Chapter 4

Solid Models and their Rendering

4.1 Solid Models and Raytracing

To draw a picture, such as a frame from an animation, or an architectural plan of a building, we must first start with some model within the computer. This model will contain all the information we require to make the picture, such as what objects are in the picture; what colours they are; what textures, reflectivity, and so on.

The kind of model strongly affects the rendering process in that certain kinds of models tend to be rendered in certain ways, but this connection is not absolute. For example, solid models fit well with ray tracing and this has led to quite sophisticated renderers. Some of the benefits of this have then been used to improve surface renderers. One of the big advantages of ray tracing is that we can model many phenomena that real light exhibits, such as reflection and refraction, and we can do this in a uniform manner.

We shall first consider solid models.

4.1.1 Constructive Solid Geometry

A solid model explicitly represents the entire body of an object, rather that just its surface. It has certain advantages, for example it directly represents how much material there is in an object; it can be sliced to reveal a genuine cross-section; there is no ambiguity about what is inside and what is outside the object. This sort of information is more difficult to get from surface models.

CSG models build on basic mathematical solid primitives, such as the sphere, cylinder, cone and cube. These shapes can all be represented by a mathematical formula, so they are examples of *implicit representations*. We don't construct the shape with a mesh but instead use implicit functions. Note that each formula can be used as an inequality, to determine solid from air: we are modelling a solid, not a surface.

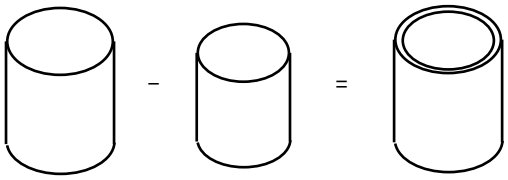
These primitive shapes are combined in various ways to make more interesting shapes, using operations, such as union, intersection, difference. It is the use of these operations with

representations of solids that gives the term, constructive solid geometry.

CSG uses a binary tree to represent the composite model: an internal node is a boolean operation; each branch has an associated transformation (scale, rotate, etc.); while a leaf node is a primitive object. The tree is hierarchical, so each sub-tree is a valid part of the model, though not necessarily connected and not necessarily as it will finally appear (because some parts of it may be cut away higher up the tree etc).

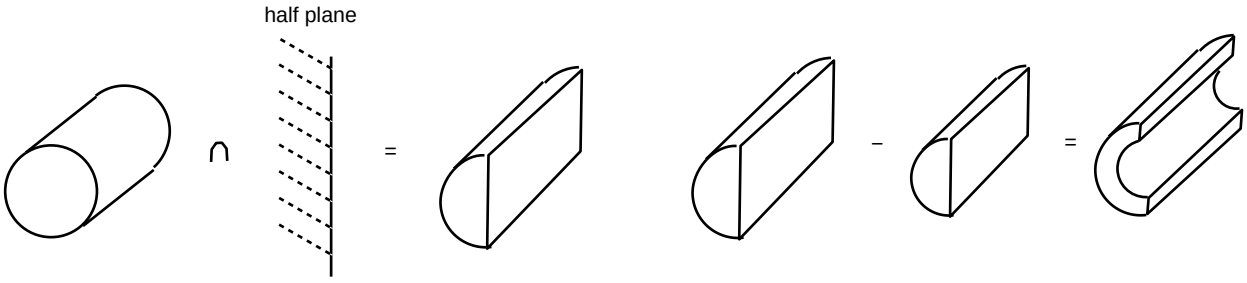
CSG is thus the boolean combination of a set of shapes.

Suppose we wish to model a cup. We can think of this as a solid cylinder with a slightly smaller cylinder cut from it

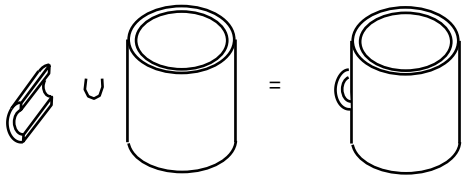


Already we see a difference from surface modelling: we have to have solid material and therefore a definite thickness. To create the interior space, we had to cut away solid, using a smaller cylinder.

Then we can add a handle that is composed of the difference of two half cylinders. A half cylinder can be made by subtracting a half plane (an infinite plane which is solid one side, air the other side) from a cylinder.



We then glue (union) the handle onto the cup.



In tree form, this looks like

```
BODY =   -                op
       /   \
     big  small           transforms
     /       \
    CYL      CYL          primitives
```

```
HALF =  INT                 op
       /   \
      ?     ?               transforms
     /       \
   CYL    HALFPLANE         primitives

HANDLE = -                  op
       /   \
      ?     ?               transforms
     /       \
   HALF     HALF

CUP =    U                  op
       /   \
      ?     ?               transforms
     /       \
   BODY    HANDLE
```

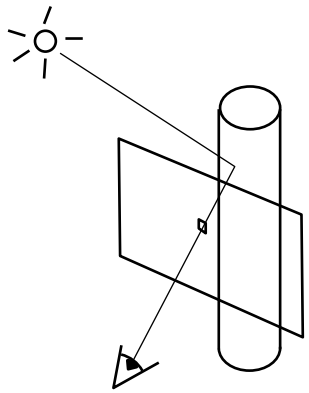
In this way we can build up complex objects. The only limitations are the original primitives.

The primitives can be represented by mathematical equations, e.g., a solid sphere is $(x - c_x)^2 + (y - c_y)^2 + (z - c_z)^2 <= r^2$. If we have to, we can derive polynomial equations to determine intersections. This may be done algebraically or numerically. However, we don't need to do this to visualise the shape: we can use ray tracing.

4.1.2 Ray Tracing

This technique is well suited to solid models as it requires knowledge of where lines intersect the objects in the model. Solid models are built from primitive shapes for which we have a mathematical formula, so calculating the intersection simply involves the simultaneous solution of that formula with the equation to the ray/line.

The basic mechanism is like the way light moves in straight lines in the real world. A line from the centre of projection CoP of the image is cast through through the current pixel and traced until it hits an object. This lets us determine the geometry (in particular, the normal of the surface) at that pixel. Then we can then trace from the object to the light source to determine the illumination.



This allows us to determine what colour the pixel should be. We repeat for all pixels on the screen. This is the essence of ray tracing. As we are about to see, it can be elaborated in

various consistent ways to produce a wide range of visual effects. For that reason it was and is hugely influential on computer graphics rendering.

A brief note on the basics. We traditionally draw an eye to represent the viewer looking at the scene. We often put this at the CoP. In fact the viewer does not strictly come into the ray-tracing calculation at all. The CoP is the optical centre of the lens system. For a simple lens, this means its physical centre and that is all we will need. We should also keep in mind that a real lens has the image on the opposite side to the objects being imaged. Furthermore, the image will be upside down in comparison to the object. In ray tracing we usually think of the rays passing from CoP, through the pixel into the scene, so the image plane (the screen) is on the same side as the objects. The image is the correct way up. We are of course tracing backwards in comparison with real light.

Sometimes it is better to go back to the optical model to help you understand what is going on. This is especially true of distributed ray-tracing (later), where the rays start on the optical axis (i.e. within the lens) but several will be off-centre. Each such bunch of rays has to pass through the pixel currently being calculated because the whole lens extent contributes towards the image at each pixel.

4.1.3 An Example of Ray Tracing

Here is a simple example of the kind of computations involved in ray tracing.

A ray is defined by a start point $\mathbf{s}$, and a direction $\mathbf{d}$, usually a unit vector:

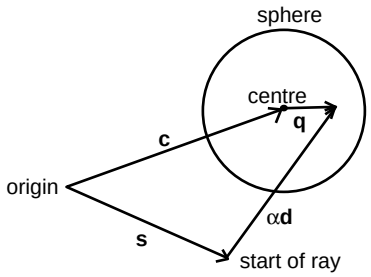
$$\mathbf{p} = \mathbf{s} + \alpha\mathbf{d},$$

with α the distance along the ray from $\mathbf{s}$. When we intersect this line with an object, the value of α gives us the distance to the object.

When a ray is reflected in a mirror or refracted when passing through glass, say, we can restart with new values of $\mathbf{s}$ and $\mathbf{d}$.

Intersection with a Sphere

We will intersect our line with a sphere, centre $\mathbf{c}$, radius r.



Suppose the intersection is at $\mathbf{q}$ on the surface of the sphere. Then

$$\mathbf{c} + \mathbf{q} = \mathbf{s} + \alpha\mathbf{d}.$$

Known are $\mathbf{c}$, r, $\mathbf{s}$ and $\mathbf{d}$. We want to find α and maybe $\mathbf{q}$.

We first rearrange this as:

$$\mathbf{q} = (\mathbf{s} - \mathbf{c}) + \alpha\mathbf{d}$$

Now we know that $\mathbf{d}$ is a unit vector. So, if we take the dot product of each side with itself, we get:

$$\mathbf{q} \circ \mathbf{q} = (\mathbf{s} - \mathbf{c}) \circ (\mathbf{s} - \mathbf{c}) + 2\alpha \mathbf{d} \circ (\mathbf{s} - \mathbf{c}) + \alpha^2,$$

But we also know that $|\mathbf{q}| = r$, the radius of the sphere. So

$$\alpha^2 + 2\mathbf{d} \circ (\mathbf{s} - \mathbf{c})\alpha + (\mathbf{s} - \mathbf{c}) \circ (\mathbf{s} - \mathbf{c}) - r^2 = 0,$$

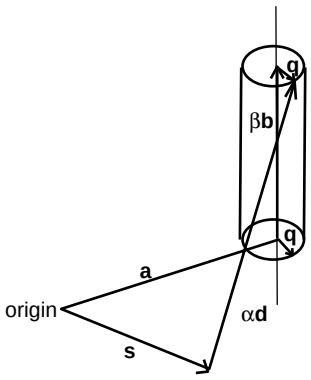
This is a scalar equation that we can solve for α, so we know how far away the intersection is: we will need this shortly. However, in general we get two (complex) roots α_1 and α_2. If both are real, it corresponds to the ray entering and leaving the sphere, with the smaller root being the entry, and the larger root the exit. Both complex means the ray misses the sphere. A single root means the ray is tangential to the sphere.

Notice that a quick test to determine if there is a real intersection is to compute the discriminant $b^2 - 4ac$ of the quadratic equation $ax^2 + bx + c$. If this is negative, both roots are imaginary; if zero, the roots are equal and real; if strictly positive, unequal and real. We have to compute this quantity to calculate the roots, so we can use it to avoid doing the full calculation when both intersections are imaginary.

The unit normal to the surface at the point of intersection is $\mathbf{q}/|\mathbf{q}|$, where $\mathbf{q}$ is given by $\mathbf{q} = \mathbf{s} + \alpha\mathbf{d} - \mathbf{c}$. This will be useful when we calculate the shading.

Intersection with a Cylinder

Only slightly harder is intersecting with a cylinder. We can define a cylinder by giving a ray (point plus direction) and a radius. The cylinder lies along the ray at the radius.



We have a cylinder base at $\mathbf{a}$, along direction $\mathbf{b}$, radius r. The ray is $\mathbf{s} + \alpha\mathbf{d}$, and we intersect at $\mathbf{q}$. Moreover, $|\mathbf{q}| = r$, $|\mathbf{b}| = |\mathbf{d}| = 1$, and $\mathbf{q} \circ \mathbf{b} = 0$.

Thus the basic equation is

$$\mathbf{a} + \beta\mathbf{b} + \mathbf{q} = \mathbf{s} + \alpha\mathbf{d}.$$

Dot with $\mathbf{b}$.

$$\mathbf{a} \circ \mathbf{b} + \beta + 0 = \mathbf{s} \circ \mathbf{b} + \alpha\mathbf{d} \circ \mathbf{b},$$

giving an equation for β in terms of α:

$$\beta = \mathbf{d} \circ \mathbf{b}\alpha + (\mathbf{s} - \mathbf{a}) \circ \mathbf{b}.$$

Take the original equation again, and rearrange

$$\beta\mathbf{b} + \mathbf{q} = \mathbf{s} - \mathbf{a} + \alpha\mathbf{d}.$$

Take the dot product of each side with itself

$$\beta^2 + 2 \times 0 + r^2 = |\mathbf{s} - \mathbf{a} + \alpha\mathbf{d}|^2,$$

or

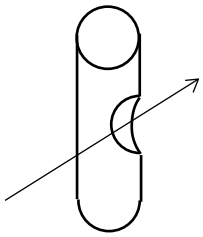
$$\beta^2 = |\mathbf{s} - \mathbf{a} + \alpha\mathbf{d}|^2 - r^2.$$

We can now substitute for β in this, giving a quadratic equation for α.

Note: there is a simple extension of this to a cone. In this case we have $|\mathbf{q}| = k\beta$, where k is a new parameter that determines the angle at the apex of the cone ($\mathbf{a}$ points at the apex).

4.1.4 Tracing the CSG Tree

In the general case of a complex model, how do we determine what part of the model the ray hit? For a single object in the world (e.g., a sphere), it is easy to find if the ray hits it or not. For a more realistic world model, many primitives will be combined using CSG and it is much harder to determine if a ray intersects. For example, we must take into account chunks of the model that are removed using a subtraction operator.



A ray within the bounds of the cylinder nevertheless misses the object because there is a chunk subtracted.

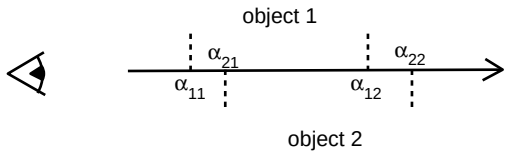
Consider the CSG tree. Suppose we intersect the ray with every primitive in the tree: these are at the leaves. This tells us all the possible intersections along the ray. So, we first notice that we only need consider the problem along the ray, not in the entire 3D space. Any single convex primitive, such as a sphere, will have upto two intersections along the ray, one where the ray enters the solid and one where it leaves. So we have a length of solid along the ray, one for each primitive which is hit by the ray.

However, we have taken no account of the way these primitives are combined to make the entire object. The rest of the tree, above the primitives, tells us how these primitives are combined; so it also tells us how the lengths of solid along the ray (which we have just identified) should be combined. We can perform CSG on these, just as we can on the entire primitives.

In practice, we traverse the tree depth first, intersecting the ray with each leaf primitive (in practice only those primitives that lie with the octree voxel or bounding volume, see below). These intersections give α distances along the ray.

Between each α pair from an object there is solid; outside of them there is space. When solids overlap along the ray we use the boolean operations in the CSG to decide what the result should be (solid or space).

For example, suppose we have two intersecting objects, the first has intercepts with the ray at α_{11} and α_{12}, while the second has α_{21} and α_{22}.



If the combining operation is

- union, then there is solid from α_{11} to α_{22}
- subtraction, there is solid from α_{11} to α_{21}
- intersection, there is solid from α_{21} to α_{12}
- symmetric difference, there is solid from α_{11} to α_{21} and α_{12} to α_{22}.

We move up the tree, modifying these spans as we go. When we reach the top (the root of the tree), we take the α of the nearest solid. This is the object we display.

Then we find the surface normal at the point of intersection. For a simple surface, this is easy. In practice we have to remember that this surface might have been revealed by taking one object (a cube say) and removing another object (a cylinder say). Then the normal might be due to the removed object but the basic colour will be that of the other object. This normal, together with the angle of the viewpoint and the angle of the light source are enough to apply *lighting models* to yield the colour of the pixel.

4.2 Lighting

Light is essential to viewing a model: with no light we can't see anything! The techniques to light a model vary from simple ambient diffuse light to more complex ideas such as spotlights or strip lights.

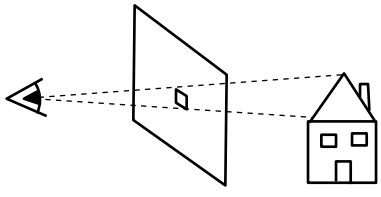
We require a model for the light. This will incorporate

- the position, intensity and colour of the light source or sources
- how incident and reflected specular light are related
- how incident and reflected diffuse light are related
- the contribution from ambient diffuse light

We also need to know the position and viewing direction of the viewer to complete the calculations.

We shall simplify and assume a single point light source. If there are more than one, the contribution from each can be computed and added. Similarly, each light will have a colour composed from red, green, and blue: we can compute each component individually and add the results.

We apply the computations for each pixel on the screen.



4.2.1 Surface Light Effects

There are several surface effects that can be individually computed, and then summed to form the result.

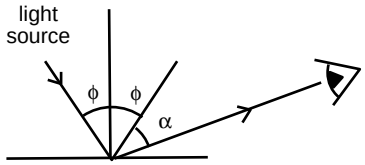
Specular Reflection

This is the cause of highlights on polished or shiny materials like glass and metal. It is

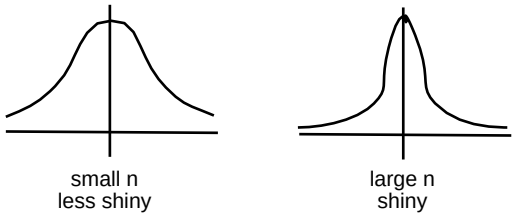
- highly directional
- the same colour as the incident light

You may like to think of it as being caused by the polish or varnish on the surface of an object, and not the object itself. For example, think of a black snooker ball: you can see highlights even though the ball is black. The perfect example of specular reflection is a mirror: you do not see the surface at all!

A simple approximation to the directionality is to use $\cos^n \alpha$, where α is the (positive) angle between view direction and ϕ the direction of perfect reflection.



The larger the n, the more rapidly the reflected light drops off as we move away from the angle ϕ, and the smaller the highlight patch. Thus large n correspond to highly shiny surfaces. Typical values are $n = 1$ for a slight sheen to $n = 20$ for polished metal.



Hence:

$$I_s = I_p K_s(\phi) \cos^n \alpha$$

is the *specular component* of the reflected light. Here I_p is the intensity of the source light, and K_s is the specular reflection coefficient: this is a function of ϕ to enable us to model material whose shininess varies with direction. Typically, $K_s = 1$ for $\phi = 90°$, and $K_s < 1$ otherwise. It is often set to a constant, values for which can be looked up in books for various actual materials.

There is just one specular coefficient (not one for each of red, green and blue) as all colours of light are reflected in the same way (otherwise specular colour would be affected by the colour of the surface).

Diffuse Reflection

Matt surfaces scatter light evenly in all directions. Thus the amount of light seen is independent of the viewer's direction. Its colour is that of the surface: indeed this is what we mean by the (perceived) colour of an object.

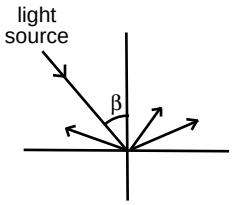
- direction independent

- colour that of the surface

A surface perpendicular to the lighting direction will scatter most light; one parallel will scatter no light. The is captured in *Lambert's cosine law*:

$$I_d = (I_p - I_s)K_d \cos\beta$$

is the *diffuse component* of reflected light. The values of I_p and I_s are as above, while K_d is a constant dependent on the material ($0 < K_d < 1$), and β is the angle between the surface normal and the incident light.



For a shiny surface $I_p - I_s$ is the amount of light that is not specularly reflected. It is as if the surface has a shiny upper layer and a matt lower layer, so that the non-specularly-reflected fraction of light is scattered by the matt layer. For a purely matt surface, $I_s = 0$.

We repeat this for each of the red, green and blue components of the light, with possibly different K_d for each.

Ambient Lighting

The basic illumination described here produces very strong contrasts since all surfaces facing away from the light source will be completely dark. It is common practice to assume that the environment scatters some light uniformly everywhere over all objects, and we model this by adding a fixed amount of diffuse (non-directional) ambient light $K_a I_a$, where I_a is the incident ambient light, and K_a the reflection coefficient for this light. Again there are red, green, and blue components.

Autophanous Facets

Some parts of the model can be self-luminous, e.g., street lights. These are not subject to the usual lighting computations, which are designed for light scatterers, not emitters. Usually, we just note them in the model, and render them assuming constant output.

Furthermore, we do not consider their effect on the rest of the environment (at this stage), as that can be quite complex.

Total Lighting

We should assume an inverse square law for light sources. In practice it is common that the light sources are well out of the scene, and effectively at infinity, so that the actual light distribution is uniform. Otherwise we would have to factor in the inverse square effect, which is quite costly.

A typical lighting model combines diffuse reflection (ambient and scattered) with some fraction of the specular component

$$I = K_a I_a + I_d + kI_s$$

The extra parameter gives us more flexibility in choosing the appearance of the object.

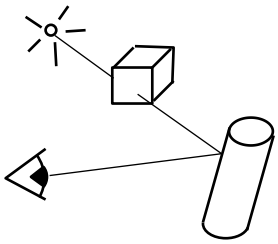
4.3 Optical Effects

The basic ray tracing idea can be extended to give a variety of optical effects that are quite difficult to get from a surface renderer.

4.3.1 Shadows

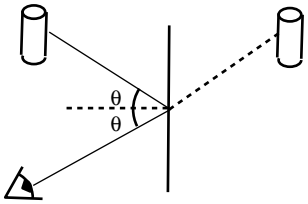
Shadows are easy with ray tracing. Once we have the intercept of the ray with the object, we can cast a ray from the intercept to the light source. If there is an object in the way, we are in shadow, and we can darken the relevant pixel.

This also works for multiple light sources: cast a ray back to each source, and sum the intensities to work out the total illumumination at that point.



4.3.2 Reflection

Mirrors are easy to simulate. Simply reflect the ray, and continue from the new start point and direction.



Angle of reflection is the angle of incidence.

If **i** is the unit vector for the incoming ray, and **n** is the surface normal, and θ is the angle of reflection, then the reflected ray is in the direction

$$\mathbf{r} = \mathbf{i} + 2\mathbf{n}cos\theta$$

and we recall that we can get the cosine from the scalar product of the incoming ray and the surface normal.

Usually we will want a specular component too: glass mirrors reflect from the front surface of the glass to give a highlight effect, as well as from the reflective layer behind the glass. Without this, it will look as though the mirror is not there at all and will be too perfect. For the same reason we tend to reduce the contribution a little on each reflection: a mirror is not 100% reflective. In other words we might assume that only 0.8 (say) of the R, G and B are reflected. This also fixes a particular problem with mirrors: parallel mirrors would reflect infinitely, so the computation would never end. If we choose a threshold for the contribution then, when it falls below the threshold we cease reflecting any further. Another, simple, fix is to count the number

of reflections a given ray has taken, and stop when they exceed a fixed number (typically no more than 10).

A tinted mirror could affect the colour, too, by reflecting R, G and B in different proportions.

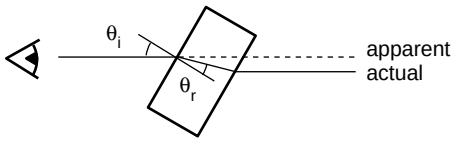
4.3.3 Transparency

Such as through glass. The simplest model is to ignore refraction, and just modify the colour components according to the material. This is OK for very thin pieces of glass (or other transparent material). There are various models of how colour might be affected: we can regard the material as a filter that selectively transmits different colours to different extents (thus modifying the red, green and blue components of the intensity of the ray).

4.3.4 Refractive Transparency

This is a more physically correct model of transparency, and looks good when the glass is more than vanishingly thin.

For refraction we use Snell's law to compute the new direction of the ray.

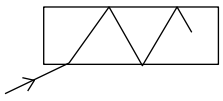


Snell's law: if θ_i is the angle to the normal of the incident ray, and θ_r is the angle of the refracted ray, then

$$\frac{\sin\theta_i}{\sin\theta_r} = \frac{\eta_r}{\eta_i},$$

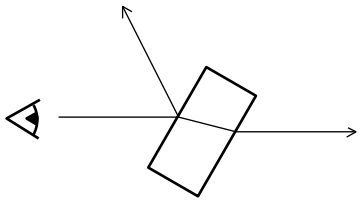
where the η are the *refractive indices* air and glass. The refractive index can vary with wavelength (colour), so the angle of refraction can vary with wavelength. In vacuum $\eta = 1$, for other materials $\eta > 1$ (so light is always bent towards the normal on moving into glass, and away when moving out).

What happens if the ray inside the glass hits the boundary at a large angle, that is, it is close to the plane of the glass? Then we might have $\sin\theta_r = \eta_i \sin\theta_i/\eta_r > 1$, as $\sin\theta_i$ is close to 1. This is not possible for a real angle θ_r, so Snell's law breaks down. Instead we get *total internal reflection*, where the ray is actually reflected back into the glass.



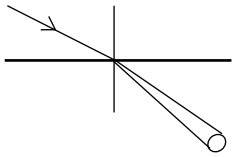
This is the effect that allows optical fibre to work. We treat the reflections just as for mirrors.

We can include the slight reflective effect of glass: the surface is shiny, so we get a slight reflection as well as transparency. This is done by generating a reflected ray as well as a refracted ray (with a small contribution from the reflected ray and the larger contribution from the refracted ray. Potentially, these split again if they hit more glass, etc., so we may get a large number of rays bouncing about than were spawned from the single original! However, we make the intensity contribution of each subsequent ray get smaller and smaller, so we can cut off as we did for mirrors.



4.3.5 Translucency

This is diffuse transmission, such as through frosted glass. Objects seem through such materials are blurred. For this, calculate the reflective lighting contribution (possibly including the colour of the material), and add it to the result of letting (a fraction of the) the ray continue.

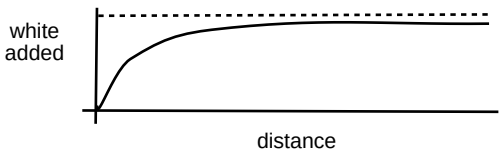


The blurring can be modelled by *specular refraction*: the ray splits into a cone of light as it refracts. We can ray trace this by using a sample set of rays lying in the cone, though now each incoming ray costs several outgoing rays.

4.3.6 Atmospherics

For outdoor scenes we may wish to allow for atmospheric scattering to give a sense of depth. This might be mist or smoke. This is done by exponentially desaturating (adding white) the colour with distance. At some point the colour becomes indistinguishable from white (the colour of the mist), and the object can no longer be seen. The rate of the exponential governs the depth of the mist.

Reference: Visual simulation of atmospheric haze (1987), P. J. Willis, Computer Graphics Forum, vol. 6, pp. 35-42.



4.4 Distributed Ray Tracing

So far we have thought in mainly in terms of a single ray from the CoP. In effect we have been assuming a pin-hole camera, one with an infinitely small aperture. The result is perfectly sharp images, one of the reasons that ray-tracing got a reputation for being "hyper-realistic". Real cameras have lenses with a definite aperture, often a centimetre or more. Rays which approach at an oblique angle can make the image locally fuzzy. Although they focus to the same image point, they did not start from the same world point. They take a longer optical path, so will not focus at the same position that more direct rays do. We can see this, for example, as depth of field: some parts of the image will be perfectly sharp, others will be out of focus. Read the supporting information on my web pages: Image Formation:Lenses.

Distributed ray-tracing is the term for a group of techniques which rely on firing several rays where classical ray-tracing would only fire one. We have met one example already: we suggested a cone of rays to help model translucency. In fact distributed ray-tracing handles other fuzzy effects, such as soft shadows. It is a powerful technique because it is a simple extension to the basic ray-tracer but produces a range of genuinely realistic effects.

To give one example, depth of field, imagine the optical lens with the image plane to its left and the object to its right. In standard ray tracing, we pass all rays through the lens centre. That is, we choose the pixel to be calculated (in the image plane) and draw a line from that pixel, through the centre of the lens and on towards the object. To get depth of field, we use a group of rays for each pixel, perhaps 16. They all start from the pixel but they diverge slightly, each ray going through a different position on the lens disc. We add up all their contributions to decide the pixel value.

Reference: Cook, Porter, Carpenter "Distributed Ray Tracing", ACM SIGGRAPH Conference Proceedings Vol 18 Number 3, pp. 137–145, July 1984

Let's examine this in more detail, thinking in terms of a conventional optical arrangement. That is we have the image (pixel) plane to the left, the lens (which we treat as infinitely thin but finite in diameter, not just as a point) in the centre and the 3D scene to the right. Choose one pixel in the image plane. Cast a ray through the centre of the lens and out into the scene: don't worry about which object it hits, if any, just think of the ray. Because this ray passes through the lens centre, it is not refracted so it follows a straight line from the pixel onwards.

On this ray, find the point which is at the focal point, measured from the lens centre towards the scene. Any object which was at the plane through that point, parallel to the lens, should appear sharply in focus. Objects nearer or further than the plane of sharpness will be progressively out of focus, depending on how far away from the focal plane they are. We can think of this plane as the image of the pixel array, produced by the lens. There will be a sample point on it corresponding to each pixel centre. Put another way, we can now forget about the pixel plane to the left of the lens and just work with this new plane to the right of the lens.

Suppose we now identify 16 (say) points on the surface of the lens. We fire one ray from each point through the focal point and out into the model. We calculate the contribution of each and then add them up. We place the result in the relevant pixel. What we have done (in effect) is model a group of rays leaving the pixel, being refracted by the lens and then being traced into the scene.

If we don't allow for this refraction, but do spread the rays over the lens surface, then we are treating the lens as plain glass. In other words, we have replaced our pin-hole camera (which produces images which are perfectly sharp everywhere) with a camera with a larger hole (which produces images which are unsharp everywhere). This gives us control over the sharpness (the bigger the aperture, the less sharp the image) without giving control of depth-related sharpness. It does illustrate however that the effective lens diameter matters: with refraction in place, the diameter is the parameter that determines how rapidly the image goes out of focus with depth. A complete solution to the problem thus offers a choice of lens focal length and a parameter to control the diameter over which the multiple rays are distributed.

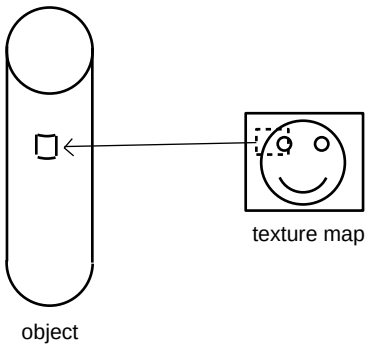
4.5 Surface Detail: Texture Maps

Real surfaces are not completely smooth and lacking in detail: they have a texture, which can range from being quite subtle to very strong. There are three common methods for producing a textured surface, *texture maps*, *bump maps*, and *displacement maps*.

A texture map regards surface texture as a modification of the surface reflection coefficient. The surface itself is still quite flat, but we can get the illusion of both texture *on* the surface

(such as woodgrain), and texturing *of* the surface (such as the tread of a tyre).

A texture is defined by providing a 2D array of colour values. This can be synthetic or a scanned real image, such as an area of a wood table top. We also define a regular mapping from this pattern onto the surface being textured. An easy mapping is a regular square tessellation.



In practice, for each pixel we determine what *area* of the texture map it corresponds to: it is vital not to point-sample texture maps because aliasing will be very visible. We then average the colour within that area, and use that as the surface colour. Then we apply lighting as usual. The sampling process thus requires the calculation of an inverse transform (from the 3D surface to the 2D texture map), so the method is sometimes called inverse-mapping.

This works well for a traditional renderer but is not fast enough for real-time applications such as games. For that, you need mip-mapping built-in to the (mesh) renderer of the PC graphics card. The quality is not as high but it is much faster. In practice, the inverse mapping is also too slow to do in real time but we don't need to. For mesh rendering we can work out where in 2D each vertex falls in the 2D texture space. Instead of doing this by inverse mapping at render time, we do it by direct association at modelling time: we know where the texture is supposed to be pinned to the surface mesh. Then we can store the 2D texture coordinates, usually called the (u, v) coordinates, with the 3D vertex in the mesh. At render time, we take the three pairs of coordinates for each triangle and use the linear interpolation between them (already needed for Gouraud shading) to index the texture map.

4.5.1 Bump Maps

This is a variant on texture maps that use the texture pattern to hold normal offsets, rather than colours. These are added to the local normal on the surface before the lighting model is applied. This gives the appearance of a non-flat surface, such as a tyre tread. This is a bit of a fudge, as we are not actually changing the flatness of the surface, merely drawing it as if it were non-flat. As long as the viewer is not too close, and you cannot see the surface in silhouette the effect is good.

4.5.2 Displacement Maps

This is a technique much better suited to surface modelling, but we mention it here for completeness. A more costly approach to adding surface detail is really to displace the surface: use

the texture map to move points on the surface. This is quite tricky to do for a solid model defined by equations. The result is a fully correct texture, but at considerable cost.

4.5.3 All the Above

We can combine all the above effects (texture, distributed rays etc etc) in a single picture, and the results can be quite impressive. The cost is in the multiple rays we have to trace (remember this is for just one pixel), and we can get into *recursive ray tracing*, where one ray generates several as it splits and reflects. Even so, it is a consistent approach: if you can trace one ray, you can trace lots, and almost all of the effects are produced by tracing more rays.

Ray tracing is especially good for specular effects, including shiny surface (poorly handled by surface renderers because they approximate surface normals), and reflections in general (not handled in surface rendering). These are the features which gave even early ray-traced pictures that "real" look than was not there in earlier 3D computer graphics.

4.6 Speeding up Ray Tracing

Ray tracing is inherently very slow. It requires much floating point computation for each pixel, and a traversal of the CSG tree for each pixel. However, because each pixel is computed independently, there is scope for ray tracing to be done in parallel: in practice the large amounts of data that needs to be interrogated makes parallelism less than 100% successful. We shall now consider some more basic ways of improving speed, on a single processor.

4.6.1 Using Object Coordinates

A scene will typically have many instances of a particular primitive. Each will have been separately transformed to fit into the model. This means the defining equation of the primitive will be quite complicated when expressed in model coordinates. Rather than try to compute intersections in model space, it is simpler to transform the ray back into object space, and solve the intersection there. For example, a general sphere might be given by $(x - c_x)^2 + (y - c_y)^2 + (z - c_z)^2 = r^2$ in model space, but represented as $x^2 + y^2 + z^2 = 1$ in object space. Solving an intersection of a straight line with the latter is numerically much simpler.

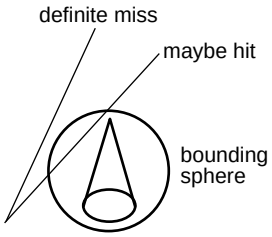
Once we have found the intersection in object coordinates, we transform it back to model coordinates.

4.6.2 Volumes of Interest

This is much like clipping in surface models. We want to find a way of rapidly rejecting primitives which are well away from the current ray, e.g., objects that are behind the point of view. We would then avoid computing putative intersections with surfaces that cannot possibly be involved.

A simple way of doing this is to enclose each primitive with a imaginary sphere (the *bounding* sphere), or box (the bounding box). We first test if the ray intersects the sphere. This is a simple and fast test. If the ray misses the sphere we know it cannot hit the object inside, no matter how complicated it is. This is a fast *culling* operation.

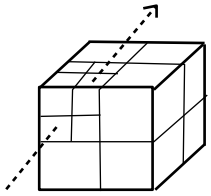
If the ray hits the sphere, we proceed as before, but we hope this happens infrequently.



4.6.3 Octrees

This is a recursive subdivision method of producing volumes of interest. We imagine that the entire model is held in an enclosing cube. We recursively subdivide the cube into eight smaller cubes (*voxels*), resulting in an *octree* structure. We keep subdividing until the contents of each voxel is "simple", e.g., entirely solid, or entirely air, or we hit some limit on voxel size. In particular this allow us to identify empty space, and home in on the surface of the solid elements.

The cost is the storage of the octree, and the tracing of a ray through it.



Ray tracing now consists of walking a ray through the octree voxels until we hit a non-empty voxel. In this way we avoid solving for intersections with objects that lie along the ray but will not be seen. We then solve for the intersection of the ray with the object(s) in this non-empty voxel, just as we do in simple ray-tracing (i.e. with the binary CSG tree to represent the model: the octree does not replace the binary tree).

A practical implementation of this is to list within each voxel a reference to the primitives which lie there. Then we use the ray to identify the relevant voxels and hence the relevant primitives. Now we can walk the CSG tree but only those parts of it which include the identified primitives.

4.7 Variant Techniques

[NOT EXAMINED]
Ray tracing only calculates local lighting effects: there is no attempt to calculate the effect of the light from one surface on another. Kajiya introduced a more rigorous approach to lighting, based on the Bidirectional Reflection Function (BRDF). To cope with this, he modified ray tracing to become *path tracing*. Here, a ray is sent into the scene. If it hits something, a contribution is calculated and a new ray is traced. This new ray starts from the object hit but is sent in a random direction away from the object. Contributions are gathered recursively (to some depth or until no object is hit) and multiplied together. The result is that the initial ray returns the colour on the object due to all the other objects along the path.

This method can be run the other way (from the light sources to the camera), when it is called *light tracing* and both can be used together (*bidirectional path tracing*).

Rays are infinitely thin. Heckbert and Hanraham introduced *beam tracing*. The beam is a pyramid shape "ray". They cast one pyramid into the entire viewing frustum. This is done

polyogon by polygon, from nearest to furthest. Every polygon hit must be visible and its shape is used to clip the beam shape (so that hidden polygons are not hit by the proceeding beam). Reflective (or refractive) polygons cause new beams to start.

The BRDF is also used in Jensen's *photon mapping*, which greatly extends the range of optical effects possible. Roughly, it uses a first stage of processing the simulate the effect of light on the scene, by sending "photons" from the light sources. The resulting photon map is then used by a second stage ray tracer, to reveal the view from the camera. The same map can be used for different camera positions.