It can be shown that this is given by:

$$F_{ij} = \frac{1}{A_i} \int_{Ai} \int_{Aj} \frac{\cos \phi_i \cos \phi_j}{\pi r^2} \, dA_j dA_i$$

where the geometric conventions are illustrated in Figure 11.1. In any practical environment A_j may be wholly or partially invisible from A_i and the integral needs to be multiplied by an occluding factor which is a binary function that depends on whether the differential area dA_i can see dA_j or not. This double integral is difficult to solve except for specific shapes.

11.2 Form factor determination

An elegant numerical method of evaluating form factors was developed in 1985 and this is known as the hemicube method. This offered an efficient method of determining form factors and at the same time a solution to the intervening patch problem.

The patch to patch form factor can be approximated by the differential area to finite area equation:

$$F_{dAiAj} = \int_{Aj} \frac{\cos \phi_i \cos \phi_j}{\pi r^2} \, dA_j$$

where we are now considering the form factor between the elemental area dA_i and the finite area A_j. dA_i is positioned at the centre point of patch *i*. The veracity of this approximation depends on the area of the two patches compared with the distance, *r*, between them. If *r* is large the inner integral does not change much over the range of the outer integral and the effect of the outer integral is simply multiplication by unity.

A theorem called the Nusselt analogue tells us that we can consider the projection of a patch *j* onto the surface of a hemisphere surrounding the elemental patch dA_i and that this is equivalent in effect to considering the patch itself. Also patches that produce the same projection on the hemisphere have the same form factor. This is the justification for the hemicube method as illustrated in Figure 11.2. Patches *A*, *B* and *C* all have the same form factor and we can evaluate the form factor of any patch *j* by considering not the patch itself, but its projection onto the faces of a hemicube.

A hemicube is used to approximate the hemisphere because flat projection planes are computationally less expensive. The hemicube is constructed around the centre of each patch with the hemicube *Z* axis and the patch normal coincident (Figure 11.3). The faces of the hemicube are divided into pixels – a somewhat confusing use of the term since we are operating in object space. Every other patch in the environment is projected onto this hemicube. Two patches that project onto the same pixel can have their depths compared and the further patch be rejected, since it cannot be seen from the receiving patch. This approach is analogous to a Z-buffer algorithm except that there is no interest in

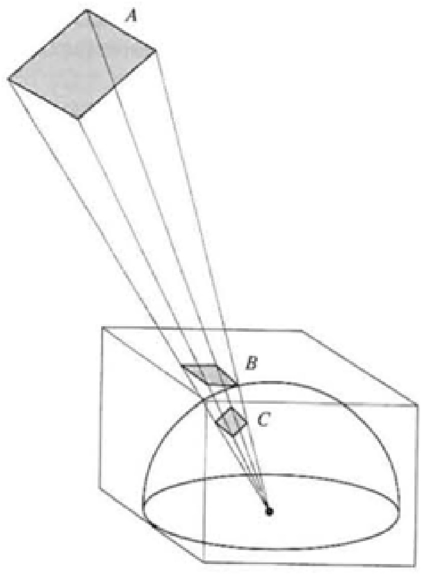


Figure 11.2
The justification for using a hemicube. Patches A, B and C have the same form factor.

intensities at this stage. The hemicube algorithm only facilitates the calculation of the form factors that are subsequently used in calculating diffuse intensities and a 'label buffer' is maintained indicating which patch is currently nearest to the hemicube pixel.

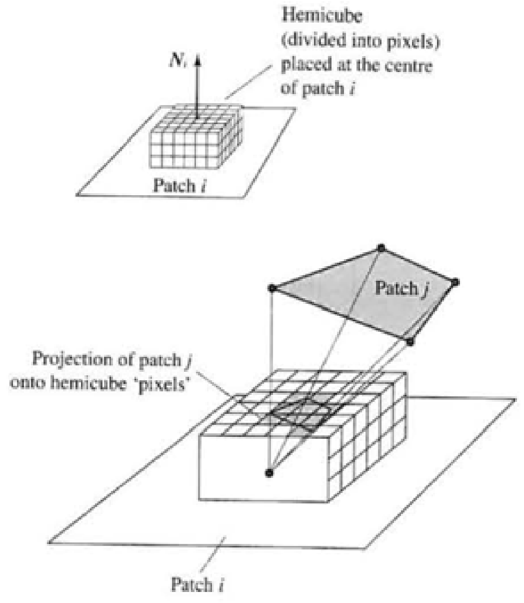

Hemicube (divided into pixels) placed at the centre of patch i
N_i
Patch i
Patch j
Projection of patch j onto hemicube 'pixels'
Patch i

Figure 11.3
Evaluating the form factor F_{ij} by projecting patch j onto the faces of a hemicube centred on patch i.

Each pixel on the hemicube can be considered as a small patch and a differential to finite area form factor, known as a delta form factor, defined for each pixel. The form factor of a pixel is a fraction of the differential to finite area form factor for the patch and can be defined as:

$$\begin{aligned}\Delta F_{dAiAj} &= \frac{\cos\phi_i \cos\phi_j}{\pi r^2}\,\Delta A \\ &= \Delta F_{ij}\end{aligned}$$

where ΔA is the area of the pixel.

These form factors are pre-calculated and stored in a look-up table. This is the foundation of the efficiency of the hemicube method. Again, using the fact that areas of equal projection onto the receiving surface surrounding the centre of patch A_i have equal form factors, we can conclude that F_{ij}, for any patch, is obtained by summing the pixel form factors onto which patch A_j projects (Figure 11.4).

Thus form factor evaluation now reduces to projection onto mutually orthogonal planes and a summation operation.

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

The method can be summarized in the following stages:

(1) Computation of the form factors, F_{ij}. Each hemicube emplacement calculates (n–1) form factors or one row in the equation.

(2) Solving the radiosity matrix equation.

(3) Rendering by injecting the results of stage (2) into a bilinear interpolation scheme.

(4) Repeating stages (2) and (3) for the colour bands of interest.

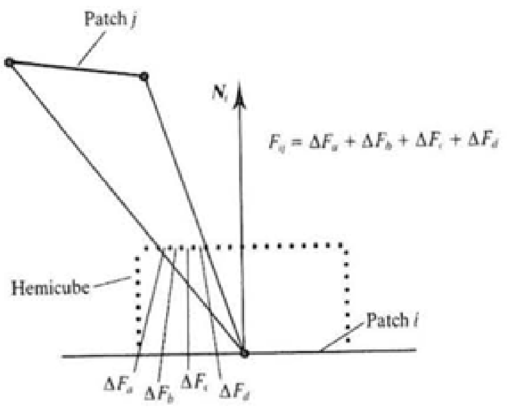


Figure 11.4
F_{ij} is obtained by summing the form factors of the pixels onto which patch i projects.

This process is shown in Figure 11.6. Form factors are a function only of the environment and are calculated once only and can be reused in stage (2) for different reflectivities and light source values. Thus a solution can be obtained for the same environment with, for example, some light sources turned off. The solution produced by stage (2) is a view-independent solution and if a different view is required then only stage (3) is repeated. This approach can be used, for example, when generating an animated walk-through of a building interior. Each frame in the animation is computed by changing the view point and calculating a new view from an unchanging radiosity solution. It is only if we change the geometry of the scene that a re-calculation of the form factors is necessary. If the lighting is changed and the geometry is unaltered, then only the equation needs resolving – we do not have to re-calculate the form factors.

Stage (2) implies the computation of a view-independent rendered version of the solution to the radiosity equation which supplies a single value, a radiosity, for each patch in the environment. From these values vertex radiosities are calculated and these vertex radiosities are used in the bilinear interpolation scheme to provide a final image. A depth buffer algorithm is used at this stage to evaluate the visibility of each patch at each pixel on the screen. (This stage should not be confused with the hemicube operation that has to evaluate inter-patch visibility during the computation of form factors.)

The time taken to complete the form factor calculation depends on the square of the number of patches. A hemicube calculation is performed for every patch (onto which all other patches are projected). The overall calculation time thus depends on the complexity of the environment and the accuracy of the solution,

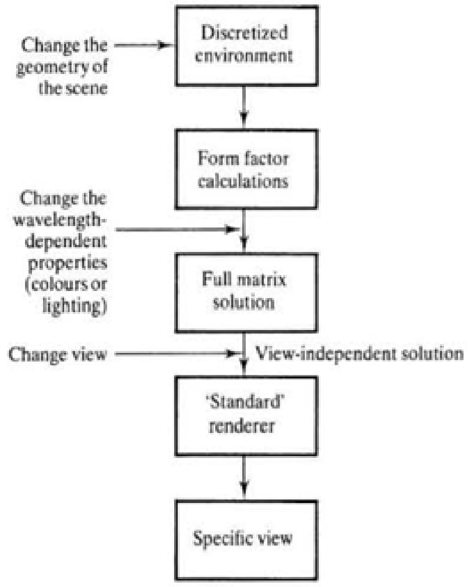

Change the geometry of the scene
Discretized environment
Form factor calculations
Change the wavelength-dependent properties (colours or lighting)
Full matrix solution
Change view
View-independent solution
'Standard' renderer
Specific view

Figure 11.6
Stages in a complete radiosity solution. Also shown are the points in the process where various modifications can be made to the image.

as determined by the hemicube resolution. Although diffuse illumination changes only slowly across a surface, aliasing can be caused by too low a hemicube resolution and accuracy is required at shadow boundaries (see Section 11.7). Storage requirements are also a function of the number of patches required. All these factors mean that there is an upward limit on the complexity of the scenes that can be handled by the radiosity method:

11.3 The Gauss–Siedel method

Cohen and Greenberg (1985) point out that the Gauss–Siedel method is guaranteed to converge rapidly for equation sets such as Equation 11.1. The sum of any row of form factors is by definition less than unity and each form factor is multiplied by a reflectivity of less than one. The summation of the row terms in Equation 11.1 (excluding the main diagonal term) is thus less than unity. The mean diagonal term is always unity ($F_{ii} = 0$ for all i) and these conditions guarantee fast convergence. The Gauss–Siedel method is an extension to the following iterative method. Given a system of linear equations:

$$\boldsymbol{Ax} = \boldsymbol{E}$$

such as Equation 11.1, we can rewrite equations for $x_1, x_2, \ldots, x_i$ in the form:

$$x_1 = \frac{E_1 - a_{12}x_2 - a_{13}x_3 - \ldots - a_{1n}\,x_n}{a_{11}}$$

which leads to the iteration:

$$x_1^{(k+1)} = \frac{E_1 - a_{12}x_2^{(k)} - a_{13}x_3 - \ldots - a_{1n}\,x_n^{(k)}}{a_{11}}$$

in general:

$$x_i^{(k+1)} = \frac{E_i - a_{i1}x_1^{(k)} - \ldots - a_{i,i-1}x_{i-1}^{(k)} - a_{i,i+1}x_{i+1}^{(k)} - \ldots - a_{in}\,x_n^{(k)}}{a_{ii}} \qquad [11.2]$$

This formula can be used in an iteration procedure:

(1) Choose an initial approximation, say:

$$x_i^{(0)} = \frac{E_i}{a_{ii}}$$

for $i = 1, 2, \ldots, n$, where E_i is non-zero for emitting surfaces or light sources only.

(2) Determine the next iterate:

$x_i^{(k+1)}$ from $x_i^{(k)}$

using Equation 11.2.

(3) If $|x_i^{(k+1)} - x_i^{(k)}| <$ a threshold

for $i = 1, 2, \ldots, n$

then stop the iteration, otherwise return to step (2).

This is known as Jacobi iteration. The Gauss–Siedel method improves on the convergence of this method by modifying Equation 11.2 to use the latest available information. When the new iterate $x_i^{(k+1)}$ is being calculated, new values:

$$x_1^{(k+1)}, x_2^{(k+1)}, \ldots, x_{i-1}^{(k+1)}$$

have already been calculated and Equation 11.2 is modified to:

$$x_i^{(k+1)} = \frac{E_i - a_{i1}x_1^{(k+1)} - \ldots - a_{i,i-1}x_{i-1}^{(k+1)} - a_{i,i+1}x_{i+1}^{(k)} - \ldots - a_{in}\, x_n^{(k)}}{a_{ii}} \qquad [11.3]$$

Note that when $i = 1$ the right-hand side of the equation contains terms with superscript k only, and Equation 11.3 reduces to Equation 11.2. When $i = n$ the right-hand side contains terms with superscript $(k+1)$ only.

Convergence of the Gauss–Siedel method can be improved by the following method. Having produced a new value $x_i^{(k+1)}$, a better value is given by a weighted average of the old and new values:

$$x_i^{(k+1)} = rx_i'^{(k+1)} + (1 - r)x_i^{(k)}$$

where r (>0) is a parameter independent of k and i. Cohen *et al.* (1988) report that a relaxation factor of 1.1 works for most environments.

11.4 Seeing a partial solution – progressive refinement

Using the radiosity method in a practical context, such as in the design of building interiors, means that the designer has to wait a long time to see a completed image. This is disadvantageous since one of the *raisons d'être* of computer-based design is to allow the user free and fast experimentation with the design parameters. A long feedback time discourages experimentation and stultifies the design process.

In 1988 the Cornell team developed an approach, called 'progressive refinement' that enabled a designer to see an early (but approximate) solution. At this stage major errors can be seen and corrected, and another solution executed. As the solution becomes more and more accurate, the designer may see more subtle changes that have to be made. We introduced this method in the previous chapter, we will now look at the details.

The general goal of progressive or adaptive refinement can be taken up by any slow image synthesis technique and it attempts to find a compromise between the competing demands of interactivity and image quality. A synthesis method that provides adaptive refinement would present an initial quickly rendered image to the user. This image is then progressively refined in a 'graceful' way. This is defined as a progression towards higher quality, greater realism etc., in a way that is automatic, continuous and not distracting to the user. Early availability of an approximation can greatly assist in the development of techniques and images, and reducing the feedback loop by approximation is a necessary adjunct to the radiosity method.

The two major cost factors in the radiosity method are the storage costs and the calculation of the form factors. For an environment of 50×10^3 patches, even although the resulting square matrix of form factors may be 90% sparse (many patches cannot see each other) this still requires 10^9 bytes of storage (at four bytes per form factor).

Both the requirements of progressive refinement and the elimination of pre-calculation and storage of the form factors are met by an ingenious restructuring of the basic radiosity algorithm. The stages in the progressive refinement are obtained by displaying the results as the iterative solution progresses. The solution is restructured and the form factor evaluation order is optimized so that the convergence is 'visually graceful'. This restructuring enables the radiosity of all patches to be updated at each step in the solution, rather than a step providing the solution for a single patch. Maximum visual difference between steps in the solution can be achieved by processing patches according to their energy contribution to the environment. The radiosity method is particularly suited to a progressive refinement approach because it computes a view-independent solution. Viewing this solution (by rendering from a particular view point) can proceed independently as the radiosity solution progresses.

In the conventional evaluation of the radiosity matrix (using, for example, the Gauss–Seidel method) a solution for one row provides the radiosity for a single patch i:

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

This is an estimate of the radiosity of patch i based on the current estimate of all other patches. This is called 'gathering'. The equation means that (algorithmically) for patch i we visit every other patch in the scene and transfer the appropriate amount of light from each patch j to patch i according to the form factor. The algorithm proceeds on a row-by-row basis and the entire solution is updated for one step through the matrix (although the Gauss–Seidel method uses the new values as soon as they are computed). If the process is viewed dynamically, as the solution proceeds, each patch intensity is updated according to its row position in the radiosity matrix. Light is gathered from every other patch in the scene and used to update the single patch currently being considered.

The idea of the progressive refinement method is that the entire image of all patches is updated at every iteration. This is termed 'shooting', where the contribution from each patch i is distributed to all other patches. The difference between these two processes is illustrated diagramatically in Figures 11.7(a) and (b). This re-ordering of the algorithm is accomplished in the following way.

A single term determines the contribution to the radiosity of patch j due to that from patch i:

$$B_j \quad \text{due to} \quad B_i = R_j B_i F_{ji}$$

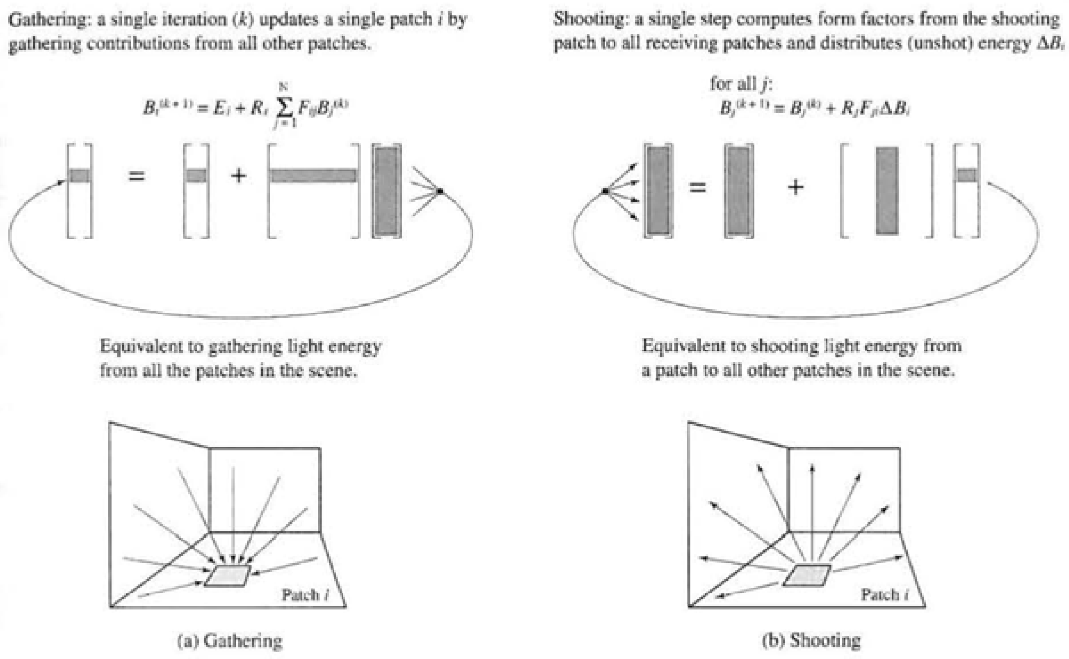

Gathering: a single iteration (k) updates a single patch i by gathering contributions from all other patches.
$B_i^{(k+1)} = E_i + R_i \sum_{j=1}^{N} F_{ij} B_j^{(k)}$
Equivalent to gathering light energy from all the patches in the scene.
Patch i
(a) Gathering
Shooting: a single step computes form factors from the shooting patch to all receiving patches and distributes (unshot) energy ΔB_i
for all j:
$B_j^{(k+1)} = B_j^{(k)} + R_j F_{ji} \Delta B_i$
Equivalent to shooting light energy from a patch to all other patches in the scene.
Patch i
(b) Shooting

Figure 11.7
(a) Gathering and
(b) shooting in radiosity solution strategies
(based on an illustration in Cohen *et al.* (1988)).

This relationship can be reversed by using the reciprocity relationship:

B_j due to $B_i = R_j B_i F_{ij} A_i / A_j$

and this is true for all patches j. This relationship can be used to determine the contribution to each patch j in the environment from the single patch i. A single radiosity (patch i) shoots light into the environment and the radiosities of all patches j are updated simultaneously. The first complete update (of all the radiosities in the environment) is obtained from 'on the fly' form factor computations. Thus an initial approximation to the complete scene can appear when only the first row of form factors has been calculated. This eliminates high start-up or pre-calculation costs.

This process is repeated until convergence is achieved. All radiosities are initially set either to zero or to their emission values. As this process is repeated for each patch i the solution is displayed and at each step the radiosities for each patch j are updated. As the solution progresses the estimate of the radiosity at a patch i becomes more and more accurate. For an iteration the environment already contains the contribution of the previous estimate of B_i and the so-called 'unshot' radiosity – the difference between the current and previous estimates – is all that is injected into the environment.

If the output from the algorithm is displayed without further elaboration, then a scene, initially dark, gradually gets lighter as the incremental radiosities are added to each patch. The 'visual convergence' of this process can be optimized by sorting the order in which the patches are processed according to the amount of energy that they are likely to radiate. This means, for example, that emitting patches, or light sources, should be treated first. This gives an early well lit solution. The next patches to be processed are those that received most light from the light sources and so on. By using this ordering scheme, the solution proceeds in a way that approximates the propagation of light through an environment. Although this produces a better visual sequence than an unsorted process, the solution still progresses from a dark scene to a fully illuminated scene. To overcome this effect an arbitrary ambient light term is added to the intermediate radiosities. This term is used only to enhance the display and is not part of the solution. The value of the ambient term is based on the current estimate of the radiosities of all patches in the environment, and as the solution proceeds and becomes 'better lit' the ambient contribution is decreased.

Four main stages are completed for each iteration in the algorithm. These are:

(1) Find the patch with the greatest (unshot) radiosity or emitted energy.
(2) Evaluate a column of form factors, that is, the form factors from this patch to every other patch in the environment.
(3) Update the radiosity of each of the receiving patches.
(4) Reduce the temporary ambient term as a function of the sum of the differences between the current values calculated in step (3) and the previous values.

An example of the progressive refinement during execution is shown in Figure 10.7 and Section 10.3.2 contains a full description of this figure.

11.5 Problems with the radiosity method

There are three significant problems associated with radiosity rendering. They are algorithm artefacts that appear in the image, the inability to deal with specular interaction and the inordinate time taken to render a scene of moderate complexity. Curiously, hardly any research effort has been devoted to the time factor, and this is perhaps the reason that radiosity has not generally migrated into applications programs. This contrasts with the situation in ray tracing research in the 1980s, where quite soon after the first ray traced imagery appeared, a large and energetic research effort was devoted to making the method faster. In the remainder of the chapter we will deal exclusively with image quality, noting in passing that it is usually related to execution time – quality can be improved by defining the scene more accurately which in the mainstream method means allowing more iterations in the program.

Developments in the radiosity method beyond the techniques described in the previous chapter have mostly been motivated by defects or artefacts that arise out of the representation of the scene as a set of largish patches. Although other factors, such as taking into account scattering atmospheres and the incorporation of specular reflection are important, addressing the visual defects due to meshing accounts for most research emphasis and it is with this aspect that we will deal.

11.6 Artefacts in radiosity images

The common artefacts in radiosity images that use the classical approach of the previous chapter are due to:

(1) Approximations in the hemicube method for determining the form factors.
(2) Using bilinear interpolation as a reconstruction of the radiosity function from the constant radiosity solution.
(3) Using a meshing or subdivision of the scene that is independent of the nature of the variations in the radiosity function.

The visibility and thus the importance of these depends, of course, on the nature of the scene; but usually the third category is the most noticeable and the most difficult to deal with. In practice the artefacts cannot be treated independently: there is little point in developing a powerful meshing strategy without also dealing with artefacts that emerge from bilinear interpolation. We will now look at these image defects detailing both the cause and the possible cure.

11.6.1 Hemicube artefacts

The serious problem of the hemicube method is aliasing caused by the regular division of the hemicube into uniform pixels. Errors occur as a function of the size of the hemicube pixels due to the assumption that patches will project exactly onto an integer number of pixels, which in general, of course, they do not. This is similar to aliasing in ray tracing. We attempt to gather information from a three-dimensional environment by looking in a fixed number of directions. In ray tracing these directions are given initially by evenly spaced eye-to-pixel rays. In the radiosity method, by projecting the patches onto hemicubes we are effectively sampling with projection rays from the hemicube origin. Figure 11.8 shows a two-dimensional analogue of the problem where a number of identical polygons project onto either one or two pixels depending on the interference between the projection rays and the polygon grid. The polygons are of equal size and equal orientation with respect to patch i. Their form factors should be different – because the number of pixels onto which each polygon projects is different for each polygon. However, as the example shows, neighbouring polygons which should have almost equal form factors will produce values in the ratio 2:1.

The geometry of any practical scene can cause problems with the hemicube method. Its accuracy depends on the distance between the patches involved in the calculation. When distances become small the method falls down. This

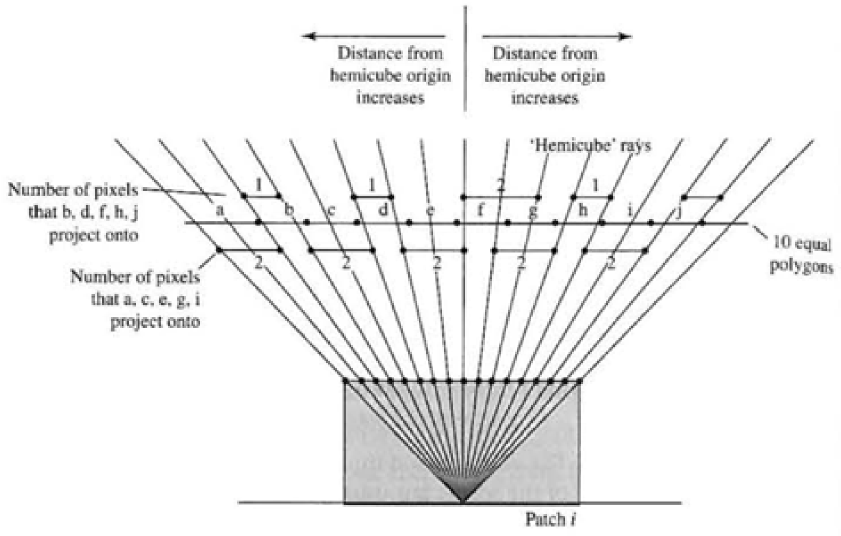

Distance from hemicube origin increases
Distance from hemicube origin increases
'Hemicube' rays
Number of pixels that b, d, f, h, j project onto
Number of pixels that a, c, e, g, i project onto
10 equal polygons
a b c d e f g h i j
1
2
Patch *i*

Figure 11.8
Interference between hemicube sampling and a set of equal polygons (after Wallace *et al.* (1989)).

situation occurs in practice, for example, when an object is placed on a supporting surface. The errors in form factors occur precisely in those regions from which we expect the radiosity technique to excel and produce subtle phenomena such as colour bleeding and soft shadows. Baum *et al.* (1989) quantify the error involved in form factor determination for proximal surfaces, and demonstrate the hemicube method is only accurate in contexts where the inter-patch distance is at least five patch diameters.

Yet another hemicube problem occurs with light sources. In scenes which the radiosity method is used to render, we are usually concerned with area sources such as fluorescent lights. As with any other surface in the environment we divide the light sources into patches and herein lies the problem. For a standard solution an environment will be discretized into patches where the subdivision resolution depends on the area of surface (and the accuracy of the solution required). However, in the case of light sources the number of hemicubes required or the number of patches required depends on the distance from the closest surface it illuminates. A hemicube operation effectively reduces an emitting patch to a point source. Errors will appear on a close surface as isolated areas of light if the light source is insufficiently subdivided. With strip lights, where the length to breadth ratio is great, insufficient subdivision can give rise to banding or aliasing artefacts that run parallel with the long axis of the light source. An example of the effect of insufficient light source subdivision is shown in Figure 11.14.

Hemicube aliasing can, of course, be ameliorated by increasing the resolution of the hemicube, but this is inefficient, increasing the computation required for all elements in the scene irrespective of whether they are aliased by the hemicube or not; exactly the same situation which occurs with conventional (context independent) anti-aliasing measures (Chapter 14).

Problems emerge from the approximation (see the previous chapter):

$F_{ij} \approx F_{dAiAj}$

The hemicube evaluates a form factor from a differential area – effectively a point – to a finite area. There are two consequences of this. Figure 11.9 illustrates a problem that can arise with intervening patches. Here the form factor from patch *i* to patch *j* is calculated as if the intervening patch did not exist because patch *j* can be seen in its entirety from the hemicube origin.

Finally, consider the sampling 'efficiency' of the hemicube. Patches that can be 'seen' from the hemicube in the normal direction are more important than patches in the horizon direction. (They project onto hemicube cells that have higher delta form factors.) If we consider distributing the computational effort evenly on the basis of importance sampling then cells nearer the horizon are less important. An investigation reported in Max and Troutman (1993) derives optimal resolution, shapes and grid cell spacings. In this work a top-face resolution 40% higher than that of the sides and a side height of 70% of the width is suggested. Note that this leads also to a reduction in aliasing artefacts caused by uniform hemicube cells.

11.6.2 Reconstruction artefacts

Reconstruction artefacts are so called because they originate from the nature of the method used to reconstruct or approximate the continuous radiosity function from the constant radiosity solution. We recall that radiosity methods can only function under the constant radiosity assumption which is that we divide the environment up into patches and solve a system of equations on the basis that the radiosity is constant across each patch.

The commonest approach – bilinear interpolation – is overviewed in Figure 11.10. Here we assume that the curved surface shown in Figure 11.10(a) will exhibit a continuous variation in radiosity value along the dotted line as shown.

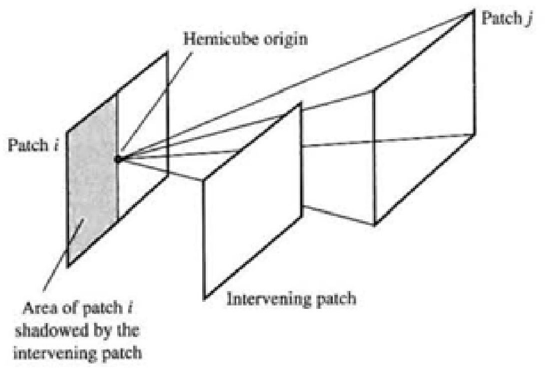


Figure 11.9
All of patch *j* can be seen from the hemicube origin and the F_{dAiAj} approximation falls down.

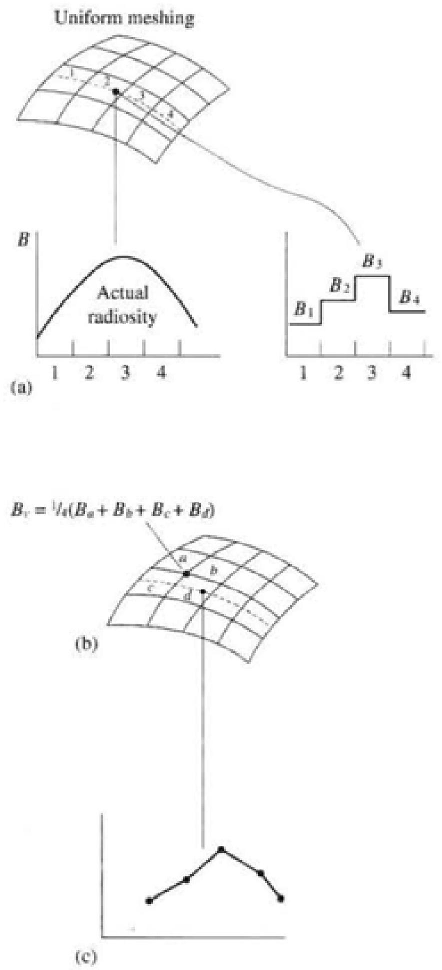


Figure 11.10
Normal reconstruction approach used in the radiosity method.
(a) Compute a constant radiosity solution.
(b) Calculate the vertex radiosities.
(c) Reconstruction by linear interpolation.

The first step in the radiosity method is to compute a constant radiosity solution which will result in a staircase approximation to the continuous function. The radiosity values at a vertex are calculated by averaging the patch radiosities that share the vertex (Figure 11.10(b)). These are then injected into a bilinear interpolation scheme and the surface is effectively Gouraud shaded resulting in the piecewise linear approximation (Figure 11.10(c)).

The most noticeable defect arising out of this process is Mach bands which, of course, we also experience in normal Gouraud shading, where the same interpolation method is used. The 'visual importance' of these can be reduced by using texture mapping but they tend to be a problem in radiosity applications because many of these exhibit large area textureless surfaces – interior walls in buildings, for example. Subdivision meshing strategies also reduce the visibility of Mach bands because by reducing the size of the elements they reduce the difference between vertex radiosities.

More advanced strategies involve surface interpolation methods (Chapter 3). Here the radiosity values are treated as samples of a continuous radiosity function and quadratic or cubic Bézier/B-spline patch meshes are fitted to these. The obvious difficulties with this approach – its inherent cost and the need to prevent wanted discontinuities being smoothed out – has meant that the most popular reconstruction method is still linear interpolation.

11.6.3 Meshing artefacts

One of the most difficult aspects of the radiosity approach, and one that is still a major research area, is the issue of meshing. In the discussions above we have simply described patches as entities into which the scene is divided with the proviso that these should be large to enable a solution which is not prohibitively expensive. However, the way in which we do this has a strong bearing on the quality of the final image. How should we do this so that the appearance of artefacts is minimized? The reason this is difficult is that we can only do this when we already have a solution, so that we can see where the problems occur. Alternatively we have to predict where the problems will occur and subdivide accordingly. We begin by looking at the nature and origin of meshing artefacts.

First some terminology:

- **Meshing** This is a general term used in the context of radiosity to describe either the initial scene subdivision or the act of further subdivision that may take place while a program is executing. The 'initial scene subdivision' may be a general scene database not necessarily created for input to a radiosity renderer. However, for reasons that will soon become apparent it is more likely to be a preprocessed version of such a database or a scene that has been specifically created for a radiosity solution.
- **Patches** These are the entities in the initial representation of the scene. In a standard radiosity solution, where subdivision occurs during the solution, patches form the input to the program.
- **Elements** These are the portions into which patches are subdivided.

The simplest type of meshing artefact – a so-called D^0 discontinuity – is a discontinuity in the value of the radiosity function. The common sources of such a discontinuity are shadow boundaries caused by a point light source and objects which are in contact. In the former case the light source suddenly becomes visible as we move across a surface and the reconstruction and meshing 'spreads' the shadow edge towards the mesh boundaries. Thus the shadow edge will tend to take the shape of the mesh edges giving it a staircase appearance. However, because we tend to use area light sources in radiosity applications the discontinuities that occur are higher than D^0. Nevertheless these still cause visible

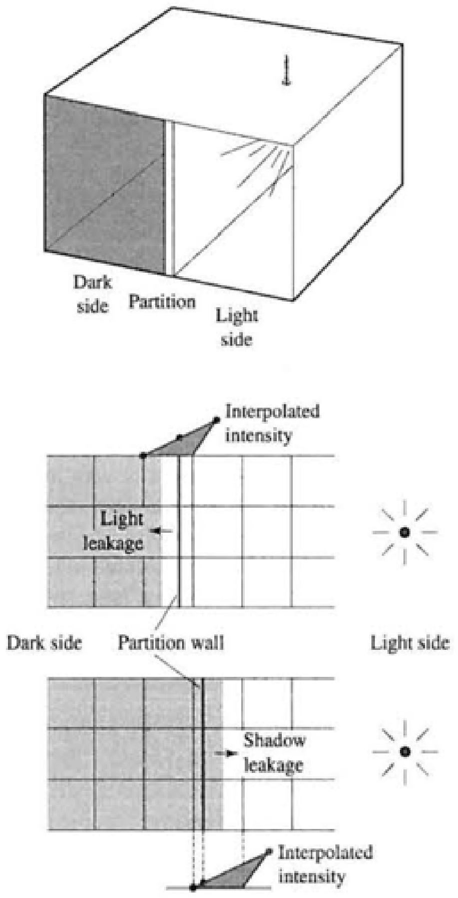


Figure 11.11
Shadow and light leakage.

artefacts. Discontinuities in the derivatives of the radiosity function occur at penumbra and umbra boundaries in shadows in scenes illuminated by area light sources. Again there is 'interference' between the boundaries and the mesh, giving a characteristic 'staircase' appearance to the shadow edges. These are more difficult to deal with than D^0 discontinuities.

When objects are in contact, then unless the intersection boundary coincides with a mesh boundary, shadow or light leakage will occur. The idea is shown in Figure 11.11 for a simple scene. Here, the room is divided by a floor-to-ceiling partition, which does not coincide with patch boundaries on the floor. One half of the room contains a light source and the other is completely dark. Depending on the position of the patch boundaries, the reconstruction will produce either light leakage into the dark region or shadow leakage into the lit region. Figure 18.16 shows the effect of shadow and light leakage for a more complex scene. Despite the fact that the representation contains many more patches than we require for a conventional computer graphics rendering (Gouraud shading) the quality is unacceptably low.

It should be apparent that further subdivision of the scene cannot entirely eliminate shadow and light leakage – it can only reduce it to an acceptable level. It can, however, be eliminated entirely by forcing a meshing along the curve of intersection between objects in contact. Figure 18.18 is the result of meshing the area around a wall light by considering the intersection between the lamp and the wall. Now the wall patch boundaries coincide with the lamp patch boundaries eliminating the leakage that occurred before this meshing.

11.7 Meshing strategies

Meshing strategies that attempt to overcome these defects can be categorized in a number of ways. An important distinction can be made on the basis of when the subdivision takes place:

(1) *A priori* – meshing is completed before the radiosity solution is invoked; that is we predict where discontinuities are going to occur and mesh accordingly. This is also called discontinuity meshing.

(2) *A posteriori* – the solution is initiated with a 'start' mesh which is refined as the solution progresses. This is also called adaptive meshing.

As we have seen, when two objects are in contact, we can eliminate shadow and light leakage by ensuring that mesh element boundaries from each object coincide, which is thus an *a priori* meshing.

Another distinction can be made depending on the geometric nature of the meshing. We can, for example, simply subdivide square patches (non-uniformly) reducing the error to an acceptable level. The commonest approach to date, Cohen and Wallace (1993) term this *h*-refinement. Alternatively we could adopt an approach where the discontinuities in the radiosity function are tracked across a surface and the mesh boundaries placed along the discontinuity boundary. A form of this approach is called *r*-refinement by Cohen and Wallace (1993) where the nodes of the initial mesh are moved in a way that equalizes the error in the elements that share the node. These approaches are illustrated conceptually in Figure 11.12.

11.7.1 Adaptive or *a posteriori* meshing

The classic adaptive algorithm, called substructuring, was described by Cohen *et al.* (1986). Reported before the development of the progressive refinement algorithm, this approach was initially incorporated into a full matrix solution. Adaptive subdivision proceeds by considering the radiosity variation at the nodes or vertices of an element and subdividing if the difference exceeds some threshold.

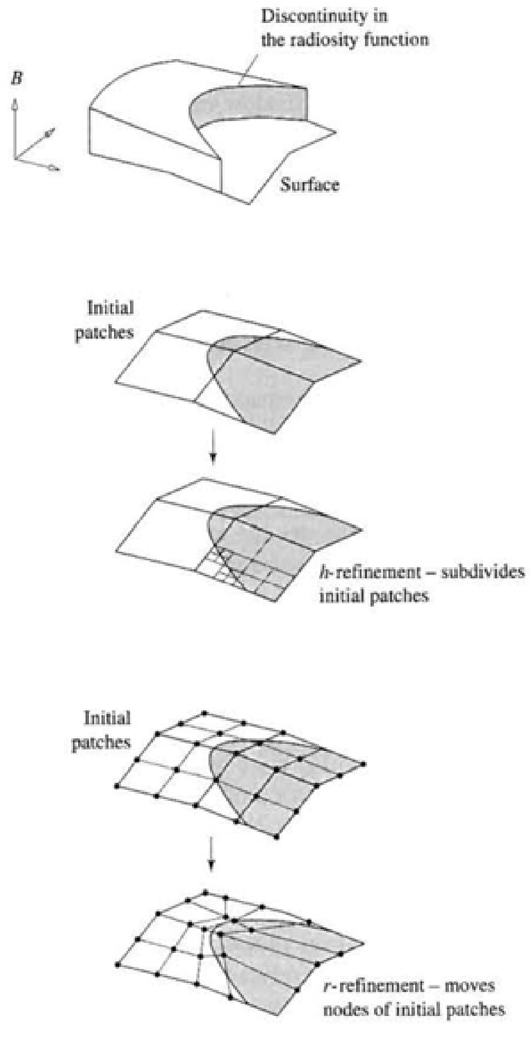


Figure 11.12
Examples of refinement strategies (*a posteriori*).

The idea is to generate an accurate solution for the radiosity of a point from the 'global' radiosities obtained from the initial 'coarse' patch computation. Patches are subdivided into elements. Element-to-patch form factors are calculated where the relationship between element-to-patch and patch-to-patch form factors is given by:

$$F_{ij} = \frac{1}{A_i} \sum_{q=1}^{R} F_{(iq)j} A_{(iq)}$$

where:

F_{ij} is the form factor from patch i to patch j
$F_{(iq)j}$ is the form factor from element q of patch i to patch j
$A_{(iq)}$ is the subdivided area of element q of patch i
R is the number of elements in patch i

Patch form factors obtained in this way are then used in a standard radiosity solution.

This increases the number of form factors from $N \times N$ to $M \times N$, where M is the total number of elements created, and naturally increases the time spent in form factor calculation. Patches that need to be divided into elements are revealed by examining the graduation of the coarse patch solution. The previously calculated (coarse) patch solution is retained and the fine element radiosities are then obtained from this solution using:

$$B_{iq} = E_q + R_q \sum_{j=1}^{N} B_j F_{(iq)j} \qquad [11.4]$$

where:

B_{iq} is the radiosity of element q
B_j is the radiosity of patch j
$F_{(iq)j}$ is the element q to patch j form factor

In other words, as far as the radiosity solution is concerned, the cumulative effect of elements of a subdivided patch is identical to that of the undivided patch; or, subdividing a patch into elements does not affect the amount of light that is reflected by the patch. So after determining a solution for patches, the radiosity within a patch is solved independently among patches. In doing this, Equation 11.4 assumes that only the patch in question has been subdivided into elements – all other patches are undivided. The process is applied iteratively until the desired accuracy is obtained. At any step in the iteration we can identify three stages:

(1) Subdividing selected patches into elements and calculating element-to-patch form factors.
(2) Evaluating a radiosity solution using patch-to-patch form factors.
(3) Determining the element radiosities from the patch radiosities.

Where stage (2) just occurs for the first iteration, the coarse patch radiosities are calculated once only. The method is distinguished from simply subdividing the environment into smaller patches. This strategy would result in $M \times M$ new form factors (rather than $M \times N$) and an $M \times M$ system of equations.

Subdivision of patches into elements is carried out adaptively. The areas that require subdivision are not known prior to a solution being obtained. These areas are obtained from an initial solution and are then subject to a form factor subdivision. The previous form factor matrix is still valid and the radiosity solution is not re-computed.

Only part of the form factor determination is further discretized and this is then used in the third phase (determination of the element radiosities from the coarse patch solution). This process is repeated until it converges to the desired degree of accuracy. Thus image quality is improved in areas that require more accurate treatment. An example of this approach is shown in Figure 11.13. Note the effect on the quality of the shadow boundary. Figure 11.14 shows the same set-up but this time the light source is subdivided to a lower and higher resolution than in Figure 11.13. Although the effect, in this case, of insufficient subdivision of emitting and non-emitting patches is visually similar, the reasons for these discrepancies differ. In the case of non-emitting patches we have changes in reflected light intensity that do not coincide with patch boundaries. We increase the number of patches to capture the discontinuity. With emitting patches the problem is due to the number of hemicube emplacements per light source. Here we increase the number of patches that represent the emitter because each hemicube emplacement reduces a light to a single source and we need a sufficiently dense array of these to represent the spatial extent of the

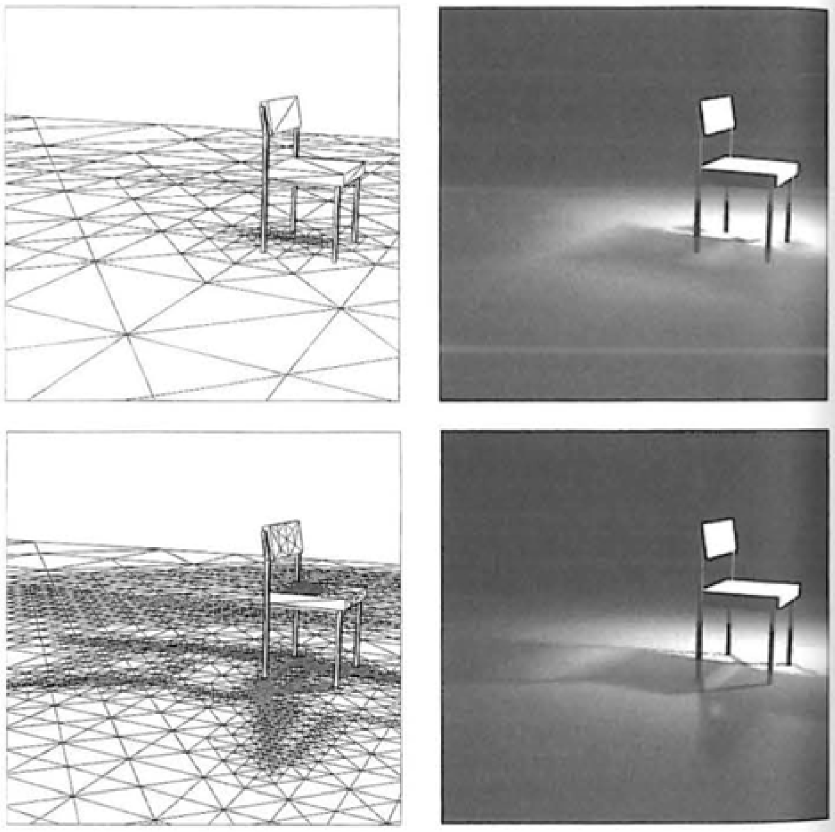


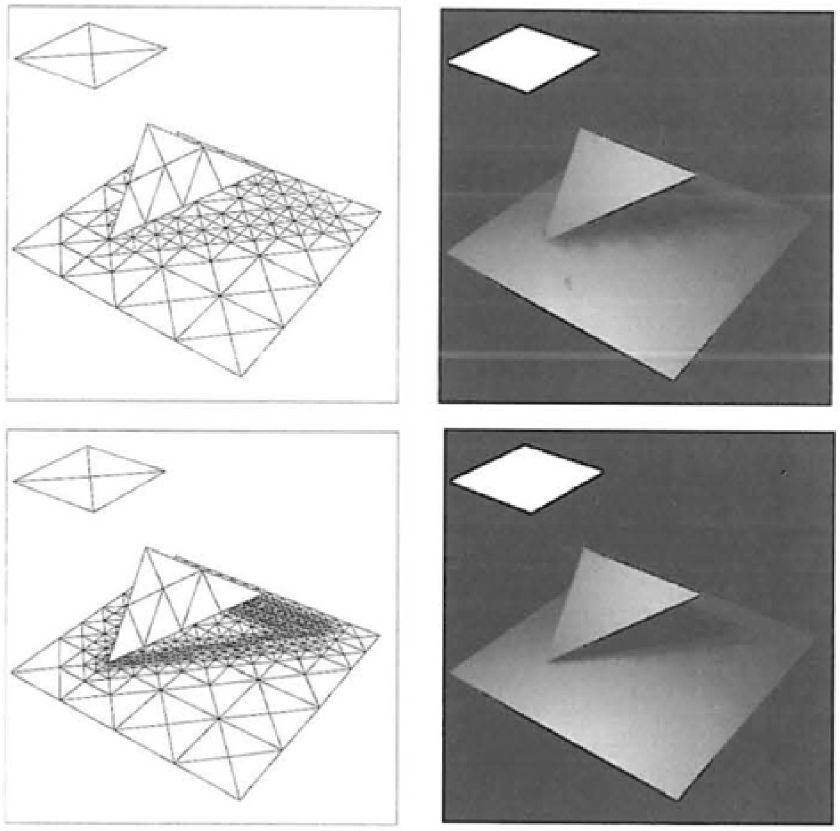
Figure 11.13
Adaptive subdivision and shadows.

(a) Shape and shadow areas do not correspond to shape of the occluder.

emitter. In this case we are subdividing a patch (the emitter) over whose surface the light intensity will be considered uniform.

Adaptive subdivision can be incorporated in the progressive refinement method. A naive approach would be to compute the radiosity gradient and subdivide based on the contribution of the current shooting patch. However, this approach can lead to unnecessary subdivisions. The sequence, shown in Figure 11.15 shows the difficulties encountered as subdivision, performed after every iteration, proceeds around one of the wall lights. Originally two large patches situated away from the wall provide general illumination of the object. This immediately causes subdivision around the light–wall boundary because the program detects a high difference between vertices belonging to the same patches. These patches have vertices both under the light and on the wall. However, this subdivision is not fine enough and as we start to shoot energy from the light source itself light leakage begins to occur. Light source patches continue to shoot energy in the order in which the model is stored in the data-

Figure 11.13 *continued*



(b) Boundary of shadow is jagged.

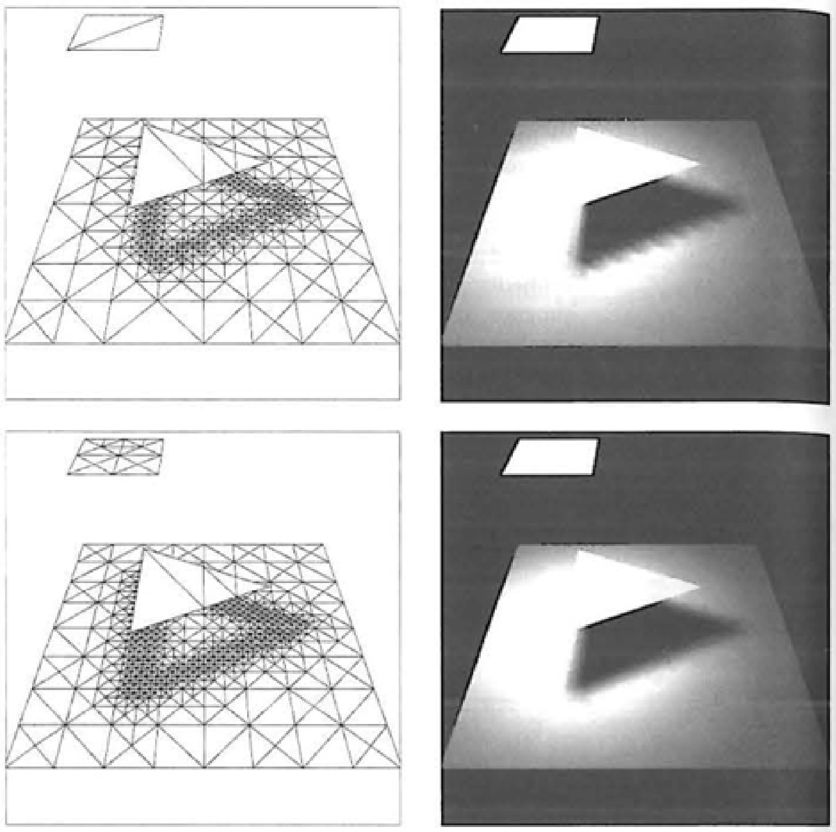


Figure 11.14
The effect of insufficient subdivision of emitters.

base and we spiral up the sphere, shooting energy onto its inside and causing more and more light leakage. Eventually the light emerges onto the wall and brightens up the appropriate patches. As the fan of light rotates above the light more and more inappropriate subdivision occurs. This is because the subdivision is based on the current intensity gradients which move on as further patches are shot. Note in the final frame this results in a large degree of subdivision in an area of highlight saturation. These redundant patches slow the solution down more and more and we are inadvertently making things worse as far as execution time is concerned.

Possible alternative strategies are:

(1) Limit the subdivision by only initiating it after every *n* patches instead of after every patch that is shot.
(2) Limit the initiation of subdivision by waiting until the illumination is representative of the expected final distribution.

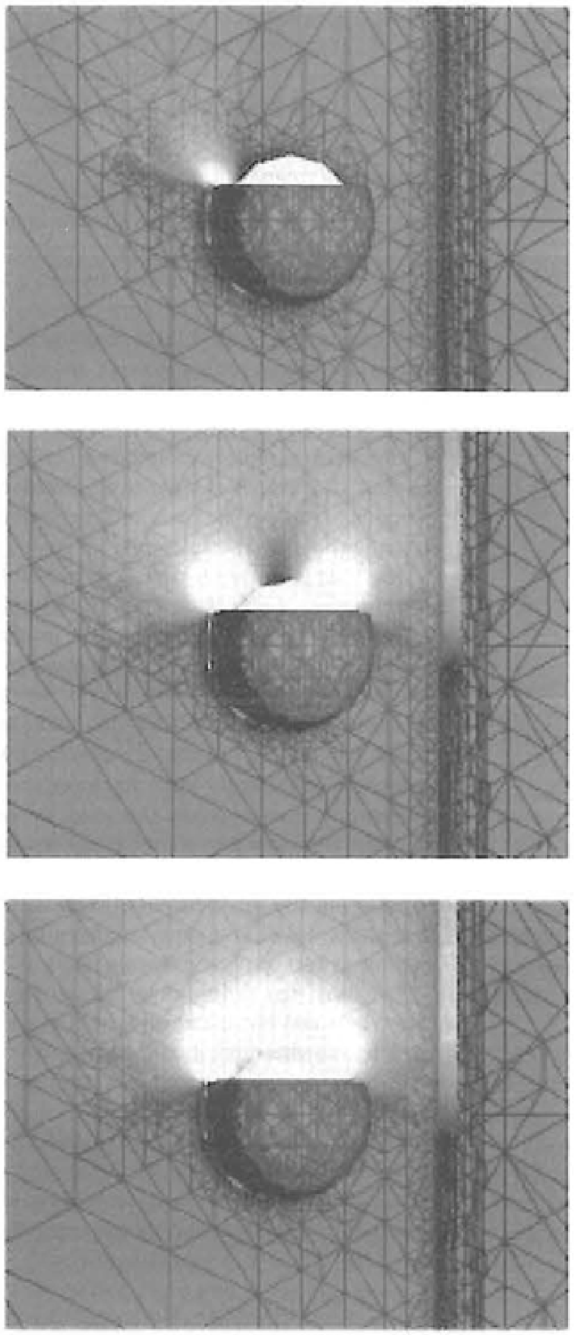


Figure 11.15
The sequence shows the difficulties encountered as subdivision, performed after every iteration, proceeds around one of the wall lights. As the fan of light rotates above the light more and more inappropriate subdivision occurs. This is because the subdivision is based on the current intensity gradients which move on as further patches are shot. Note in the final frame this results in a large degree of subdivision in an area of high light saturation. These redundant patches slow the solution down more and more and we are inadvertently making things worse as far as execution time is concerned.

11.7.2 *A priori* meshing

We will now look at two strategies for *a priori* meshing – processing a scene before it is used in a radiosity solution.

Hierarchical radiosity

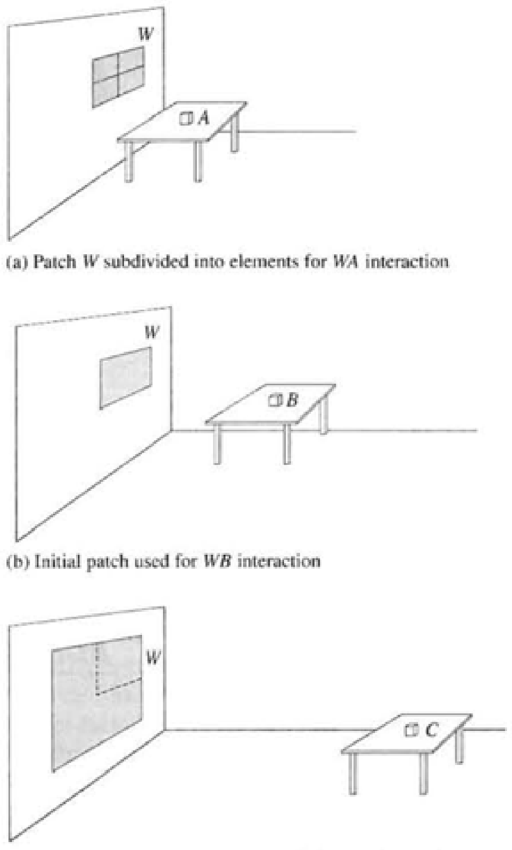
Adaptive subdivision is a special case of an approach which is nowadays known as hierarchical radiosity. Hierarchical radiosity is a generalization of adaptive subdivision – a two-level hierarchy – to a subdivision employing a continuum of levels. The interaction between surfaces is then computed using a form factor appropriate to the geometric relationship between the two surfaces. In other words, the hierarchical approach attempts to limit form factor calculation time by limiting the accuracy of the calculation to an amount determined by the distance between patches.

Hierarchical radiosity can be embedded in a progressive refinement approach making an *a posteriori* algorithm; alternatively the hierarchical subdivision can be made on an *a priori* basis and the system then solved. In what follows we will describe the *a priori* framework.

The idea is easily illustrated in principle. Figure 11.16 shows a wall patch *W* and three small objects *A*, *B* and *C* located at varying distances from *W*. The distance from *W* to *A* is comparable to its dimension and we assume that *W* has to be subdivided to calculate the changes in illumination in the vicinity of *A* due to light emitted or reflected from *W*. In the case of *B* we assume that the whole of patch *W* can be used. Detailed variation of the radiosity in the vicinity of *B* due to patch *W* is not unduly affected by subdividing *B*. The distance to *C*, we assume, is sufficiently large to make the form factor between *W* and *C* correspondingly small, and in this case we can consider a larger area on the wall merging *W* into a patch four times its area. Thus for the three interactions we use either a subdivided *W*, the whole of *W* or *W* as part of a larger entity when considering the interaction between the wall and the objects *A*, *B* and *C*. Note that this implies not only subdivision of patches but the opposite process – agglomeration of patches into groups.

The idea, first proposed by Hanrahan *et al.* (1991), proposes that if the form factor between two patches currently under consideration exceeds a threshold, then to use these patches at their current size will introduce an unacceptable error into the solution and the patches should be subdivided. Compared with the strategy in the previous section we are taking our differential threshold one stage further back in the overall process. Instead of comparing the difference between the calculated radiosity of neighbouring patches and subdividing and re-calculating form factors if necessary, we are looking directly at the form factors themselves and subdividing until the form factor falls below a threshold. This idea is easily demonstrated for the simple case of two patches sharing a common border. Figure 11.17 shows the geometric effect of subdivision based on a form factor threshold. The initial form factor estimate is large and the patches

Figure 11.16
Hierarchical radiosity: scene subdivision is determined by energy interchange.



(a) Patch W subdivided into elements for WA interaction

(b) Initial patch used for WB interaction

(c) Patch W merged into a larger patch for WC interaction

are subdivided into four elements. At the next level of subdivision only two out of 16 form factor estimates exceed the threshold and they are subdivided. It is easily seen from the illustration that in this example the pattern of subdivision 'homes into' the common edge.

A hierarchical subdivision strategy starts with an (initial) large patch subdivision of n patches. This results in $n(n-1)/2$ form factor calculations. Pairs of patches that cannot be used at this initial level are then subdivided as suggested by the previous figure, the process continuing recursively. Thus each initial patch is represented by a hierarchy and links. The structure contains both the geometric subdivision and links that tie an element to other elements in the scene. A node in the hierarchy represents a group of elements and a leaf node a single element. To make this process as fast as possible a crude estimate of the form factor can be used. For example, the expression inside the integral definition of the form factor:

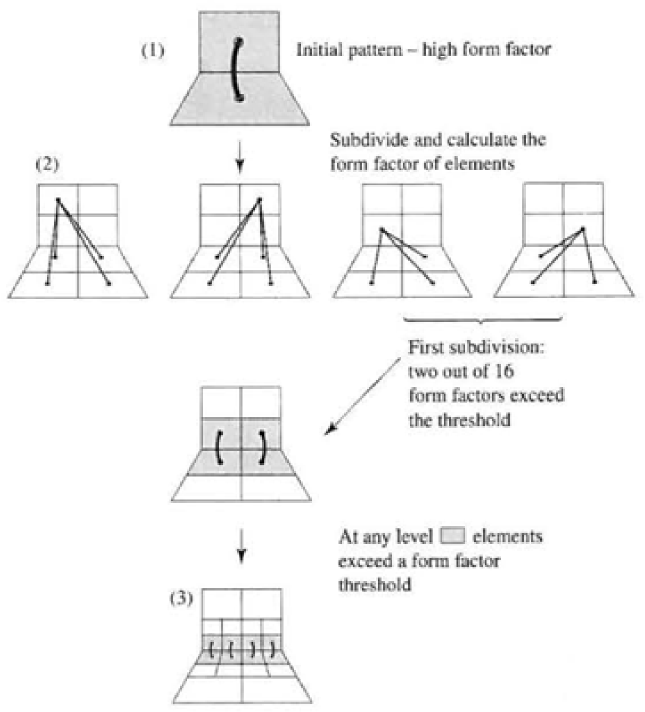

(1)
Initial pattern – high form factor
Subdivide and calculate the form factor of elements
(2)
First subdivision: two out of 16 form factors exceed the threshold
At any level ▨ elements exceed a form factor threshold
(3)

Figure 11.17
Hierarchical radiosity: the geometric effect of subdivision of two perpendicular patches (after Hanrahan *et al.* (1991)).

$$\frac{\cos \phi_i \cos \phi_j}{\pi r^2}$$

can be used but note that this does not take into account any occluding patches.

Thus the stages in establishing the hierarchy are:

(1) Start with an initial patch subdivision. This would normally use much larger patches than those required for a conventional solution.

(2) Recursively apply the following:
   (a) Use a quick estimate of the form factor between pairs of linked surfaces.
   (b) If this falls below a threshold or a subdivision limit is reached, record their interaction at that level.
   (c) Subdivide the surfaces.

It is important to realize that two patches can exhibit an interaction between any pair of nodes at any level in their respective hierarchies. Thus in Figure 11.18 a link is shown between a leaf node in patch *A* and an internal node in patch *C*. The tree shown in the figure for *A* represents the subdivisions necessary for its interaction with patch *B* and that for patch *C* represents its interactions with some other patch *X*. This means that energy transferred from *A* to the internal node in *C* is inherited by all the child nodes below the destination in *C*.

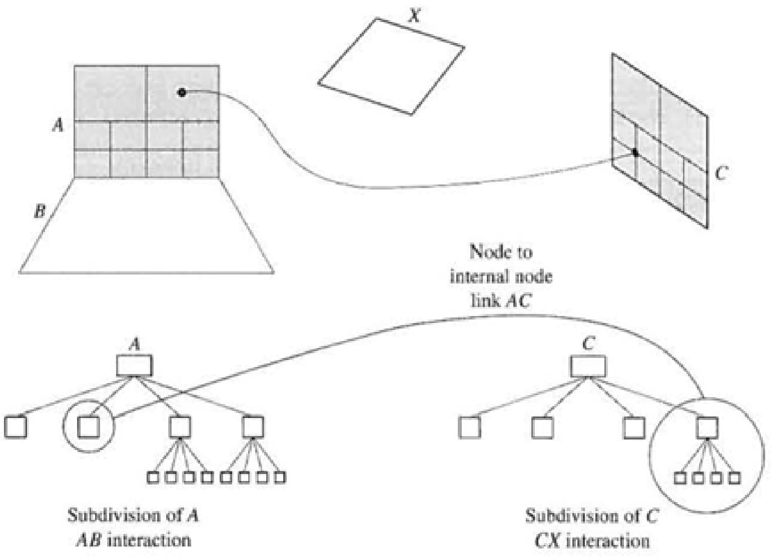

X
A
B
C
Node to internal node link AC
A
C
Subdivision of A AB interaction
Subdivision of C CX interaction

Figure 11.18
Interaction between two patches can consist of energy interchange between any pair of nodes at any level in their respective hierarchy.

Comparing this formulation with the classical full matrix solution we have now replaced the form factor matrix with a hierarchical representation. This implies that a 'gathering' solution proceeds from each node by following all the links from that node and multiplying the radiosity found at the end of the link by the associated form factor. Because links have, in general, been established at any level, the hierarchy needs to be followed in both directions from a node.

The iterative solution proceeds by executing two steps on each root node until a solution has converged. The first step is to gather energy over each incoming link. The second step, known as 'pushpull' pushes a node's reflected radiosity down the tree and pulls it back up again. Pushing involves simply adding the radiosity at each child node. (Note that since radiosity has units of power/unit area the value remains undiminished as the area is subdivided.) The total energy received by an element is the sum of the energy received by it directly plus the sum of the energy received by its parents. When the leaves are reached the process is reversed and the energy is pulled up the tree, the current energy deposited at each node is calculated by averaging the child node contributions.

In effect this is just an elaboration of the Gauss–Siedel relaxation method described in Section 11.3. For a particular patch we are gathering contributions from all other patches in the scene to enable a new estimate of the current patch. The difference now is that the gathering process involves following all links out of the current hierarchy and the hierarchy has to be updated correctly with the bi-directional traversal or pushpull process.

The above algorithm description implies that at each node in the quadtree data structure the following information is available:

gathering and shooting radiosity (B_g and B_s)
emission value (E)
area (A)
reflectivity (ρ)
pointer to four children
pointer to list of (gathering) links (L)

The algorithm itself has a simple elegant structure and following the excellent treatment given in Cohen and Wallace (1993) can be expressed in terms of the following pseudocode:

```
while not converged
   for all surfaces (every root node p)
         GatherRad(p)
   for all surfaces
         PushPull(p,0.0)
```

The top level procedure is a straightforward iterative process which gathers all the energy from the incoming links, pushes it down the structure then pulls the radiosity values back up the hierarchy.

GatherRad(p) calculates the radiosity absorbed and then reflected at node p. It is as follows:

```
GatherRad(p)
   $p.B_g := 0$
   for each link L into p
   $p.B_g := p.B_g + p.\rho(L.F_{pq} * L.q.B_s)$
   for each child r of p
     GatherRad(r)
```

Here q is the element linked to p by L, F_{pq} is the form factor for this link and ρ is the reflectivity of the element p. The radiosity at the destination/shooter end of the link is B_s. This is converted into the reflected radiosity B_g at the source/gatherer end of the link by multiplying it by the form factor F_{pq} and the reflectivity ρ.

PushPullRad(p,B) can be viewed as a procedure that distributes the energy correctly throughout the hierarchy balancing the tree. It is as follows:

```
PushPullRad(p,$B_{down}$)
   if p is a leaf then $B_{up} := p.E + p.B_g + B_{down}$
   else $B_{up} := 0$; for each child node r of p
                 $B_{up} := B_{up} + (r.A/p.A)$ * PushPullRad(r, $p.B_g + B_{down}$)
   $p.B_s := B_{up}$
   return $B_{up}$
```

The procedure is first called at the top of the hierarchy with the gathered radiosity at that level. The recursion has the effect of passing or pushing down this radiosity onto the child nodes. At each internal node the gathered power is added to the inherited power accumulated along the downwards path. When a leaf node is reached any emission is added into the gathered radiosity for that node and the result assigned to the shooting radiosity for that node. The recursion then unwinds pulling the leaf node radiosity up the tree and performing an area weighting at each node.

Although hierarchical radiosity is an efficient method and one that can be finely controlled (the accuracy of the solution depends on the form factor tolerance and the minimum subdivision area) it still suffers from shadow leaks and jagged shadow boundaries because it subdivides the environment regularly (albeit non-uniformly) without regard to the position of shadow boundaries. Reducing the value of the control parameters to give a more accurate solution can still be prohibitively expensive. This is the motivation of the approach described in the next section.

Finally we can do no better than to quote from the original paper, in which the authors give their inspiration for the approach:

The hierarchical subdivision algorithm proposed in this paper is inspired by methods recently developed for solving the *N*-body problem. In the *N*-body problem, each of the n particles exerts a force on all the other n–1 particles, implying $n(n-1)/2$ pairwise interactions. The fast algorithm computes all the forces on a particle in less than quadratic time, building on two key ideas:

(1) Numerical calculations are subject to error, and therefore, the force acting on a particle need only be calculated to within the given precision.

(2) The force due to a cluster of particles at some distant point can be approximated, within the given precision, with a single term – cutting down on the total number of interactions.

Discontinuity meshing

The commonest, and simplest, type of *a priori* meshing is to take care of the special case of interpenetrating geometry (D^0) as we suggested at the beginning of this section. This is mostly done semi-manually when the scene is constructed and disposes of shadow and light leakage – the most visible radiosity artefact. The more general approaches attend to higher-order discontinuities. D^1 and D^2 discontinuities occur when an object interacts with an area light source – the characteristic penumbra–umbra transition within a shadow area – as described in Chapter 9.

As we have seen, common *a posteriori* methods generally approach the problem by subdividing in the region of discontinuities in the radiosity function and can only eliminate errors by resorting to higher and higher meshing densities. The idea behind discontinuity meshing is to predict where the discontinuities are going to occur and to align the mesh edges exactly with the path of the discontinuity. This approach is by definition an *a priori* method. We predict where the discontinuities will occur and mesh, before invoking the solution phase so that when the solution proceeds there can be no artefacts present due to the non-alignment of discontinuities and mesh edges.

To predict the position of the discontinuities shadow detection algorithms are used and the problem is usually couched in terms of visual events and critical surfaces. Two types of visual events can be considered VE and EEE. VE or vertex–edge events occur when a vertex of a source 'crosses' an edge of an occluding polygon known in this context as a receiver. Figure 11.19 shows the interaction between a vertex of a triangular source and an edge of a rectangular occlude. The edge and vertex together form a critical surface whose intersection with a receiving surface forms part of the outer penumbra boundary. For each edge of the occluder a critical surface can be defined with respect to each vertex in the source. We can also define EV events which occur due to the interaction of a source edge with a receiver polygon.

VE events can cause both D^1 and D^2 discontinuities as Figures 11.20 and 11.21 demonstrate. Figure 11.20 shows the case of a D^1 discontinuity. Here there is the coincidence that the edge of the occluder and the source are parallel. Both vertices V_1 and V_2 contribute to the penumbra. As we travel outwards from the umbra along path *xy*, the visible area of the source increases linearly and the radiance exhibits piecewise linearity or D^1 discontinuities. A D^2 discontinuity caused by a VE event is shown in Figure 11.21. In this case, a single vertex of the light source is involved along the path *xy*. As we travel outwards from the umbra the visible area of the source increases quadratically and the radiance exhibits D^2 discontinuities.

EEE or edge–edge–edge events occur when we have multiple occluders. The important difference here is that the boundary of the penumbra – the critical curve – is no longer a straight line as it was in the previous VE examples but a conic. The corresponding discontinuities in the radiance function along the curve are D^2. Also the critical surface is no longer a segment of a plane but is a (ruled) quadric surface.

Visual events can occur for any edges and vertices of any object in the scene. For a scene with n objects there can be $O(n^2)$ VE critical surfaces and $O(n^3)$ EEE critical surfaces. Because of the cost and the higher complexity of EEE events approaches have concentrated on detecting VE events.

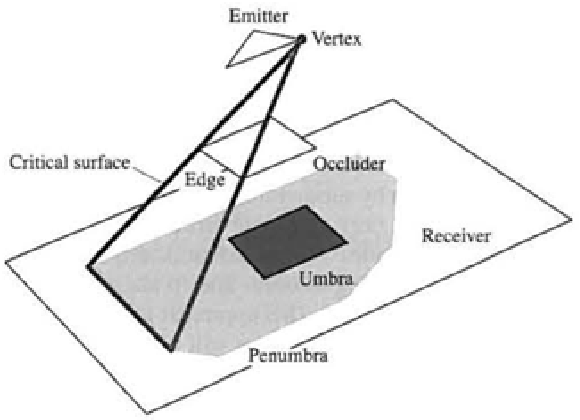


Figure 11.19
VE event: the edge of the occluder and a vertex of the emitter form a critical surface whose intersection with the receiver forms the outer boundary of the penumbra (after Nishita and Nakamae (1985)).

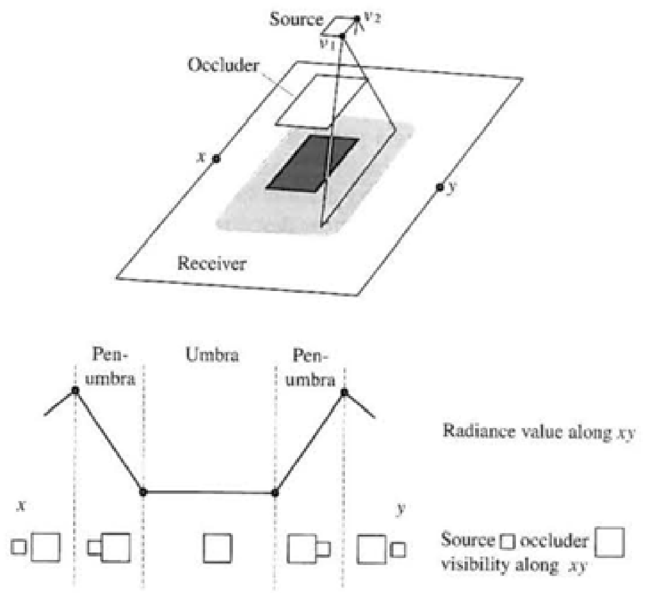


Figure 11.20
VE event causing a D^1 discontinuity (after Lischinski *et al.* (1992)).

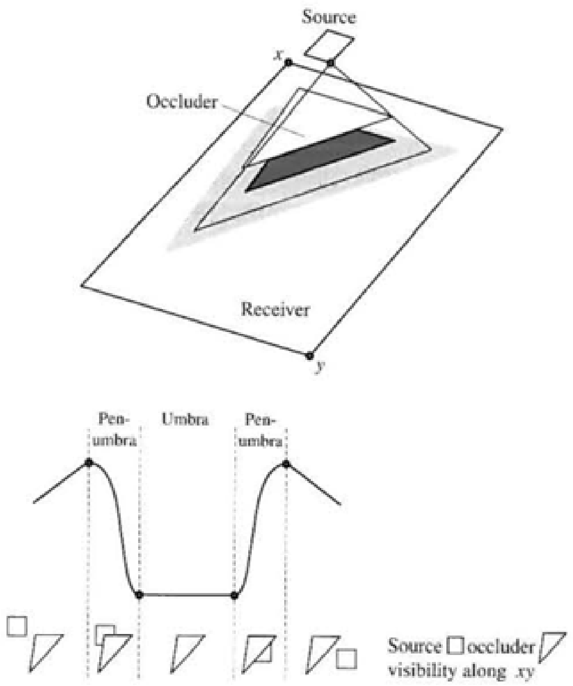


Figure 11.21
VE event causing a D^2 discontinuity (after Lischinski *et al.* (1992)).

A straightforward approach by Nishita and Nakamae (1985) explicitly determines penumbra and umbra boundaries by using a shadow volume approach. For each object a shadow volume is constructed from each vertex in the light source. Thus, there is a volume associated with each light source vertex just as if it were a point source. The intersection of all volumes on the receiving surface forms the umbra and the penumbra boundary is given by the convex hull containing the shadow volumes. An example is shown in Figure 11.22.

We will now describe in some detail a later and more elaborate approach by Lischinski *et al.* (1992) to discontinuity meshing. This integrates discontinuity meshing into a modified progressive refinement structure and deals only with VE (and EV) events. This particular algorithm is representative in that it deals with most of the factors that must be addressed in a practical discontinuity meshing approach including handling multiple light sources and reconstruction problems.

Lischinski *et al.* build a separate discontinuity mesh for each source, accumulating the results into a final solution. The scene polygons are stored as a BSP tree which means that they can be fetched in front-to-back order from a source vertex. For a source the discontinuities that are due to single VE events are located as follows. Figure 11.23 shows a single VE event generating a wedge defined by the vertex and projectors through the end points of the edge, *E*. The event is processed by fetching the polygons in the order *A*, *B* and *C*. *A* is nearer to the source than *E* and is thus not affected by the event. If a surface (*B* and *C*) faces the source then the intersection of the wedge with the surface adds a

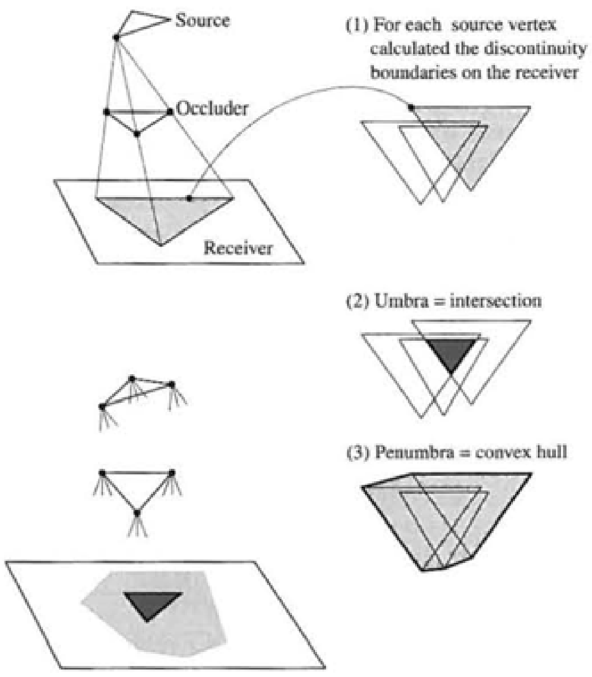

Source
Occluder
Receiver
(1) For each source vertex calculated the discontinuity boundaries on the receiver
(2) Umbra = intersection
(3) Penumbra = convex hull

Figure 11.22
Umbra and penumbra from shadow volumes formed by VE events.

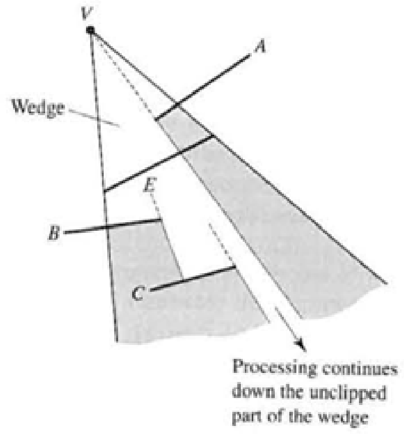


Figure 11.23
Processing a VE wedge (after Lischinski *et al.* (1992)).

discontinuity to that surface. The discontinuity is 'inserted' into the mesh of the surface. As each surface is processed it clips out part of the wedge and the algorithm proceeds 'down' only the unclipped part of the wedge. When the wedge is completely clipped the processing of that particular VE event is complete.

The insertion of the discontinuity into the mesh representing the surface is accomplished by using a DM tree which itself consists of two components – a two-dimensional BSP tree connecting into a winged edge data structure (Mantyla 1988) representing the interconnection of surface nodes. The way in which this works is shown in Figure 11.24 for a single example of a vertex generating three VE events which plant three discontinuity/critical curves on a receiving surface. If the processing order is *a*, *b*, *c* then the line equation for *a* appears as the root node and splitting it into two regions as shown. *b*, the next wedge to be processed is checked against region R_1 which splits into R_{11} and R_{12} and so on.

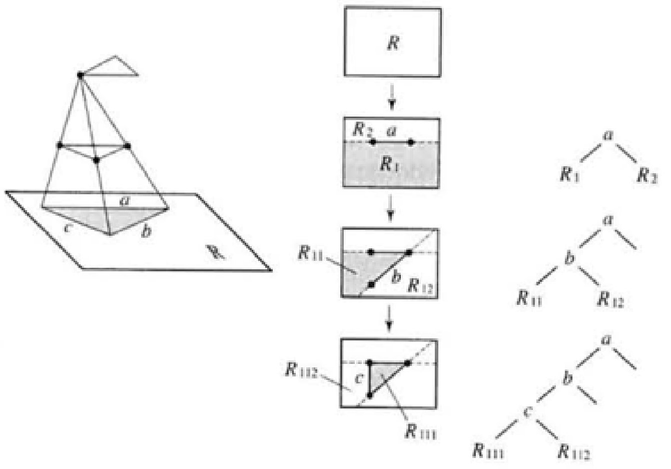


Figure 11.24
Constructing a DM tree for a single VE event (after Lischinski *et al.* (1992)).

12 Ray tracing strategies

Introduction – Whitted ray tracing

In Chapter 10 we gave a brief overview of Whitted ray tracing. We will now describe this popular algorithm in detail. Although it was first proposed in by Appel (1968), ray tracing is normally associated with Whitted's classic paper (1980). We use the term 'Whitted ray tracing' to avoid the confusion that has arisen due to the proliferation of adjectives such as 'eye', forward' and 'backward' to describe ray tracing algorithms.

Whitted ray tracing is an elegant partial global illumination algorithm that combines the following in a single model:

- Hidden surface removal.
- Shading due to direct illumination.
- Global specular interaction effects such as the reflection of objects in each other and refraction of light through transparent objects.
- Shadow computation (but only the geometry of hard-edged shadows is calculated).

It usually 'contains' a local reflection model such as the Phong reflection model, and the question arises: why not use ray tracing as the standard approach to rendering, rather than using a Phong approach with extra algorithms for hidden surface removal, shadows and transparency? The immediate answer to this is cost. Ray tracing is expensive, particularly for polygon objects because effectively each polygon is an object to be ray traced. This is the dilemma of ray tracing. It can only function in reasonable time if the scene is made up of 'easy' objects. Quadric objects, such as spheres are easy, and if the object is polygonal the number of facets needs to be low for the ray tracer to function in reasonable time. If the scene is complex (many objects each with many polygons) then the basic algorithm needs to be burdened with efficiency schemes whose own cost tends to be a function of the complexity of the scene. Much of the research into ray tracing in the 1980s concentrated on the efficiency issue. However, we are just about at a point in hardware development where ray tracing is a viable alternative for a practical renderer and the clear advantages of the algorithm are beginning to overtake the cost penalties. In this chapter we will develop a program to ray trace spheres. We will then extend the program to enable polygonal objects to be dealt with.

12.1 The basic algorithm

12.1.1 Tracing rays – initial considerations

We have already seen that we trace infinitesimally thin light rays through the scene, following each ray to discover perfect specular interactions. Tracing implies testing the current ray against objects in the scene – intersection testing – to find if the ray hits any of them. And, of course, this is the source of the cost in ray tracing – in a naive algorithm, for each ray we have to test it against all objects in the scene (and all polygons in each object). At each boundary, between air and an object (or between an object and air) a ray will 'spawn' two more rays. For example, a ray initially striking a partially transparent sphere will generate at least four rays for the object – two emerging rays and two internal rays (Figure 10.5). The fact that we appropriately bend the transmitted ray means that geometric distortion due to refraction is taken into account. That is, when we form a projected image, objects that are behind transparent objects are appropriately distorted. If the sphere is hollow the situation is more complicated – there are now four intersections encountered by a ray travelling through the object.

To perform this tracing we follow light beams in the reverse direction of light propagation – we trace light rays from the eye. We do this eye tracing because tracing rays by starting at the light source(s) would be hopelessly expensive. This is because we are only interested in that small subset of light rays which pass through the image plane window.

At each hit point the same calculations have to be made and this implies that the easiest way to implement a simple ray tracer is as a recursive procedure. The recursion can terminate according to a number of criteria:

- It always terminates if a ray intersects a diffuse surface.
- It can terminate when a pre-set depth of trace has been reached.
- It can terminate when the energy of the ray has dropped below a threshold.

The behaviour of such an approach is demonstrated in Figure 18.11 (Colour Plate). Here the trace is terminated at recursives depths of 2, 3 and 4 and unassigned pixels (pixels which correspond to a ray landing on a pure specular surface with no diffuse contribution) are coloured grey. You can see that the grey region 'shrinks into itself' as a function of recursive depth.

12.1.2 Lighting model components

At each point $\boldsymbol{P}$ that a ray hits an object, we spawn in general, a reflected and a transmitted ray. Also we evaluate a local reflection model by calculating $\boldsymbol{L}$ at that point by shooting a ray to the light source which we consider as a point. Thus at each point the intensity of the light consists of up to three components:

- A local component.
- A contribution from a global reflected ray that we follow.
- A contribution from a global transmitted ray that we follow.

We linearly combine or add these components together to produce an intensity for point $\boldsymbol{P}$. It is necessary to include a local model because there may be direct illumination at a hit point. However, it does lead to this confusion. The use of a local reflection model does imply empirically blurred reflection (spread highlights); however, the global reflected ray at that point is not blurred but continues to discover any object interaction along an infinitesimally thin path. This is because we cannot afford to blur global reflected rays – we can only follow the 'central' ray. This results in a visual contradiction in ray traced images, which is that the reflection of the light source in an object – the specular highlight – is blurred, but the images of other objects are perfect. The reason for this is that we want objects to look shiny – by having them exhibit a specular highlight – and include images of other objects. Thus most algorithms use a local and a global specular component.

It is also necessary to account for local diffuse reflection, otherwise we could not have coloured objects. We cannot in ray tracing handle diffuse interaction as we did in radiosity. This would mean spawning, for every hit, a set of diffuse rays that sampled the hemispherical set of diffuse rays that occurs at the hit point on the surface of the object, if it happens to be diffuse. Each one of these rays would have to be followed and may end up on a diffuse surface and a combinatorial explosion would develop that no machine could cope with. This problem is the motivation for the development of Monte Carlo methods such as path tracing, as we saw in Chapter 10.

If a ray hits a pure diffuse surface then the trace is terminated. Thus we have the situation where the result of the local model computation at each hit point is passed up the tree along with the specular interaction.

12.1.3 Shadows

Shadows are easily included in the basic ray tracing algorithm. We simply calculate $\boldsymbol{L}$, the light direction vector, and insert it into the intersection test part of the algorithm. That is, $\boldsymbol{L}$ is considered a ray like any other. If $\boldsymbol{L}$ intersects any objects, then the point from which $\boldsymbol{L}$ emanates is in shadow and the intensity of direct illumination at that point is consequently reduced (Figure 12.1). This generates hard-edged shadows with arbitrary intensity. The approach can also lead to great expense. If there are n light sources, then we have to generate n intersection tests. We are already spawning two rays per hit point plus a shadow ray, and for n light sources this becomes $(n + 2)$ rays. We can see that as the number of light sources increases shadow computations are quickly going to predominate since the major cost at each hit point is the cost of the intersection testing.

In an approach by Haines and Greenberg (1986) a 'light buffer' was used as a shadow testing accelerator. Shadow testing times were reduced, using this procedure, by a factor of between 4 and 30. The method pre-calculates for each light source, a light buffer which is a set of cells or records, geometrically disposed as two-dimensional arrays on the six faces of a cube surrounding a point light source (Figure 12.2). To set up this data structure all polygons in the scene are cast or projected onto each face of the cube, using as a projection centre the position of the light source. Each cell in the light buffer then contains a list of polygons that can be seen from the light source. The depth of each polygon is calculated in a local coordinate system based on the light source, and the records are sorted in ascending order of depth. This means that for a particular ray from the eye, there is immediately available a list of those object faces that may occlude the intersection point under consideration.

Shadow testing reduces to finding the cell through which the shadow feeler ray passes, accessing the list of sorted polygons, and testing the polygons in the list until occlusion is found, or the depth of the potentially occluding polygon is greater than that of the intersection point (which means that there is no occlusion because the polygons are sorted in depth order). Storage requirements are prodigious and depend on the number of light sources and the resolution of the

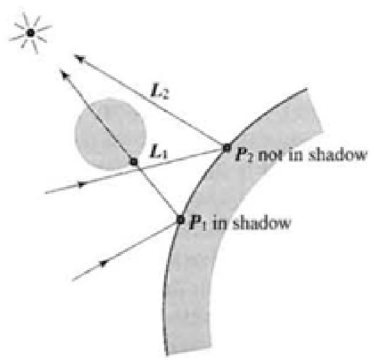


Figure 12.1
Shadow shape is computed by calculating $\boldsymbol{L}$ and inserting it into the intersection tester.

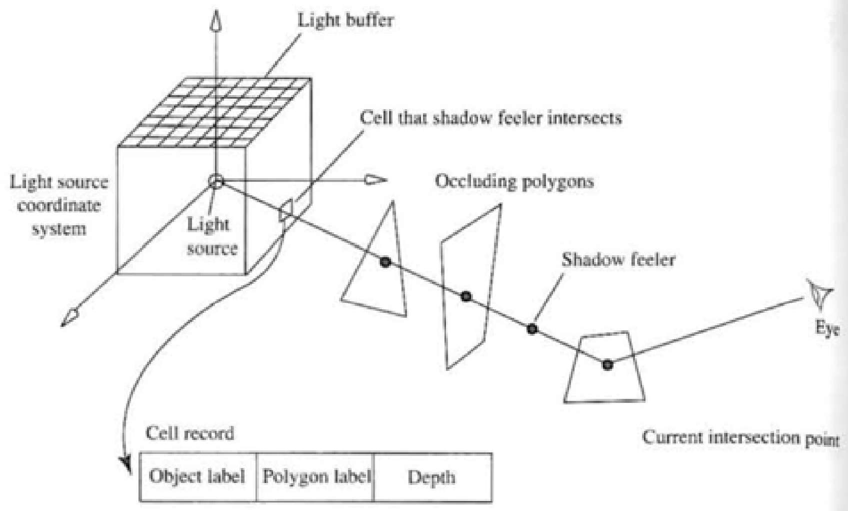


Figure 12.2
Shadow testing accelerator of Haines and Greenberg (1986).

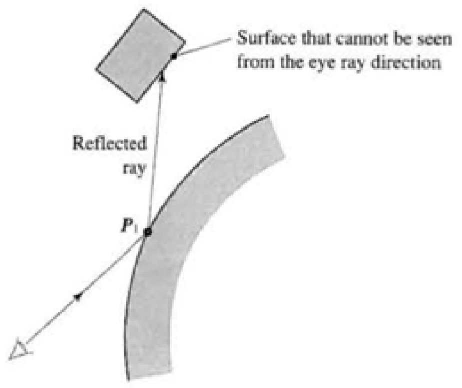
light buffers. (Note the similarity between the light buffer and the radiosity hemicube described in Chapter 11.)

Apart from the efficiency consideration, the main problem with shadows in Whitted ray tracing is that they are hard edged due to the point light source assumption and the light intensity within a shadow area has to be a guess. This is, of course, not inconsistent with the perfect specular interactions that result from tracing a single infinitesimally thin ray from each hit point for each type of interaction. Just as distributed ray tracing (described in Section 10.6) deals with 'blurry' interaction by considering more than one ray per interaction, so it implements soft shadow by firing more than one ray towards a (non-point) light source.

12.1.4 Hidden surface removal

Hidden surface removal is 'automatically' included in the basic ray tracing algorithm. We test each ray against all objects in the scene for intersection. In general this will give us a list of objects which the ray intersects. Usually the intersection test will reveal the distance from the hit point to the intersection and it is simply a matter of looking for the closest hit to find, from all the intersections, the surface that is visible from the ray-initiating view point. A certain subtlety occurs with this model, which is that surfaces hidden, from the point of view of a standard rendering or hidden surface approach, may be visible in ray tracing. This point is illustrated in Figure 12.3 which shows that a surface, hidden when viewed from the eye ray direction, can be reflected in the object hit by the incident ray.

Figure 12.3
A reflected 'hidden' surface.



12.2 Using recursion to implement ray tracing

We will now examine the working of a ray tracing algorithm using a particular example. The example is based on a famous image, produced by Turner Whitted in 1980, and it is generally acknowledged as the first ray traced image in computer graphics. An imitation is shown in Figure 12.4 (reproduced as a monochrome image here and colour image in the Colour Plate section).

First some symbolics. At every point $\boldsymbol{P}$ that we hit with a ray we consider two major components a local and a global component:

$$\begin{aligned} I(\boldsymbol{P}) &= I_{\text{local}}(\boldsymbol{P}) + I_{\text{global}}(\boldsymbol{P}) \\ &= I_{\text{local}}(\boldsymbol{P}) + k_{\text{rg}}\, I(\boldsymbol{P}_{\text{r}}) + k_{\text{tg}} I(\boldsymbol{P}_{\text{t}}) \end{aligned}$$

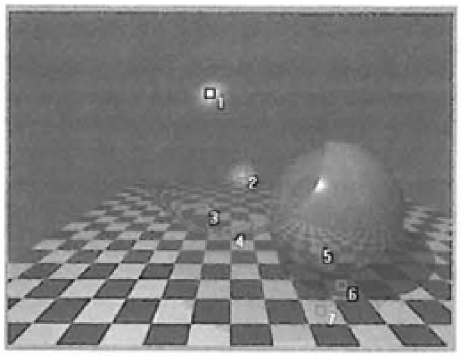
where:

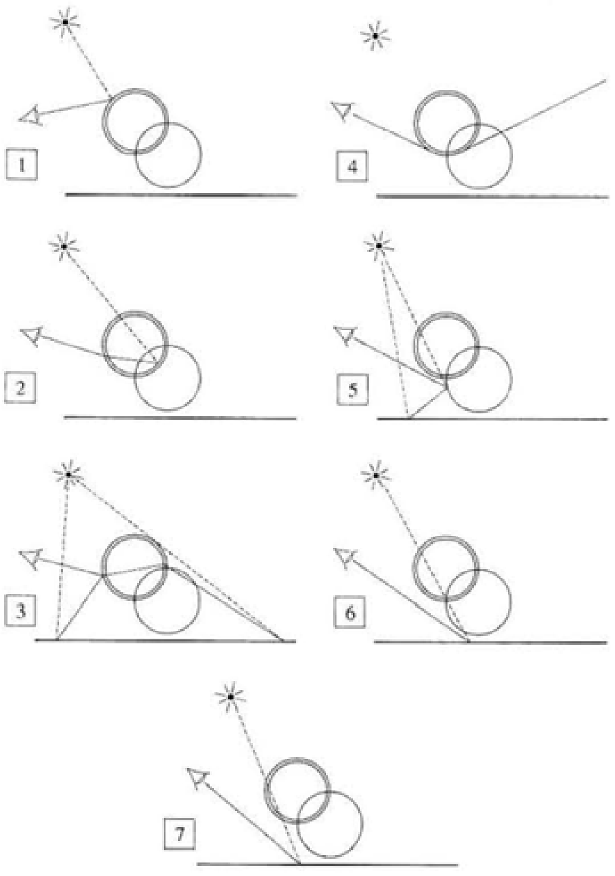
$\boldsymbol{P}$ is the hit point
$\boldsymbol{P}_{\text{r}}$ is the hit point discovered by tracing the reflected ray from $\boldsymbol{P}$
$\boldsymbol{P}_{\text{t}}$ is the hit point discovered by tracing the transmitted ray from $\boldsymbol{P}$
k_{rg} is the global reflection coefficient
k_{tg} is the global transmitted coefficient

This recursive equation emphasizes that the illumination at a point is made up of three components, a local component, which is usually calculated using a Phong local reflection model, and a global component, which is evaluated by finding $\boldsymbol{P}_{\text{r}}$ and $\boldsymbol{P}_{\text{t}}$ and recursively applying the equation at these points. The overall process is sometimes represented as a tree as we indicated in Figure 10.5.

A procedure to implement ray tracing is easily written and has low code complexity. The top-level procedure calls itself to calculate the reflected and transmitted rays. The geometric calculation for the reflected and transmitted ray directions are given in Chapter 1, and details of intersection testing a ray with a sphere will also be found there.

Figure 12.4
The Whitted scene (see also Colour Plate section).





The basic control procedure for a ray tracer consists of a simple recursive procedure that reflects the action at a node where, in general, two rays are spawned. Thus the procedure will contain two calls to itself, one for the transmitted and one for the reflected ray. We can summarize the action as:

```
ShootRay (ray structure)
    intersection test
    if ray intersects an object
        get normal at intersection point
        calculate local intensity ($I_{local}$)
        decrement current depth of trace
        if depth of trace > 0
            calculate and shoot the reflected ray
            calculate and shoot the refracted ray
```

where the last two lines imply a recursive call of ShootRay(). This is the basic control procedure. Around the recursive calls there has to be some more detail which is:

```
Calculate and shoot reflected ray elaborates as
    if object is a reflecting object
        calculate reflection vector and include in the ray structure
        Ray Origin := intersection point
        Attenuate the ray (multiply the current $k_{rg}$ by its value at the
        previous invocation)
        ShootRay(reflected ray structure)
        if reflected ray intersects an object
            combine colours ($k_{rg}$ $I$) with $I_{local}$
Calculate and shoot refracted ray elaborates as
    if object is a refracting object
        if ray is entering object
            accumulate refractive index
            increment number of objects that the ray is currently
            inside
            calculate refraction vector and include in refracted ray
            structure
        else
            de-accumulate refractive index
            decrement number of objects that the ray is currently
            inside
            calculate refraction vector and include in refracted ray
            structure
        Ray origin := intersection point
        Attenuate ray ($k_{tg}$)
        if refracted ray intersects an object
            combine colours ($k_{tg}$ $I$) with $I_{local}$
```

The ray structure needs to contain at least the following information:

- Origin of the ray.
- Its direction.
- Its intersection point.
- Its current colour at the intersection point.
- Its current attenuation.
- The distance of the intersection point from the ray origin.
- The refractive index the ray is currently experiencing.
- Current depth of the trace.
- Number of objects we are currently inside.

Thus the general structure is of a procedure calling itself twice for a reflected and refracted ray. The first part of the procedure finds the object closest to the ray start. Then we find the normal and apply the local shading model, attenuating the light source intensity if there are any objects between the intersection point ***P*** and the object. We then call the procedure recursively for the reflected and transmitted ray.

The number of recursive invocations of ShootRay() is controlled by the depth of trace parameter. If this is unity the scene is rendered just with a local reflection model. To discover any reflections of another object at a point ***P*** we need a depth of at least two. To deal with transparent objects we need a depth of at least three. (The initial ray, the ray that travels through the object and the emergent ray have to be followed. The emergent ray returns an intensity from any object that it hits.)

12.3 The adventures of seven rays – a ray tracing study

Return to Figure 12.4. We consider the way in which the ray tracing model works in the context of the seven pixels shown highlighted. The scene itself consists of a thin walled or hollow sphere, that is almost perfectly transparent, together with a partially transparent white sphere, both of which are floating above the ubiquitous red and yellow chequerboard. Everywhere else in object space is a blue background. The object properties are summarized in Table 12.1. Note that this model allows us to set k_s to a different value from k_{rg} – the source of the contradiction mentioned in Section 12.1.2; reflected rays are treated differently depending on which component (local or global) is being considered.

Consider the rays associated with the pixels shown in Figure 10.4.

Ray 1
This ray is along a direction where a specular highlight is seen on the highly transparent sphere. Because the ray is near the mirror direction of $\boldsymbol{L}$, the contribution from the specular component in $I_{\text{local}}(\boldsymbol{P})$ is high and the contributions

Table 12.1

Very transparent hollow sphere						
k_d (local)	0.1	0.1	0.1			(low)
k_s (local)	0.8	0.8	0.8			(high)
k_{rg}	0.1	0.1	0.1			(low)
k_{tg}	0.9	0.9	0.9			(high)
Opaque (white) sphere						
k_d (local)	0.2	0.2	0.2			(white)
k_s (local)	0.8	0.8	0.8			(white)
k_{rg}	0.4	0.4	0.4			(white)
k_{tg}	0.0	0.0	0.0			
Chequerboard						
k_d (local)	1.0	0.0	0.0/1.0	1.0	0.0	(high red or yellow)
k_s (local)	0.2	0.2	0.2			
k_{rg}	0					
k_{tg}	0					
Blue background						
k_d (local)	0.1	0.1	1.0			(high blue)
Ambient light	0.3	0.3	0.3			
Light	0.7	0.7	0.7			

from $k_{rg}I(\boldsymbol{P}_r)$ is low. For this object k_d, the local diffuse coefficient is low (it is multiplied by 1 – transparency value) and k_s is high with respect to k_{rg}. However, note that the local contribution only dominates over a very small area of the surface of the object. Also note that, as we have already mentioned, the highlight should not be spread. But if we left it as occupying a single pixel it would not be visible.

Ray 2

Almost the same as ray 1 except that the specular highlight appears on the inside wall of the hollow sphere. This particular ray demonstrates another accepted error in ray tracing. Effectively the ray from the light travels through the sphere without refracting (that is, we simply compare $\boldsymbol{L}$ with the local value of $\boldsymbol{N}$ and ignore the fact that we are now inside a sphere). This means that the specular highlight is in the *wrong* position but we simply accept this because we have no intuitive expectation of the correct position anyway. We simply accept it to be correct.

Ray 3

Ray 3 also hits the thin-walled sphere. The local contribution at all hits with the hollow sphere are zero and the predominant contribution is the chequerboard.

This is subject to slight distortion due to the refractive effect of the sphere walls. The red (or yellow) colour comes from the high k_d in $I_{local}(\boldsymbol{P})$ where $\boldsymbol{P}$ is a point on the chequerboard. k_{rg} and k_{tg} are zero for this surface. Note, however, that we have a mix of two chequerboards. One is as described and the other is the superimposed reflection on the outside surface of the sphere.

Ray 4
Again this hits the thin-walled sphere, but this time in a direction where the distance travelled through the glass is significant (that is, it only travels through the glass it does not hit the air inside) causing a high refractive effect and making the ray terminate in the blue background.

Ray 5
This ray hits the opaque sphere and returns a significant contribution from the local component due to a white k_d (local). At the first hit the global reflected ray hits the chequerboard. Thus there is a mixture of:

white (from the sphere's diffuse component)
red/yellow (reflected from the chequerboard)

Ray 6
This ray hits the chequerboard initially and the colour comes completely from the local component for that surface. However, the point is in shadow and this is discovered by the intersection of the ray $\boldsymbol{L}$ and the opaque sphere.

Ray 7
The situation with this ray is exactly the same as for ray 6 except that it is the thin walled sphere that intersects $\boldsymbol{L}$. Thus the shadow area intensity is not reduced by as much as the previous case. Again we do not consider the recursive effect that $\boldsymbol{L}$ would in fact experience and so the shadow is in effect in the wrong place.

12.4 Ray tracing polygon objects – interpolation of a normal at an intersection point in a polygon

Constraining a modelling primitive to be a sphere or at best a quadric solid is hopelessly restrictive in practice and in this section we will look at ray tracing polygonal objects. Extending the above program to cope with general polygon objects requires the development of an intersection test for polygons (see Section 1.4.3) and a method of calculating or interpolating a normal at the hit point $\boldsymbol{P}$. We remind ourselves that the polygonal facets are only approximations to a curved surface and, just as in Phong shading we need to interpolate, from the vertex normals, an approximation to the surface normal of the 'true' surface that

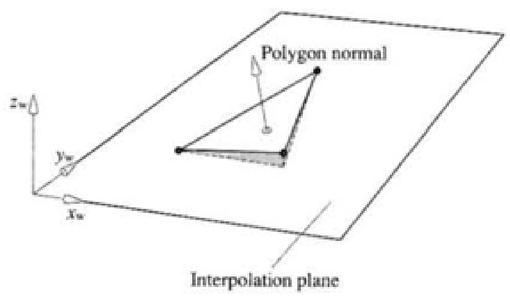


Figure 12.5
A polygon that lies almost in the $x_w y_w$ plane will have a high z_w component. We choose this plane in which to perform interpolation of vertex normals.

the facet approximates. This entity is required for the local illumination component and to calculate reflection and refraction. Recall that in Phong interpolation (see Section 6.3.2) we used the two-dimensional component of screen space to interpolate, pixel by pixel, scan line by scan line, the normal at each pixel projection on the polygon. We interpolated three of the vertex normals using two-dimensional screen space as the interpolation basis. How do we interpolate from the vertex normals in a ray tracing algorithm, bearing in mind that we are operating in world space? One easy approach is to store the polygon normal for each polygon as well as its vertex normals. We find the largest of its three components x_w, y_w and z_w. The largest component identifies which of the three world coordinate planes the polygon is closest to in orientation, and we can use this plane in which to interpolate using the same interpolation scheme as we employed for Phong interpolation (see Section 1.5). This plane is equivalent to the use of the screen plane in Phong interpolation. The idea is shown in Figure 12.5. This plane is used for the interpolation as follows. We consider the polygon to be represented in a coordinate system where the hit point $\boldsymbol{P}$ is the origin. We then have to search the polygon vertices to find the edges that cross the 'medium' axis. This enables us to interpolate the appropriate vertex normals to find $\boldsymbol{N}_a$ and $\boldsymbol{N}_b$ from which we find the required normal $\boldsymbol{N}_p$ (Figure 12.6). Having found the interpolated normal we can calculate the local illumination component and the reflected and the refracted rays. Note that because we

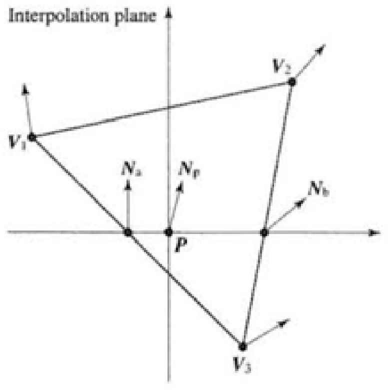


Figure 12.6
Finding an interpolated normal at a hit point $\boldsymbol{P}$.

are 'randomly' interpolating we lose the efficiency advantages of the Phong interpolation, which was incremental on a pixel by pixel, scan line by scan line basis.

We conclude that in ray tracing polygonal objects we incur two significant costs. First, the more overwhelming cost is that of intersection testing each polygon in an object. Second, we have the cost of finding an interpolated normal on which to base our calculations.

12.5 Efficiency measures in ray tracing

12.5.1 Adaptive depth control

The trace depth required in a ray tracing program depends upon the nature of the scene. A scene containing highly reflective surfaces and transparent objects will require a higher maximum depth than a scene that consists entirely of poorly reflecting surfaces and opaque objects. (Note that if the depth is set equal to unity then the ray tracer functions exactly as a conventional renderer, which removes hidden surfaces and applies a local reflection model.)

It is pointed out in Hall and Greenberg (1983) that the percentage of a scene that consists of highly transparent and reflective surfaces is, in general, small and it is thus inefficient to trace every ray to a maximum depth. Hall and Greenberg suggest using an adaptive depth control that depends on the properties of the materials with which the rays are interacting. The context of the ray being traced now determines the termination depth, which can be any value between unity and the maximum pre-set depth.

Rays are attenuated in various ways as they pass through a scene. When a ray is reflected at a surface, it is attenuated by the global specular reflection coefficient for the surface. When it is refracted at a surface, it is attenuated by the global transmission coefficient for the surface. For the moment, we consider only this attenuation at surface intersections. A ray that is being examined as a result of backward tracing through several intersections will make a contribution to the top level ray that is attenuated by several of these coefficients. Any contribution from a ray at depth n to the colour at the top level is attenuated by the product of the global coefficients encountered at each node:

$$k_1k_2 \ldots k_{n-1}$$

If this value is below some threshold, there will be no point in tracing further.

In general, of course, there will be three colour contributions (RGB) for each ray and three components to each of the attenuation coefficients. Thus when the recursive procedure is activated it is given a cumulative weight parameter that indicates the final weight that will be given at the top level to the colour returned for the ray represented by that procedure activation. The correct weight for a new procedure activation is easily calculated by taking the cumulative weight for the ray currently being traced and multiplying it by the reflection or transmission coefficient for the surface intersection at which the new ray is being created.

Another way in which a ray can be attenuated is by passing for some distance through an opaque material. This can be dealt with by associating a transmittance coefficient with the material composing an object. Colour values would then be attenuated by an amount determined by this coefficient and the distance a ray travels through the material. A simple addition to the intersection calculation in the ray tracing procedure would allow this feature to be incorporated.

The use of adaptive depth control will prevent, for example, a ray that initially hits an almost opaque object spawning a transmitted ray that is then traced through the object and into the scene. The intensity returned from the scene may then be so attenuated by the initial object that this computation is obviated. Thus, depending on the value to which the threshold is pre-set, the ray will, in this case, be terminated at the first hit.

For a highly reflective scene with a maximum tree depth of 15, Hall and Greenberg report (1983) that this method results in an average depth of 1.71, giving a large potential saving in image generation time. The actual saving achieved will depend on the nature and distribution of the objects in the scene.

12.5.2 First hit speed up

In the previous section it was pointed out that even for highly reflective scenes, the average depth to which rays were traced was between one and two. This fact led Weghorst *et al.* (1984) to suggest a hybrid ray tracer, where the intersection of the initial ray is evaluated during a preprocessing phase, using a hidden surface algorithm. The implication here is that the hidden surface algorithm will be more efficient than the general ray tracer for the first hit. Weghorst *et al.* (1984) suggest executing a modified Z-buffer algorithm, using the same viewing parameters. Simple modifications to the Z-buffer algorithm will make it produce, for each pixel in the image plane, a pointer to the object visible at that pixel. Ray tracing, incorporating adaptive depth control then proceeds from that point. Thus the expensive intersection tests associated with the first hit are eliminated.

12.5.3 Bounding objects with simple shapes

Given that the high cost of ray tracing is embedded in intersection testing, we can greatly increase the efficiency of a recursive ray tracer by making this part of the algorithm as efficient as possible. An obvious and much used approach is to enclose the object in a 'simple' volume known as a bounding volume. Initially we test the ray for intersection with a bounding volume and only if the ray enters this volume do we test for intersection with the object. Note that we also used this approach in the operation of culling against a view volume (see Chapter 6) and in collision detection (see Chapter 17).

Two properties are required of a bounding volume. First, it should have a simple intersection test – thus a sphere is an obvious candidate. Second, it should efficiently enclose the object. In this aspect a sphere is deficient. If the object is long and thin the sphere will contain a large void volume and many rays will pass the bounding volume test but will not intersect the object. A rectangular solid, where the relative dimensions are adjustable, is possibly the best simple bounding volume. (Details of intersection testing of both spheres and boxes are given in Chapter 1.)

The dilemma of bounding volumes is that you cannot allow the complexity of the bounding volume scheme to grow too much, or it obviates its own purpose. Usually for any scene, the cost of bounding volume calculations will be related to their enclosing efficiency. This is easily shown conceptually. Figure 12.7 shows a two-dimensional scene containing two rods and a circle representing complex polygonal objects. Figure 12.7(a) shows circles (spheres) as bounding volumes with their low enclosing efficiency for the rods. Not only are the spheres inefficient, but they intersect each other, and the space occupied by other objects. Using boxes aligned with the scene axes (axis aligned bounding boxes, or AABBs) is better (Figure 12.7(b)) but now the volume enclosing the sloping rod is inefficient. For this scene the best bounding volumes are boxes with any orientation (Figure 12.7(c)); the cost of testing the bounding volumes increases from spheres to boxes with any orientation. These are known as OBBs.

Weghorst *et al.* (1984) define a 'void' area, of a bounding volume, to be the difference in area between the orthogonal projections of the object and bounding volume onto a plane perpendicular to the ray and passing through the origin of the ray (see Figure 12.8). They show that the void area is a function of object, bounding volume and ray direction and define a cost function for an intersection test:

$$T = b^*B + i^*I$$

where:

T is the total cost function
b is the number of times that the bounding volume is tested for intersection
B is the cost of testing the bounding volume for intersection
i is the number of times that the item is tested for intersection (where $i \le b$)
I is the cost of testing the item for intersection

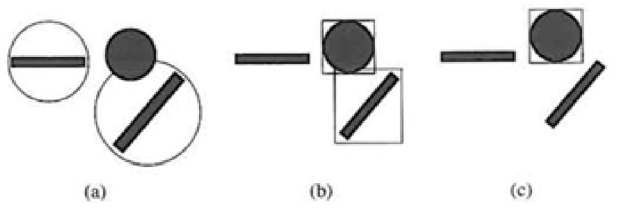
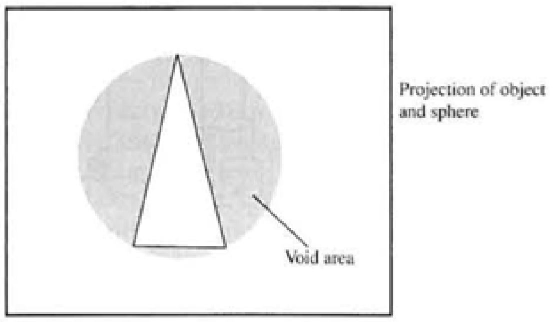


Figure 12.7
Three different bounding volumes, going from (a) to (c). The complexity cost of the bounding volume increases together with its enclosing efficiency.
(a) Circles (spheres) as bounding volumes;
(b) rectangles (boxes) as bounding volumes;
(c) rectangles (boxes) at any orientation.

Figure 12.8
The void area of a bounding sphere.



It is pointed out by the authors that the two products are generally interdependent. For example, reducing B by reducing the complexity of the bounding volume will almost certainly increase i. A quantitive approach to selecting the optimum of a sphere, a rectangular parallelepiped and a cylinder as bounding volumes is given.

12.5.4 Secondary data structures

Another common approach to efficiency in intersection testing is to set up a secondary data structure to control the intersection testing. The secondary data structure is used as a guide and the primary data structure – the object database – is entered at the most appropriate point.

Bounding volume hierarchies

A common extension to bounding volumes, first suggested by Rubin and Whitted (1980) and discussed in Weghorst *et al.* (1984), is to attempt to impose a hierarchical structure of such volumes on the scene. If it is possible, objects in close spatial proximity are allowed to form clusters, and the clusters are themselves enclosed in bounding volumes. For example, Figure 12.9 shows a container (a) with one large object (b) and four small objects (c_1, c_2, c_3 and c_4) inside it. The tree represents the hierarchical relationship between seven boundary extents: a cylinder enclosing all the objects, a cylinder enclosing (b), a cylinder enclosing (c_1, c_2, c_3, c_4) and the bounding cylinders for each of these objects. A ray traced against bounding volumes means that such a tree is traversed from the topmost level. A ray that happened to intersect c_1 in the above example would, of course, be tested against the bounding volumes for c_1, c_2, c_3 and c_4, but only because it intersects the bounding volume representing that cluster. This example also demonstrates that the nature of the scene should enable reasonable clusters of adjacent objects to be selected, if substantial savings over a non-hierarchical bounding scheme are to be achieved. Now the intersection test is implemented as a recursive process, descending through a hierarchy, only from

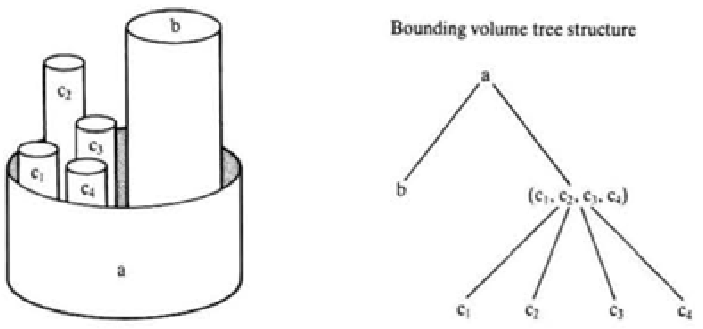


Figure 12.9
A simple scene and the associated bounding cylinder tree structure.

those nodes where intersections occur. Thus a scene is grouped, where possible, into object clusters and each of those clusters may contain other groups of objects that are spatially clustered. Ideally, high-level clusters are enclosed in bounding volumes that contain lower-level clusters and bounding volumes. Clusters can only be created if objects are sufficiently close to each other. Creating clusters of widely separated objects obviates the process. The potential clustering and the depth of the hierarchy will depend on the nature of the scene: the deeper the hierarchy the more the potential savings. The disadvantage of this approach is that it depends critically on the nature of the scene. Also, considerable user investment is required to set up a suitable hierarchy.

Bounding volume hierarchies used in collision detection are discussed in Chapter 17. Although identical in principle, collision detection requires efficient testing for intersection between pairs of bounding volumes, rather than ray/volume testing. OBB hierarchies have proved useful in this and are described in Section 17.5.2.

The use of spatial coherence

Currently, spatial coherence is the only approach that looks like making ray tracing a practical proposition for routine image synthesis. For this reason it is discussed in some detail. Object coherence in ray tracing has generally been ignored. The reason is obvious. By its nature a ray tracing algorithm spawns rays of arbitrary direction anywhere in the scene. It is difficult to use such 'random' rays to access the object data structure and efficiently extract those objects in the path of a ray. Unlike an image space scan conversion algorithm where, for example, active polygons can be listed, there is no *a priori* information on the sequence of rays that will be spawned by an initial or view ray. Naive ray tracing algorithms execute an exhaustive search of all objects after each hit, perhaps modified by a scheme such as bounding volumes, to constrain the search.

The idea behind spatial coherence schemes is simple. The space occupied by the scene is subdivided into regions. Now, rather than check a ray against all objects or sets of bounded objects, we attempt to answer the question: is the region, through which the ray is currently travelling, occupied by any objects? Either there is nothing in this region, or the region contains a small subset of the objects. This group of objects is then tested for intersection with the ray. The size of the subset and the accuracy to which the spatial occupancy of the objects is determined varies, depending on the nature and number of the objects and the method used for subdividing the space.

This approach, variously termed spatial coherence, spatial subdivision or space tracing has been independently developed by several workers, notably Glassner (1984), Kaplan (1985) and Fujimoto *et al.* (1986). All of these approaches involve pre-processing the space to set up an auxiliary data structure that contains information about the object occupancy of the space. Rays are then traced using this auxiliary data structure to enter the object data structure. Note that this philosophy (of pre-processing the object environment to reduce the computational work required to compute a view) was first employed by Schumaker *et al.* (1969) in a hidden surface removal algorithm developed for flight simulators (see Section 6.6.10). In this algorithm, objects in the scene are clustered into groups by subdividing the space with planes. The spatial subdivision is represented by a binary tree. Any view point is located in a region represented by a leaf in the tree. An on-line tree traversal for a particular view point quickly yields a depth priority order for the group clusters. The important point about this algorithm is that the spatial subdivision is computed off-line and an auxiliary structure, the binary tree representing the subdivision, is used to determine an initial priority ordering for the object clusters. The motivation for this work was to speed up the on-line hidden surface removal processing and enable image generation to work in real time.

Dissatisfaction with the bounding volume or extent approach, to reducing the number of ray object intersection tests, appears in part to have motivated the development of spatial coherence methods (Kaplan 1985). One of the major objections to bounding volumes has already been pointed out. Their 'efficiency' is dependent on how well the object fills the space of the bounding volume. A more fundamental objection is that such a scheme may increase the efficiency of the ray–object intersection search, but it does nothing to reduce the dependence on the number of objects in the scene. Each ray must still be tested against the bounding extent of every object and the search time becomes a function of scene complexity. Also, although major savings can be achieved by using a hierarchical structure of bounding volumes, considerable investment is required to set up an appropriate hierarchy, and depending on the nature and disposition of objects in the scene, a hierarchical description may be difficult or impossible. The major innovation of methods described in this section is to make the rendering time constant (for a particular image space resolution) and eliminate its dependence on scene complexity.

The various schemes that use the spatial coherence approach differ mainly in the type of auxiliary data structure used. Kaplan (1985) lists six properties that a practical ray tracing algorithm should exhibit if the technique is to be used in routine rendering applications. Kaplan's requirements are:

(1) Computation time should be relatively independent of scene complexity (number of objects in the environment, or complexity of individual objects), so that scenes having realistic levels of complexity can be rendered.

(2) Per ray time should be relatively constant, and not dependent on the origin or direction of the ray. This property guarantees that overall computation time for a shaded image will be dependent only on overall image resolution (number of first-level rays traced) and shading effects (number of second-level and higher level rays traced). This guarantees predictable performance for a given image resolution and level of realism.

(3) Computation time should be 'rational' and 'interactive' (within a few minutes) on affordable processor systems.

(4) The algorithm should not require the user to supply hierarchical object descriptions or object clustering information. The user should be able to combine data generated at different times, and by different means, into a single scene.

(5) The algorithm should deal with a wide variety of primitive geometric types, and should be easily extensible to new types.

(6) The algorithm's use of coherence should not reduce its applicability to parallel processing or other advanced architectures. Instead, it should be amenable to implementation on such architectures.

Kaplan summarizes these requirements by saying, 'in order to be really usable, it must be possible to trace a large number of rays in a complex environment in a rational, predictable time, for a reasonable cost'.

Two related approaches to an auxiliary data structure have emerged. These involve an octree representation (Fujimoto *et al.* 1986; Glassner 1984) and a data structure called a BSP (binary space partitioning). The BSP tree was originally proposed by Fuchs (1980) and is used in Kaplan (1985).

Use of an octree in ray tracing

An octree (see Chapter 2) is a representation of the objects in a scene that allows us to exploit spatial coherence – objects that are close to each other in space are represented by nodes that are close to each other in the octree.

When tracing a ray, instead of doing intersection calculations between the ray and every object in the scene, we can now trace the ray from subregion to subregion in the subdivision of occupied space. For each subregion that the ray passes through, there will only be a small number of objects (typically one or two) with which it could intersect. Provided that we can rapidly find the node in the octree that corresponds to a subregion that a ray is passing through, we have immediate access to the objects that are on, or close to, the path of the ray. Intersection calculations need only be done for these objects. If space has been subdivided to a level where each subregion contains only one or two objects, then the number of intersection tests required for a region is small and does not tend to increase with the complexity of the scene.

Tracking a ray using an octree

In order to use the space subdivision to determine which objects are close to a ray, we must determine which subregion of space the ray passes through. This involves tracking the ray into and out of each subregion in its path. The main operation required during this process is that of finding the node in the octree, and hence the region in space, that corresponds to a point (x, y, z).

The overall tracking process starts by detecting the region that corresponds to the start point of the ray. The ray is tested for intersection with any objects that lie in this region and if there are any intersections, then the first one encountered is the one required for the ray. If there are no intersections in the initial region, then the ray must be tracked into the next region through which it passes. This is done by calculating the intersection of the ray with the boundaries of the region and thus calculating the point at which the ray leaves the region. A point on the ray a short distance into the next region is then used to find the node in the octree that corresponds to the next region. Any objects in this region are then tested for intersections with the ray. The process is repeated as the ray tracks from region to region until an intersection with an object is found or until the ray leaves occupied space.

The simplest approach to finding the node in the octree that corresponds to a point (x, y, z) is to use a data structure representation of the octree to guide the search for the node. Starting at the top of the tree, a simple comparison of coordinates will determine which child node represents the subregion that contains the point (x, y, z). The subregion, corresponding to the child node, may itself have been subdivided and another coordinate comparison will determine which of its children represents the smaller subregion that contains (x, y, z). The search proceeds down the tree until a terminal node is reached. The maximum number of nodes traversed during this search will be equal to the maximum depth of the tree. Even for a fairly fine subdivision of occupied space, the search length will be short. For example, if the space is subdivided at a resolution of $1024 \times 1024 \times 1024$, then the octree will have depth 10 $(= \log_8(1024 \times 1024 \times 1024))$.

So far we have described a simple approach to the use of an octree representation of space occupancy to speed up the process of tracking a ray. Two variations of this basic approach are described by Glassner (1984) and Fujimoto *et al.* (1986). Glassner describes an alternative method for finding the node in the octree corresponding to a point (x, y, z). In fact, he does not store the structure of the octree explicitly, but accesses information about the voxels via a hash table that contains an entry for each voxel. The hash table is accessed using a code number calculated from the (x, y, z) coordinates of a point. The overall ray tracking process proceeds as described in our basic method.

In Fujimoto *et al.* (1986) another approach to tracking the ray through the voxels in the octree is described. This method eliminates floating point multiplications and divisions. To understand the method it is convenient to start by ignoring the octree representation. We first describe a simple data structure representation of a space subdivision called SEADS (Spatially Enumerated Auxiliary Data Structure). This involves dividing all of occupied space into equally sized voxels regardless of occupancy by objects. The three-dimensional grid obtained in this way is analogous to that obtained by the subdivision of a two-dimensional graphics screen into pixels. Because regions are subdivided regardless of occupancy by objects, a SEADS subdivision generates many more voxels than the octree subdivision described earlier. It thus involves 'unnecessary' demands for storage space. However, the use of a SEADS enables very fast tracking of rays from region to region. The tracking algorithm used is an extension of the DDA (Digital Differential Analyzer) algorithm used in two-dimensional graphics for selecting the sequence of pixels that represent a straight line between two given end points. The DDA algorithm used in two-dimensional graphics selects a subset of the pixels passed through by a line, but the algorithm can easily be modified to find all the pixels touching the line. Fujimoto *et al.* (1986) describe how this algorithm can be extended into three-dimensional space and used to track a ray through a SEADS three-dimensional grid. The advantage of the '3D-DDA' is that it does not involve floating point multiplication and division. The only operations involved are addition, subtraction and comparison, the main operation being integer addition on voxel coordinates.

The heavy space overheads of the complete SEADS structure can be avoided by returning to an octree representation of the space subdivision. The 3D-DDA algorithm can be modified so that a ray is tracked through the voxels by traversing the octree. In the octree, a set of eight nodes with a common parent node represents a block of eight adjacent cubic regions forming a $2 \times 2 \times 2$ grid. When a ray is tracked from one region to another within this set, the 3D-DDA algorithm can be used without alteration. If a ray enters a region that is not represented by a terminal node in the tree, but is further subdivided, then the subregion that is entered is found by moving down the tree. The child node required at each level of descent can be discovered by adjusting the control variables of the DDA from the level above. If the 3D-DDA algorithm tracks a ray out of the $2 \times 2 \times 2$ region currently being traversed, then the octree must be traversed upwards to the parent node representing the complete region. The 3D-DDA algorithm then continues at this level, tracking the ray within the set of eight regions containing the parent region. The upward and downward traversals of the tree involve multiplication and division of the DDA control variables by 2, but this is a cheap operation.

Finally, we summarize and compare the three spatial coherence methods by listing their most important efficiency attributes:

- Octrees: are good for scenes whose occupancy density varies widely – regions of low density will be sparsely subdivided, high density regions will be finely subdivided. However, it is possible to have small objects in large regions. Stepping from region to region is slower than with the other two methods because the trees tend to be unbalanced.

- SEADS: stepping is faster than an octree but massive memory costs are incurred by the secondary data structure.
- BSP: the depth of the tree is smaller than an octree for most scenes because the tree is balanced. Octree branches can be short, or very long for regions of high spatial occupancy. The memory costs are generally lower than those of an octree. Void areas will tend to be smaller.

12.5.5 Ray space subdivision

In this unique scheme, suggested by Arvo and Kirk (1987), instead of subdividing object space according to occupancy, ray space is subdivided into five-dimensional hypercubic regions. Each hypercube in five-dimensional space is associated with a candidate list of objects for intersection. That stage in object space subdivision schemes where three-space calculations have to be invoked to track a ray through object space is now eliminated. The hypercube that contains the ray is found and this yields a complete list of all the objects that can intersect the ray. The cost of the intersection testing is now traded against higher scene pre-processing complexity.

A ray can be considered as a single point in five-dimensional space. It is a line with a three-dimensional origin together with a direction that can be specified by two angles in a unit sphere. Instead of using a sphere to categorize direction, Arvo and Kirk (1987) use a 'direction cube'. (This is exactly the same tool as the light buffer used by Haines and Greenberg (1986) – see Section 12.1.3.) A ray is thus specified by the 5-tuple (x, y, z, u, v), where x, y, z is the origin of the ray and u, v the direction coordinates; together with a cube face label that indicates which face of the direction cube the ray passes through. Six copies of a five-dimensional hypercube (one for each direction cube face) thus specify a collection of rays having similar origins and similar directions.

This space is subdivided according to object occupancy and candidate lists are constructed for the subdivided regions. A 'hyper-octree' – a five-dimensional analog of an octree – is used for the subdivision.

To construct candidate lists as five-dimensional space is subdivided, the three-dimensional equivalent of the hypercube must be used in three-space. This is a 'beam' or an unbounded three-dimensional volume that can be considered the union of the volume of ray origins and the direction pyramid formed by a ray origin and its associated direction cell (Figure 12.10). Note that the beams in three-space will everywhere intersect each other, whereas their hypercube equivalents in five-space do not intersect. This is the crux of the method – the five-space can be subdivided and that subdivision can be acheived using binary partitioning. However, the construction of the candidate lists is now more difficult than with object space subdivision schemes. The beams must be intersected with the bounding volumes of objects. Arvo and Kirk (1987) report that detecting polyhedral intersections is too costly and suggest the approximation where beams are represented or bounded by cones interacting with spheres as object bounding volumes.

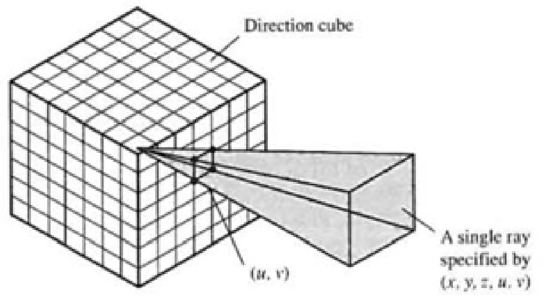


Figure 12.10
A ray (or beam) as a single point in (x, y, z, u, v) space.

12.6 The use of ray coherence

Up to now we have considered a ray to be infinitesimally thin and looked at efficiency measures that attempt to speed up the basic algorithm. It is easy to see that a major source of inefficiency that we have not touched on until now is the lack of use of ray coherence. This simply means that if the ray tracing algorithm generates a ray for each pixel and separately traces every such ray we are taking no account whatever of the fact that adjacent initial rays will tend to follow the same path. We will now look at ways in which we can 'broaden' a ray into a geometric entity.

Heckbert and Hanrahan (1984) exploit the coherence that is available from the observation that, for any scene, a particular ray has many neighbours each of which tends to follow the same path. Rather than tracing single rays, then, why not trace groups of parallel rays, sharing the intersection calculations over a bundle of rays? This is accomplished by recursively applying a version of the Weiler–Atherton hidden surface removal algorithm (Weiler and Atherton 1977). The Weiler–Atherton algorithm is a projection space subdivision algorithm involving a preliminary depth sort of polygons followed by a sort of the fragments generated by clipping the sorted polygons against each other. Finally, recursive subdivision is used to sort out any remaining ambiguities. This approach restricts the objects to be polygonal, thus destroying one of the important advantages of a ray tracer which is that different object definitions are easily incorporated due to the separation of the intersection test from the ray tracer.

The initial beam is the viewing frustum. This beam or bundle of rays is traced through the environment and is used to build an intersection tree, different from a single ray tree in that a beam may intersect many surfaces rather than one. Each node in the tree now contains a list of surfaces intersected by the beam.

The procedure is carried out in a transformed coordinate system called the beam coordinate system. Initially this is the view or eye coordinate system. Beams are volumes swept out as a two-dimensional polygon in the xy plane is translated along the z axis.

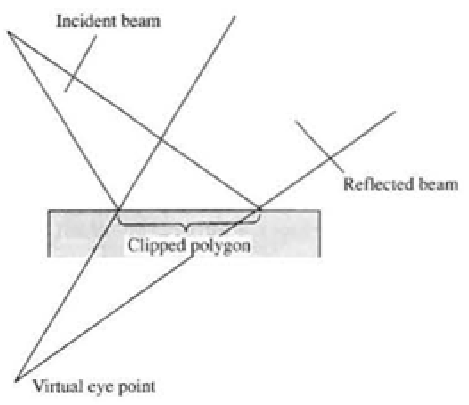


Figure 12.11
Reflection in beam tracing.

Reflection (and refraction) are modelled by calling the beam tracer recursively. A new beam is generated for each beam–object intersection. The cross-section of any refelected beam is defined by the area of the polygon clipped by the incident beam and a virtual eye point (Figure 12.11).

Apart from the restriction to polygonal objects the approach has other disadvantages. Beams that partially intersect objects change into beams with complex cross-sections. A cross-section can become disconnected or may contain a hole (Figure 12.12). Another disadvantage is that refraction is a non-linear phenomenon and the geometry of a refracted beam will not be preserved. Refraction therefore, has to be approximated using a linear transformation.

Another approach to beam tracing is the pencil technique of Shinya *et al.* (1987). In this method a pencil is formed from rays called 'paraxial rays'. These are rays that are near to a reference ray called an axial ray. A paraxial ray is represented by a four-dimensional vector in a coordinate system associated with the axial ray. Paraxial approximation theory, well known in optical design and electromagnetic analysis, is then used to trace the paraxial rays through the environment. This means that for any rays that are near the axial ray, the pencil transformations are linear and are 4 × 4 matrices. Error analysis in paraxial theory supplies functions that estimate errors and provide a constraint for the spread angle of the pencil.

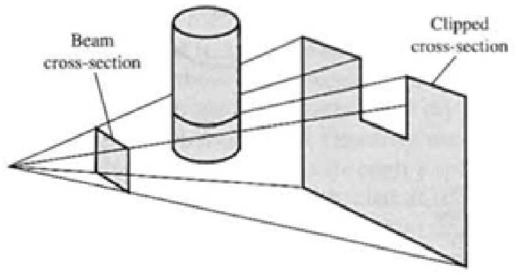


Figure 12.12
A beam that partially intersects an object produces a fragmented cross-section.

The 4 × 4 system matrices are determined by tracing the axial ray. All the paraxial rays in the pencil can then be traced using these matrices. The paraxial approximation theory depends on surfaces being smooth so that a paraxial ray does not suddenly diverge because a surface discontinuity has been encountered. This is the main disadvantage of the method.

An approach to ray coherence that exploits the similarity between the intersection trees generated by successive rays is suggested by Speer *et al.* (1986). This is a direct approach to beam tracing and its advantage is that it exploits ray coherence without introducing a new geometrical entity to replace the ray. The idea here is to try to use the path (or intersection tree) generated by the previous ray, to construct the tree for the current ray (Figure 12.13). As the construction of the current tree proceeds, information from the corresponding branch of the previous tree can be used to predict the next object hit by the current ray. This means that any 'new' intervening object must be detected as shown in Figure 12.14. To deal with this, cylindrical safety zones are constructed around each ray in a ray set. A safety zone for ray$_{r-2}$ is shown in Figure 12.15. Now if the current ray does not pierce the cylinder of the corresponding previous ray, and this ray intersects the same object, then it cannot intersect any new intervening objects. If a ray does not pierce a cylinder, then new intersection tests are required as in standard ray tracing, and a new tree that is different from the previous tree, is constructed.

In fact, Speer *et al.* (1986) report that this method suffers from the usual computational cost paradox – the increase in complexity necessary to exploit the ray coherence properties costs more than the standard ray tracing as a function of scene complexity. This is despite the fact that two-thirds of the rays behave coherently. The reasons given for this are the cost of maintaining and pierce-checking the safety cylinders, whose average radius and length decrease as a function of scene complexity.

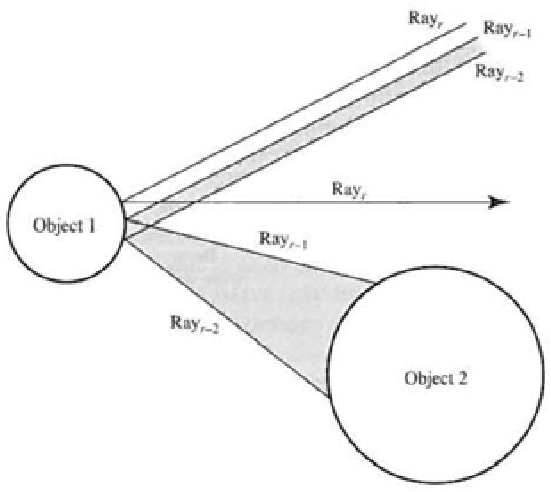


Figure 12.13
Ray coherence: the path of the previous ray can be used to predict the intersections of the current ray.

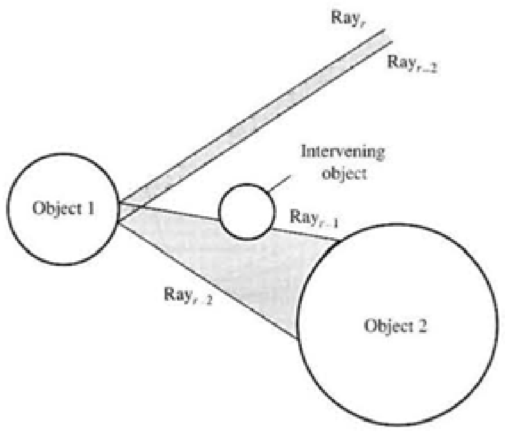


Figure 12.14
'Intervening' object in the path of the ray *r*.

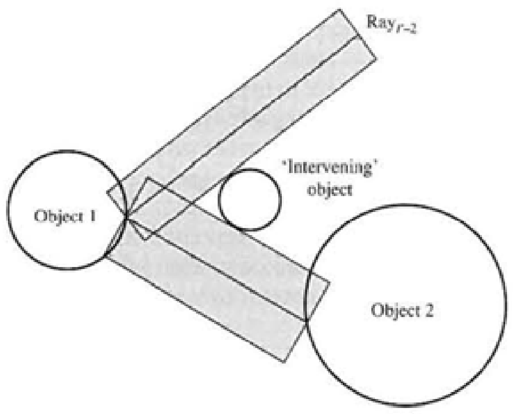


Figure 12.15
Cylindrical safety zones.

12.7 A historical digression – the optics of the rainbow

Many people associate the term 'ray tracing' with a novel technique but, in fact, it has always been part of geometric optics. For example, an early use of ray tracing in geometric optics is found in René Descartes' treatise, published in 1637, explaining the shape of the rainbow. From experimental observations involving a spherical glass flask filled with water, Descartes used ray tracing as a theoretical framework to explain the phenomenon. Descartes used the already known laws of reflection and refraction to trace rays through a spherical drop of water.

Rays entering a spherical water drop are refracted at the first air–water interface, internally reflected at the water–air interface and finally refracted as they emerge from the drop. As shown in Figure 12.16, horizontal rays entering the

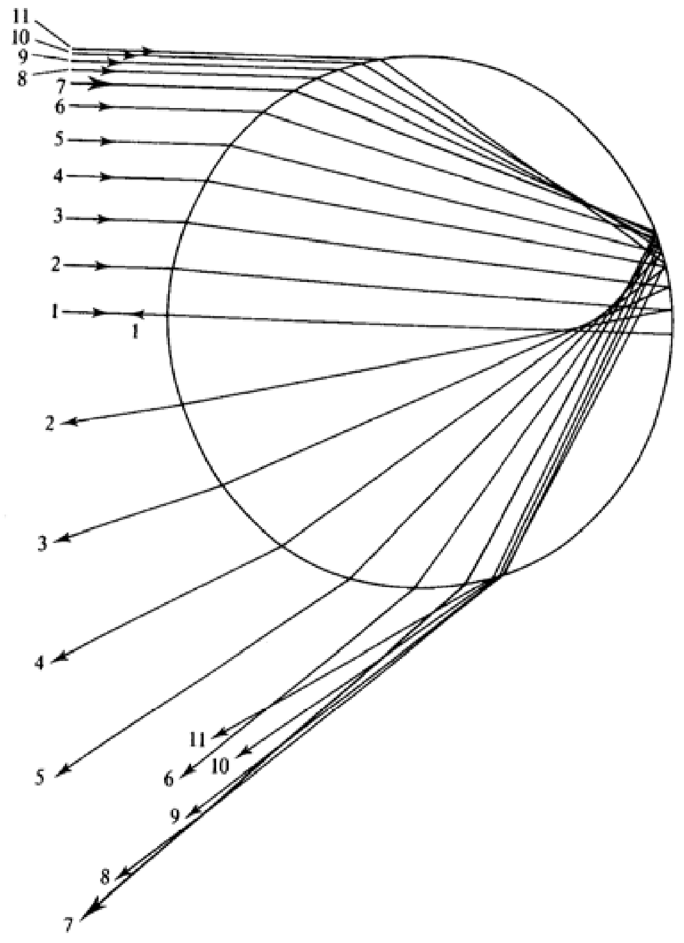


Figure 12.16
Tracing rays through a spherical water drop (ray 7 is the Descartes ray).

drop above the horizontal diameter emerge at an increasing angle with respect to the incident ray. Up to a certain maximum the angle of the exit ray is a function of the height of the incident ray above the horizontal diameter. This trend continues up to a certain ray, when the behaviour reverses and the angle between the incident and exit ray decreases. This ray is known as the Descartes ray, and at this point the angle between the incident and exit ray is 42°. Incident rays close to the Descartes ray emerge close to it and Figure 12.16 shows a concentration of rays around the exiting Descartes ray. It is this concentration of rays that makes the rainbow visible.

Figure 12.17 demonstrates the formation of the rainbow. An observer looking away from the sun sees a rainbow formed by '42°' rays from the sun. The paths of such rays form a 42° 'hemicone' centred at the observer's eye. (An interesting consequence of this model is that each observer has his own personal rainbow.)

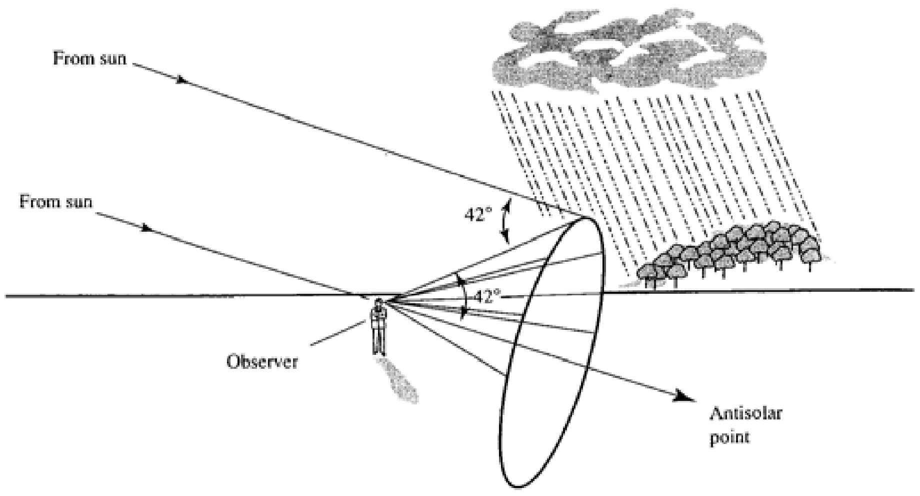


Figure 12.17
Formation of a rainbow.

This early, elegant use of ray tracing did not, however, explain that magical attribute of the rainbow – colour. Thirty years would elapse before Newton discovered that white light contained light at all wavelengths. Along with the fact that the refractive index of any material varies for light of different wavelengths, Descartes' original model is easily extended. About 42° is the maximum angle for red light, while violet rays emerge after being reflected and refracted through 40°. The model can then be seen as a set of concentric hemicones, one for each wavelength, centred on the observer's eye.

This simple model is also used to account for the fainter secondary rainbow. This occurs at 51° and is due to two internal reflections inside the water drops.

13

Volume rendering

Introduction

Volume rendering means rendering or visualizing voxel-based data. In Chapter 2 we introduced data representation techniques that are based on labelling all voxels in a region of object space with object occupancy. We saw that in applications where large homogeneous objects may occupy hundreds or thousands of voxels we may impose a hierarchical structure, such as an octree, on the data. On the other hand, in applications like medical imaging a data structure may be a vast three-dimensional array of voxels that emerges from a scanner. In this chapter we consider visualizing such large unstructured sets of voxels.

In the last decade or so a new discipline, ViSC, or Visualization in Scientific Computing, has emerged. One of the major application areas in this field is the visualization of scalar functions of three-spatial variables. Such data, prior to the availability of hardware and software for volume rendering, was visualized using 'traditional' techniques such as iso-contours in cross-sectional planes. The advent of volume rendering has meant that the data can be considered as a computer graphics object and all three dimensions displayed. Scalar functions of three-spatial variables abound in science and engineering. Engineers are concerned with designing three-dimensional objects and analyzing their potential behaviour. Calculations may produce predictions relating to temperature and stress, for example.

A voxel volume is produced either by a mathematical model, such as in computational fluid dynamics, or the voxels are collected from the real world as in medical imaging. Visualization software generally treats both types in the same way. The major practical distinction between different data sources is the shape of the volume element. In medical imagery the voxels are rectangular or cubic. In other applications this may not be the case. In the example shown in Figure 13.1 (Colour Plate) the volume elements were wedge shaped, that is, a cylinder divided up in a 'slice of cake' manner.

Medical imaging has turned out to be one of the most common applications of volume rendering. It has enabled data, collected from a tomographic system as a set of parallel planes, to be viewed as a three-dimensional computer graphics object. The material in this chapter is mostly based on this particular application. Although certain context-dependent considerations are necessary, the medical imaging problem is quite general and any strategy developed for this will easily adapt to other applications.

In medical imaging three-dimension data are available from stacks of parallel CT (computed tomography) data. These systems reconstruct or collect data in sets of planes according to some particular property, the original modality being the X-ray absorption coefficient at each point in the plane. The basic medical system enables a clinician to view the information in each plane. With visualization the entire stack of planes is considered as volume data and rendered accordingly. A very simplified illustration of a tomographic imaging system is shown in Figure 13.2. From this we should note that information is sampled in many two-dimensional planes of zero thickness. Voxel values are inferred from these data. The data exhibit the characteristic that the resolution within a plane (typically 512 × 512) is much greater than the resolution between planes. Scans are typically taken at distances of the order of 0.5 cm. These data are then interpreted as a set of voxels where each voxel exhibits an X-ray absorption coefficient and it is this data set that is volume rendered. Currently the systems that reconstruct the tomograms, which are used routinely for diagnosis, and the systems for volume visualization are separate – a point we return to in the next section.

One of the most remarkable projects in medicine is the Visible Human Project (1998). This is a 15 gigabytes voxel data set (male) and 40 gigabytes (female) consisting of MRI, X-ray CT and anatomical images obtained from cadavers. The initial aim of the Visible Human Project was to acquire transverse CT, MRI and cryosection images of a representative male and female cadaver at an average of 1 mm intervals. The corresponding transverse sections in each of the three modalities were to be registered with one another.

The Visible Human Male data set consists of axial MRI images of the head and neck and longitudinal sections of the rest of the body were obtained at 4 mm intervals. The MRI images are 256 × 256 pixels × 12 bits resolution. The CT data consist of axial CT scans of the entire body taken at 1 mm intervals at a resolution of 512 × 512 pixels × 12 bits. The axial anatomical images are 2048 × 1216 pixels × 24 bits. These are also at 1 mm intervals and coincide with the CT axial images. There are 1871 cross-sections for each mode, CT and anatomical.

The Visible Human Female data set has the same characteristics as the male cadaver with one exception. The axial anatomical images were obtained at 0.33

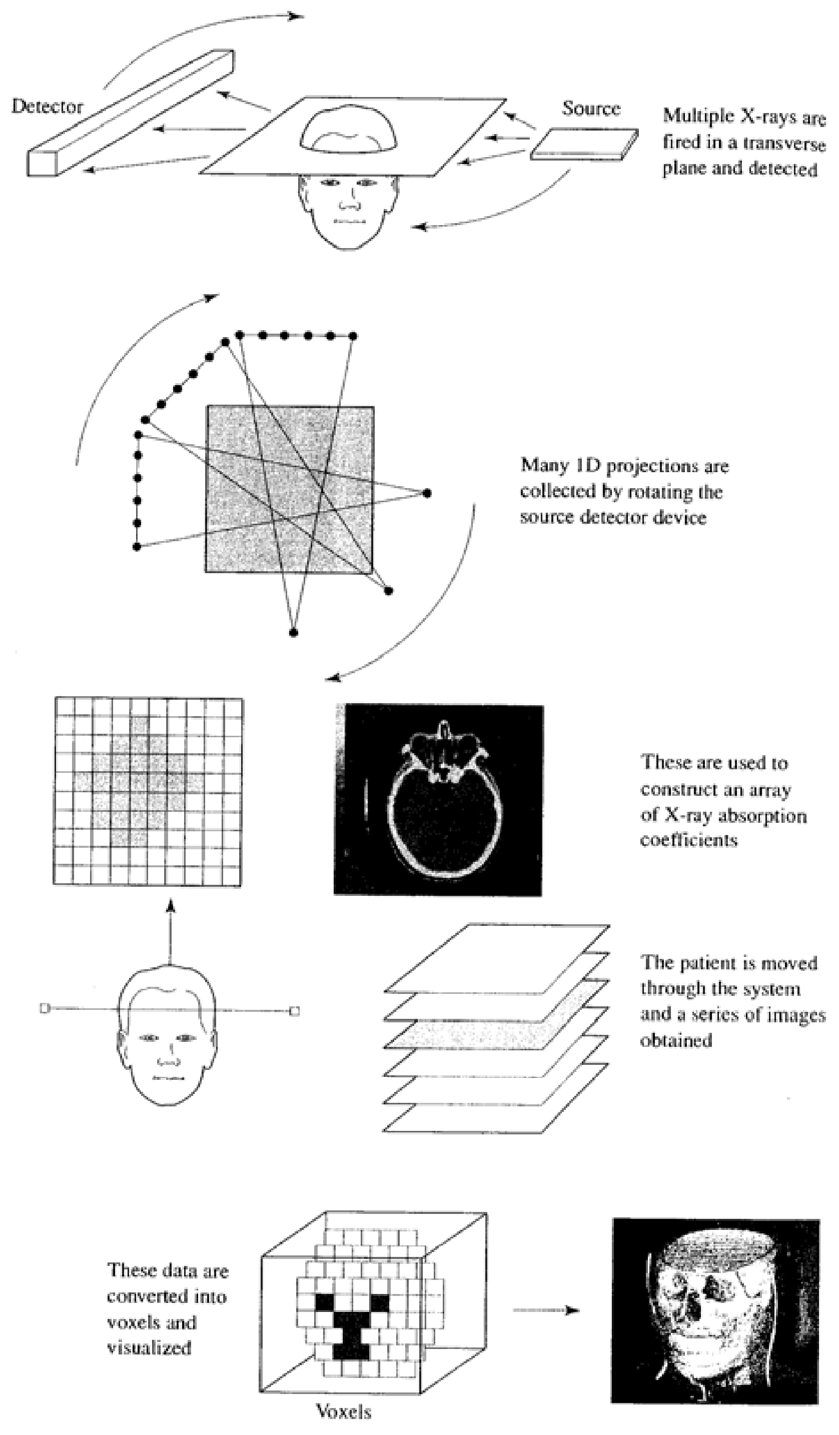

Detector
Source
Multiple X-rays are fired in a transverse plane and detected
Many 1D projections are collected by rotating the source detector device
These are used to construct an array of X-ray absorption coefficients
The patient is moved through the system and a series of images obtained
These data are converted into voxels and visualized
Voxels

Figure 13.2
X-ray computer tomography and volume rendering.

mm intervals instead of 1 mm intervals to enable cubic voxels. This resulted in over 5000 anatomical images.

The long-term goal of the project is stated as:

The Visible Human Project data sets are designed to serve as a common reference point for the study of human anatomy, as a set of common public domain data for testing medical imaging algorithms, and as a test bed and model for the construction of image libraries that can be accessed through networks. The data sets are being applied to a wide range of educational, diagnostic, treatment planning, virtual reality, artistic, mathematical and industrial uses by over 1000 licensees in 41 countries. But key issues remain in the development of methods to link such image data to text-based data. Standards do not currently exist for such linkages. Basic research is needed in the description and representation of image-based structures, and to connect image-based structural–anatomical data to text-based functional–physiological data. This is the larger, long-term goal of the Visible Human Project: to transparently link the print library of functional–physiological knowledge with the image library of structural–anatomical knowledge into one unified resource of health information.

13.1 Volume rendering and the visualization of volume data

The basic idea of volume rendering is that a viewer should be able to perceive the data volume from a rendered projection on the view plane. In medical imaging we may want to view a surface, or the volume, or just part of the volume.

Thus we view the extraction and display of 'hard' surfaces that exist in the data as part of the volume rendering problem. In many cases we may have a volume data set from which we have to extract and display surfaces that exist anywhere within the volume. Rather than bounding surfaces of an object, we may be dealing with an object that possesses many 'nested' surfaces – like the skin of an onion. If such surfaces are extractable by some unique property then we can render them visible by making them 100% opaque and all other data in the volume 100% transparent.

We will now extend the medical example and consider techniques for visualizing the stack of CT slices as a three-dimensional volume of data. All of the techniques described in this chapter apply to most volume data. They are more or less completely general. It is simply easier to consider the different possibilities in the context of a particular application area.

As we mentioned previously, our reconstructed CT data consist of a number of infinitely thin slices or two-dimensional arrays, where the inter-slice distance is, in practice, greater than a pixel dimension within the slice. To turn this stack into a regular three-dimensional array of cubic voxels we have to invoke some form of interpolation. We can then consider the various possibilities, or modes of displaying this volume data.

For any application, because we are dealing with volume data, the options available are much greater than with rendering the surface of an object; and the particular mode of display will depend on the applications. The nature of these requirements and also the nature of the data determines the algorithm that is used.

The three basic options available for displaying a volume data set on a two-dimensional display surface are:

(1) To slice the data set with a cross-sectional plane. This is clearly the easiest option and is trivial if the plane is parallel to one of the coordinate planes of the volume data set. It is also the least ambitious as it effectively displays only two dimensions of the data.

(2) To extract an object that is 'known' to exist within the data set and render it in the normal way. Thus an internal organ of a body can be displayed in isolation just as if it had been dissected. This implies, first, that the object can be segmented from the remainder of the data and, second, that the segmented form can be converted into a computer graphics representation.

(3) To assign transparency and colour to voxels within the object then view the entire set from any angle. This is usually known as volume rendering. Alternatively in medical applications it is sometimes called a computed X-ray as it is analogous to a conventional X-ray. In other words it is possible to generate a computed X-ray from any viewing angle, including angles that may be physically impossible with conventional X-ray equipment. As well as having freedom to select any viewing angle we can also change the opacities in any way we require.

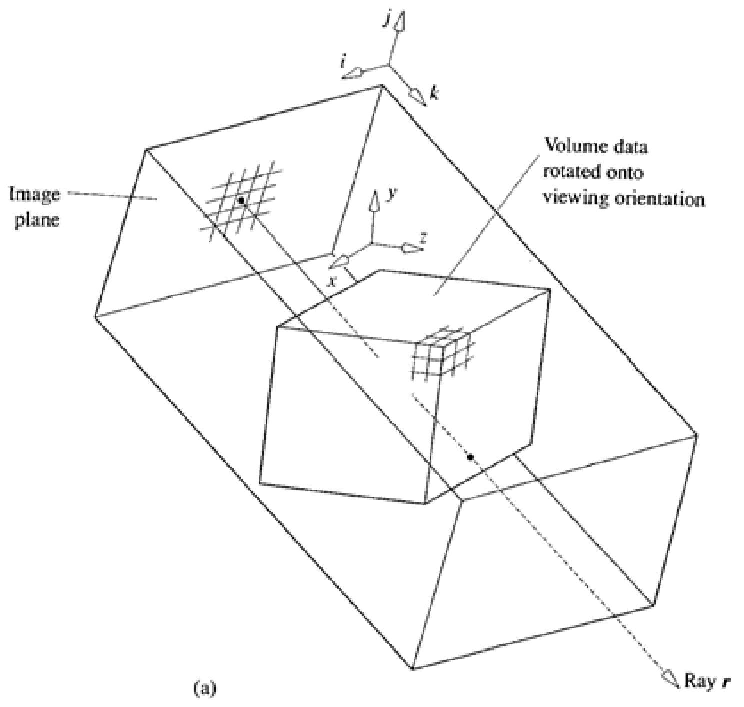
Currently, the main application of the visualization of volume data is in medical illustration – in the form of interactive atlases for medical education, medical research, surgical planning and computer graphics research. For diagnostic applications clinicians appear to prefer examining the original tomographic slices side by side. This is due in part to inadequacies in the process that, as we shall see, involve interpolative methods which may interfere with the integrity of the original data. For example, in the second method – extracting an object from the data – small holes in a surface may be filled in.

The connection with medical illustration is reinforced by the fact that many quality issues can be resolved in the original data collection stage – the scanning of the body – but there are usually limits associated with this. In the case of X-ray CT scanning, the X-ray dosage received by the patient is a function of the resolution, both in terms of the spatial resolution within the reconstruction plane and the number of planes collected. To increase the resolution means subjecting the patient to a higher X-ray dosage which is already higher than that for a conventional X-ray. This has meant that the high quality imagery has been generated from data obtained from cadavers.

The development of medical atlases from volume data sets has led to a variety of creative combinations of the above three display options. Examples of common combinations are shown in Figures 13.3 (Colour Plate). The first two examples show an extracted object(s) embedded in a transparent surround of the skull. The extracted structures have been turned into computer graphics objects and rendered normally. They are then effectively re-embedded in the three-dimensional data volume which is displayed with the surrounding voxels set to some semi-transparent value. The semi-transparent voxels can be set to grey scale – to simulate an X-ray – or any desired colour. This can be effected simply by using method 3 and setting the object voxels to be opaque, but a better result is normally obtained, at least for the purposes of medical illustration, by rendering the objects of interest conventionally. The motivation of this type of illustration is obvious – it highlights the object and orients it with respect to the body or skin. Another popular combination (the second two examples) is to cut away a rendered version of the skin to show internal organs as a cross-section positioned within a three-dimensional model. Here, the organs are assigned an appropriate pseudo-colour simply to highlight their shape. Such colours can be 'pure' false colours that identify or label the structure of interest and they can relate to the values associated with the voxels on some understood basis. A standard hue circle set of colours could reflect the value of the absorption coefficient, for example.

The overall idea of volume rendering is shown in Figure 13.4 as a ray casting algorithm which shows a volume data set, represented as a cube, rotated into a desired viewing orientation and intersected by a bundle of parallel rays – one for each pixel. (The term 'ray casting' is used to distinguish the method from ray tracing – in this context the rays continue as a parallel bundle through the volume instead of diverging after a hit.) Such an approach is a useful conceptual starting point; in practice, there are many different ways of implementing this approach. We will now discuss the following general options and considerations with respect to such an algorithm:

- **What properties of the data do we want to see in the image plane?** We may want to see the external boundary surface as a shaded object. In medical imagery this would be the skin surface and this implies that we have to 'find' this surface and shade it. In the ray casting case this would simply involve terminating the ray when it strikes the first non-zero voxel, evaluating a surface normal for the voxel and applying a local shading model. Alternatively we may want to visualize an internal object and shade it. In medical imagery we might want to see bone structure underneath the skin/flesh layer. This implies that we have to extract such a surface for the data set before we can render it as a computer graphics object. We may want to move a cross-sectional cutting plane through the data as shown in Figure 13.4(b) and view the contents of the intersection of the cutting plane with the data as it moves. We may want to see both bones, such as the rib cage, and the organs contained within. This could be accomplished either by rendering the bones as opaque so that the viewer sees the organs through the gaps in the bones, or by rendering the bones as partially transparent. Other possibilities are easily imagined. We could compose a projection that, for each pixel, was the maximum data value encountered along the ray. A less obvious mode is to display the sum along each ray path. This will then give an image analogous to a conventional X-ray, giving us the facility of being able to generate a (virtual) X-ray-type



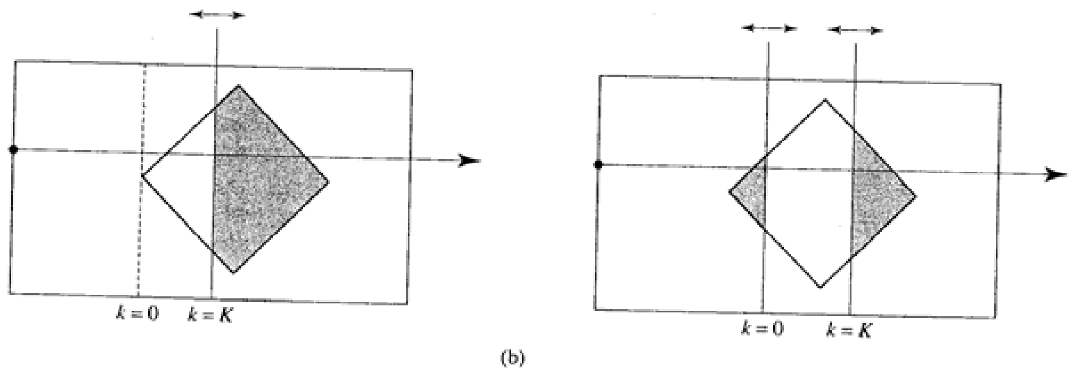


Figure 13.4
(a) Volume rendering by casting parallel rays from each pixel (after Levoy (1990)). (b) Using planes parallel to the view plane to construct a view volume of the data set.

image from viewing angles that would be impossible with conventional equipment.

- **What is the relationship between the reality and the data?** Our volume data set will consist, in general, of a three-dimensional array of points, representing a three-dimensional sampling of the reality. This may be a very large data set, say 512^3. We associate the single sample with the entire voxel volume, just as a sample in two-dimensional image processing is associated with a square pixel extent. But what does that single sample represent? Here, the simpler case to consider is binary occupancy. We assume that the voxel resolution is fine enough that any voxel contains only a single material, or it contains nothing. Alternatively, we could consider that a voxel contained a mix of materials. In medical imagery it may be that the physical extent of a voxel corresponds to a region which straddles both bone and tissue. Do we consider the value of a voxel to be constant throughout its extent, or do we consider that the value varies throughout? If the latter, what model do we use to interpolate the variation between neighbouring voxels?

- **What are the implications of voxel size?** Unlike conventional surface rendering, where we have a definition associated with an object for each pixel, it is likely that the projection of a voxel extent onto the image plane will occupy many pixels.

We will now look at these considerations in greater detail.

13.2 'Semi-transparent gel' option

The most general viewing option is somehow to give a viewer the facility to see all the data. No voxel is considered completely opaque and all the data are therefore seen. The physical analogue is an object that is made of different coloured transparent gels. All other options can be considered a particularization of this method. Each voxel is assigned a colour, $\mathbf{C}$, and a transparency, α. The colour associated with the material type can be chosen 'aesthetically'. In the CT example, white could be chosen for bone and the transparency would be made proportional to density so that bone could be made almost completely opaque.

We then cast a ray from each pixel into the data volume which has been rotated into the desired viewing orientation and perform a compositing operation. This accumulates a resultant colour and opacity for that pixel. The process is like considering the volume to be made up of a semi-transparent gel of different colours and opacities. It is as if behind the volume we had diffuse white light and we are looking into it from the front side. The process is analogous to taking a conventional X-ray of the volume in the viewing direction; but now we are transmitting parallel beams of light through a volume whose opacity relates to tissue density and displaying the result.

In clinical application at the Johns Hopkins Medical Institution, Ney *et al.* (1990) state:

The images generated using this unshaded rendering process are reminiscent of a conventional radiographic image. These images are particularly useful for examining bony abnormalities. The bones are semi-transparent and therefore internal detail is visible, as well as surface detail. Unfortunately the unshaded technique does not work well for imaging soft tissue. The high variability of bone density causes the unshaded algorithm to produce the perceived detail. Soft tissue attenuation values are confined to a far narrower spectrum, making it more difficult to separate, for example, a vessel or node from adjacent muscle.

Thus we see that this visualization involves a number of steps:

(1) Classify each voxel in the original data and assign desired colour and opacity values.
(2) Transform the (now classified) volume data into the viewing direction.
(3) For each pixel cast a ray and find, by compositing along the ray, a colour for that pixel.

We now describe each of these steps separately.

13.2.1 Voxel classification

Considering the more general case of a voxel containing more than one tissue type, a typical classification scheme was introduced by Drebin *et al.* (1988) (for the particular case of X-ray CT data). In this scheme voxels are classified into four types according to the value of the X-ray absorption coefficient. The types are: air, fat, soft tissue and bone. The method is termed 'probabilistic classification' and it assumes that two, but not more than two, materials can exist in a voxel. Thus voxels can consist of seven types: air, air and fat, fat, fat and soft tissue, soft tissue, soft tissue and bone, and bone. Mixtures are only possible between neighbouring materials in the absorption coefficient scale – air, for example, is never adjacent to bone.

The classification scheme uses a piecewise linear 'probability' function (Figure 13.5). Consider a specific material assigned such a function. There will exist a particular CT number that is most likely to represent this material (point A in Figure 13.5(a)). Points B_1 and B_2 represent the maximum deviation in CT number from point A that is still considered this material. Any CT number less than B_1 or greater than B_2 and contained within the limits defined by C_1 and C_2 is classified as a mixture of 'neighbouring' materials. A complete scheme is shown in Figure 13.5(b). Voxels are assigned (R, G, B, α) values according to some

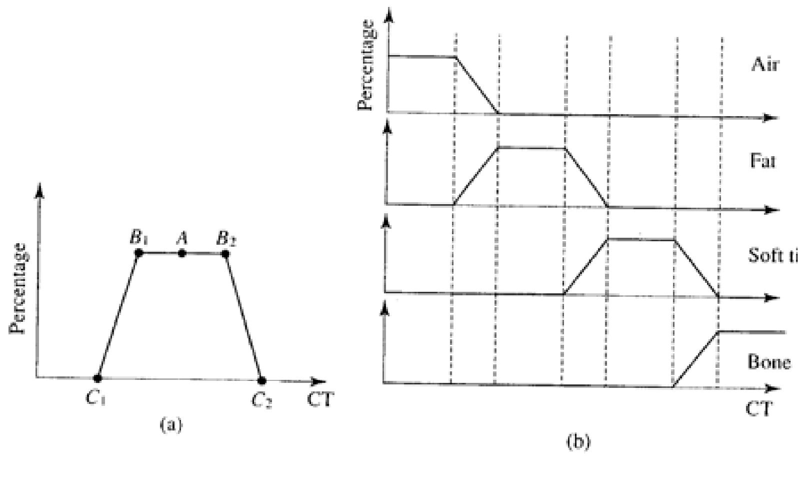


Figure 13.5
Material classification in CT data (due to Drebin *et al.* (1988)). (a) Trapezoid classification function for one material. (b) Classification functions.

scheme and if a mixture of two materials is present in a voxel the two colours are mixed in the same proportion as the materials.

13.2.2 Transforming into the viewing direction

Theoretically a simple process, this step produces difficulties. A simple illustration of the viewing process is shown in Figure 13.6. In general, the data volume can be rotated into any desired orientation and when pixel rays are cast into the rotated volume this involves a resampling operation and aliasing has to be considered. One of the main options in the overall construction of a volume rendering algorithm is the way in which this transformation is performed and its position in the order of the three stages described in Section 13.2.

In the CT example it is only useful to rotate about the *z* axis (spinal rotation) and about the *x* axis (somersault rotation). This means that the rotation of the volume can be performed by rotating two-dimensional planes perpendicular to these axes.

13.2.3 Compositing pixels along a ray

The simplest compositing operation (Figure 13.7) is the recursive application of the formula:

$$\boldsymbol{C}_{\text{out}} = \boldsymbol{C}_{\text{in}}\,(1 - \alpha) + \boldsymbol{C}\,\alpha$$

where:

$\boldsymbol{C}_{\text{out}}$ is the accumulated colour emerging from a voxel
$\boldsymbol{C}_{\text{in}}$ is the accumulated colour into that voxel
α is the opacity of the current voxel
$\boldsymbol{C}$ is the colour of the current voxel

Note that this form is just an extension of the **over** operation defined in Section 6.6.3 for compositing two images. The direction implied by $\boldsymbol{C}_{\text{out}}$ and $\boldsymbol{C}_{\text{in}}$ is from back to front with respect to the view plane. That is, we start the operation with the voxel furthest from the view plane.

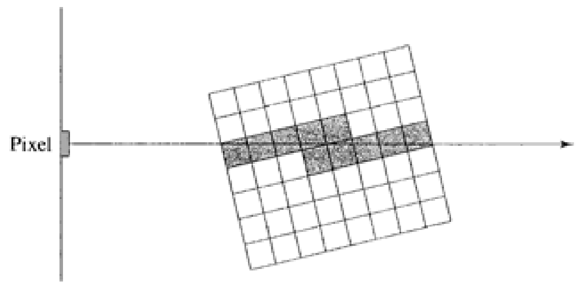


Figure 13.6
Ray casting implies resampling the data. A ray will not, in general, intercept voxel centres.

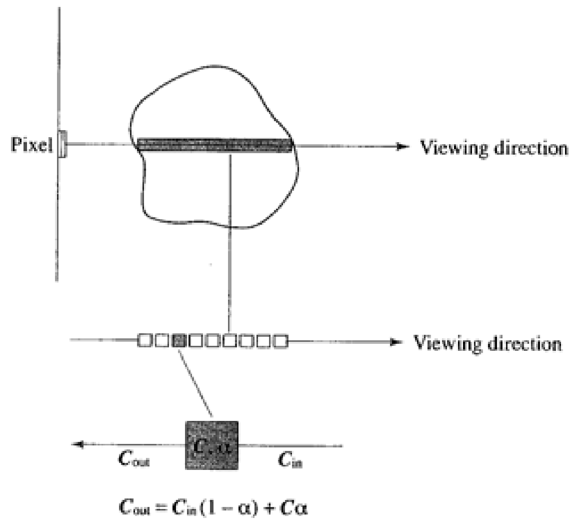


Figure 13.7
The ray compositing operation.

It does not matter in this model where the light comes from. We simply note that any light exiting from the voxel of interest along the viewing direction has the colour of that voxel plus the product of the incoming light and $(1 - \alpha)$. There are elaborations that can be made on this simple model. For example, α should in reality be a vector quantity since it will differ according to the R, G or B component of the colour of the voxel. The effect of this operation is to make voxels with high α values predominate, obscuring voxels that are behind them and being made visible through voxels in front of them.

13.3 Semi-transparent gel plus surfaces

If we assume that opaque surfaces are present in the data volume then we supplement the previous scheme with a shading scheme, and present the surfaces as part of the display according to the various options that we described in Section 13.1. Assuming that a voxel can contain part of a surface we can evaluate a normal, and a shading component is calculated as a function of this normal and the direction of the illuminating source. This shading component can then replace $\boldsymbol{C}$ in the compositing operation.

The shape of surfaces is now perceived in the normal way as the lighting model enhances the details in the surface. Various options now emerge. We can display just those voxels that contain, say, bone together with its surface shape detail, visible through a fuzzy cloud of soft tissue. Bone can then be made completely opaque or still be given an opacity so that detail behind the bone is still visible.

A surface is detected by evaluating a normal using the volume gradient. The components of this normal are:

$$\mathbf{N}_x = R(x+1, y, z) - R(x-1, y, z)$$
$$\mathbf{N}_y = R(x, y+1, z) - R(x, y-1, z)$$
$$\mathbf{N}_z = R(x, y, z+1) - R(x, y, z-1)$$

where for each voxel, R is evaluated by summing the products of the percentage of each material in the voxel times its assigned density. If a material is homogeneous these differences evaluate to zero and the voxel under consideration is deemed not to contain a surface segment. This scheme is illustrated diagrammatically in Figure 13.8.

The presence of a surface is quantified by the magnitude of the surface normal – the larger this magnitude the more likely it is that a surface exists. The magnitude or 'strength' of the surface ($|\mathbf{N}|$) can be used to weight the contribution of the shaded component. No binary decision is taken on the presence or absence of a surface. A normalized version of the surface normal is calculated and can then be used in a shading equation such as the Phong reflection model. We should bear in mind that this technique is purely for the purposes of visualization. It has absolutely no relation to physical reality. We assume that each

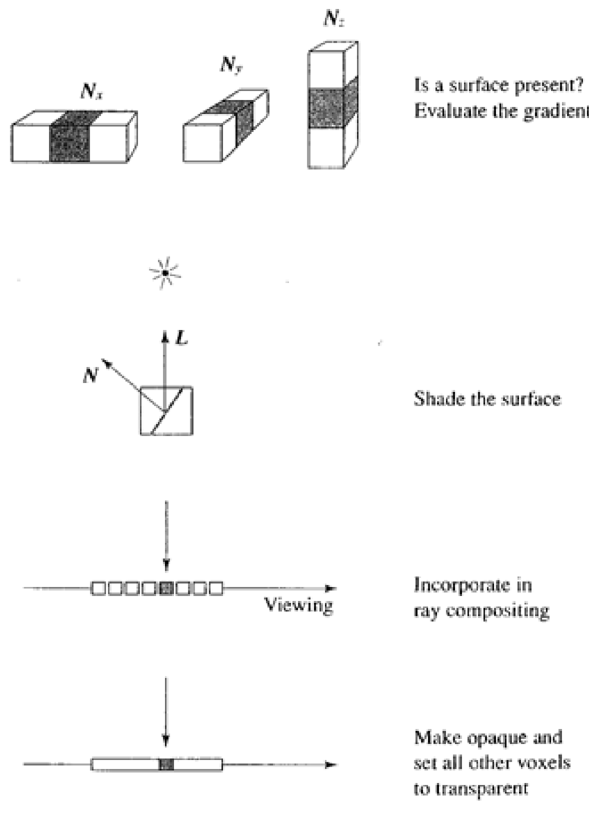


Figure 13.8
Surface detection and shading.

voxel has an uninterrupted view of the light source even though it may be buried in the middle of a volume.

The localness of this operation means that it is sensitive to noise. This can be diminished by reducing the localness. In the above formula the gradient is evaluated by considering six neighbouring voxels. We can extend this to 18 or even 24 voxels.

We have shaded surfaces by calculating the interaction of a normal of the voxel containing the surface with a light source. Then the surface shape detail becomes visible. We can either incorporate the shaded surface in the semi-transparent gel model or we can make the surface opaque and remove all voxels that do not contain a surface. This makes the first surface the ray hits the surface that is seen by the viewer. These options are indicated schematically by Figure 13.8.

It is important to realize that the surface detection is local and is evaluated for single voxels. No decision has to be taken about the existence or otherwise of a surface if the shading component is included in the semi-transparent gel model. This is important in medical applications where clinicians are (rightly) suspicious of methods where binary decisions on the existence of a surface are made. There are, however, applications where such an approach – explicit extraction of an (assumed) continuous surface – is desirable, as we describe in the next section.

13.3.1 Explicit extraction of isosurfaces

If the volume data is such that it is known to contain continuous isosurfaces, then these surfaces can be explicitly extracted and converted into polygon mesh structures and rendered in the normal way. Such an approach finds one or more appropriate polygons for each voxel and produces a continuous set of such polygons from the set of voxels comprising the surface.

So why go to the trouble of finding a polygon mesh surface when we can find and shade surfaces in the volume by using the density gradient? One of the motivations is that conventional rendering techniques can be used if the surface is represented with conventional graphics primitives and volume rendering then reduces to a preprocessing operation of surface extraction.

The technique used is known as the marching cubes algorithm reported by Lorenson and Cline (1987). An actual surface is built up by fitting a polygon or polygons through each voxel that is deemed to contain a surface. A voxel possesses eight vertices and if we assume at the outset that a voxel can sit astride a surface, then we can assign a polygon to the voxel in a way that depends on the configuration of the values at the vertices. By this is meant the distribution of those vertices that are inside and outside the surface over the eight vertices of the cube. If certain assumptions are made, then there happens to be 256 possibilities. From considerations of symmetry these cases can be reduced to 15 and these are shown in Figure 13.9. The final position and orientation of each polygon within each voxel type is determined by the strength of the field values at the vertices. A surface is built up that consists of a normal polygon mesh and the

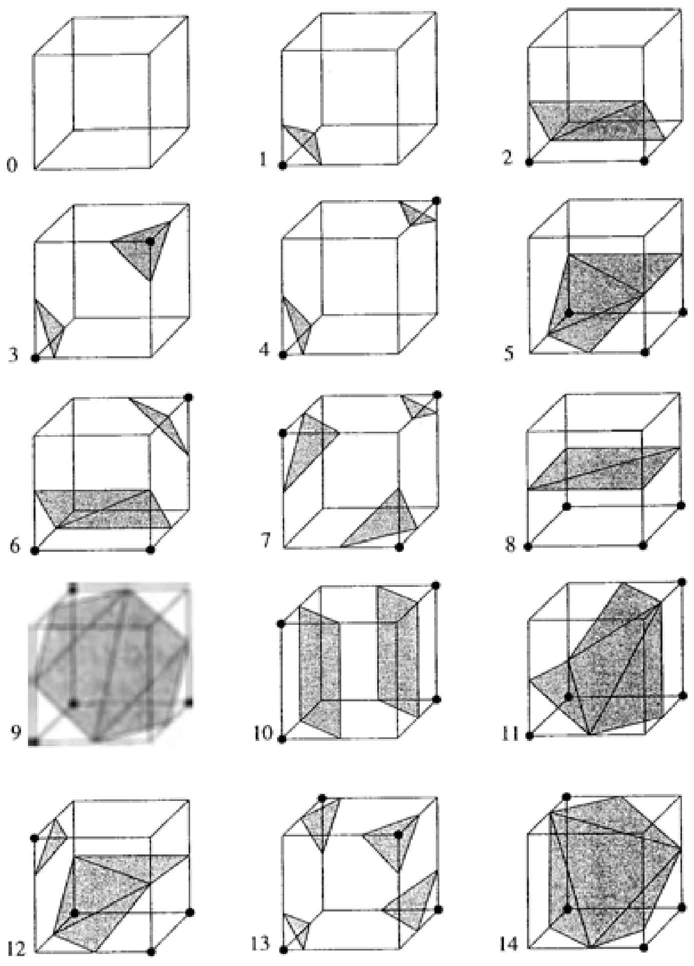


Figure 13.9
The 15 possibilities in the marching cubes algorithm. Dot (•) used in the figure represents a vertex that is inside a surface.

difference in quality between rendering such a surface and effecting surface extraction by appropriate zero–one opacity assignment in volume rendering is due to what is effectively an inferior resolution in the volume rendering method. In the volume rendering method a surface may exist somewhere within the voxel. The opacity of such a voxel is set to one and the information on the position and orientation of the surface fragment is reduced to the surface normal. In the marching cubes algorithm the surface fragment (or fragments) is positioned and oriented accurately within the voxel – at least within the limitations of the interpolation method used. However, explicit surface extraction methods sometimes make errors by making the assumption that a surface exists across neighbouring voxels. They can fit a surface over what in reality are neighbouring surface fragments. In other words, they make a binary decision that may be erroneous. Another problem with the marching cubes algorithm is the sheer volume of primitives that can be generated. This can run into millions where many primitives project onto the same pixel.

Figures 13.10 (Colour Plate) and 13.11 (Colour Plate) compare the two main approaches for rendering an object of interest. The original data are 23 planes of X-ray CT data with a 512 × 512 resolution in each plane. Figure 13.10 shows a skull rendered using the marching cubes algorithms. The second illustration (Figure 13.11) is exactly the same data but this time they are rendered using a volume rendering algorithm with the bone opacity set to unity. Although it may not be too apparent in the reproductions, the marching cube version appears to be of higher quality or resolution – this is an illusory consequence of the algorithm; it is accessing the same data but creating an explicit computer graphics model of one or more polygons per voxel. The volume rendering algorithm is simply assigning normals to each voxel based on local information.

13.4 Structural considerations in volume rendering algorithms

There are many options in setting up a volume rendering algorithm. As we have seen, the process of viewing a volume data is conceptually simple involving as it does the rotation of the volume into the viewing orientation, then ray casting (or an equivalent operation) into the volume to discover a suitable value for each pixel. The main research thrust in volume rendering arises out of the importance of efficient hardware implementation. Interactivity and animation are important in most application areas because of their contribution to the interpretation of the data. Because we are generally dealing with very large data sets – routinely in the order of 512^3 – the relationship between the algorithm design and available hardware (such as parallel processors) becomes of critical importance if interactivity/animation demands are to be met.

The terminology used to describe algorithmic options in volume rendering is somewhat confusing. The confusion seems to arise out of what names to give to the main categories. There are two main categories:

(1) Ray casting methods (with two variants). Also called image or pixel space traversal or back projection.

(2) Voxel projection methods (with two variants). Also called object or voxel space traversal or forward projection.

These options are illustrated diagrammatically in Figure 13.12. In ray casting we can either transform and resample the volume data so that it is oriented with a coordinate axis parallel to the image plane, or we can leave it untransformed. If the data are transformed prior to ray casting then we generate a set of rays parallel to rows (or columns) of the transformed data. For untransformed data the ray set is subject to the inverse viewing transform. Ray casting methods are also categorized as image space methods in that the outermost loop of the algorithm traverses image space.

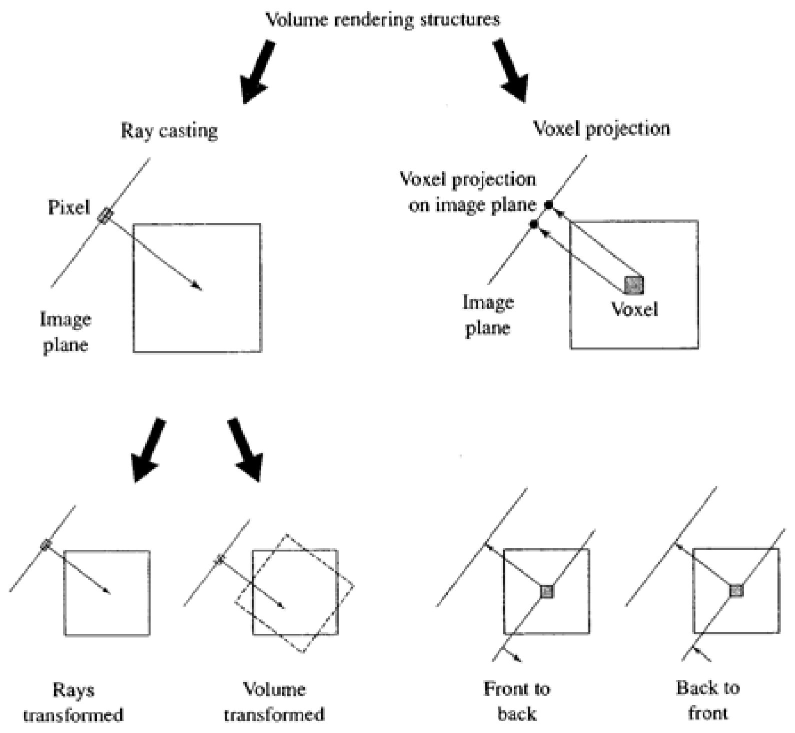


Figure 13.12
A taxonomy of volume rendering structures.

Although at first sight it would seem that ray casting methods can be implemented in parallel, memory bottleneck problems arise. If arbitrary viewing directions are allowed there is no way to distribute voxels in memory to ensure that no contentions occur.

A potential problem with forward projection is that holes may arise in the image plane. For voxel projection methods we have to bear in mind that in most applications a single voxel will form a projection in the image plane that spreads over many pixels. (This has been called a footprint.) If we ignore perspective projections then this footprint is the same for all voxels – for a given view – and such coherence can be used to advantage for fast implementation and efficient anti-aliasing. We will now consider these options in greater detail. The important difference between the methods are manifest in the suitability for parallel implementation and how resampling is accomplished.

13.4.1 Ray casting (untransformed data)

In ray casting we traverse image space and cast a ray from each pixel to find a single colour for that pixel by the compositing operation previously described. (The method bears little or no relationship to ray tracing which traces a pixel ray in any direction through the scene depending on the geometry and nature of the objects that are hit. In volume rendering we cast a set of parallel pixel rays which all remain travelling in the same direction.) To do this two non-trivial tasks have to be performed. First, we have to find these voxels through which the ray passes and, second, we have to find a value for each of the voxels from the classified data set.

Consider the first problem. This in itself breaks down into two parts. Finding the voxels through which a pixel ray passes is a well-worked-out problem – we simply use a 3DDA (three-dimensional differential analyzer) an extension into three-dimensional space of knowledge worked out over the years to deal with the two-dimensional line/pixel problem. However, once we find these voxels, how do we deal with their values? How do we obtain values to insert into our compositing scheme? Using the basic values of each encountered voxel is wrong. One reason is obvious. The path lengths through each voxel will vary from a very small distance, for a ray that just cuts the corner of a voxel, to a large distance for a ray that is close to the diagonal across opposite corners. We are effectively viewing along a ray and a long journey through a voxel should produce a higher contribution to the compositing than a short one. This is, of course, one of the consequences of sampling a practical volume data set with an infinitely thin ray – or more precisely resampling. It is a three-dimensional problem of the equivalent resampling process in image processing. We start with sampled data, rotate them into a new orientation, and resample them. We have to filter when we are resampling to avoid aliasing. The complication in volume rendering is that the data are three-dimensional and the resampling is in three-dimensional space. An appropriate way to proceed therefore, is to measure equal points along the ray and find a resampled value at these points by filtering over a three-dimensional region, using the equally spaced ray sample points as a centre for the three-dimensional filter kernel.

The algorithm is sometimes described as an image space traversal algorithm and the outermost loop is usually defined as 'cast a ray for each pixel'. However, we need to recognize that we can do no better than cast a parallel set of rays into the volume that pass through every voxel in the data. A simple scheme to achieve this is shown in cross-section in Figure 13.13. The ray set is constructed by passing each ray through the centre point of each voxel in the front face of the data set.

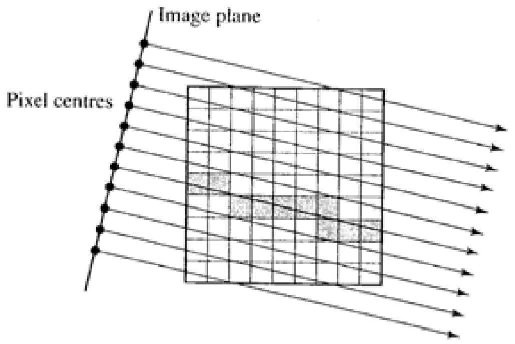


Figure 13.13
An appropriate set of rays in a ray casting algorithm.

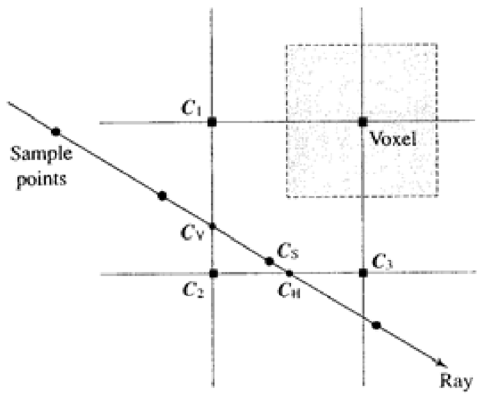


Figure 13.14
$\mathbf{C}_S$, the value at a sample point on the ray, is evaluated by bilinear interpolation. $\mathbf{C}_V$ is evaluated from $\mathbf{C}_1$ and $\mathbf{C}_2$. $\mathbf{C}_H$ is evaluated from $\mathbf{C}_2$ and $\mathbf{C}_3$. $\mathbf{C}_S$ is evaluated from $\mathbf{C}_V$ and $\mathbf{C}_H$.

The same concept is used by Yagel *et al.* (1992), who use the idea of a ray 'template'. The ray template method adopts the simple approach of moving the ray one voxel at a time along a line called the base plane. Thus the ray, or ray template, is computed once only and stored in a data structure. All rays are then followed by obtaining the appropriate displacements from this information. The shaded voxels in Figure 13.13 form a ray template. In effect, this approach is exploiting the coherency between rays.

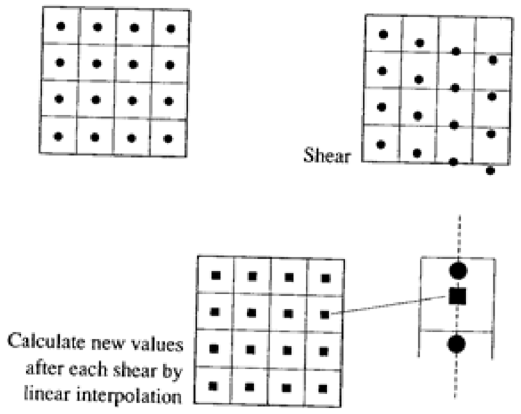
We now consider the question of resampling. If the volume is left undisturbed, then the rendering (or compositing) process and resampling process are merged into one operation. We step along the ray at equal sample points and evaluate, for each sample point, a $\mathbf{C}$ to be used in the compositing. We could simply use a value for $\mathbf{C}$ that was the value of the voxel that contained the sample point. But normally the more accurate process of trilinear interpolation is used. This is shown in cross-section in Figure 13.14 where it becomes in two dimensions bilinear interpolation. To evaluate $\mathbf{C}_S$ we interpolate from the surrounding grid points, evaluating first the horizontal and vertical intersects of the ray with the voxel grid lines. We can then find the value of $\mathbf{C}_S$. The process is a simplified version of bilinear interpolation used in polygon shading (see Chapter 1) where the polygon is a square.

13.4.2 Ray casting (transformed data)

The second variant of ray casting involves pre-transforming the data into the desired orientation. The geometry of the actual ray casting is then trivial (or eliminated) in that we simply composite along rows or columns of the transformed data.

To transform the data a three-pass (all shear) decomposition described in Wolberg (1990) can be used. A viewing transformation then becomes a sequence of pure shears – three for each axis. So a general transformation is a set of nine shears. The importance of a shear-only process lies in its implementation in

Figure 13.15
Resampling is performed during *each* shear.


Shear
Calculate new values after each shear by linear interpolation

special-purpose hardware. In particular, it possesses the property that every voxel in a single shear is moved through a constant amount.

The significant difference in the two ray casting variants is involved in the resampling. Now resampling must be performed during each shear and the process of resampling is performed *before* the compositing. Resampling during a shear involves simple linear interpolation (Figure 13.15).

In ray casting methods an important efficiency enhancement is ignoring empty space in the data volume. A cast ray advances through empty space until it encounters an object. It penetrates the object until sufficient opacity has accumulated, and for high opacity this may be a short distance compared with the traversal through empty space. The empty space does not contribute to the final image and because of the large number of voxels, it is important to implement some space-skipping procedure. This can be based on a bounding volume, just as in speed-up schemes in conventional ray tracing, the traversal of the data set starting from the surface of the bounding volume.

13.4.3 Voxel projection method

This variant of volume rendering possibilities involves traversing the data set and projecting each voxel onto the image plane, as we indicated in Figure 13.12. If we move a plane through the data as shown in this figure, then the frame buffer is used as an accumulator and all pixels are updated simultaneously until all the data are completely traversed and the pixels have their final values.

We can traverse the data from either front to back or back to front. The significant difference between these two variants is that with back-to-front traversal we only need to accumulate colour, while with front-to-back traversal we need to accumulate both colour and transparency. (This is exactly equivalent to saying that with front-to-back traversal we require a Z-buffer.)

Voxel projection algorithms are important because they are more easy to parallelize. At each point in the process, that is, at each voxel, we only need knowledge about a small surrounding neighbourhood. This contrasts with ray casting into untransformed data where we generally require the entire data set when we cast a single ray.

Possibly the most well known voxel projection algorithm is due to Westover (1990) and is termed 'splatting'. This strange word is used to describe the effect that one voxel has in the image plane. In effect, the algorithm considers how the contribution of a voxel should be spread or splatted in the image plane. Consider Figure 13.16. A point in the data at the centre of a particular voxel projects onto a single pixel. To determine what the value of the pixel should be we can calculate a contribution by filtering over the three-dimensional region surrounding the sampled voxel. Alternatively, we can take the sample voxel value and spread this over a number of pixels in the image plane. Both approaches are equivalent. If we consider the filter function to be a three-dimensional Gaussian then this projects into the image plane as a circular function. Thus we can project and filter the data by taking the voxel value and splatting it into the image plane by

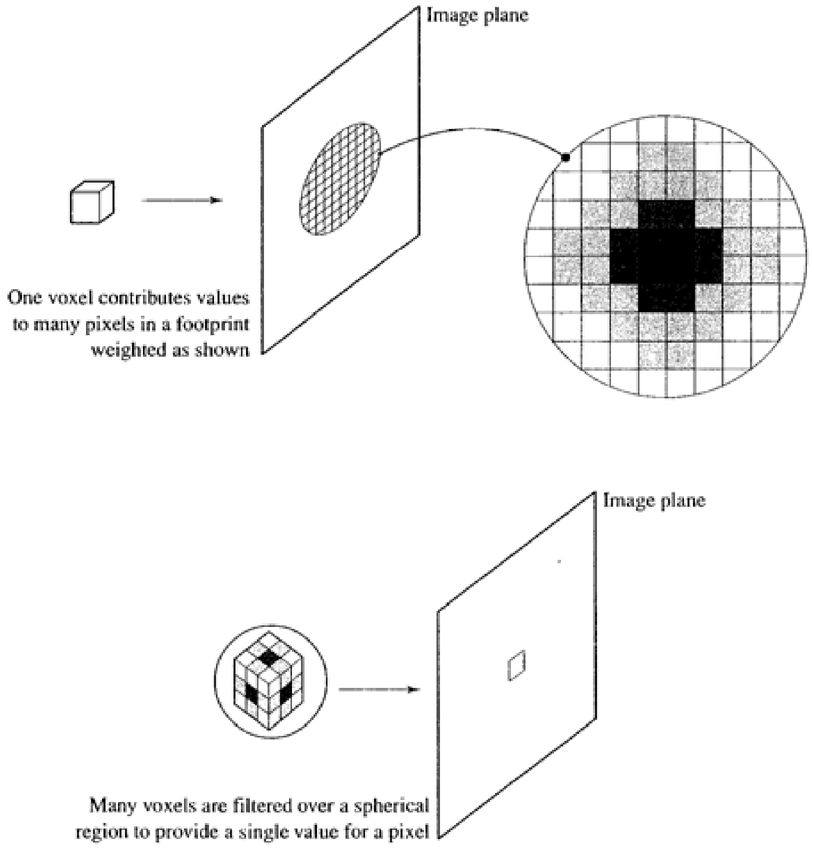


Image plane
One voxel contributes values to many pixels in a footprint weighted as shown
Image plane
Many voxels are filtered over a spherical region to provide a single value for a pixel

Figure 13.16
Filtering in voxel projection.

multiplying it with the filter weights and accumulating these values. This set of values is called the footprint of the voxel and for a parallel projection all footprint weights are the same and can be stored in a look-up table.

13.5 Perspective projection in volume rendering

So far we have not mentioned the issue of perspective projection. In the case of medical imaging it may be that a perspective projection is not required. The volume data in medical applications usually has limited spatial extent of some centimetres and we would not expect to perceive significant perspective clues over this distance. Also it is not usually the shape of the overall structure that is important to the viewer, but some detail such as a fracture or a tumour and its relationship to surrounding structures. Some specific applications in medicine do require a perspective projection. An example is the construction of a 'beam's eye view' in radiation therapy planning. Here, the clinician requires a view of the volume looking down a treatment beam. Treatment beams diverge and so a perspective projection is required.

A number of obvious difficulties occur in constructing a perspective projection in a volume renderer. The most serious results from the divergence of rays from the centre of projection (Figure 13.17). If the ray density is such that the nearest plane in the volume data is sampled with one ray per voxel, then in the example shown, this will quickly drop to one ray per two voxels and small detail can be missed. Another problem is anti-aliasing during resampling. If we consider travelling along the four rays that pass through each of the four corners of a pixel and the centre of projection, the geometry of the volume at the centre of the neighbourhood over which we must filter is no longer a cubic voxel but a truncated pyramid.

One of the easiest ways of implementing perspective projection is to augment the voxel projection or footprint algorithm. Full details of this are given by Westover (1990).

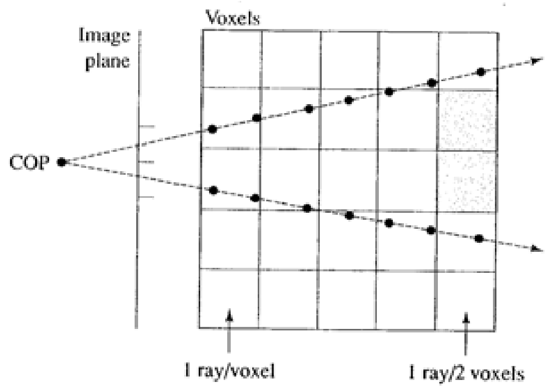


Figure 13.17
Ray density and perspective projection (after Novins *et al.* (1990)). One ray/pixel results in decreasing sampling rate.

13.6 Three-dimensional texture and volume rendering

Since a volume data set can be considered as a three-dimensional texture, then volume rendering can be carried out by a three-dimensional texture mapping facility (Section 8.7). The algorithm (Haeberli and Segal 1993) consists of first calculating the set of parallel polygons that are normal to the viewing direction. This entails finding the intersections between a set of parallel planes and the bounding planes of the data volume. The polygon vertices are then texture mapped and the entire set of polygons is composited in back-to-front order. Since we are now sampling the data with parallel planes, rather than by stepping in equal distances along individual rays, then for a perspective projection the planes will produce unequal sample intervals for rays emanating from the view point. In that case, we need to sample using segments of a sphere rather than planes as shown in Figure 13.18.

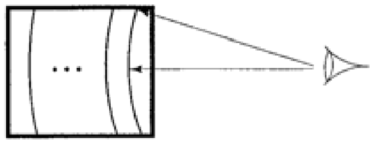


Figure 13.18
Volume data 'sampled' using segments of a sphere centred on the eye point.

14 Anti-aliasing theory and practice

Note
This chapter discusses the classical approach to anti-aliasing and requires some understanding of Fourier theory. A brief introduction, sufficient for an intuitive appreciation of this is given in Section 14.9 at the end of the chapter.

Introduction

The final quality of computer graphics imagery depends on many varied factors. Artefacts arise out of modelling and other factors that are a consequence of operations in the particular rendering algorithm that was used to generate the image. For example, consider the many image defects in polygon mesh scenes. We have modelling artefacts sometimes called geometric aliasing – the visibility of piecewise linearities on the silhouette edge of a polygon mesh object. There are artefacts that emerge from the shading algorithm such as Mach bands and inadequacies due to the interpolation method (see Chapter 18 for a discussion of these). In the case of the radiosity method the view-independent phase throws up difficult quality problems which are not dealt with by general anti-aliasing approaches as we have already discussed in Chapter 11.

Anti-aliasing is the general term given to methods that deal with discrepancies that arise from undersampling and it is this issue which we deal with in this chapter. Such methods are used in conventional rendering approaches like those discussed in Chapter 6 for polygon mesh objects, ray tracing in Chapter 12 and in the Monte Carlo techniques discussed in Chapter 10. Anti-aliasing in texture mapping is discussed in Chapter 8 for the reason that, although it is a classical approach, the particular implementation – mip-mapping – is used exclusively with texture mapping.

14.1 Aliases and sampling

We first consider the term 'alias'. In theory this refers to a particular image artefact that is mostly visible in texture maps when the periodicity in the texture approaches the dimension of a pixel. This is easily demonstrated and Figure 14.1(a) is the classic example of this effect – an infinite chequerboard. Towards the top of the image the squares reduce, then apparently increase in size, causing a glaring visual disturbance. This is due to undersampling. The notion of sampling in computer graphics comes from the fact that we are calculating a single colour or value for each pixel; we are sampling a solution at discrete points in a solution space. This is a space that is potentially continuous in the sense that, because computer graphics images are generated from abstractions, we can calculate samples anywhere or everywhere in the image plane.

We will now look at a simple one-dimensional example which will relate undersampling, aliases and the notion of spatial frequencies. Consider using a sine wave to represent an information signal (although a sine wave does not contain any information anyway, this does not matter for our purposes). Figure 14.2 shows a sine wave being sampled at different rates (with respect to the frequency of the sine wave). Undersampling the sine wave and reconstructing a

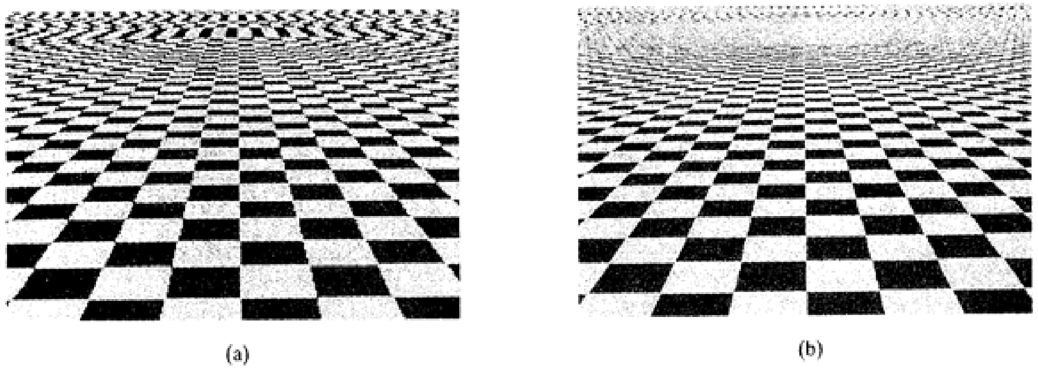


Figure 14.1
The pattern in (b) is a supersampled version of that in (a). Aliases still occur but appear at a higher spatial frequency.

continuous signal from the samples (dotted line in the figure) produces an 'alias' of the original signal – another sine wave at a lower frequency than the one being sampled. We can say that this happens because the coherence or regularity of the sampling pattern is interfering with the regularity of the information. To avoid aliasing artefacts we have to sample at an appropriately high frequency with respect to the signal or image information and we normally consider the process of calculating an image function at discrete points in the image plane to be equivalent to sampling.

The defects that arise in computer graphics that are due to insufficient calculations or samples and which are easily modelled by an image plane sampling model are coherent patterns breaking up – the case that we have already discussed – and small fragments that are missed because they fall between two sample points.

Consider the chequerboard example again. The pattern units approach the size of a pixel very quickly and the pattern 'breaks up'. High spatial frequencies are aliasing as lower ones and forming new visually disturbing coherent patterns. Now consider Figure 14.1(b) where we render the same image onto a view plane with double the resolution of the previous one. Aliasing artefacts still appear but at a higher spatial frequency. In theoretical terms we have increased the sampling frequency, but the effect persists except that it happens at a higher spatial frequency. This demonstrates two important facts. Spatial frequencies in a computer graphics image are unlimited because they originate from a mathematical definition. You cannot get rid of aliases by simply increasing the pixel resolution.

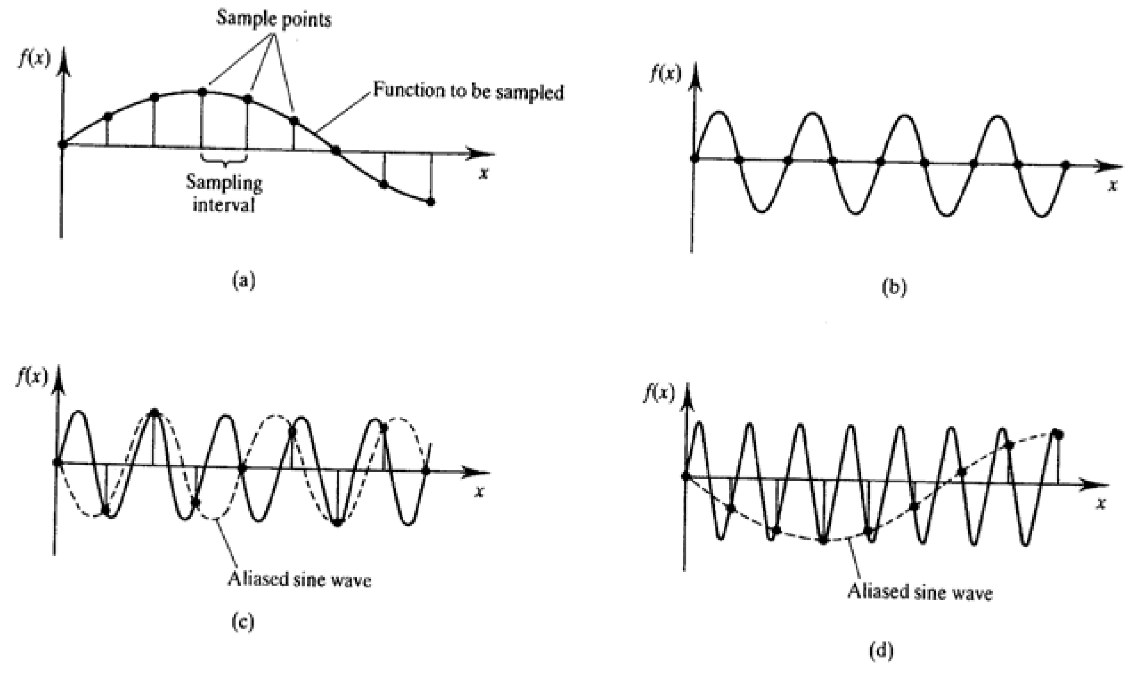


Figure 14.2
Space domain representation of the sampling of a sine wave. (a) Sampling interval is less than one-half the period of the sine wave. (b) Sampling interval is equal to one-half the period of the sine wave. (c) Sampling interval is greater than one-half the period of the sine wave. (d) Sampling interval is much greater than one-half the period of the sine wave.

The artefacts simply occur at a higher spatial frequency. But they are, of course, less noticeable.

Now, the example in Figure 14.2 can be generalized by considering these cases in the frequency domain for an $f(x)$ that contains information, that is not a pure sine wave. We now have an $f(x)$ that is any general variation in x and may, for example, represent the variation in intensity along a segment of a scan line. The frequency spectrum of $f(x)$ will exhibit some 'envelope' (Figure 14.3(a)) whose limit is the highest frequency component in $f(x)$, say, $f_{\max}$. The frequency spectrum of a sampling function (Figure 14.3(b)) is a series of lines, theoretically extending to infinity, separated by the interval f_s (the sampling frequency). Sampling in the space domain involves multiplying $f(x)$ by the sampling function. The equivalent process in the frequency domain is convolution and the frequency spectrum of the sampling function is convolved with $f(x)$ to produce the frequency spectrum shown in Figure 14.3(c) – the spectrum of the sampled version of $f(x)$. This sampled function is then multiplied by a reconstructing filter to reproduce the original function. A good example of this process, in the time domain, is a modern telephone network. In its simplest form this involves sampling a speech waveform, encoding and transmitting

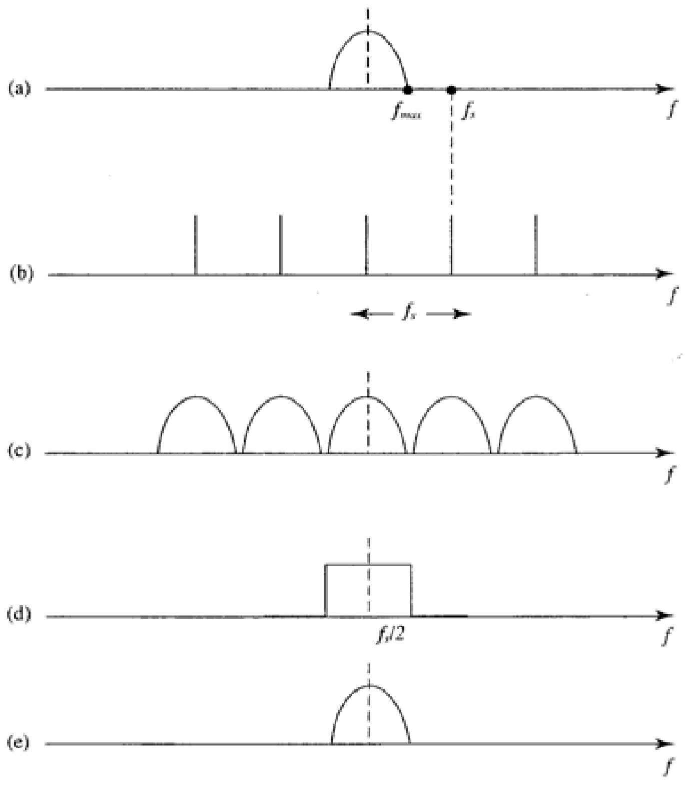


Figure 14.3
Frequency domain representation of the sampling process when $f_s > 2f_{max}$. (a) Frequency spectrum of $f(x)$.
(b) Frequency spectrum of the sampling function.
(c) Frequency spectrum of the sampled function (convolution of (a) and (b)).
(d) Ideal reconstruction filter. (e) Reconstructed $f(x)$.

digital versions of each sample over a communications channel, then reconstructing the original signal from the decoded samples by using a reconstructing filter.

Note that the reconstruction process, which is multiplication in the frequency domain, is convolution in the space domain. In summary, the process in the space domain is multiplication of the original function with the sampled function, followed by convolution of the sampled version of the function with a reconstructing filter.

Now in the above example the condition:

$$f_s > 2f_{max}$$

is true. In the second example (Figure 14.4) we show the same two processes of multiplication and convolution but this time we have:

$$f_s < 2f_{max}$$

Incidentally, $f_s/2$ is known as the Nyquist limit. Here the envelopes, representing the information in $f(x)$, overlap. It is as if the spectrum has 'folded' over a line defined by the Nyquist limit (Figure 14.4(e)). This folding is an information-destroying process; high frequencies (detail in images) are lost and

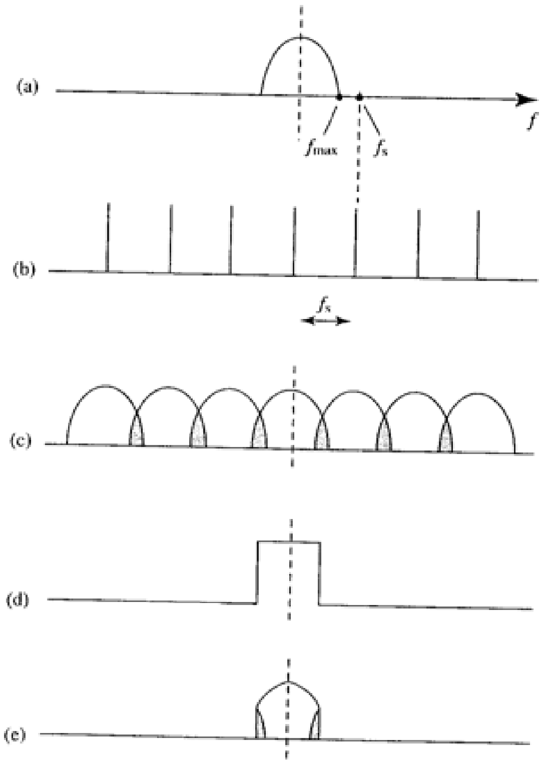


Figure 14.4
Frequency domain representation of the sampling process when $f_s < 2f_{max}$. (a) Frequency spectrum of $f(x)$. (b) Frequency spectrum of the sampling function. (c) Frequency spectrum of the sampled function. (d) Ideal reconstruction filter. (e) Distorted $f(x)$.

appear as interference (aliases) in low frequency regions. This is precisely the effect shown in Figure 14.1 where low spatial frequency structures are emerging in high frequency regions.

The sampling theorem extends to two-dimensional frequencies or spatial frequencies. The two-dimensional frequency spectrum of a graphics image in the continuous generation domain is theoretically infinite. Sampling and reconstructing in computer graphics is the process of calculation of a value at the centre of a pixel and then assigning that value to the entire spatial extent of that pixel.

Aliasing artefacts in computer graphics can be reduced by increasing the frequency of the sampling grid (that is increasing the spatial resolution of the pixel array). There are two drawbacks to this approach: the obvious one that there is both an economic and a technical limit to increasing the spatial resolution of the display (not to mention the computational limits on the cost of the image generation process) and, since the frequency spectrum of computer graphics images can extend to infinity, increasing the sampling frequency does not necessarily solve the problem. When, for example, we applied the increased resolution approach to coherent texture in perspective, we simply shifted the effect up the spatial frequency spectrum.

14.2 Jagged edges

The most familiar defects in computer graphics are called jaggies. These are produced by the finite size of a (usually) square pixel when a high contrast edge appears in the image. These are particularly troublesome in animated images where their movement gives them the appearance of small animated objects and makes them glaringly visible. These defects are easier to get rid of because they do not arise out of the algorithm *per se* – they are simply a consequence of the resolution of the image plane.

Jagged edges are recognized by everyone and described in all computer graphics textbooks; but they are not aliasing defects in the classical sense of an aliased spatial frequency, where a high spatial frequency appears as a disruptive lower one. They are defects produced by the final limiting effect of the display device. We can certainly ameliorate their effect by, for example, calculating an image at a resolution higher than the pixel resolution; in other words increasing the sampling frequency deals with both aliases and jaggies. In the case of jaggies, edge information is 'forced' into the horizontal and vertical edges of the pixels. Consider Figure 14.5 which shows a perfect rectangle and a pixelized version. The Fourier transform for the perfect rectangle maps the edge information into high energy components along directions corresponding to the orientation of the edges in the image. The Fourier version of the pixel version also contains this information together with high energy components along the axes corresponding to the false or pixel edges. Jaggies do not arise because of high spatial frequencies aliasing as lower ones.

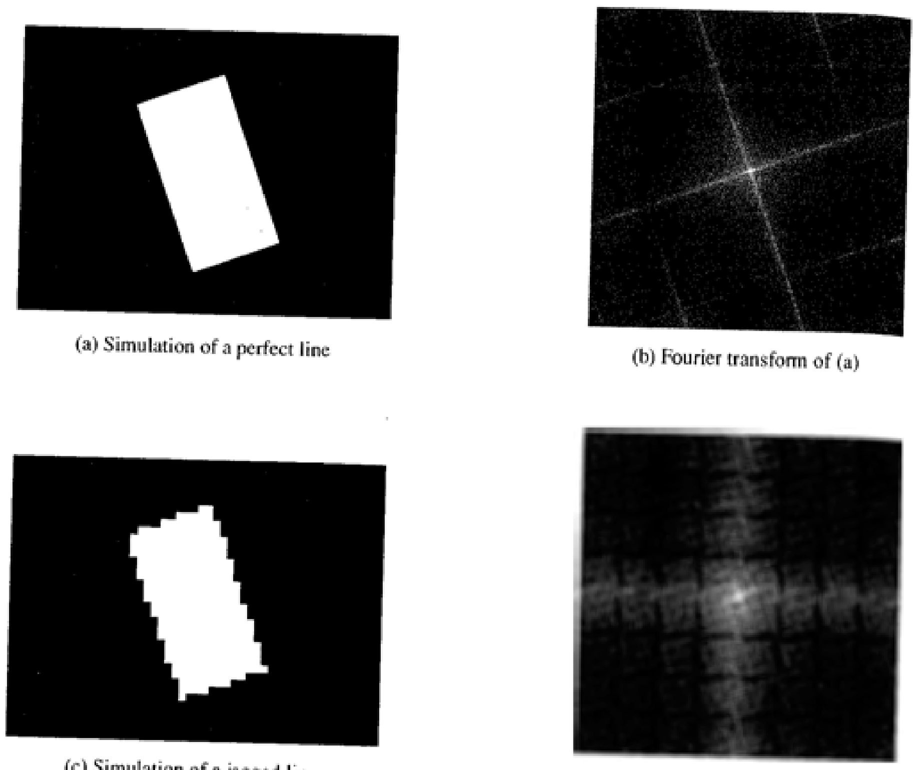


Figure 14.5
The effect of jaggies is to rotate high energy components onto the horizontal and vertical axes in the Fourier domain.

(a) Simulation of a perfect line

(b) Fourier transform of (a)

(c) Simulation of a jagged line

(d) Fourier transform of (c)

14.3 Sampling in computer graphics compared with sampling reality

Let us now return in more detail to the notion of sampling in the image plane. In image synthesis what we are doing is performing, for each pixel, a number of (sometimes very complicated) operations that eventually calculate, for that pixel, a constant value. Usually we calculate a value at the centre of the pixel and 'spread' that value over the pixel extent.

We assume that, in principle, this is no different from having a continuous image in the view plane and sampling this with a discrete two-dimensional array of sample points (one for each pixel). We say that this assumption is valid because we can approach such an image by increasing and increasing the sample resolution and calculating a value for the image at more and more points in the image plane. However, it is important to bear in mind that we do not have access to a continuous image in computer graphics and this limits and conditions our approaches to anti-aliasing measures.

In fact, both the terms 'sampling' and 'reconstruction' – another term borrowed from digital signal processing – are used indiscriminately and, we feel, somewhat confusingly in computer graphics, and we will now emphasize the difference between an image processing system, where their usage is wholly appropriate, and their somewhat artificial use in computer graphics.

Consider Figure 14.6 which shows a schematic diagram for an image processor and a computer graphics system. In the image processor a sampler converts a two-dimensional continuous image into an array of samples. Some operations are then performed on the digital image and a reconstruction filter converts the processed samples back into an analogue signal.

Not so in image synthesis. Sampling does not exist in the same sense – the operations involved in assigning a value to a pixel depend on the rendering algorithm used and we can only ever calculate the value of an image function at these points.

Reconstruction, in image synthesis does not mean generating a continuous image from a digital one but may mean, for example, generating a low (pixel) resolution image from an image stored at a higher (undisplayable) resolution. We are not reconstructing an image since a continuous image never existed in the first place. An appreciation of these differences will avoid confusion. (In reality we do reconstruct a continuous image for display on a computer graphics monitor, but this is done by fixed electronics that operate on the image produced in the framestore by a graphics program. A comprehensive approach to anti-aliasing would need to take the transfer characteristics of the conversion electronics into account but we will not do so in this text.)

To return to the problem of aliasing artefacts. Fourier theory tells us that aliasing occurs because we sample a continuous image (or the equivalent operation in computer graphics) and we do not do this at a high enough resolution to capture the high spatial frequencies or detail in the image. The sampling theorem states that if we wish to sample an image function without loss of information then our (two-dimensional) sampling frequency must be at least twice as high as the highest frequency component in the image.

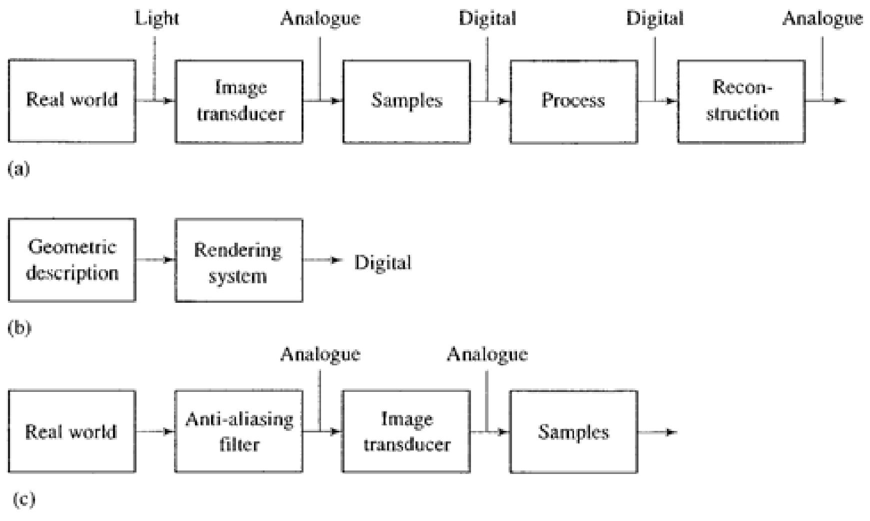


Figure 14.6
Sampling, reconstruction and anti-aliasing in image processing and image synthesis. (a) Image capture and processing. (b) Image synthesis. (c) Anti-aliasing in image capture.

So what does this mean in terms of practical computer graphics? Just this: if we consider we are sampling a continuous image in the view plane with a grid of square pixels, then the highest frequency that can appear along a scan line is:

$$f = 1/2d$$

where d is the distance between pixel centres.

Having fixed these concepts it is easy to see why anti-aliasing is so difficult in computer graphics. The problem stems from two surprising facts. There is no limit to the value of the high frequencies in computer graphics – we have already discussed this using the example of the infinite chequerboard – and there is no direct way to limit (the technical term is band-limit) these spatial frequencies.

This is easily seen by comparing image synthesis with image capture through a device like a TV camera (Figure 14.6c). Prior to sampling a continuous image we can pass it through a band limiting filter (or an anti-aliasing filter). Higher frequencies that cannot be displayed are simply eliminated from the image before it is sampled. We say that the image is pre-filtered. In such systems aliasing problems are simply not allowed to occur.

In image synthesis our scene database exists as a mathematical description or as a set of points connected by edges. Our notion of sampling is inextricably entwined with rendering. We sample by evaluating the projection of the scene at discrete points. We cannot band limit the image because no image exists – we can only define its existence at the chosen points.

14.4 Sampling and reconstruction

In Figure 14.3 we saw that provided the sampling theory is obeyed then reconstruction of the information from the samples is obtained by using a reconstruction filter in the shape of a box. However, this is a Fourier domain representation and in computer graphics all our operations have to take place in the space domain. Therefore the reconstruction process is convolution in the space or image domain. In computer graphics this implies (usually) filtering a rendered image in some way. If the rendered image was continuous then our reconstruction filter would consist of a sinc function $h(x, y)$ – which is the transform of the Fourier domain equivalent of a circular step function (Figure 14.7).

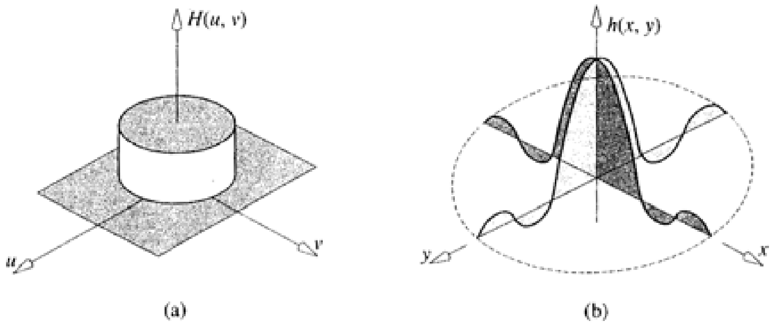


Figure 14.7
Ideal filters in the Fourier and space domains.
(a) An ideal low pass (multiplicative) filter $H(u,v)$.
(b) The equivalent (convolving) filter $h(x,y)$.

There are, however, practical difficulties associated with this. The filter cannot have unlimited extent – it has to be truncated at some point and the way in which this is done is an important aspect of the design of the filter.

14.5 A simple comparison

We will now consider the anti-aliasing options in computer graphics briefly in the form of a comparative overview. Figure 14.8 shows four main approaches.

(1) Pre-filtering – 'infinite' samples per pixel
Here we calculate the precise contribution of fragments of projected object structure as it appears in a pixel. This single value is taken as the pixel colour. The practical effect of this approach is simply a reduction of the 'infinite' resolution to the finite resolution of the pixel display. If the physical extent of a pixel is small this is a high quality but totally impractical method. However, note that although this method assumes accurate geometry we assume that the light intensity is constant across any fragment. Effectively what we are doing with this algorithm is pre-filtering – that is, filtering *before* sampling using a box filter.

This is the method which approaches the anti-aliasing filter in Figure 14.6(c). It effectively removes those high frequencies that manifest as sub-pixel detail but because the calculations are continuous it is doing this before sampling.

(2) No filtering – one sample per pixel
In the second case we consider only one sample per pixel. This becomes equivalent to the first case if, and only if, the projection is such that a pixel only ever contains a single geometric structure and all structure boundaries in the projection coincide with pixel edges – impossible constraints in practice. This 'do nothing' approach is extremely common in real-time animation. It is also used as a preview method in off-line production where a final anti-aliased image is generated only when a creator is satisfied with the preview.

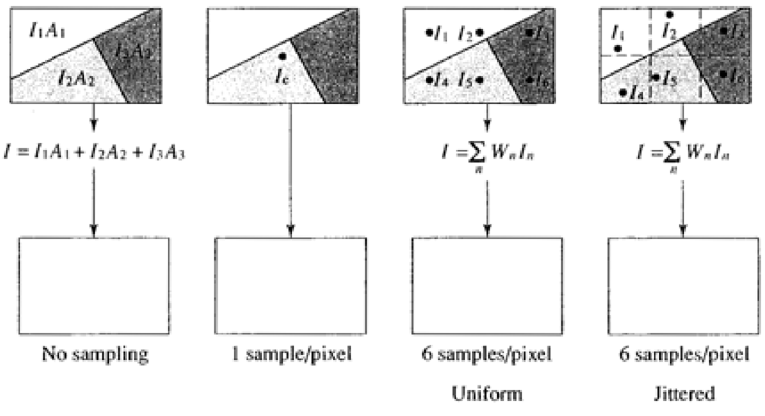


Figure 14.8
A comparison of four approaches to calculating a single value for a pixel.

(3) Post filtering – n uniform samples per pixel
This is the commonest approach to anti-aliasing and involves rendering a virtual image at *n* times the resolution of the final screen image. This is an approximation to the notion of a continuous image. The final image is then produced by sampling the virtual image and reconstructing it by a convolution operation. Both operations are combined into a single operation. The effectiveness of this approach depends on the number of supersamples and the relationship between the image structure within a pixel and the sampling grid point. Note that although we can regard this approach as an approximation to the first case, the samples that relate to the same fragment can now have different intensities.

(4) Post-filtering – stochastic samples
This approach can be seen as a simple alteration of the previous – instead of uniformly sampling within a pixel we now jitter the samples according to some scheme. This approach has already been discussed in Chapter 10 (see Figure 10.9) as an integral part of Monte Carlo methods. In this chapter we will look at why it functions as a 'pure' anti-aliasing technique.

14.6 Pre-filtering methods

The originator of this technique was Catmull (Catmull 1978). Although Catmull's original algorithm is prohibitively expensive, it has spawned a number of more practical successors.

The algorithm essentially performs sub-pixel geometry in the continuous image generation domain and returns, for each pixel, an intensity which is computed by using the areas of visible sub-pixel fragments as weights in an intensity sum. This is equivalent to convolving the image with a box filter and using the value of the convolution integral at a single point as the final pixel value. (Note the width of the filter is less than ideal and a wider filter using information from neighbouring regions would give a lower cut-off frequency.) Another way of looking at the method is to say that it is an area sampling method.

We can ask the question: what does performing 'sub-pixel geometry' mean in practical computer graphics terms? To do this we inevitably have to use a practical approximation. (To reiterate an earlier point, we have no access to a continuous image. In computer graphics we can only define an image at certain points.) This means that the distinction between sampling techniques and supersampling is somewhat artificial and indeed the A-buffer approach (described shortly), usually categorized as an area sampling technique, could equally well be seen as supersampling.

Catmull's method is incorporated in a scan line renderer. It proceeds by dividing the continuous image generation domain into square pixel extents. An intensity for each square is computed by clipping polygons against the square pixel boundary. If polygon fragments overlap within a square they are sorted in z and clipped against each other to produce visible fragments. A final intensity is computed by multiplying the shade of a polygon by the area of its visible fragment and summing.

The origin of the severe computational overheads inherent in this method is obvious. The original method was so expensive that it was only used in two-dimensional animation applications involving a few largish polygons. Here, most pixels are completely covered by a polygon and the recursive clipping process of polygon fragment against polygon fragment is not entered.

Recent developments have involved approximating the sub-pixel fragments with bit masks (Carpenter 1984; Fiume *et al.* 1983). Carpenter (1984) uses this approach with a Z-buffer to produce a technique known as the A-buffer (anti-aliased, area averaged, accumulator buffer). The significant advantage of this approach is that floating point geometry calculations are avoided. Coverage and area weighting are accomplished by using bitwise logical operators between the bit patterns or masks representing polygon fragments. It is an efficient area sampling technique, where the processing per pixel square will depend on the number of visible fragments.

Another efficient approach to area sampling, due to Abram *et al.* (1985), pre-computes contributions to the convolution integral and stores these in look-up tables indexed by the polygon fragments. The method is based on the fact that the way in which a polygon covers a pixel can be approximated by a limited number of cases. The algorithm is embedded in a scan line renderer. The convolution is not restricted to one pixel extent but more correctly extends over, say, a 3 × 3 area. A pixel acts as an accumulator whose final value is correct when all fragments that can influence its value have been taken into account.

Consider a 3 × 3 pixel area and a 3 × 3 filter kernel (Figure 14.9). A single visible fragment in the centre pixel will contribute to the convolution integral when the filter is centred on each of the nine squares. The nine contributions that such a fragment makes can be pre-computed and stored in a look-up table. The two main stages in the process are:

(1) Find the visible fragments and identify or categorize their shape.

(2) Index a pre-computed look-up table which gives the nine contributions for each shape. A single multiplication of the fragment's intensity by the pre-computed contribution weighting gives the desired result.

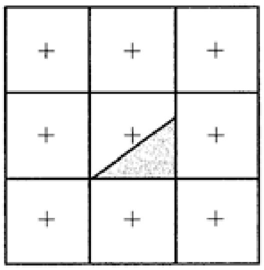


Figure 14.9
A single fragment in the centre pixel will cause contributions to filtering on each of the nine squares.

0424

Abram assumes that the shapes fall into one of seven categories:

(1) There is no fragment in the pixel.

(2) The fragment completely covers the pixel.

(3) The fragment is trapezoidal and splits the pixel along opposite edges.

(4) The fragment is triangular and splits the pixel along adjacent edges.

(5) The complement of (4) (a pentagonal fragment).

(6) The fragment is an odd shape that can be described by the difference of two or more of the previous types.

(7) The fragment cannot be easily defined by these simple types.

14.7 Supersampling or post-filtering

Supersampling is the most common form of anti-aliasing and is usually used with polygon mesh rendering. It involves calculating a virtual image at a spatial resolution higher than the pixel resolution and 'averaging down' the high resolution image to a lower (pixel) resolution. In broad terms, subject to the previous reservations about the use of the term 'sampling', we are increasing the sampling frequency. The advantage of the method is trivial implementation which needs to be set against the high disadvantage of cost and increased Z-buffer memory. In terms of Fourier theory we can:

(1) Generate a set of samples of $I(x, y)$ at some resolution (higher than the pixel resolution).

(2) Low pass filter this image which we regard as an approximation to a continuous image.

(3) Re-sample the image at the pixel resolution.

Steps 2 and 3 (often confusingly referred to as reconstruction) are carried out simultaneously by convolving a filter with the virtual image and using as steps in the convolution intervals of pixel width. That is, for a 3 × virtual image, the filter would be positioned on (super) pixels in the virtual image, using a step length of three super-pixels. Figure 14.10 is a representation of the method working and two examples of filters tabulated as weights (note that these are normalized – the filter weights must sum to unity). For an (odd) scaling factor S and a filter h of dimension k:

$$I'(i, j) = \sum_{p=Si-k}^{Si+k} \sum_{q=Sj-k}^{Sj+k} I(p, q)h(Si - p, Sj - q)$$

This method works well with most computer graphics images and is easily integrated into a Z-buffer algorithm. It does not work with images whose spectrum energy does not fall off with increasing frequency. (As we have already mentioned, supersampling is not, in general a theoretically correct method of anti-aliasing.)

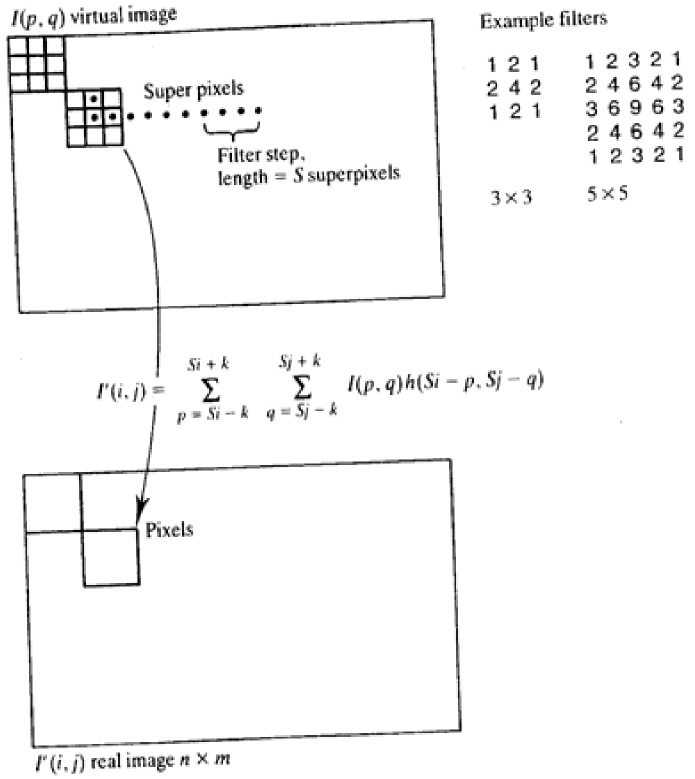

$I(p, q)$ virtual image
Super pixels
Filter step, length = S superpixels
Example filters
1 2 1
2 4 2
1 2 1
3×3
1 2 3 2 1
2 4 6 4 2
3 6 9 6 3
2 4 6 4 2
1 2 3 2 1
5×5
$I'(i, j) = \sum_{p = Si - k}^{Si + k} \sum_{q = Sj - k}^{Sj + k} I(p, q)h(Si - p, Sj - q)$
Pixels
$I'(i, j)$ real image $n \times m$

Figure 14.10
'Reducing' a virtual image by convolution.

Supersampling methods differ trivially in the value of n and the shape of the filter used. For, say, a medium resolution image of 512×512 it is usually considered adequate to supersample at 2048×2048 ($n = 4$). The high resolution image can be reduced to the final 512×512 form by averaging and this is equivalent to convolving with a box filter. Better results can be obtained using a shaped filter, a filter whose values vary over the extent of its kernel. There is a considerable body of knowledge on the optimum shape of filters with respect to the nature of the information that they operate on (see, for example, Oppenheim and Shafer (1975)). Most of this work is in digital signal processing and has been carried out with functions of a single variable $f(t)$. Computer graphics has unique problems that are not addressed by conventional digital signal processing techniques. For example, space variant filters are required in texture mapping. Here, both the weights of the filter kernel and its shape have to change.

To return to supersampling and non-varying filters, Crow (1981) used a Bartlett window, three of which are shown in Table 14.1.

Digital convolution is easy to understand and implement but is computationally expensive. A window is centred on a supersample and a weighted sum of products is obtained by multiplying each supersample by the corresponding

Table 14.1 Bartlett windows used in post-filtering a supersampled image

3 × 3	5 × 5	7 × 7
1 2 1	1 2 3 2 1	1 2 3 4 3 2 1
2 4 2	2 4 6 4 2	2 4 6 8 6 4 2
1 2 1	3 6 9 6 3	3 6 9 12 9 6 3
	2 4 6 4 2	4 8 12 16 12 8 4
	1 2 3 2 1	3 6 9 12 9 6 3
		2 4 6 8 6 4 2
		1 2 3 4 3 2 1

weight in the filter. The weights can be adjusted to implement different filter kernels. The digital convolution proceeds by moving the window through n supersamples and computing the next weighted sum of products. Using a 3 × 3 window means that nine supersamples are involved in the final pixel computation. On the other hand, using the 7 × 7 window means a computation of 49 integer multiplications. The implication of the computation overheads is obvious. For example, reducing a 2048 × 2048 supersampled image to 512 × 512, with a 7 × 7 filter kernel, requires 512 × 512 × 49 multiplications and additions.

An inevitable side-effect of filtering is blurring. In fact, we could say that we trade aliasing artefacts against blurring. This occurs because information is integrated from a number of neighbouring pixels. This means that the choice of the spatial extent of the filter is a compromise. A wide filter has a lower cut-off frequency and will be better at reducing aliasing artefacts. It will, however, blur the image more than a narrower filter which will exhibit a higher cut-off frequency.

Finally, the disadvantages of the technique should be noted. Supersampling is not a suitable method for dealing with very small objects. Also it is a 'global' method – the computation is not context dependent. A scene that exhibited a few large-area polygons would be subject to the same computational overheads as one with a large number of small-area polygons. The memory requirements are large if the method is to be used with a Z-buffer. The supersampled version of the image has to be created and stored before the filtering process can be applied. This increases the storage requirements of the Z-buffer by a factor of n^2, making it essentially a virtual memory technique.

14.8 Non-uniform sampling – some theoretical concepts

Non-uniform sampling has become of great interest in computer graphics because it addresses the high cost problem of conventional anti-aliasing techniques. It does this by getting away from the idea of uniform sampling and allows us to address the issue of context-sensitive anti-aliasing measures, or devoting computing resources to those parts of the image that need attention. The way in which this is done invariably means that we study algorithms where there is no separation between the rendering part and the anti-aliasing part.

We cannot, as we did above with supersampling, render without using the anti-aliasing strategy.

Another benefit of considering non-uniform sampling is that it enables algorithms where we can convert aliases into noise. That is, we can design algorithms in such a way that, for a given pixel resolution, the algorithm produces noise where a conventional algorithm would produce aliases. Approaches that do this are called stochastic sampling methods and they function by making uniform intervals between samples irregular.

Ideally we wish to generate an image using most effort in busy regions and least in regions where the illumination is changing slowly. The crux of the matter in image synthesis is: how do we know which regions to devote most attention to before we have generated the image? This consideration leads us naturally to the most common strategy which is to generate a low resolution image, examine it, and use this to generate a higher resolution image in those areas of the low resolution image that appear to need further attention. We can go on repeating this process recursively until we come up against some pre-specified limit. This is called adaptive refinement (an example of this technique is shown in Figure 18.13).

A simple, but by no means complete, taxonomy of non-uniform sampling would be the two main categories of non-uniform subdivision and stochastic sampling. There are many subdivisions – different ways of effecting the stochastic sampling and ways of combining the two approaches into a single sampling strategy. For example, a stochastic sampling pattern may be generated at different scales (number of samples per unit area) so that it can be incorporated in an adaptive refinement scheme.

The approaches are represented schematically in Figure 14.11. Both these methods are applied after an initial sampling of the image plane has taken place. Most commonly in computer graphics this is uniform sampling, usually but not necessarily at pixel level. The techniques then become non-uniform supersampling in that the non-uniform strategy operates at sub-pixel level.

Non-uniform subdivision is a general strategy that appears in many algorithms in computer science. It naturally fits into an adaptive refinement scheme in image synthesis which consists of dividing the image plane into a grid of initial (say square) sampling boxes, then recursively subdividing these into squares until a resolution limit is reached. There is another subtle problem with such methods. This is that the output from an algorithm that uses this kind of

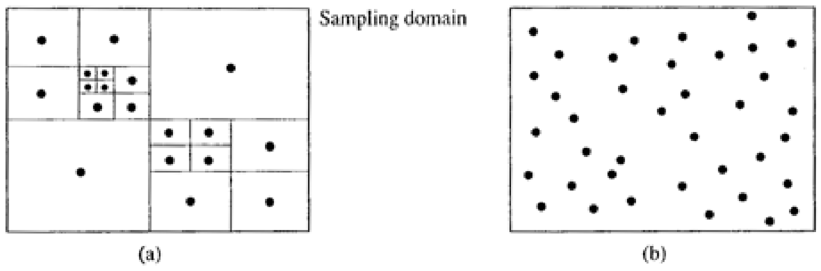


Figure 14.11
The two main non-uniform sampling techniques.
(a) Non-uniform subdivision;
(b) stochastic sampling.

strategy is going to be a set of non-uniform samples. These have to be converted into a uniform set of (pixel) samples prior to display. Alternatively we can say that we have to reconstruct the image from non-uniform samples and then re-sample at a uniform rate. There is no worked out theory that encompasses reconstruction from non-uniform samples and a variety of ad hoc techniques exist. A simple scheme is shown in Figure 14.12.

Stochastic sampling seems at first sight a strange idea but an intuitive explanation of its efficacy is straightforward. Aliases appear in an image as a direct consequence of the regularity of the sampling pattern 'beating' with regularities or coherences in the image. If we make the samples irregular then the higher frequency coherences in the image will appear as noise rather than aliases. This perturbation of regular sampling, and consequent trade-off of aliasing against noise is stochastic sampling.

An easy demonstration of the functioning of this trade-off is to return to our sine wave example. Figure 14.13 shows a sine wave, again being sampled by a regular sampling pattern. Now we can invoke a stochastic sampling technique by 'jittering' each sample by some random amount about the regular sampling instant. Consider the effect of doing this on a sine wave whose frequency is below the Nyquist limit (Figure 14.13(a)). Here our procedure will sample the sine wave inaccurately, introducing amplitude perturbations, or noise, that depends on the extent of the sample instant jitter. For a sine wave whose

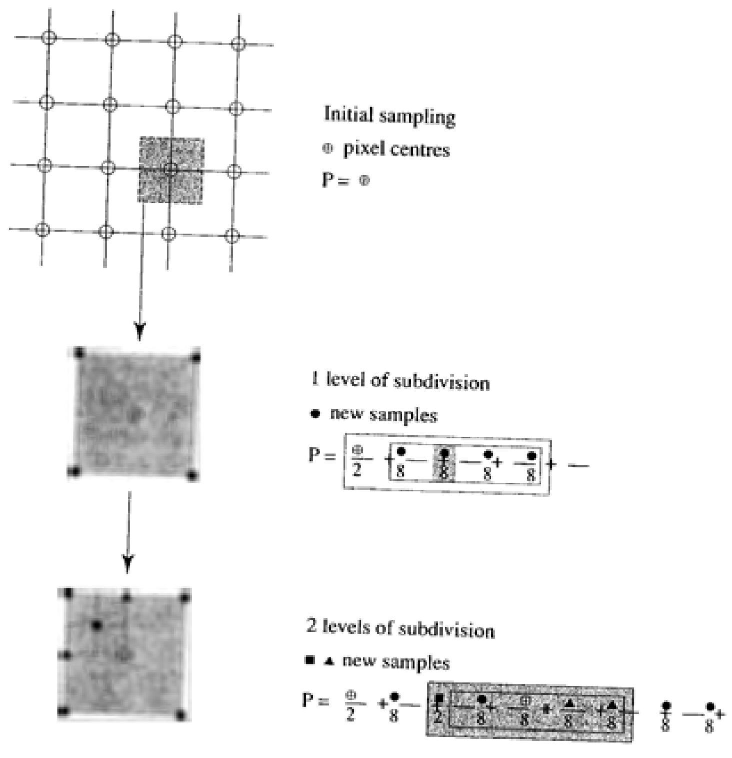


Figure 14.12
Simple reconstruction for non-uniform subdivision.

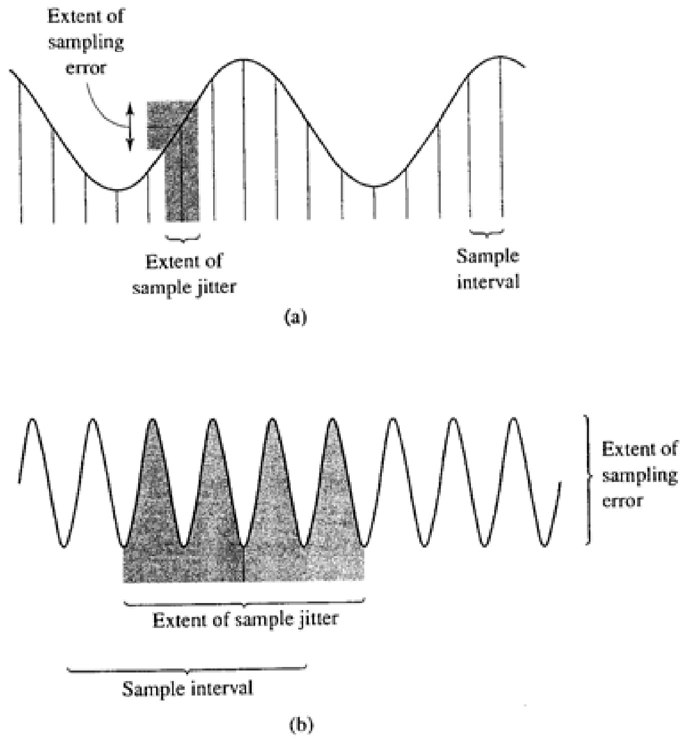


Figure 14.13
Sampling a sine wave whose frequency is (a) below and (b) above the Nyquist limit (after Cook).

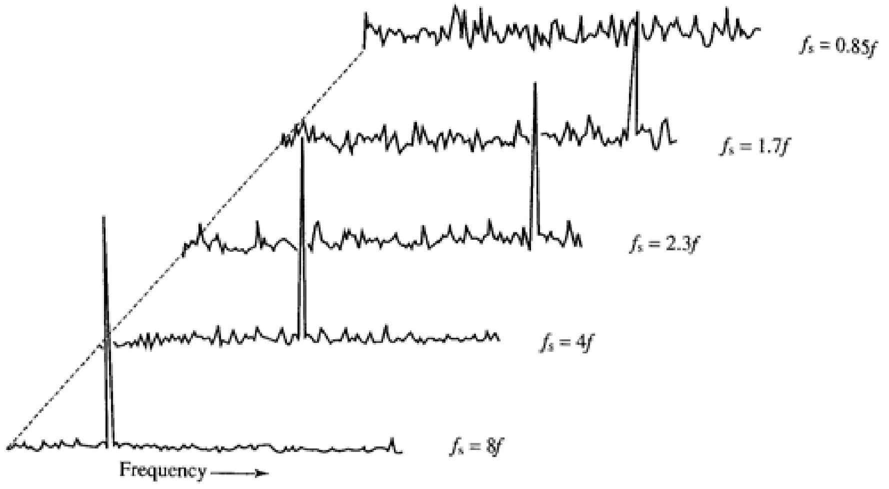
frequency is well above the Nyquist limit (Figure 14.13(b)), the sample jitter extent encompasses many cycles and the effect of sampling successively such packets of waves will simply be to produce a set of random numbers. Thus the aliased sine wave that would be produced by a regular sampling interval is exchanged for noise.

Jittering is easily carried out within a two-dimensional area, such as a pixel, by starting with a uniform grid and applying two component jitters in the x and y directions. This is cheap and easy to do and for this reason it is probably the most common strategy in computer graphics.

Stochastic sampling has an interesting background. In 1982, Yellot (Yellot 1982) pointed out that the human eye contains an array of non-uniformly distributed photoreceptors and he suggested that this is the reason that the human eye does not produce its own aliasing artefacts. Photoreceptor cells in the fovea are tightly packed and the lens acts as an anti-aliasing filter. However, in the region outside the fovea, the spatial density of photoreceptors is much lower and for this reason the cells are non uniformly distributed

These factors are easily demonstrated in the frequency domain by considering the spectrum of a sine wave sampled by this method and again varying the frequency about the Nyquist limit (Figure 14.14). As the sampling frequency is reduced with respect to the sine wave the amplitude of the sine wave spike diminishes and the noise amplitude increases. Eventually the sine wave peak

Figure 14.14
Varying the frequency of a sine wave (f) with respect to a perturbed sampling frequency (f_s).



disappears. The point of the illustration is that no alias spikes appear. The information represented by the sine wave eventually disappears but instead of aliasing we get noise. The perturbation can range in x over a minimum of half a cycle (where the sine wave frequency is at the Nyquist limit) and will in general range over a number of complete cycles. If the range encompasses a number of cycles exactly then, for white noise jitter, the probability of sampling each part of the sine wave tends to be equal and the energy in the samples appears as white noise. A mathematical treatment of the attenuation due to white noise jitter and Gaussian jitter is given in Balakrishnan (1962).

One of the problems of this method is that it is only easily incorporated into methods where independent calculations are made for each sample. This is certainly the case in ray tracing, where rays are spawned in the continuous object space domain, and are, in effect, samples in this space. They can easily be jittered. In 'standard' image synthesis methods, using, say, interpolative shading in the context of a Z-buffer or scan line algorithm, introducing jitter presents much more of a difficulty. The algorithms are founded on uniform incremental methods in screen space and would require substantial modification to have the effect of two-dimensional sampling perturbation. Although such algorithms are equivalent to image generation in a continuous domain succeeded by two-dimensional sampling, in practice the sampling and generation phases are not easily unmeshed.

A major rendering system, called REYES (Cook *et al.* 1987) does, however, integrate a Z-buffer-based method with stochastic sampling. This works by dividing initial primitives, such as bi-cubic parametric patches into (flat) 'micropolygons' (of approximate dimension in screen space of half a pixel). All shading and visibility calculations operate on micropolygons. Shading occurs prior to

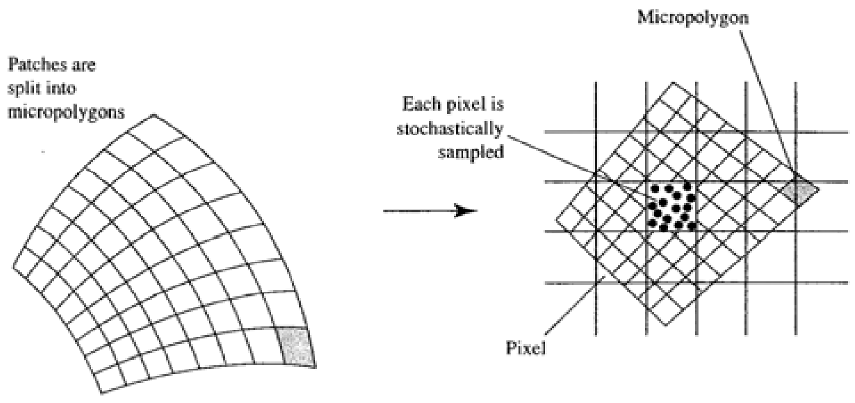

Patches are split into micropolygons
Each pixel is stochastically sampled
Micropolygon
Pixel

Figure 14.15 Graphical primitives are subdivided into micropolygons. These are shaded and visibility calculations are perfomred by stochastically sampling the micropolygons in screen space (after Cook *et al.* (1987)).

visibility calculations and is constant over a micropolygon. The micropolygons are then stochastically sampled from screen space, the *Z* value of each sample point calculated by interpolation and the visible sample hits filtered to produce pixel intensities (Figure 14.15). Thus shading is carried out at micropolygon level and visibility calculations at the stochastic sampling level.

This method does away with the coherence of 'classical' rendering methods, by splitting objects into micropolygons. It is most suitable for objects consisting of bi-cubic parametric patches because they can be easily subdivided.

14.9 The Fourier transform of images

Fourier theory is not used to any extent in computer graphics except in specialized applications such as generating terrain height fields using Fourier synthesis. However, an intuitive understanding of it is vital to understanding the effects of and the cure for image defects due to undersampling.

The Fourier transform is one of the fundamental tools of modern science and engineering and it finds applications in both analogue and digital electronics, where information is represented (usually) as a continuous function of time and in work associated with computer imagery where the image $I(x, y)$ is represented as an intensity function of two spatial variables.

Calculating the Fourier transform of an image, $I(x, y)$, means that the image is represented as a weighted set of spatial frequencies (or weighted sinusoidally undulating surfaces) and this confers, as far as certain operations are concerned, particular advantages. The individual spatial frequencies are known as basis functions.

Any process that uses the Fourier domain will usually be made up of three main phases. The image is transformed into the Fourier domain. Some operation is performed on this representation of the image and it is then transformed back into its normal representation – known as the space domain. The transformations are called forward and reverse transforms. Fourier transforms are important, and this is reflected in the fact that the algorithms which perform the transformations are implemented in hardware in image-processing computers.

There is no information lost in transforming an image into the Fourier domain – the visual information in the image is just represented in a different way. For the non-mathematically minded it is, at first sight, a strange beast. One point in the Fourier domain representation of an image contains information about the entire image. The value of the point tells us how much of a spatial frequency is in the image.

We define the Fourier transform of an image $I(x, y)$:

$$F(u, v) = \frac{1}{2\pi} \iint I(x, y) e^{-j(ux+vy)} \mathrm{d}x\mathrm{d}y$$

and the reverse transform as:

$$I(x, y) = \frac{1}{2\pi} \iint F(u, v) e^{j(ux+vy)} \mathrm{d}u\mathrm{d}v$$

The Fourier transform is a complex quantity and can be expressed as a real and imaginary part:

$$F(u, v) = \mathrm{Real}(u, v) + j\,\mathrm{Imag}(u, v)$$

and we can represent $F(u, v)$ as two functions known as the amplitude and phase spectrum respectively:

$$|F(u, v)| = (\mathrm{Real}^2(u, v) + \mathrm{Imag}^2\,(u, v))^{1/2}$$

$$\varphi(u, v) = \tan^{-1}(\mathrm{Imag}(u, v)/\mathrm{Real}(u, v))$$

Now it is important to have an intuitive idea of the nature of the transform and, in particular, the physical meaning of a spatial frequency. We first consider the easier case of a function of a single variable $I(x)$. If we transform this into the Fourier domain then we have the transform $F(u)$. The amplitude spectrum, $|F(u)|$, specifies a set of sinusoids that, when added together, produce the original function $I(x)$ and the phase spectrum specifies the phase relationship of each sinusoid (the value of the sinusoid at $x = 0$). That is each point in $|F(u)|$ specifies the amplitude and frequency of a single sine wave component. Another way of putting it is to say that any function $I(x)$ decomposes into a set of sine wave coefficients. This situation is shown in Figure 14.16. The first part of the figure shows the amplitude spectrum of a single sinusoid which is just a single point (actually a pair of points symetrically disposed about the origin) in the Fourier domain. The second example shows a function that contains information – it could be a speech signal. This exhibits a spectrum that has extent in the Fourier domain. The spread from the minimum to the maximum frequency is called the bandwidth.

A 2D function $I(x, y)$ – an image function – decomposes into a set of spatial frequencies $|F(u, v)|$. A spatial frequency is a surface – a sinusoidal 'corrugation' whose frequency or rate of undulation is given by the distance of the point (u, v) from the origin:

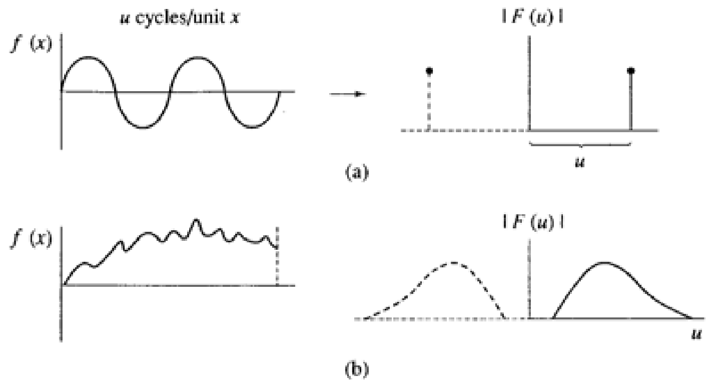


Figure 14.16
One-dimensional Fourier transform. (a) A sine wave maps into a single point. (b) A 'window' of an 'information wave' maps into a frequency spectrum.

$\sqrt{u^2 + v^2}$

and whose orientation – the angle the peaks and troughs of the corrugation make with the x axis is given by:

$\tan^{-1}(u/v)$

A single point $F(u, v)$ tells us how much of that spatial frequency is contained by the image. Figure 14.17 is a two-dimensional analogue of Figure 14.16. Here, a sinusoid has spatial extent and maps into a single point (again, actually a pair of points) in the Fourier domain. If we now consider an image $I(x, y)$, this maps into a two-dimensional frequency spectrum that is a function of the two variables u and v. Different categories of images exhibit different categories of Fourier transforms as we shall demonstrate shortly by example. However, most images have Fourier representations with the amplitude characteristic peaking at (0, 0) and decreasing with increasing spatial frequency. Images of natural scenes tend to exhibit Fourier representation that contain no coherent structures. Images of man-made scenes generally exhibit coherences in the Fourier domain reflecting the occurrence of coherent structures (roads, buildings etc.) in the original scene. Computer graphics images often have high energy in high spatial frequency components, reflecting the occurrence of detailed texture in the image.

A property of the Fourier representation that is of importance in image processing is that the circumference of a circle centred on the origin specifies a set of spatial frequencies of identical rate of undulation:

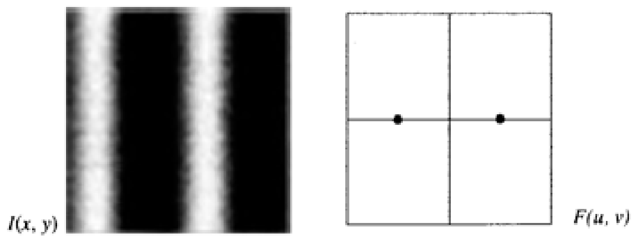


Figure 14.17
An image made up of a single spatial frequency and its Fourier transform.

$r = \sqrt{u^2 + v^2}$

having every possible orientation.

We will now look at the nature of the transform qualitatively by examining three different examples of amplitude spectra.

Figure 14.18(a)

Figure 14.18(a) is an image from nature. It produces a Fourier transform that exhibits virtually no coherences. Despite the fact that there is much line structure manifested in the edges of the leaves, the lines are at every possible orientation and no coherence is visible in the Fourier domain.

Figure 14.18(b)

Figure 14.18(b) is an image of a man-made scene. There is obvious structure in the Fourier domain that relates to the scene. First, there is the line structure that originates from the tramline discontinuity (top of the arches). Second, there is the discontinuity between the upper and lower arches that manifests as another line in the Fourier domain. There are coherences around the v axis that are due to the horizontal edges of the structure. Because the orientation of these lines varies

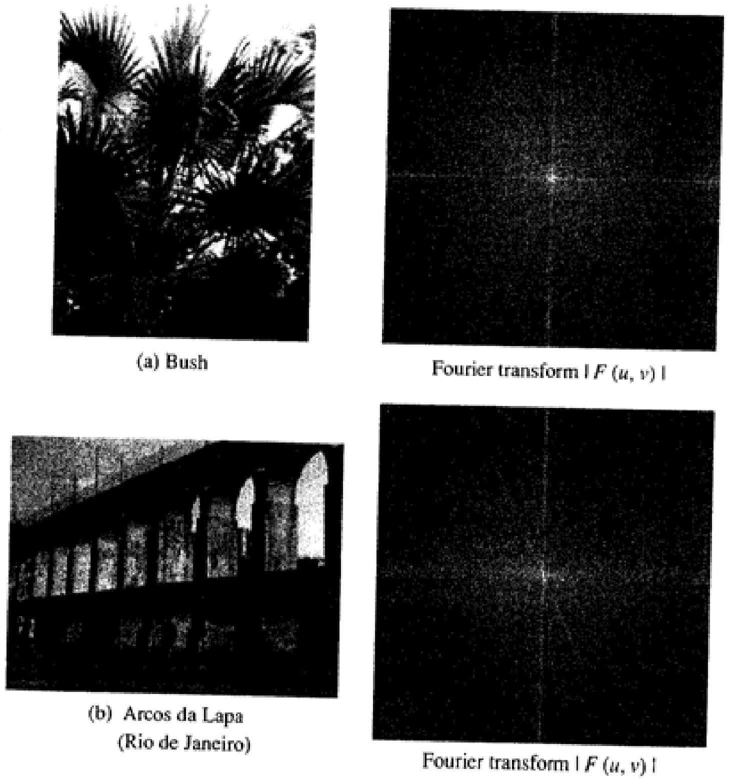
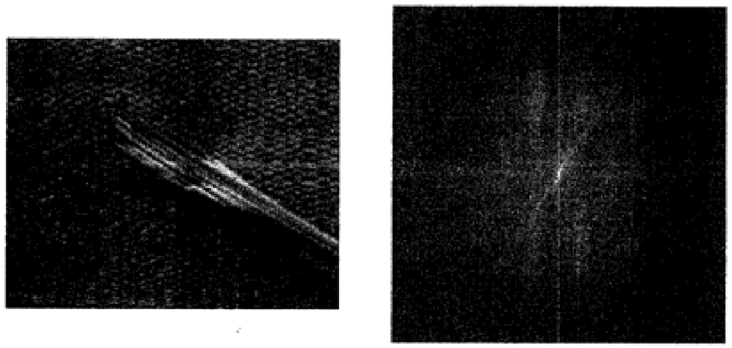

(a) Bush
Fourier transform $|F(u, v)|$
(b) Arcos da Lapa (Rio de Janeiro)
Fourier transform $|F(u, v)|$

Figure 14.18
Fourier transforms of natural and man-made scenes.

about the vertical, due to the camera perspective, they map into non-vertical lines in the Fourier domain. There is a vertical coherence in the Fourier domain that relates to scan lines in the data collection device and also due to horizontal discontinuities manifested by the long shadows. The remainder of the contributions in the Fourier domain originate from the natural components in the image such as the texture on the arch walls.

Figures 14.19(a) and 14.19(b)
Figures 14.19(a) and 14.19(b) are two man-made textures. The relationships between the coherences of the texture and the structures in the Fourier domain should be clear. In both cases the textures have been overlaid with a leaf, which manifests as a blurry 'off-vertical' line in the Fourier domain.

What can we conclude from these examples? A very important observation is that information that is 'spread' throughout the space domain separates out in the Fourier domain. In particular, we see that in the second example the coherences in the image structure are reflected in the Fourier domain as lines or spokes that pass through the origin. In the third example, the texture, produces components that are strictly localized in the Fourier domain at their predomi-


(a)

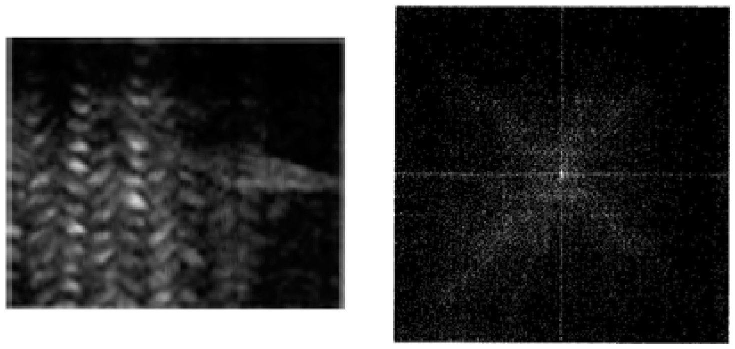

(b)

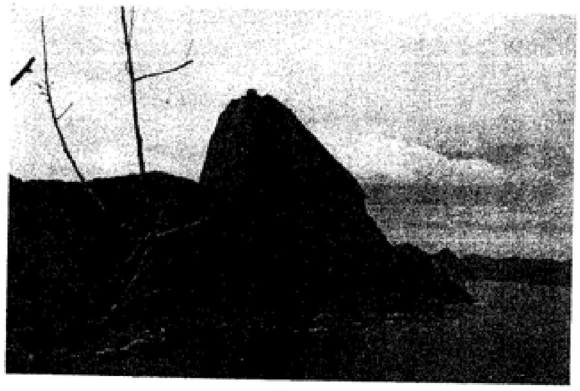
Figure 14.19
Fourier transforms of textures.

nant spatial frequencies. This property of the Fourier domain is probably the most commonly used and accounts for spatial filtering, where we may want to enhance some spatial frequencies and diminish others to effect particular changes to the image. It is also used in image compression where we encode or quantize the transform of the image, rather than the image itself. This gives us the opportunity to use less information to encode those components of the transform that we know have less 'importance'. This is a powerful approach and it happens that much less information can be used to encode certain parts of the transform without any significant fall in image quality. The original information in the image is reordered in the transform in a way that enables us to make easy judgements about its relative importance in the image domain.

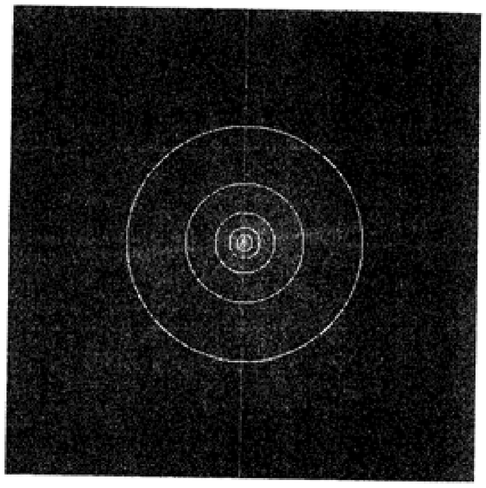
An extremely important property of the Fourier domain is demonstrated in Figure 14.20. This shows that most of the image power is concentrated in the low frequency components. The figure shows circles superimposed at different radii on the Fourier transform of the image shown in the figure. If we calculate the proportion of the total sum of $|F(u, v)|^2$ over the entire domain contained within each circle, then we find the relationship shown in Figure 14.20(b):

A property of the Fourier transform pair that is fundamental in image processing is known as the convolution theorem. This can be written as:

Figure 14.20
The percentage of image power enclosed in concentric circles of increasing radius.



Pão de Açúcar
(Rio de Janeiro)



Fourier transform
$|F(u, v)|$

(a)

Radius (pixels)	% image power
8	95
16	97
32	98
64	99.4
128	99.8

(b)

$$I(x, y) * h(x, y) = \Im^{-1}(F(u, v)H(u, v))$$

where:

* means convolution

In words: the convolution of the image function $I(x, y)$ with $h(x, y)$ in the space domain is equivalent to (or the inverse transform of) the multiplication of $F(u, v)$ and $H(u, v)$ in the Fourier domain, where:

$$I(x, y) = \Im^{-1}(F(u, v))$$

and:

$$h(x, y) = \Im^{-1}(H(u, v))$$

Analogously we have:

$$I(x, y)h(x, y) = \Im^{-1}(F(u, v) * H(u, v))$$

Both of these results are known as the convolution theorem. Convolution, and its special case – cross-correlation – is the operation that we perform on a computer graphics image when we filter a supersampled image down to screen resolution.

15 Colour and computer graphics

Introduction

This chapter is concerned with quantitative aspects of colour. Most treatment of colour in practical computer graphics has been qualitative. In setting up a scene database we tend to choose object colours more or less arbitrarily. However, certain applications are emerging in computer graphics where the accurate simulation of light–object interaction, in terms of colour, is required. Also, in the field of visualization, colour is used to impart numeric information and suitable numeric information to colour mappings must be considered in conjunction with knowledge of the subtle underlying psycho-physical mechanisms of the human colour vision system.

It is curious that an industry which has devoted major research effort to photo-realism has all but ignored a rigorous approach to colour. After all, frame stores whose pixels are capable of displaying any of 16 million colours have been commonplace for many years. We suspect three reasons for this:

(1) The dominance of the RGB or three equation approach in rendering methods such as Phong shading, ray tracing and radiosity, and the high cost of evaluating these models at more than three wavelengths.
(2) The rendering models themselves have obvious shortcomings that are visually far more serious than the unsubtle treatment of colour (spatial domain aliasing is visible, colour domain aliasing is generally invisible).
(3) The lack of demand from applications that require an accurate treatment of colour.

With some exceptions (see, for example, Hall and Greenberg (1983) and Hall (1989) little research into rendering with accurate treatment of colour has been carried out. There are, however, a growing number of applications that would benefit from accurate colour simulation, and a rendering method exists (the radiosity method) that is subtle enough in its treatment of light–object interaction to benefit from such an approach. Clearly this will be one of the major developments in CAAD (computer aided architectural design) in the future. A computer graphics visualization of an architectural design, either interior or exterior, is usually recognizable as such. We know that the image is not a photograph. This appears to be due predominantly to the lack of fine geometric detail. Modelling costs are high and approximations are made. In the radiosity method a coarse detail model is mandatory. So we first notice the inadequate geometry. However, 'second order' effects are no doubt just as important and such aspects as unrealistic shadows and light that 'doesn't look quite right' contribute to the immediate visible signature of a computer graphics image.

Another area where colour is of critical importance is volume rendering in ViSC (Chapter 13). Here colour is used to enable a viewer to perceive variations in data values in three space which may be extremely subtle. In this context it is important that the colours used communicate the information in an optimal way. This topic relies on perceptual colour models.

If we decide that accurate colour simulation is important, this throws up other problems apart from the cost implication in extending from three wavelengths to *n* wavelengths. These are:

(1) What descriptive colour system or model do we use to categorize colour? Clearly we could simply work with sampled functions of wavelength for the reflectivity characteristics of objects and the intensity of a light source. Although this may be convenient (and necessary) in the calculation domain, it will be useless to an architect, say, who wishes to specify a paint colour in a standard system using a colour label or a triple. What colour space should be used for the storage and the communication of images? It would be extremely impractical to store the results of *n* wavelength calculations.

(2) A major problem in using accurate colour exists in reproduction and viewing. Two colours specified in a standard system should look the same to a viewer. But this is only true if they are reproduced on carefully calibrated computer graphics monitors that are viewed under identical conditions. Although colour can be measured locally with precision, by using a colourimeter, such perceptual shifts due to, for example, contrast with surrounding colours, will always occur. This practical problem is not easy to overcome and unless it is dealt with it mitigates against the use of accurate colour simulation.

15.1 Colour sets in computer imagery

To deal with colour in computer imagery we need to quantify it in some way and this gives us the notion of a colour space or domain. This is a three-dimensional