Practical Ray Tracing

Geoff Wyvill
University of Otago

Part 2: Professional Ray Tracing

There is a logical gap between the two halves of this course. In the first half ray tracing is introduced for beginners. This part is really directed to those who have written a ray tracer and are familiar with some of the practical problems. If you have taken Part One as a beginner, there will be things here you do not completely understand. But it is not possible to convey in a short course the kind of information that one accumulates from practical experience. What you miss now will become clearer when you have that experience.

This part of the course is a ragged collection of bits of information that collectively protect you from many of the mistakes that make ray tracers buggy or inefficient. This collection is necessarily incomplete and the choice of material is largely based on my experience of adapting a ray tracer written as a research exercise to be successful in a competitive commercial environment. The section, equation and figure numbers follow on from Part One but these notes can also be read independently.

10. Polygons again

The use of polygons as the primary modelling tool is so firmly established that we are going to have to handle such models for a long time to come. Extensive polygon databases already describe engineering models and they are used in popular animation systems. Even systems that use NURBS or trimmed Bézier patches as their primary representation seem to convert these to the ubiquitous polygon mesh for rendering. So here are a few notes and problems specifically on polygons.

10.1 Do the inside/outside test first

In Part 1, we showed that point **p** is inside the triangle, (**a**, **b**, **c**) if the three expressions:
(**b** - **a**) × (**p** - **a**).**n**, (**c** - **b**) × (**p** - **b**).**n** , (**a** - **c**) × (**p** - **c**).**n** have the same sign. But there is another way to express this. Suppose we project the triangle (**a**, **b**, **c**) onto a plane perpendicular to the ray, **u** + **v**t (Fig. 18). In that projection, the vector (**p** - **a**) is identical to (**u** - **a**) because all points along the ray, occupy the same point in the projection plane. To project a point onto this plane we remove its component in the direction of the ray, **v**. That is:

$$\text{Projection}(\mathbf{k}, \mathbf{v}) = \mathbf{k} - (\mathbf{v}.\mathbf{k})\,\mathbf{v} / \mathbf{v}^2 \qquad (26)$$

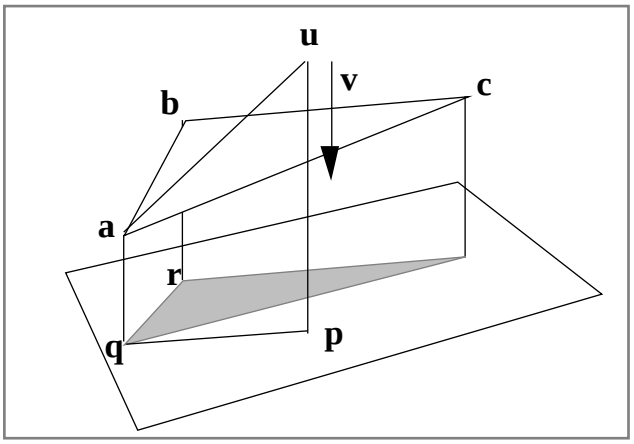

Fig. 18. Projection of triangle onto plane
$(\mathbf{r} - \mathbf{q}) \times (\mathbf{p} - \mathbf{q}) = (\mathbf{b} - \mathbf{a}) \times (\mathbf{u} - \mathbf{a}).\mathbf{v}\ \mathbf{v}$

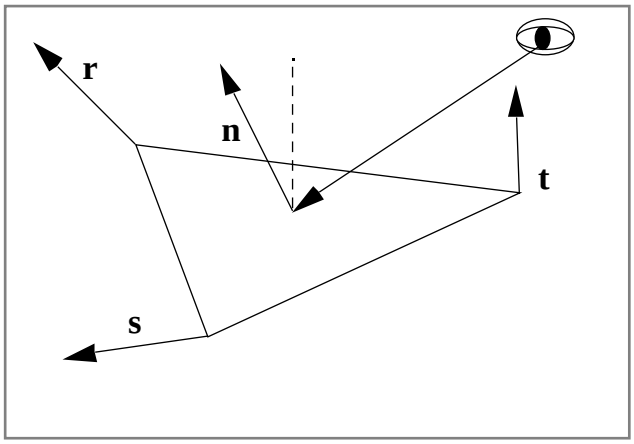

Fig. 19. Instead of the true normal (dotted) **n** is a weighted average of **r**, **s**, **t**

The properties of vectors are, of course, independent of the coordinate system. Given any three perpendicular unit vectors, **i**, **j**, **k**, any vector can be separated into three components along the **i**, **j**, **k**, axes:

$$\mathbf{p} = \mathbf{i}\,(\mathbf{p}.\mathbf{i}) + \mathbf{j}\,(\mathbf{p}.\mathbf{j}) + \mathbf{k}\,(\mathbf{p}.\mathbf{k}) \qquad (27)$$

The cross product

$$\mathbf{p} \times \mathbf{q} = \mathbf{i}\,((\mathbf{p}.\mathbf{j})(\mathbf{q}.\mathbf{k}) - (\mathbf{q}.\mathbf{j})(\mathbf{p}.\mathbf{k})) + \mathbf{j}\,((\mathbf{p}.\mathbf{k})(\mathbf{q}.\mathbf{i}) - (\mathbf{q}.\mathbf{k})(\mathbf{p}.\mathbf{i})) + \mathbf{k}\,((\mathbf{p}.\mathbf{i})(\mathbf{q}.\mathbf{j}) - (\mathbf{q}.\mathbf{i})(\mathbf{p}.\mathbf{j})) \qquad (28)$$

If we set the **j** components of **p** and **q** to zero the first and last terms in (28) will also be zero so the cross product of the projections:

$$\text{Projection}(\mathbf{p}, \mathbf{j}) \times \text{Projection}(\mathbf{q}, \mathbf{j}) = \mathbf{j}\,((\mathbf{p}.\mathbf{k})(\mathbf{q}.\mathbf{i}) - (\mathbf{q}.\mathbf{k})(\mathbf{p}.\mathbf{i})) \qquad (29)$$

which is simply the **j** component of $\mathbf{p} \times \mathbf{q} = \mathbf{j}\,(\mathbf{p} \times \mathbf{q}) . \mathbf{j}$

In Fig. 18 we are looking along the ray $\mathbf{u} + \mathbf{v}t$ so the plane of the diagram is perpendicular to **v**. The product $(\mathbf{u} - \mathbf{a}) \times (\mathbf{b} - \mathbf{a}) . \mathbf{v}$ will be positive iff the point **u** is on the left of the directed line $\mathbf{b} - \mathbf{a}$ in the projection. This means we can replace (11) by the more general form:

$$(\mathbf{b} - \mathbf{a}) \times (\mathbf{u} - \mathbf{a}).\mathbf{v},\ (\mathbf{c} - \mathbf{b}) \times (\mathbf{u} - \mathbf{b}).\mathbf{v}\ ,\ (\mathbf{a} - \mathbf{c}) \times (\mathbf{u} - \mathbf{c}).\mathbf{v} \qquad (30)$$

and the ray will pass through the triangle a, b, c iff these three products have the same sign. The neat thing about this test is that it can be done *before* finding the ray/plane intersection. The intersection is necessary only for those polygons that pass the test. On the other hand, if we perform the intersection calculation first we must do it for all triangles. Except for the

special case of a ray parallel to a plane, the intersection test always produces a result so we do just as many inside/outside tests.

10.2 Normals and smoothing

Very often, simple curved surfaces are approximated by a large mesh of polygons, usually triangles. An approximate model of a sphere needs at least 1000 triangles before it looks anything like round and a simple model of a human face needs over 15000. One popular way to improve the appearance of a polygon model is to smooth the shading across each facet. To do this, we usually store a false surface normal at each vertex. These vertex normals are then interpolated across the polygon to create the illusion of a smoothly curving surface. When a ray intersects the polygon the illumination is calculated using this interpolated normal, Fig. 19. This works quite well most of the time but it leads to some odd effects, particularly where the light meets the surface at a low angle.

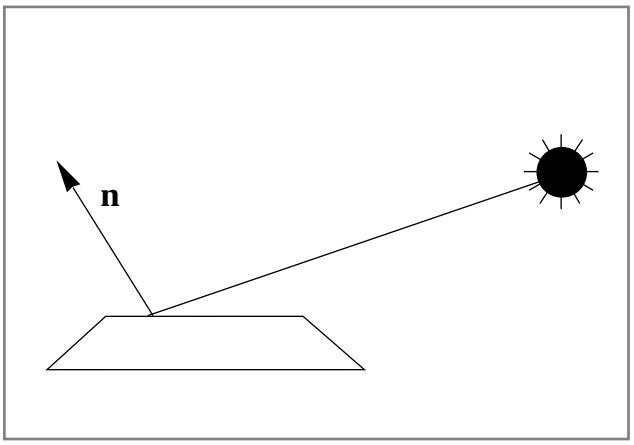

Fig. 20. Smoothed normal points away from light.

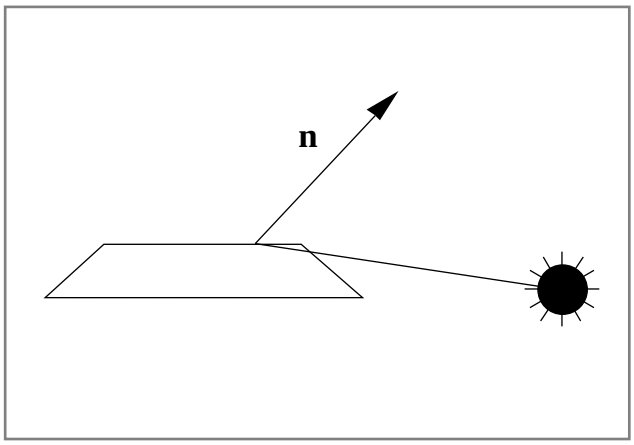

Fig. 21. Surface is in shadow even though normal suggests otherwise.

In Fig. 20, the smoothed normal actually points away from the light source. This means that careless application of Lambertian or Phong shading can produce negative illumination causing colours to overflow. In Fig. 21, the surface shadows itself. The point of interest is not illuminated although the curved surface suggested by the normal would be. Some systems attempt to fix these problems by using the true normal when the interpolated one gives trouble. But there is no proper fix. The surface shape and normals are inconsistent and will give problems at some angles. Reflected rays can also be created that enter the surface they are reflected from and refracted rays may not lie inside the surface that they should have passed through. The commonest solution is to adjust the position of lights or objects to eliminate the ‘bug’.

11. Constructive Solid Geometry

The best way to deal with the problems associated with polygon models is simply not to use polygons. One of the most attractive alternatives is Constructive Solid Geometry (CSG). The basic idea of CSG is that every 3D object consists of a set of points in space and we define more complex objects in terms of simpler ones by adding and subtracting sets from other sets.

We can think of addition as equivalent to gluing or welding objects together and subtraction as equivalent to cutting. This is illustrated in Fig. 22.

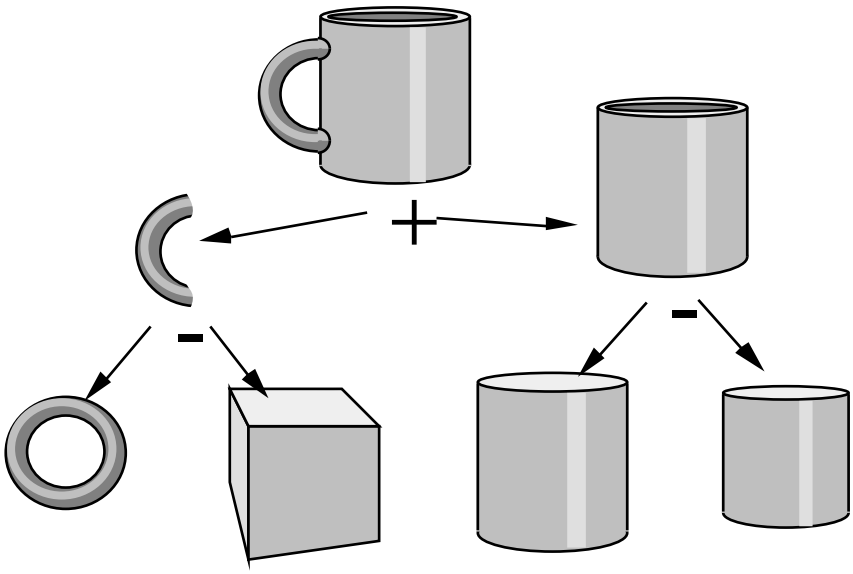

Fig. 22. CSG construction of a coffee mug.

The data structure used to represent this kind of model is a directed graph. As with any hierarchical structure for 3D models, the nodes of the graph contain matrices of transformation. They also contain information to say how the sub-graphs are to be combined. In Fig. 22, each node specifies addition or subtraction but other set operations like intersection can also be used.

11.1 Intersection with CSG models

Again, in principle, we have to find the ray intersections with every primitive in the CSG model. Again they are ordered according to their distance from the eye. How can we use the CSG tree to determine the correct intersection point?

This is illustrated in Fig. 23. The ray from **p** to **q** finds intersections with the various primitives at **a**, **b**, **c**, **d**, **e**. The CSG graph is shown on the right. Which is the correct intersection point? Points **a** and **b** are inside the subtracted plane primitive and **c** is inside the smaller, subtracted sphere. Point **d** is where the ray enters the solid and **e** where it leaves. The nearest valid intersection is, therefore, **d**. To determine this, we must traverse the CSG structure applying a set of logical rules at each node. The rules are easy to understand. They are applied recursively so if they work for primitives and they handle the joining of sub-graphs correctly, they will work in all cases.

CSG is not nearly as popular in ray tracers as you might expect. This is probably because a lot of popular implementations are not very efficient. But with correct design, a CSG ray tracer performs at least as well as its polygon based cousin and uses far fewer primitive objects to do the same job. All of the animation examples shown in this course were made using “Katachi”, a modelling and ray tracing system that primarily uses CSG to define its models.

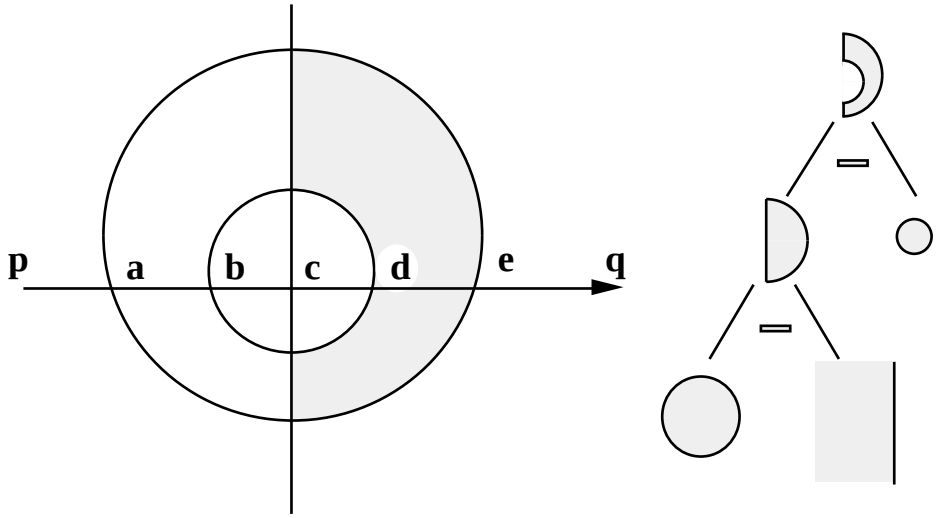

Fig. 23. Unraveling the CSG structure.

PLUS Node

Left branch	Right branch: IN	OUT	BORDER
IN	IN	IN	IN
OUT	IN	OUT	BORDER
BORDER	IN	BORDER	

MINUS Node

Left branch	Right branch: IN	OUT	BORDER
IN	OUT	IN	BORDER
OUT	OUT	OUT	OUT
BORDER	OUT	BORDER	

Fig. 24. Combining rules.

The combining rules expressed in Fig. 24, work in a system of three value logic. Each primitive has associated with it an inside/outside test which tells us whether a point is inside its volume or not. The special value, border, is not returned by any of these routines. It is given only to the particular instance of the primitive with which a particular intersection test has been done.

The algorithm is simple. If the current node represents a primitive, then apply the primitive test (below). Otherwise get the values from the sub-nodes and combine according to the table. Here we deal only with “+” and “-” nodes. Similar rules can be made for intersection nodes.

11.2. Primitive inside/outside tests

The primitive objects in a CSG system are all represented by simple, algebraic inequalities. The sphere, for example is represented by:

$$x^2 + y^2 + z^2 \leq 1 \qquad (31)$$

If (31) is satisfied for a point (x, y, z) then that point lies inside the sphere. Thus at the leaves of the CSG tree, the inside/outside test is just a matter of inserting the point to be tested into the equation and testing a sign. No equation solving is required.

12. Efficiency of primitive intersection routines

The simplest way to do intersection tests on transformed objects is to use the inverse transformation on the ray. But it is also possible to transform the equation of a primitive object into the world space. I have frequently heard the claim that this provides the most efficient way to find intersections and this claim is interesting to examine.

12.1. Transformation of quadratic equations

In what follows, $\mathbf{u}'$ means the transpose of $\mathbf{u}$. This will be a row vector, (x, y, z, 1). M´ means the transpose of the matrix M. We start by observing that the most general *quadratic* function of the variables, x, y, z has the form,

$$ax^2 + by^2 + cz^2 + dxy + eyz + fxz + gx + hy + iz + k = 0 \qquad (32)$$

We can write this in the form:

$$\mathbf{u}' M \mathbf{u} = 0 \qquad (33)$$

$$\text{where } M = \begin{pmatrix} a & d/2 & f/2 & g/2 \\ d/2 & b & e/2 & h/2 \\ f/2 & e/2 & c & i/2 \\ g/2 & h/2 & i/2 & k \end{pmatrix} \qquad (34)$$

$$\text{For example, if we choose: } M = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -r^2 \end{pmatrix} \qquad (35)$$

Equation (32) becomes the equation of a sphere of radius, r:

$$x^2 + y^2 + z^2 - r^2 = 0 \qquad (36)$$

Because equation (33) is the most general quadratic form, it describes a family of surfaces that includes, spheres, ellipsoids, cylinders cones and hyperboloids. Such surfaces are known as *quadrics*. Another way to write (35) is

$$\mathbf{u}' M \mathbf{u} = r^2 \qquad (37)$$

where

$$M = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \tag{38}$$

This form suggests that we might rewrite (37):

$$\mathbf{u}' M \mathbf{u} = q \tag{39}$$

where -q is the constant term taken out of the matrix. The value of q is analogous to r^2 for a sphere but can be calculated for any quadric.

In (39), the vector, **u**, represents a point on the surface of one of these quadric shapes. Let us define another vector,

$$\mathbf{v} = T\mathbf{u} \tag{40}$$

where T is a 4x4 transformation matrix. Mutiplying both sides of (40) by the inverse of T, T^{-1},

$$T^{-1}\mathbf{v} = \mathbf{u} \tag{41}$$

and taking the transpose of both sides,

$$\mathbf{v}'T^{-1\prime} = \mathbf{u}' \tag{42}$$

This gives us an expression for **u** and **u**′ which we may substitute in (39):

$$\mathbf{v}'T^{-1\prime} M T^{-1}\mathbf{v} = q \tag{43}$$

This equation holds for any point, **v**, that is a transformation of a corresponding point **u** on the original quadric. In other words, this is the equation of the transformed surface. Not only that, the matrix,

$$N = T^{-1\prime} M T^{-1} \tag{44}$$

is just another 4 × 4 matrix. Equation (43) has the same form as (39). We have established a procedure for applying a general transformation matrix, T, to an implicit equation. The result is another equation that describes the transformed shape.

12.2 In which space to intersect

If we substitute the point **u** + **v**t into (32) and collect the terms, we get a quadratic equation in t. The actual calculation of the coefficients of this quadratic needs 49 multiplications and 20 additions. Substitution of **u** + **v**t into (36) needs ten multiplications and six additions. But first we must convert **u** and **v** into the original, 'primitive' space in which (36) defines the sphere. Conversion of **u** takes nine additions and nine multiplications. Conversion of **v** takes six plus

nine because the shift component is not needed for a direction vector. For practical purposes on modern computers, floating point additions take much the same time as multiplications, so the cost of setting up the quadratic equation is 69 flops in world space and 49 in primitive space.

Solving the equation and finding the intersection point costs the same in each space. The surface normal is found in world space from the derivatives of (32). The x-component, for example is:

$$(2\,a\,v_x + d\,v_y + f\,v_z)\,t + 2\,a\,u_x + d\,u_y + f\,u_z + g \qquad (45)$$

If we precalculate 2a, this needs seven multiplications and six additions. By contrast, the normal in primitive space is the same as the hit point, $\mathbf{u} + \mathbf{v}t$, but it requires conversion back into world space and that requires another fifteen flops. Solving the equation takes seven flops for the first root plus three for the second plus one square root. In summary:

Costs (flops)	World Space	Primitive Space
u conversion	—	18
v conversion	—	15
Setting up Equation	69	16
Find discriminant	4	4
Total setup	73	53
Solving Equation	8	8
Surface normal	39	15
Total	120	76

It actually is *less* work to transform the ray into the primitive space than to use the more sophisticated solution. Not only that, it is less work whether a solution is found or not. How can this be? Well, equation (32) contains a lot more information. It could potentially describe a different kind of quadratic surface, a hyperboloid for example. But when we transform the ray vector into primitive space, we convert the knowledge that the object is a transformed sphere into a lot of zero coefficients in the equation.

By contrast, it is slightly faster to deal with planes in world space. I have no information on higher order surfaces like the torus.

13. Rounding errors

Every area of computing is affected by rounding errors although there are remarkably few publications that deal with this area. Ray tracing is no exception and in some ways the algorithm is particularly vulnerable. We will assume that the numbers that represent our coordinate system are fixed point fractions. Then continuous space is represented by a finite grid of points and any position can only be represented by its nearest grid point. The use of

floating point numbers does not change this. It merely complicates it because the grid no longer has a uniform spacing.

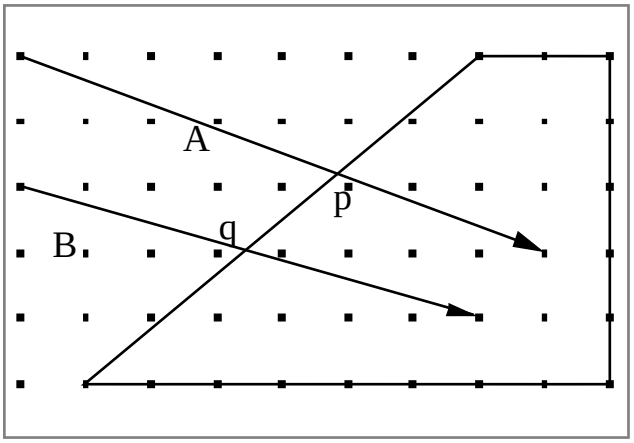

Fig. 25. Intersections with rounding errors.

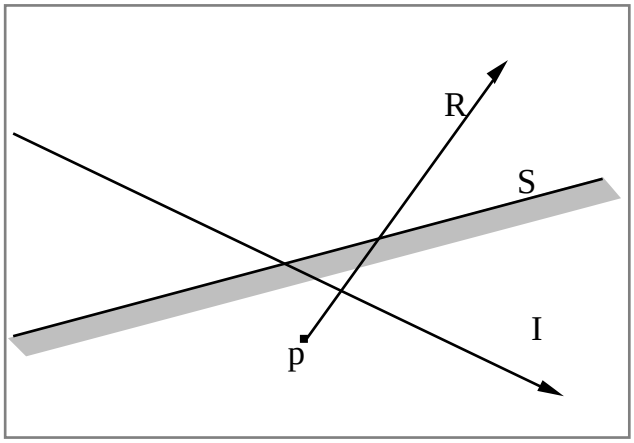

Fig. 26. False shadowing.

Because of the rounding errors strange things occur. Figure 25 shows two rays, A and B, directed at a plane surface. Representable points are shown by dots. If the intersection routines find the nearest representable point to the ideal intersection, then the intersection, p, with A lies inside the surface while the intersection, q, with B lies outside the surface. This is the usual case and our algorithms must expect it. Notice that these errors are present even though the end points of the rays, and the vertices of the object fall exactly on representable points.

In Fig. 26, an intersection point, p, lies inside the object S. A light seeking ray, R, from the intersection point also intersects S. The result is that the surface can appear falsely to be in shadow. This ‘error’ appears apparently at random points producing a speckled effect on the surface. The usual fix for this is to shift the point, p by an amount large compared with the rounding errors but nonetheless small compared with the dimensions of the objects. Ideally we shift p outside the surface to generate reflected rays or shadow rays but inside the surface for refracted rays. Some authors displace p back or forth along the ray, some along the surface normal. There are arguments for each approach but there are also odd exceptional cases that produce a false result whatever you choose. The only really robust solution is to keep a record of which surfaces the ray has crossed and use a strictly logical test to disallow the false intersections. This is described in detail in [Wyvill 1992].

14. Sampling and Filtering, Jittering

The root of all aliasing problems in computer generated imagery is that an image built from pixels can only represent a level of detail commensurate with the size and number of pixels. This is best understood in terms of sampling theory and the well-known Shannon/Nyquist sampling theorem tells us that an image can be reconstructed unambiguously from samples only if it contains no frequency component higher than half the sample frequency. This means that the ideal image that our pixels represent must be sufficiently blurred that it contains no detail

smaller than two-pixels in width. The best way, therefore, to avoid aliasing is to blur the image before turning it into pixels.

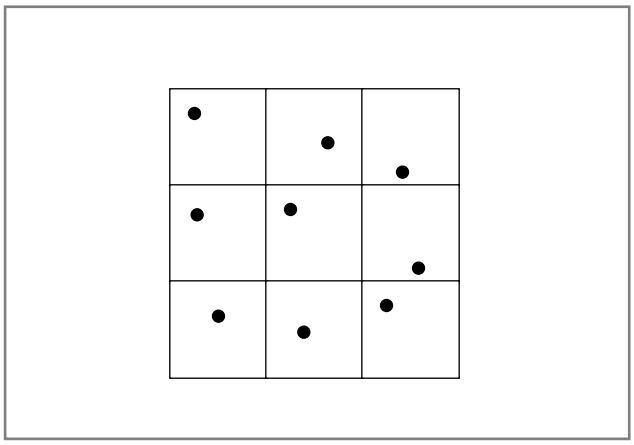

Fig. 27 Jittering pattern

Unfortunately, when an image is constructed by ray tracing, there is no such ideal image to blur. The only information we have is from samples represented by our primary rays. If we know something about the image in advance, we can use various tricks to locate edges and avoid the most obvious artefacts, but in general, we cannot be sure that our particular choice of rays hasn’t missed an important detail.

What we can do, is to trade aliasing for noise. Most images with a lot of detail contain some kind of regularity. If we cast rays in a regular grid pattern, this pattern is very likely to interfere with some regular pattern in the scene and produce ugly artefacts. If instead we cast rays in directions with small random disturbances, it is likely that their average will be close to the ‘true’ colour of that part of the picture. A good and simple way to do this is to decide in advance how many rays are needed for each pixel. Nine is a good choice for a cheap picture, twenty five for better quality. Fig. 27 shows a typical pattern for throwing these rays. The pixel is divided into nine equal areas and one ray is thrown at a random point in each area. Fig. 28 shows the classical aliasing produced by a chess board pattern receding into the distance. Fig. 29 shows the same picture created with jittered sampling in the pattern of Fig. 27.

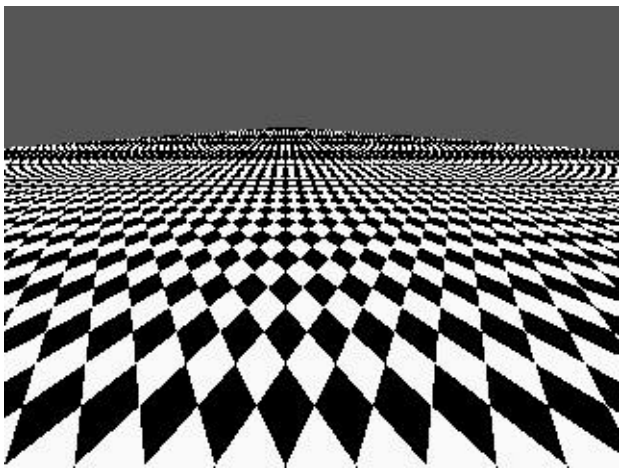

Fig. 28. Aliasing

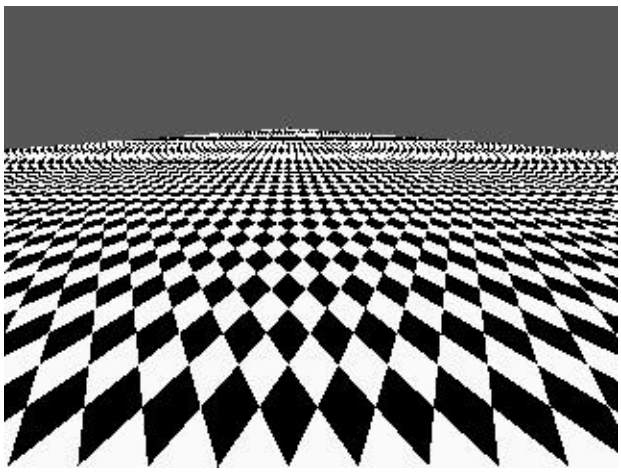

Fig. 29. Jittering

The main problem with jittering is that a typical picture has 500,000 to 1,000,000 pixels. Each pixel has a certain probability of returning a ‘wrong’ answer. With twenty samples per pixel, the probability of getting a totally wrong answer (black instead of nearly white) is about one part in 1,000,000. With 1,000,000 pixels, this will still happen once in every image on average.

Another way to disguise the aliasing artefacts is to blur the picture artificially after the ray tracing is all done. This is called post-filtering. It amounts to adding in a little of each pixel value to the surrounding pixels. Once again, it is only a disguise. None of these techniques attacks the prime cause of the problem that the picture is built only from point samples.

15. Distributed ray tracing

Jittering is only one kind of randomisation, useful in ray tracing. Using rays that are in some way randomised and averaging the result enables us to represent, approximately, a variety of phenomena that add to the richness of the images we can create. This is known as distributed ray tracing and the rays can be distributed (randomised) in time or space. The principal effects we can capture with this technique are depth of field, motion blur and soft shadows.

15.1 Depth of field

Classical ray tracing emulates the effect of a pinhole camera. Real cameras use lenses that enable them to collect more light but also produce side effects. The most important of these is known as ‘depth of field’. An ideal thin lens focuses every point in an object plane onto a single point in an image plane. Points that are not in the object plane are out of focus. Light from these points arrives on different parts of the image plane. In Fig. 30, the point, A, on the object plane is accurately focused at point, a, on the image plane. B, however, is focused at point, b, which does not lie on the image plane. Rays from B through different parts of the lens arrive at different points on the image plane resulting in blurring.

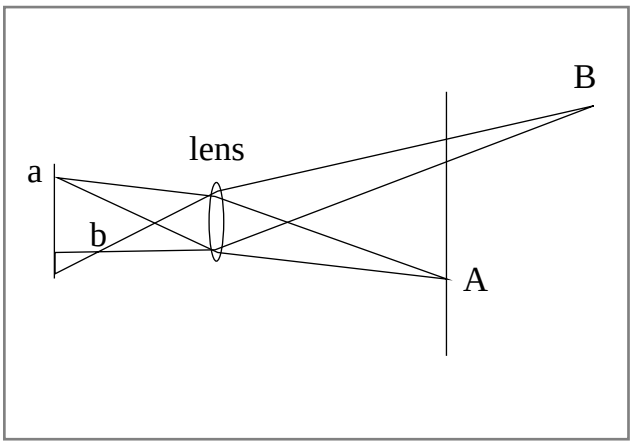

Fig.30. Depth of field

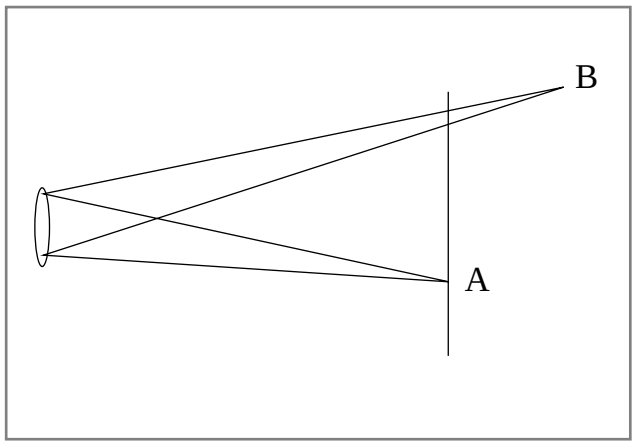

Fig. 31. Simplified camera geometry

Fig. 31 shows how to emulate the effect of depth of field without explicitly simulating the lens. The image is formed in the object plane itself and rays emanating from random points on the lens are sent through each pixel. Two such rays meeting at B pass through different points of

the image plane and thus give rise to blurring. As with jittering, we get a better result using randomised rays than simply by using many samples from fixed points on the lens. as with jittering, this is accompanied by the introduction of noise.

15.2 Motion blur.

Moving objects normally appear blurred in the direction of motion. If we make animation without motion blur, fast moving objects appear to have copies of their edges superimposed on the picture. In extreme cases, we get true temporal aliasing effects like the ‘reversing’ of the motion of spoked wheels in early movies. Motion blur is expensive in ray tracing because the only really general way to do it is to make each frame repeatedly at tiny time increments and then average the result in to the final frame to be displayed. Distributed ray tracing offers a cheaper alternative. Each ray is cast at a different time, randomised in the interval that the frame represents.

This works well for simple scenes but there are complications when we try to emulate motion blur with many objects. This is because the space division algorithm has to be extended to deal with objects that move. If an object moves very quickly, it appears in many cells of the space division and much of the advantage is lost.

15.3 Soft shadows

Soft shadows result from extended light sources. A point light source is either visible or not but an extended light source produces more illumination at a point from which more of it is visible. A cheap and effective way to produce soft shadows is to cast a bundle of light seeking rays to different parts of the extended source. The proportion of these rays that find the source is then used as a measure of the effective brightness at the point of interest. Once again, if these rays are distributed stochastically we get a better result on average than if they are cast in a regular pattern.

15.4 Economy of distributed ray tracing

The error in any kind of stochastic method is a function of the total number of samples being averaged. If we want motion blur, soft shadows, depth of field and stochastic antialiasing all together in one image, the cost need be no more than for any one of these effects. Instead of distributing each ray in time and then for each secondary ray, spawning a bundle of samples to an extended light source, we can simply start with our jittered rays for antialiasing and then send each ordinary light seeking ray to a different, randomly chosen part of the light source. The quality of the final picture is a function of the total number of rays cast not of the number of ways in which they are distributed.

16. Texture mapping

There is little to be said of texture mapping except that it can be used in a ray tracer just as in any other rendering system. However, there are a few trick uses of texture in ray tracing that are not really applicable to other rendering methods. A texture function can be used to alter any of the surface properties of an object. It can specify that part of an object is effectively invisible so the ray intersection can be discarded. This is particularly useful in very complicated scenes where some objects can be rendered separately and turned into a texture painted onto a plane within the scene. While the camera angle does not change, the false objects are indistiguishable from the original yet moving objects can pass behind or in front of them with the correct appearance.

One of the most cunning uses of texture is to cause an effective displacement of the object’s surface. The leaves in Fig. 32 have been made this way. Each leaf is represented by a cylindrical surface bounded by an ellipsoid. The ellipsoid is a sphere transformed using the mathematics of Section 12. The geometry of this ellipsoid is described by an equation of the form of (39). A 3D noise texture is added to the ellipsoid function at the intersection point and if this result is greater than q, the intersection is discarded. This simple strategy produces the wavy edges on the leaves. At the same time, the noise function is used to vary the surface normals in a bump map, producing the appearance of a crinkled surface.

Fig. 32 Leaves

17. Object Oriented Design

One of the recurrent problems that people have in the professional environment is that their ray tracer cannot easily be modified. The designer will have allowed for a range of alternative primitive objects in the model and (if you are lucky) for a range of alternative lighting models. But what happens if you want to introduce an object that behaves totally differently from the ordinary. In a recent research project we introduced a fuzzy object into the Katachi system. Fuzzy objects represented clouds of small water droplets. As such, they had no surface normals but a set of rules defining how they would appear when lit from different directions. They were also required to cast fuzzy shadows.

The key to designing systems that allow for easy expansion is to associate the code for each kind of object with the objects. Thus the ray tracing process looks something like this:

Cast_rays:	Choose which primary rays to cast according to the image requirements. Jittering would be done at this stage.
Shoot_a_ray:	For a given primary ray find the appropriate intersection. This calls different intersection routines stored with each different

kind of primitive.

Find illumination: Execute the illumination routine associated with the object intersected. This action will call Shoot_a_ray to find light sources, shadows and reflections.

In the original design of Katachi the CSG primitive routines had a very simple interface. they consisted of just three routines:

Intersect: Finds all the intersections with a given ray and the given primitive.

Point_Inside: Tells whether the given point is inside or outside of the primitive.

Voxel_test: Tells whether the given primitive completely contains the given voxel, is completely outside the given voxel or has a part of its surface within the given voxel.

These routines proved sufficient to sort CSG trees into a divided space, find intersections and unravel the CSG structure. Currently the system supports spheres, planes, cylinders, cones, tori and ‘soft objects’ as standard objects. But for special purposes, primitives have been added that do special things. There is a ‘height field’ primitive that aids the representation of realistic terrain. There are fuzzy objects. There are a variety of specially curved implicit surfaces of fourth and fifth order. And finally there is a new kind of interpolated volumetric primitive that handles everything from clouds to penguins.

18. References and Bibliography

In this section are listed the main sources of the material in this course. Clearly the subject is too large for this to be anywhere near complete

John G. Cleary and Geoff Wyvill, Analysis of an Algorithm for Fast Ray Tracing Using Uniform Space Subdivision *The Visual Computer*,Volume 4 Number 2, July 1988, pp 65-83
(This is the only place I know where all the details of cell skipping have been published.)

Chattopadhyay, S. and Fujimoto, A.
Bi-Directional Ray Tracing.
Computer Graphics 1987: Proceedings of CG International '87, Springer-Verlag, 335-343.

Cook, R.L.
Stochastic Sampling in Computer Graphics.
acm Transactions on Graphics, January 1986, Vol. 5, No. 1, 51-72.

Cook, R.L., Porter, T. and Carpenter, L.
Distributed Ray Tracing.
Computer Graphics (SIGGRAPH '84 Conference Proceedings), Vol. 18, No. 3, July 1984, pp. 137-145.

Fujimoto, A. and Iwata, K.
Accelerated Ray Tracing.
Computer Graphics Visual Technology and Art: Proceedings of Computer Graphics Tokyo '85, Springer-Verlag 1985, 41-65.

Fujimoto, A., Perrott, C.G. and Iwata, K.
Environment for Fast Elaboration of Constructive Solid Geometry.
Advanced Computer Graphics: Proceedings of Computer Graphics Tokyo '86, Springer-Verlag 1986, 20-33.

Glassner, A.S.
Space Subdivision for Fast Ray Tracing.
IEEE Computer Graphics and applications, Vol. 4, No. 10, October 1984, 15-22 .

Glassner, A.S.
Spacetime Ray Tracing for Animation.
IEEE Computer Graphics and applications, Vol. 8, No. 2, March 1988, pp. 60-70.

Ohta, M. and Maekawa, M.
Ray Coherence Theorem and Constant Time Ray Tracing Algorithm.
Computer Graphics 1987: Proceedings of CG International '87, Springer-Verlag, 303-314.

Scott. D. Roth, Ray Casting for Modeling Solids, *Computer Graphics and Image Processing*, Vol. 18, 109-144, 1982
(The original method of ray tracing CSG models. Still worth reading.)

Turner Whitted, An Improved Illumination Model for Shaded Display, *Comm. ACM*, Vol. 23, No. 6, 343-349, 1980
(The original paper on making ray tracing work)

Wyvill, G. and Sharp, P.
Volume and Surface Properties in CSG.
New Trends in Computer Graphics: Proceedings of CG International '88, Springer-Verlag 1988, 257-266. Also Item 17, Department of Computer Science, University of Otago, New Zealand.

Wyvill, G. and Trotman, A.
Exact Ray Tracing of CSG Models by Preserving Boundary Information.
Visual Computing — Integrating Computer Graphics with Computer Vision — [Proceedings of CG International '92], Springer-Verlag 1992, 411-428.

Wyvill, G., Kunii, T.L. and Shirai, Y.
Space Division for Ray Tracing in CSG.
IEEE Computer Graphics and applications, Vol. 6, No. 4, April 1986, 28-34.

Wyvill, G., Ward, A. and Brown, T.
Sketches by Ray Tracing.
Computer Graphics 1987: Proceedings of CG International '87, Springer-Verlag, 315-333. Also Item 15, Department of Computer Science, University of Otago, New Zealand.

Wyvill, G. and Sharp, P.
Fast Antialiasing of Ray Traced Images.
New Advances in Computer Graphics: Proceedings of CG International
'89, Springer-Verlag 1989, 579-588. Also Item 16, Department of Computer Science, University of Otago, New Zealand.