampled.

pixel. The primary ray and all its descendants have the same index. This means that a ray emerging from a first hit along a direction relative to $\boldsymbol{R}$, will emerge from all other hits in the same relative $\boldsymbol{R}$ direction for each object (Figure 10.16). This ensures that each pixel intensity, which is finally determined from 16 samples, is based on samples that are distributed, according to the importance sampling criterion, across the complete range of the specular reflection functions associated with each object. Note that there is nothing to prevent a look-up table being two-dimensional and indexed also by the incoming angle. This enables specular reflection functions that depend on angle of incidence to be implemented. Finally, note that transmission is implemented in exactly the same way using specular transmission functions about the refraction direction.

In summary we have:

(1) The process of distributing rays means that stochastic anti-aliasing becomes an integral part of the method (Chapter 14).

(2) Distributing reflected rays produces blurry reflections.

(3) Distributing transmitted rays produces convincing translucency.

(4) Distributing shadow rays results in penumbrae.

(5) Distributing ray origins over the camera lens area produces depth of field.

(6) Distributing rays in time produces motion blur (temporal anti-aliasing).

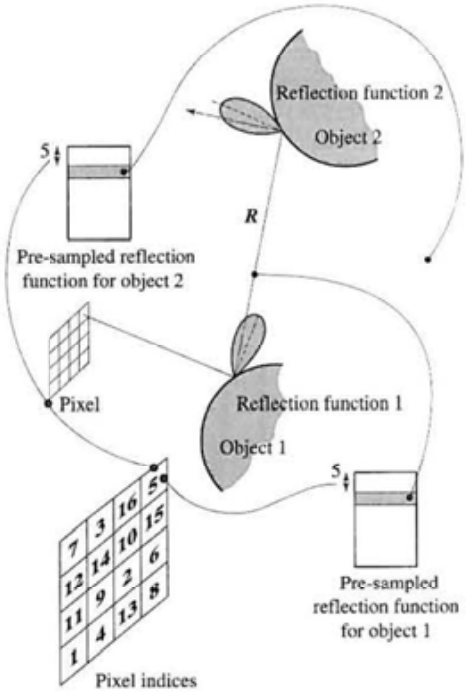


Figure 10.16
Distributed ray tracing and reflected rays.

10.7 Two-pass ray tracing

Two-pass ray tracing (or bi-directional ray tracing) was originally developed to incorporate the specular-to-diffuse transfer mechanism into the general ray tracing model. This accounts for caustics which is the pattern formed on a diffuse surface by light rays being reflected through a medium like glass or water. One can usually see on the bottom and sides of a swimming pool beautiful elliptical patterns of bright light which are due to sunlight refracting at the wind-disturbed water surface causing the light energy to vary across the diffuse surface of the pool sides. Figure 10.17 shows a ray from the scene in Figure 10.4 emanating from the light source refracting through the sphere and contributing to a caustic that forms on the (diffuse) table top. This is an LSSDE path.

Two-pass ray tracing was first proposed by Arvo (1986). In Arvo's scheme, rays from the light source were traced through transparent objects and from specular objects. Central to the working of such a strategy is the question of how information derived during the first pass is communicated to the second. Arvo suggests achieving this with a light or illumination map, consisting of a grid of data points, which is pasted onto each object in the scene in much the same way that a conventional texture map would be.

In general, two-pass ray tracing simulates paths of type LS*DS*E. The algorithm 'relies' on there being a single D interaction encountered from both the light source and the eye. The first pass consists of shooting rays from the light source and following them through the specular interactions until they hit a diffuse surface (Figure 10.17). The light energy from each ray is then deposited or cached on the diffuse surface, which has been subdivided in some manner, into elements or

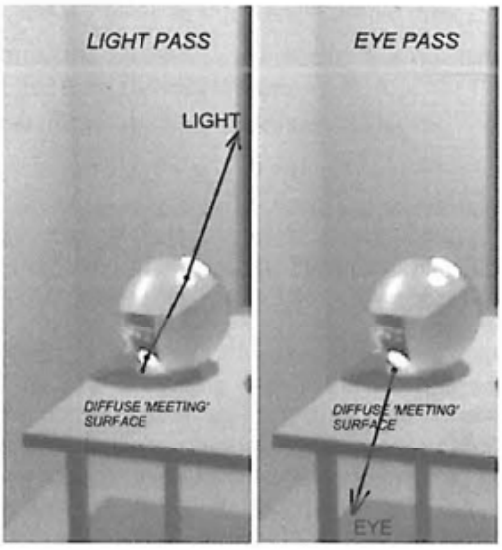


Figure 10.17
Two-pass ray tracing for the LSSDE path in Figure 10.4.

bins. In effect the first pass imposes a texture map or illumination map – the varying brightness of the caustic – on the diffuse surface. The resolution of the illumination map is critical. For a fixed number of shot light rays, too fine a map may result in map elements receiving no rays and too coarse a map results in blurring.

The second pass is the eye trace – conventional Whitted ray tracing – which terminates on the diffuse surface and uses the stored energy in the illumination map as an approximation to the light energy that would be obtained if diffuse reflection was followed in every possible direction from the hit point. In the example shown, the second pass simulates a DE path (or ED path with respect to the trace direction). The 'spreading' of the illumination from rays traced in the first pass over the diffuse surface relies on the fact that the rate of change of diffuse illumination over a surface is slow. It is important to note that there can only be one diffuse surface included in any path. Both the eye trace and the light trace terminate on the diffuse surface – it is the 'meeting point' of both traces.

It is easy to see that we cannot simulate LS*D paths by eye tracing alone. Eye rays do not necessarily hit the light and we have no way of finding out if a surface has received extra illumination due to specular to diffuse transfer. This is illustrated for an easy case of an LSDE path in Figure 10.18.

The detailed process is illustrated in Figure 10.19. A light ray strikes a surface at $\boldsymbol{P}$ after being refracted. It is indexed into the light map associated with the

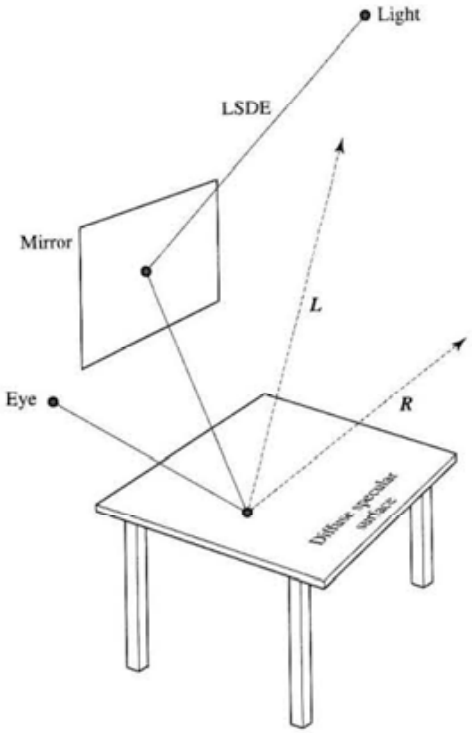


Figure 10.18
An example of an LSDE path (see also Figures 10.4 and 10.17 for examples of SDE paths). An eye ray can 'discover' light ray *L* and reflected ray *R* but cannot find the LSDE path.

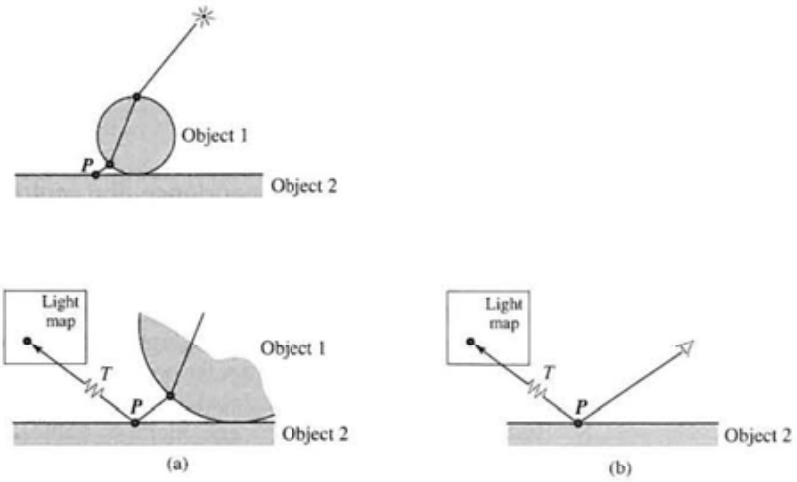


Figure 10.19
Two-pass ray tracing and light maps. (a) First pass: light is deposited in a light map using a standard texture mapping *T*. (b) Second pass: when object 2 is conventionally eye traced extra illumination at ***P*** is obtained by indexing the light map with *T*.

object using a standard texture mapping function ***T***. During the second pass an eye ray hits ***P***. The same mapping function is used to pick up any illumination for the point ***P*** and this contribution weights the local intensity calculated for that point.

An important point here is that the first pass is view independent – we construct a light map for each object which is analogous in this sense to a texture map – it becomes part of the surface properties of the object. We can use the light maps from any view point after they are completed and they need only be computed once for each scene.

Figure 10.20(a) and (b) (Colour Plate) shows the same scene rendered using a Whitted and two-pass ray tracer. In this scene there are three LSD paths:

(1) Two caustics from the red sphere – one directly from the light and one from the light reflected from the curved mirror.

(2) One (cusp) reflected caustic from the cylindrical mirror.

(3) Secondary illumination from the planar mirror (a non-caustic LSDE path).

Figures 10.20(c)–(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, these 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.

Two-pass ray tracing, as introduced by Arvo (1986) was apparently the first algorithm to use the idea of caching illumination. This approach has subsequently been taken up by Ward in his RADIANCE renderer and is the basis of most recent approaches to global illumination (see Section 10.9). We have also introduced the idea of light maps in Chapter 8. The difference between light maps in the context of this chapter and those in Chapter 8 is in their application. In global illumination they are used as part of the rendering process to

make a solution more efficient or feasible. Their application in Chapter 8 was as a mechanism for by-passing light calculations in real time rendering. In that context they function as a means of carrying pre-calculated rendering operations into the real time application.

10.8 View dependence/independence and multi-pass methods

In Section 10.5 we introduced path tracing as a method that implemented full L(DlS)* E interaction but pointed out that this is an extremely costly approach to solving the global illumination problem. In this section we will look at approaches which have combined established partial solutions such as ray tracing and radiosity and these are termed multi-pass methods.

A multi-pass method in global illumination most commonly means a combination of a view-independent method (radiosity) with a view-dependent method (ray tracing). (Although we could categorize two-pass ray tracing as a multi-pass method we have chosen to consider it as an extension to ray tracing.) Consider first the implications of the difference between a view-dependent and a view-independent approach. View-independent solutions normally only represent view-independent interactions (pure diffuse–diffuse) because they are mostly solutions where the light levels at every point in the scene are written into a three-dimensional scene data structure. We should bear in mind, however, that in principle there is nothing to stop us computing a view-independent solution that stores specular interactions, we would simply have to increase the dimensionality of the solution to calculate/store the direction of the light on a surface as well as its intensity. We return to this point in Section 10.11.

A pure view-independent algorithm evaluates only sufficient global illumination to determine the final image and if a different view is required the algorithm starts all over again. This is obvious. A more subtle point is that, in general, view-dependent algorithms evaluate an independent solution for each pixel. This is wasteful in the case of diffuse interaction because the illumination on large diffuse surfaces changes only slowly. It was this observation that led to the idea of caching illumination.

View-independent algorithms on the other hand are generally more expensive and do not handle high frequency changes such as specular interaction without significant cost in terms of computation and storage. Multi-pass algorithms exploit the advantages of both approaches by combining them.

A common approach is to post-process a radiosity solution with a ray-tracing pass. A view-independent image with the specular detail added is then obtained. However, this does not account for all path types. By combining radiosity with two-pass ray tracing the path classification, LS*DS*E can be extended to LS*(D*)S*E, the inclusion of radiosity extending the D component to D*. This implies the following ordering for an extended radiosity algorithm. Light ray tracing is employed first and light rays are traced from the source(s) through all specular transports until a diffuse surface is reached and the light energy is

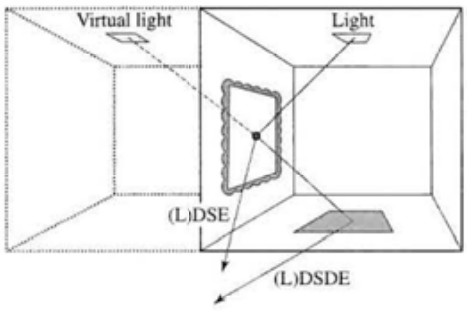


Figure 10.21
The virtual environment method for incorporating DSD paths in the radiosity method.

deposited. This accounts for the LS* paths. A radiosity solution is then invoked using these values as emitting patches and the deposited energy is distributed through the D* chain. Finally an eye pass is initiated and this provides the final projection and the ES* or ES*D paths.

Comparing the string LS*(D*)S*E with the complete global solution, we see that the central D* paths should be extended to (D*S*D*)* to make LS*(D*S*D*)*S*E which is equivalent to the complete global solution L(S|D)*E. Conventional or classical radiosity does not include diffuse-to-diffuse transfer that takes place via an intermediate specular surface. In other words once we invoke the radiosity phase we need to include the possibility of transfer via an intermediate specular path DSD.

The first and perhaps the simplest approach to including a specular transfer into the radiosity solution was based on modifying the classical radiosity algorithm for flat specular surfaces, such as mirrors, and is called the virtual window approach. This idea is shown in Figure 10.21. Conventional radiosity calculates the geometric relationship between the light source and the floor and the LDE path is accounted for by the diffuse–diffuse interaction between these two surfaces. (Note that since the light source is itself an emitting diffuse patch we can term the path LDE or DDE). What is missing from this is the contribution of light energy from the LSD or DSD path that would deposit a bright area of light on the floor. The DSD path from the light source via the mirror to the floor can be accounted for by constructing a virtual environment 'seen' through the mirror as a window. The virtual light source then acts as if it was the real light source reflected from the mirror. However, we still need to account for the LSE path which is the detailed reflected image formed in the mirror. This is view dependent and is determined during a second pass ray tracing phase. The fact that this algorithm only deals with what is, in effect, a special case illustrates the inherent difficulty of extending radiosity to include other transfer mechanisms.

10.9 Caching illumination

Caching illumination is the term we have given to the scheme of storing three- or five-dimensional values for illumination, in a data structure associated with

the scene, as a solution progresses. Such a scheme usually relates to view-dependent algorithms. In other words the cached values are used to speed up or increase the accuracy of a solution; they do not comprise a view-dependent solution in their own right. We can compare such an approach with a view-independent solution such as radiosity where final illumination values are effectively cached on the (discretized) surfaces themselves. The difference between such an approach and the caching methods described in this section is that the storage method is independent of the surface. This means that the meshing problems inherent in surface discretization methods (Chapter 11) are avoided. Illumination values on surfaces are stored in a data structure like an octree which represents the entire three-dimensional extent of the scene.

Consider again the simplified form of the radiance equation:

$$L_{\text{surface}} = \int \rho L_{\text{in}}$$

The BRDF is known but L_{in} is not and this, as we pointed out in Section 10.4, limits the efficacy of importance sampling. An estimate of L_{in} can be obtained as the solution proceeds and this requires that the values are stored. The estimate can be used to improve importance sampling and this is the approach taken by Lafortune and Williams (1995) in a technique that they call adaptive importance sampling. Their method is effectively a path tracing algorithm which uses previously calculated values of radiance to guide the current path. The idea is shown in Figure 10.22 where it is seen that a reflection direction during a path trace is chosen according to both the BRDF for the point and the current value of the field radiance distribution function for that point which has been built up

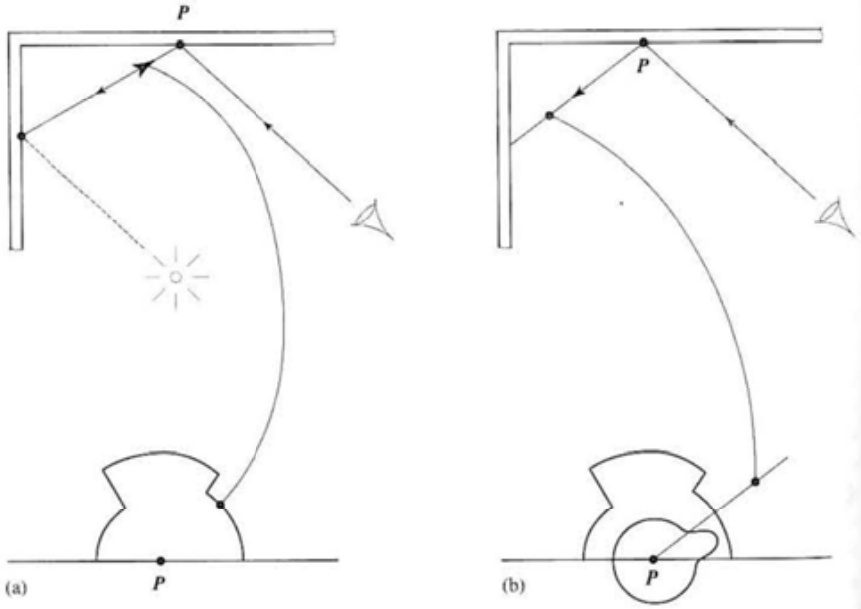


Figure 10.22
Adaptive importance sampling in path tracing (after Lafortune and Williams (1995)).
(a) Incoming radiance at a point ***P*** is cached in a 5D tree and builds up into a distribution function.
(b) A future reflected direction from ***P*** is selected on the basis of both the BRDF and the field radiance distribution function.

from previous values. Lafortune and Williams (1995) cache radiance values in a five-dimensional tree – a two-dimensional extension of a (three-dimensional) octree.

The RADIANCE renderer is probably the most well-known global illumination renderer. Developed by Ward (1994) over a period of nine years, it is a strategy, based on path tracing, that solves a version of the rendering equation under most conditions. The emphasis of this work is firmly on the accuracy required for architectural simulations under a variety of lighting conditions varying from sunlight to complex artificial lighting set-ups. The algorithm is effectively a combination of deterministic and stochastic approaches and Ward (1994) describes the underlying motivations as follows:

> The key to fast convergence is in deciding *what* to sample by removing those parts of the integral we can compute deterministically and gauging the importance of the rest so as to maximise the payback from our ray calculations.

Specular calculations are made separately and the core algorithm deals with indirect diffuse interaction. Values resulting from (perfect) diffuse interaction are cached in a (three-dimensional) octree and these cached values are used to interpolate a new value if a current hit point is sufficiently close to a cached point. This basic approach is elaborated by determining the 'irradiance gradient' in the region currently being examined which leads to the use of a higher-order (cubic) interpolation procedure for the interpolation. The RADIANCE renderer is a path tracing algorithm that terminates early if the cached values are 'close enough'.

Finally Ward expresses some strong opinions about the practical efficacy of the radiosity method. It is unusual for such criticisms to appear in a computer graphics paper and Ward is generally concerned that the radiosity method has not migrated from the research laboratories. He says:

> For example, most radiosity systems are not well automated, and do not permit general reflectance models or curved surfaces . . .. Acceptance of physically based rendering is bound to improve, but researchers must first demonstrate the real-life applicability of their techniques. There have been few notable successes in applying radiosity to the needs of practising designers. While much research has been done on improving efficiency of the basic radiosity method, problems associated with more realistic complicated geometries, have only recently got the attention they deserve. For whatever reason it appears that radiosity has yet to fulfil its promise, and it is time to re-examine this technique in the light of real-world applications and other alternatives for solving the rendering equation.

An example of the use of the RADIANCE renderer is given in the comparative image study in Chapter 18 (Figure 18.19).

10.10 Light volumes

Light volume is the term given to schemes that cache a view-independent global illumination by storing radiance or irradiance values at sample points over all space (including empty space). Thus they differ from the previous schemes

which stored values only at point on surfaces. In Chapter 16 we also encounter light volumes (therein termed light fields). Light fields are the same as light volumes and differ only in their intended application. They are used to efficiently store a pre-calculated view-independent rendering of a scene. In global illumination, however, they are used to facilitate a solution in some way.

An example of the application of light volumes is described by Greger *et al.* (1998). Here the idea is to use a global illumination solution in 'semi-dynamic' environments. Such an environment is defined as one wherein the moving objects are small compared to the static objects, implying that their position in the scene does not affect the global illumination solution to any great extent. A global solution is built up in a light volume and this is used to determine the global illumination received by a moving object – in effect the moving object travels through a static light volume receiving illumination but not contributing to the pre-processed solution.

10.11 Particle tracing and density estimation

In this final section we look at a recent approach (Walter *et al.* 1997) whose novelty is to recognize that it is advantageous to separate the global problem into light transport and light representation calculations. In this work rather than caching illumination, particle histories are stored. The reason for this is that light transport – the flow of light between surfaces – has high global or inter-surface complexity. On the other hand the representation of light on the surface of an object has high local or intra-surface complexity. Surfaces may exhibit shadows, specular highlights caustics etc. (A good example of this is the radiosity algorithm where transport and representation are merged into one process. Here the difficulty lies in predicting the meshing required – for example, to define shadow edges – to input into the algorithm. That is, we have to decide on a meshing prior to the light transport solution being available.)

The process is divided into three sequential phases:

- **Particle tracing** In a sense this is a view-independent form of path tracing. Light-carrying particles are emitted from each light source and travel through the environment. Each time a surface is hit this is recorded (surface identifier, point hit and wavelength of the particle) and a reflection/transmission direction computed according to the BRDF of the surface. Each particle generates a list of information for every surface it collides with and a particle process terminates after a minimum of interactions or until it is absorbed. Thus a particle path generates a history of interactions rather than returning a pixel intensity.
- **Density estimation** After the particle tracing process is complete each surface possesses a list of particles and the stored hit points are used to construct the illumination on the surface based on the spatial density of the hit points. The result of this process is a Gouraud shaded mesh of triangles.

- **Mesh optimization** The solution is view independent and the third phase optimizes or decimates the mesh by progressively removing meshes as long as the resulting change due to the removal does not drop below a (perceptually based) threshold. The output from this phase is an irregular mesh whose detail relates to the variation of light over the surface.

Walter *et al.* (1997) point out that a strong advantage of the technique is that its modularity enables optimization for different design goals. For example, the light transport phase can be optimized for the required accuracy of the BRDFs. The density phase can vary its criteria according to perceptual accuracy and the decimation phase can achieve high compression while maintaining perceptual quality.

A current disadvantage of the approach is that it is a three-dimensional view-independent solution which implies that it can only display diffuse–diffuse interaction. However, Walter *et al.* (1997) point out that this restriction comes out of the density estimation phase. The particle tracing module can deal with any type of BRDF.

11 The radiosity method

Introduction

Ray tracing, the first computer graphics model to embrace global interaction, or at least one aspect of it – suffers from an identifying visual signature: you can usually tell if an image has been synthesized using ray tracing. It only models one aspect of the light interaction – that due to perfect specular reflection and transmission. The interaction between diffusely reflecting surfaces, which tends to be the predominant light transport mechanism in interiors, is still modelled using an ambient constant (in the local reflection component of the model). Consider, for example, a room with walls and ceiling painted with a matte material and carpeted. If there are no specularly reflecting objects in the room, then those parts of the room that cannot see a light source are lit by diffuse interaction. Such a room tends to exhibit slow and subtle changes of intensity across its surfaces.

In 1984, using a method whose theory was based on the principles of radiative heat transfer, researchers at Cornell University, developed the radiosity method (Goral *et al.* 1984). This is now known as classical radiosity and it simulates LD*E paths, that is, it can only be used, in its unextended form, to render scenes that are made up in their entirety of (perfect) diffuse surfaces.

To accomplish this, every surface in a scene is divided up into elements called patches and a set of equations is set up based on the conservation of light energy.

A single patch in such an environment reflects light received from every other patch in the environment. It may also emit light if it is a light source – light sources are treated like any other patch except that they have non-zero self-emission. The interaction between patches depends on their geometric relationship. That is distance and relative orientation. Two parallel patches a short distance apart will have a high interaction. An equilibrium solution is possible if, for each patch in the environment, we calculate its interaction between it and every other patch in the environment.

One of the major contributions of the Cornell group was to invent an efficient way – the hemicube algorithm – for evaluating the geometric relationship between pairs of patches; in fact, in the 1980s most of the innovations in radiosity methods have come out of this group.

The cost of the algorithm is $O(N^2)$ where N is the number of patches into which the environment is divided. To keep processing costs down, the patches are made large and the light intensity is assumed to be constant across a patch. This immediately introduces a quality problem – if illumination discontinuities do not coincide with patch edges artefacts occur. This size restriction is the practical reason why the algorithm can only calculate diffuse interaction, which by its nature changes slowly across a surface. Adding specular interaction to the radiosity method is expensive and is still the subject of much research. Thus we have the strange situation that the two global interaction methods – ray tracing and radiosity – are mutually exclusive as far as the phenomena that they calculate are concerned. Ray tracing cannot calculate diffuse interaction and radiosity cannot incorporate specular interaction. Despite this, the radiosity method has produced some of the most realistic images to date in computer graphics.

The radiosity method deals with shadows without further enhancement. As we have already discussed, the geometry of shadows is more-or-less straightforward to calculate and can be part of a ray tracing algorithm or an algorithm added onto a local reflection model renderer. However, the intensity within a shadow is properly part of diffuse interaction and can only be arbitrarily approximated by other algorithms. The radiosity method takes shadows in its stride. They drop out of the solution as intensities like any other. The only problem is that the patch size may have to be reduced to delineate the shadow boundary to some desired level of accuracy. Shadow boundaries are areas where the rate of change of diffuse light intensity is high and the normal patch size may cause visible aliasing at the shadow edge.

The radiosity method is an object space algorithm, solving for the intensity at discrete points or surface patches within an environment and not for pixels in an image plane projection. The solution is thus independent of viewer position. This complete solution is then injected into a renderer that computes a particular view by removing hidden surfaces and forming a projection. This phase of the method does not require much computation (intensities are already calculated) and different views are easily obtained from the general solution.

11.1 Radiosity theory

Elsewhere in the text we have tried to maintain a separation between the algorithm that implements a method and the underlying mathematics. It is the case, however, that with the radiosity method, the algorithm is so intertwined with the mathematics that it would be difficult to try to deal with this in a separate way. The theory itself consists of nothing more than definitions – there is no manipulation. Readers requiring further theoretical insight are referred to the book by Siegel and Howell (1984).

The radiosity method is a conservation of energy or energy equilibrium approach, providing a solution for the radiosity of all surfaces within an enclosure. The energy input to the system is from those surfaces that act as emitters. In fact, a light source is treated like any other surface in the algorithm except that it possesses an initial (non-zero) radiosity. The method is based on the assumption that all surfaces are perfect diffusers or ideal Lambertian surfaces.

Radiosity, B, is defined as the energy per unit area leaving a surface patch per unit time and is the sum of the emitted and the reflected energy:

$$B_i dA_i = E_i dA_i + R_i \int_j B_j F_{ji} dA_j$$

Expressing this equation in words we have for a single patch i:

radiosity × area = emitted energy + reflected energy

E_i is the energy emitted from a patch. The reflected energy is given by multiplying the incident energy by R_i, the reflectivity of the patch. The incident energy is that energy that arrives at patch i from all other patches in the environment; that is we integrate over the environment, for all j ($j \neq i$), the term $B_j F_{ji} dA_j$. This is the energy leaving each patch j that arrives at patch i. F_{ji} is a constant, called a form factor, that parametrizes the relationship between patches j and i.

We can use a reciprocity relationship to give:

$$F_{ij} A_i = F_{ji} A_j$$

and dividing through by dA_i, gives:

$$B_i = E_i + R_i \int_j B_j F_{ij}$$

For a discrete environment the integral is replaced by a summation and constant radiosity is assumed over small discrete patches, giving:

$$B_i = E_i + R_i \sum_{j=1}^{n} B_j F_{ij}$$

Such an equation exists for each surface patch in the enclosure and the complete environment produces a set of n simultaneous equations of the form:

$$\begin{bmatrix} 1-R_1F_{11} & -R_1F_{12} & \dots & -R_1F_{1n} \\ -R_2F_{21} & 1-R_2F_{22} & \dots & -R_2F_{2n} \\ \vdots & \vdots & \dots & \vdots \\ R_nF_{n1} & -R_nF_{n2} & \dots & 1-R_nF_{nn} \end{bmatrix} \begin{bmatrix} B_1 \\ B_2 \\ \vdots \\ B_n \end{bmatrix} = \begin{bmatrix} E_1 \\ E_2 \\ \vdots \\ E_n \end{bmatrix} \quad [11.1]$$

Solving this equation is the radiosity method. Out of this solution comes B_i the radiosity for each patch. However, there are two problems left. We need a way of computing the form factors. And we need to compute a view and display the patches. To do this we need a linear interpolation method – just like Gouraud shading – otherwise the subdivision pattern – the patches themselves – will be visible.

The E_is are non-zero only at those surfaces that provide illumination and these terms represent the input illumination to the system. The R_is are known and the F_{ij}s are a function of the geometry of the environment. The reflectivities are wavelength-dependent terms and the above equation should be regarded as a monochromatic solution; a complete solution being obtained by solving for however many colour bands are being considered. We can note at this stage that $F_{ii} = 0$ for a plane or convex surface – none of the radiation leaving the surface will strike itself. Also from the definition of the form factor the sum of any row of form factors is unity.

Since the form factors are a function only of the geometry of the system they are computed once only. The method is bound by the time taken to calculate the form factors expressing the radiative exchange between two surface patches A_i and A_j. This depends on their relative orientation and the distance between them and is given by:

$$F_{ij} = \frac{\text{Radiative energy leaving surface } A_i \text{ that strikes } A_j \text{ directly}}{\text{Radiative energy leaving surface } A_i \text{ in all directions in the hemispherical space surrounding } A_i}$$

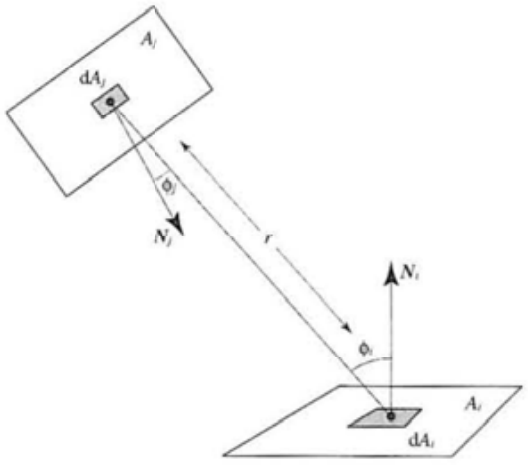


Figure 11.1
Form factor geometry for two patches i and j (after Goral *et al.* (1984)).