Computer Graphics International 1995[1] Course Notes

Practical Ray Tracing

Geoff Wyvill
University of Otago

Part 1: Basic Ray Tracing

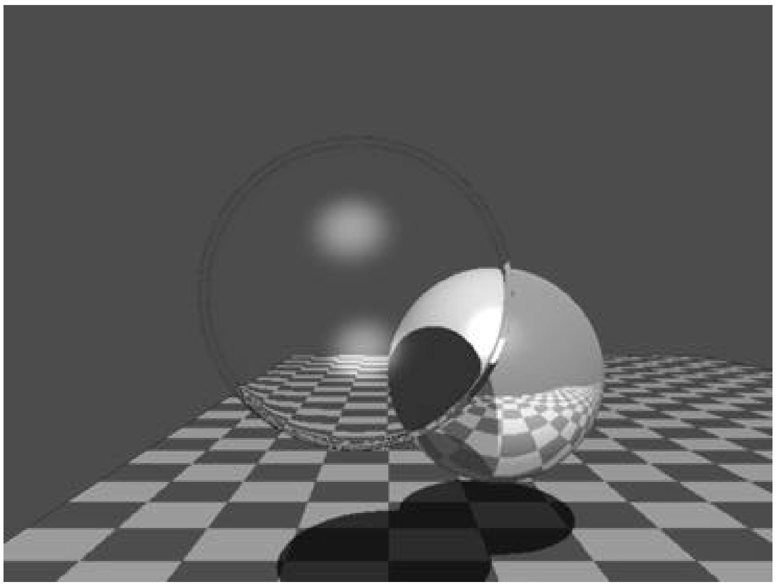


Fig. 1. A classical example of ray tracing.

1. Introduction

In 1979, Turner Whitted [Whitted 1980] revolutionised our thinking about high quality picture production. He didn't actually invent ray tracing, but he suggested a practical illumination model that made synthetic images of photographic quality easy to compute from basic geometrical models. The images produced by ray tracers typically show variation of shading on curved surfaces, shadows and reflections. Because of the high quality of these images, a lot of people have wrongly gained the impression that ray tracing is 'difficult'. It is easier to write a ray tracer than a Z-buffer or A-buffer rendering system. The real cost of ray traced images is in the computer time needed to create them and even this is not as great as is popularly supposed.

The underlying idea for ray tracing is to model the basic property of light, that it travels in straight lines. This was known to the ancient Greeks although many seem to have believed that the 'rays' of sensation were sent out from the eye. Euclid evidently understood perspective as

[1] Copyright © 1995 Computer Graphics Society

early as 300 B.C. A pinhole camera was described by della Porta in 1558 and this is often seen as the basic model for ray tracers.

Of course, we understand now that light is energy that travels from some source, ultimately to be observed by our eyes. But it is clearly not efficient to trace rays from a light source since most of them do not end up in the picture. Instead, we trace the rays backwards from the picture through a hypothetical pinhole and into the scene we have modelled.

And that is it. One simple principle that enables us to make glorious pictures from three dimensional scenes in our imagination. In this tutorial, the principle is developed in detail. In part one, the basic ideas are developed. In part two, we discuss some of the refinements that turn a picture making toy into a serious professional tool.

1.1 The basic process

This is illustrated in Fig. 2. The output of the ray tracer is a set of pixel values that represent the picture of the scene projected on to a picture plane. Each pixel of the output, corresponds to a point on this ideal picture plane. A single ray is cast from the eyepoint through each of these points and the colour found for that ray is the colour given to the corresponding pixel. This geometry produces the same picture as an ideal pinhole camera. In the camera the image appears upside down on a picture plane behind the pinhole. We prefer to use a plane in front of the eye. Fig. 3 shows the equivalence of these approaches.

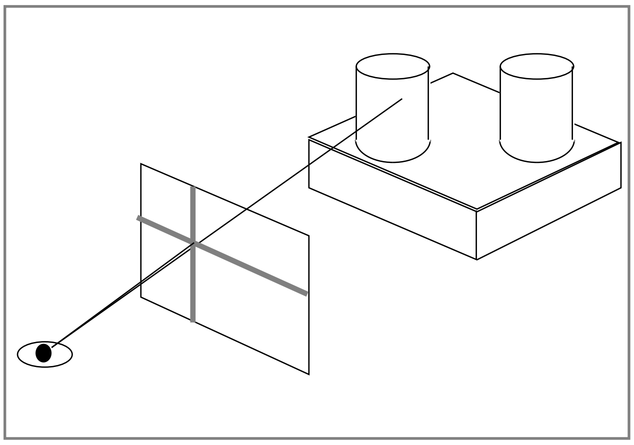


Fig. 2. Eye and picture plane

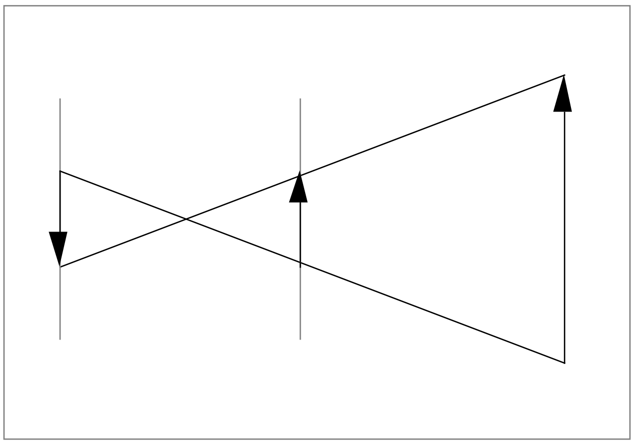


Fig. 3. Equivalence to pinhole camera

So building a picture is reduced to finding the colour of each pixel by casting a ray into the scene. Most of this tutorial is devoted to what happens to one ray. Putting them together is easy. Finding the colour from a ray requires firstly that we find which object it hits and then finding the colour and illumination of that object at that point. The illumination depends on the angle between the surface and the direction of incoming light and this is usually expressed as the angle between the incoming light and the surface normal at the hit point.

2. Intersection tests

To find the intersection of a ray and an object requires solving an equation. The exact form of the equation depends on the kind of object. Because it is easiest to understand we start with a sphere. For all intersection tests we represent the ray in vector form as:

$$\mathbf{p} = \mathbf{u} + \mathbf{v}t \qquad (1)$$

u is the starting point and **v** is the direction of the ray. The variable t indicates how far down the ray the point **p** is.

2.1 Intersection with a sphere

The equation of a sphere of unit radius centred at the origin of coordinates is:

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

We can also express this in vector form:

$$\mathbf{p}^2 = 1 \qquad (3)$$

Where the ray meets the sphere $\mathbf{p} = \mathbf{u} + \mathbf{v}t$ satisfies (3) and we have:

$$(\mathbf{u} + \mathbf{v}t)^2 = 1 \qquad (4)$$

or

$$\mathbf{u}^2 + 2\mathbf{u}\,.\mathbf{v}t + \mathbf{v}^2t^2 = 1 \qquad (5)$$

The dot products, $\mathbf{u}^2, \mathbf{u}\,.\mathbf{v}, \mathbf{v}^2$ are all scalars so (5) is no longer a vector equation and we can solve for t in the ordinary way. It is worth noting that the ordinary quadratic equation formula:

$$t = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A} \text{ where } A = v^2\text{ , } B = 2u\,.v\text{ , and } C = u^2 - 1 \qquad (6)$$

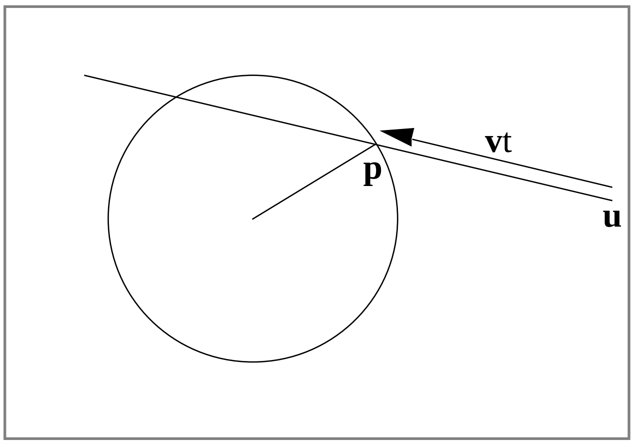


Fig. 4. Intersection with sphere

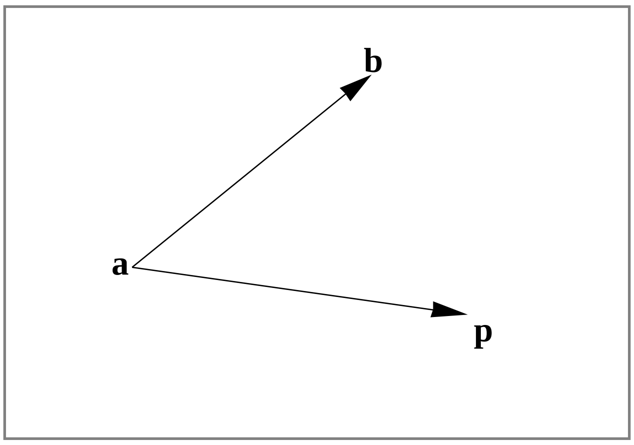


Fig. 5. Right or left of line

produces large rounding errors when $B^2 >> 4AC$. So we actually solve it in three steps as follows:

1. If $B^2 \leq 4AC$ There is no solution. When $B^2 = 4AC$ the ray meets the sphere tangentially. The best way to handle this is to treat it as missing the sphere.
2. If $B > 0$ then $t_1 = \frac{-B - \sqrt{B^2 - 4AC}}{2A}$ else $t_1 = \frac{-B + \sqrt{B^2 - 4AC}}{2A}$
3. $t_2 = \frac{C}{At_1}$

The two values, t_1, t_2 can be put into (1) to find the intersection points, **p**. For the special case of the unit sphere centred at the origin, the point, **p**, also gives us the surface normal, **n**, because the radius of a sphere meets the surface everywhere at right angles. In this case, **n** is of unit length already. Many of the calculations within a ray tracer use unit length normals.

2.2 Intersection with a triangle

Most people who write ray tracers start with the sphere because it is simple and quick to get working to see that all-important first picture. If you want to be able to ray trace more general shapes, the next step is to handle polygons. There are many published models described as polygon facets and the most popular among polygons is the triangle. More complicated polygons can always be divided into triangles so if you can ray trace triangles, you can make pictures from a terrific range of models.

A triangle is described by the positions of its vertices, $\mathbf{a}, \mathbf{b}, \mathbf{c}$. Any three points lie in a plane and the cross product of the vectors $(\mathbf{b} - \mathbf{a}) \times (\mathbf{b} - \mathbf{c})$ is normal to this plane.

$$\mathbf{n} = (\mathbf{b} - \mathbf{a}) \times (\mathbf{b} - \mathbf{c}) \tag{7}$$

for any point, p, in the plane:

$$\mathbf{n}.(\mathbf{p} - \mathbf{b}) = (\mathbf{b} - \mathbf{a}) \times (\mathbf{b} - \mathbf{c}).(\mathbf{p} - \mathbf{b}) = 0 \tag{8}$$

and this is the equation of the plane. Where the ray meets the plane $\mathbf{p} = \mathbf{u} + \mathbf{v}t$ satisfies (8) and:

$$\mathbf{n}.(\mathbf{u} + \mathbf{v}t - \mathbf{b}) = 0 \tag{9}$$

$$t = \mathbf{n}.(\mathbf{u} - \mathbf{b}) / \mathbf{n}.\mathbf{v} \tag{10}$$

If the ray is parallel to the plane, $\mathbf{n}.\mathbf{v} = 0$. we can't solve (10) and there is no intersection. Otherwise we have a single intersection with the plane, $\mathbf{p} = \mathbf{u} + \mathbf{v}t$ and the surface normal is **n**. But **p** may still be outside the triangle. If the three products:

$$(\mathbf{b} - \mathbf{a}) \times (\mathbf{p} - \mathbf{a}).\mathbf{n}, \ (\mathbf{c} - \mathbf{b}) \times (\mathbf{p} - \mathbf{b}).\mathbf{n}, \ (\mathbf{a} - \mathbf{c}) \times (\mathbf{p} - \mathbf{c}).\mathbf{n} \tag{11}$$

all have the same sign then **p** is inside the triangle. Otherwise the ray misses the triangle. To understand this see Fig. 5. Since **a**, **b**, **p** are in the plane of the triangle, (**b** - **a**) × (**p** - **a**) is perpendicular to that plane. If **p** is on the right of the line (**b** - **a**) the vector (**b** - **a**) × (**p** - **a**) will be directed towards us (out of the paper). If it is on the left, then (**b** - **a**) × (**p** - **a**) will be directed away from us. The dot product of **n** with each of the cross products (11) tells us whether **p** is to the right or to the left of each directed triangle edge. An internal point is either on the left of all three edges or on the right of all three edges. Notice that all of this is still true if we make the normal have unit length: Replace **n** by $\mathbf{n} / \sqrt{\mathbf{n}.\mathbf{n}}$.

3. Illumination fundamentals

The perceived colour of a surface depends on two things. The properties of the materials and the nature of the light falling on the surface. The colour does not depend on the distance between the object and the observer. Although less light reaches the eye from more distant objects, the objects appear smaller in the distance and the brightness (light per perceived unit area) remains constant. The brightness of an object *does* depend on the distance between the object and the source of light and this dependence is complicated. Fo now, we shall ignore this effect because we can make entirely satisfactory pictures without taking it into account.

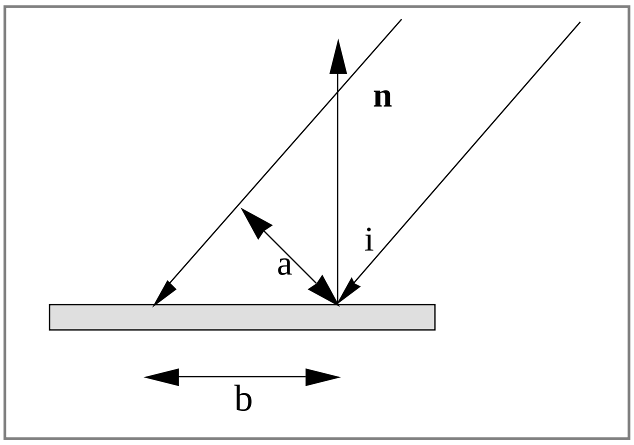


Fig. 6. Lambert's law

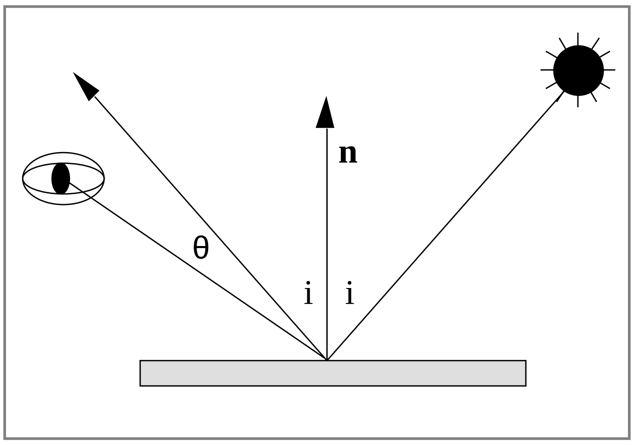


Fig. 7. Phong illumination

3.1 Diffuse reflection

Light falling on a surface at an oblique angle has less effect than the same light would have at a steeper angle. This is because the energy of the light is spread over a wider area. This is illustrated in Fig. 6 for the case of parallel light. A beam of width, a, strikes a surface at angle, i, measured from the normal. The energy of the beam is spread over an area proportional to b. So if the intensity of the beam is I, the intensity on the surface is:

$$Ia/b = I\cos(i) = I\,\mathbf{n}.\mathbf{i} \qquad (12)$$

where **n**, **i** are the surface normal and the direction to the light source expressed as unit vectors. This is sometimes called Lambert's law. The simplest illumination model assumes that the light reflected from a surface is diffused equally in all directions. In this case, we get the brightness of the surface from (12) by multiplying by a constant, k_d, known as the “coefficient of diffuse illumination”.

$$\text{Brightness} = k_d\, I\, \mathbf{n.i} \tag{13}$$

3.2 Highlights

Most surfaces are not ideal diffusers. A mirror for example reflects light in one direction only for a given incoming ray. Very often a diffusing surface, like white paper, reflects more light in directions close to the direction of mirror reflection. If the angle, θ, is the angle between the ray from the eye and the direction of mirror reflection of the light source in the surface. The illumination is approximated by:

$$\text{Brightness} = I \cos^n(\theta) \tag{14}$$

where n is chosen to characterize the shininess of the surface. A value, n = 1, produces a dull surface; n = 50 makes the surface look quite shiny, like metal.

3.3 Ambient light

Ambient light means light from the surroundings. In reality, everything in a scene is illuminated from light reflected by many other surfaces in the scene. Dark corners are never completely dark. But in our simple lighting model surfaces are lit only by specific light sources. Any surface that is not lit will appear totally black. To avoid this we include a constant term in the lighting. This is known as the ambient term and is intended to approximate the effect of all the multiple reflections of light around the scene. The brightness of a surface can then be expressed as:

$$\text{Brightness} = A\, k_a + \sum_j I_j\, k_d\, n.i + I_j\, k_p \tag{15}$$

where k_a, k_d, k_p are the coefficients for ambient, diffuse and Phong reflection, A is the constant amount of ambient light and $\sum_j$ indicates that we sum the other terms for each light source of intensity I_j.

3.4 Colour

The interaction of coloured light with surfaces in the real world is very complicated. The colour of light can be described, physically, by specifying the mixture of frequencies it contains. One good way to simulate it is to use a separate version of equation (15) for each of a number of frequency bands. Thus you would have a set of different coefficients, k_a, k_d, k_p for each frequency. A reasonably accurate colour system would need about twenty frequencies and that makes the colour model rather expensive. Most simple ray tracers use three colours that correspond to the red, green and blue (RGB) that make up a TV colour display. This is sufficient to produce some pretty effects, especially if each light source has a colour represented by different amounts of RGB.

4. Shadows, reflection and refraction

The heart of any ray tracer consists of two routines: one finds the next intersections between a ray and an object in the scene, the other determines the colour returned by a ray. The real power of ray tracing as a rendering technique becomes apparent when these routines can be called recursively. Suddenly, effects like shadows and mirror reflections can be rendered with hardly any additional code.

4.1 Shadows

At the top level the intersection routine is called to find the object responsible for the colour of a pixel. Having found the intersection, the illumination routine is called. The illumination routine needs to apply equation (15) to determine the brightness and colour of the surface. But what if the surface is in shadow? Quite simply, the illumination routine sets up a new ray from the intersection point to each light source and calls the intersection routine. If an intersection is found closer than the light source, then there is something in the way, the original intersection point is in shadow and the illumination routine can ignore that light source. This gets a bit more complicated when we have to deal with transparent objects, but that is all there is to the simpler cases.

4.2 Reflections

Mirror like reflections are made by calling both the intersection and illumination routines as part of the illumination calculation. We create a reflected ray in the correct direction and call both the intersection routine and the illumination routine for that ray. In effect we are asking what can be seen in that direction from this surface. Having got back the colour of this ray, our illumination routine has to include it in the total colour calculation for the surface. We introduce another coefficient, k_s, which is multiplied by the brightness of the reflected ray and added to the total for the current surface.

The original and reflected rays make equal angles with the surface normal. The direction of the reflected ray can be found easily using vectors. Assuming that the normal is represented by a unit vector, **n**, the component of the ray direction, **v**, in the direction of the normal is:

$$(\mathbf{v}.\mathbf{n})\,\mathbf{n} \tag{16}$$

Reflection has the effect of reversing this component without changing the component parallel to the surface. So the reflected ray is:

$$\mathbf{p} + (\mathbf{v} - 2(\mathbf{v}.\mathbf{n})\,\mathbf{n})t \tag{17}$$

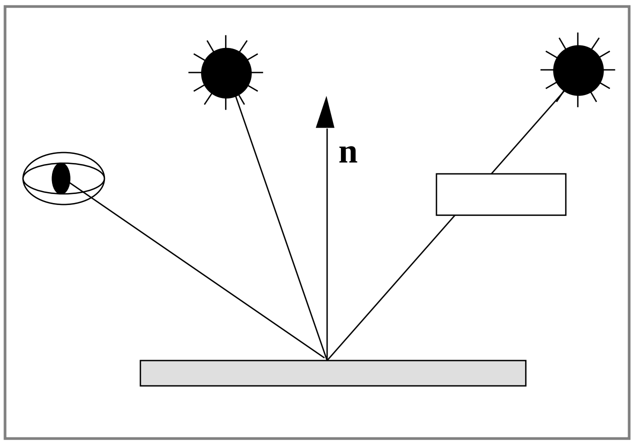


Fig. 8. Shadow calculation

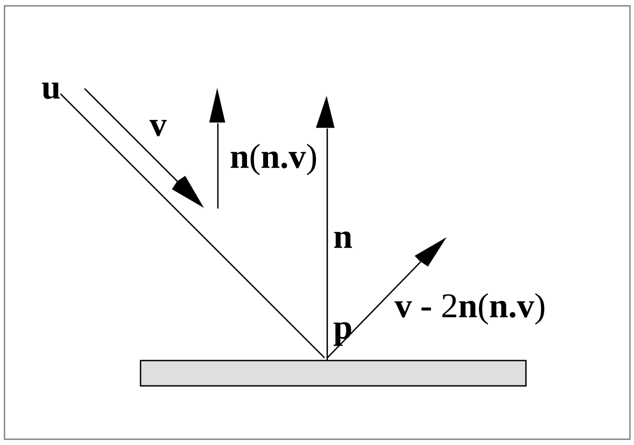


Fig. 9. reflected ray direction

5. Modelling and hierarchy

In most applications, complex objects and scenes are built from simpler ones. The four wheels of a car are all the same. The natural way to represent collections of objects like this is with a directed graph. Each object is represented by a collection of components and each component is represented by a pointer to a previously defined object together with a matrix which describes where to put that sub-object.

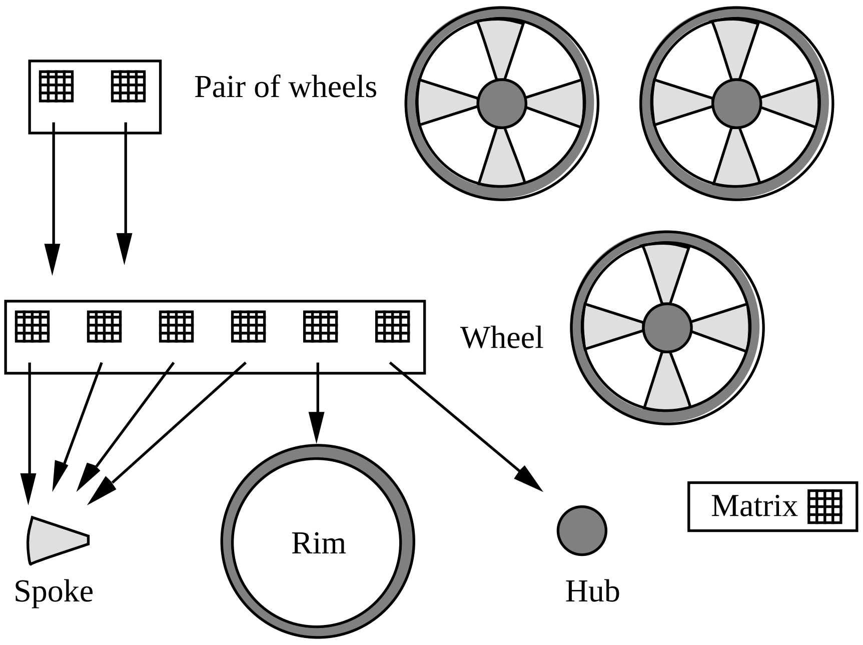


Fig. 10. Hierarchy.

Fig. 10 shows a pair of wheels with the associated data structure. Where is the top spoke of the left hand wheel? The pair definition contains a matrix, L, to position the left hand wheel in the pair. The wheel definition contains a matrix, T, to position the top spoke. The position of the spoke in the pair is defined by L T.

In a hierarchical modelling system, we can represent objects of arbitrary complexity. Everything is defined in terms of something else until, at the lowest level, we must use some more basic object that doesn't need the hierarchy to describe it. In hierarchical systems we call these bottom level objects primitives. In ray tracing, the primitives can be polygons, geometric objects such as the sphere or more complicated elements such as parametric patches.

5.1 Intersections with transformed objects

The transformation matrices are capable of expressing rotation, stretching and shifting operations. How then do we make an intersection test with a shifted, stretched and rotated sphere? By far, the simplest way is to apply an inverse transformation to the ray and then perform the calculation in the original space. Suppose we start with a unit sphere at the origin as in section 2.1. If the sphere is a primitive at the bottom level of a hierarchy, its final position rotation and stretch will all be described by some matrix, S which is made by multiplying, in order, the matrices encountered as the hierarchy is traversed. The matrix, S, describes the operation of transforming an arbitrary point from the 'primitive' space in which the sphere has been defined into the 'world' space in which the ray tracing takes place. The inverse matrix S^{-1} transforms points in the world space into points in primitive space.

First find

$$\mathbf{u}_p = S^{-1}\,\mathbf{u} \quad \text{and}\; \mathbf{v}_p = S^{-1}\,\mathbf{v} \tag{18}$$

Then find t_1, t_2 as described in 2.1 using $\mathbf{u}_p + \mathbf{v}_p t$ as the ray. The values of t_1, t_2 are the same in world space as in primitive space, so the intersection points can then be calculated as before, directly in world space.

Calculating the surface normal is a little more tricky because although the transformations preserve straight lines and planes, they do not preserve angles. Equation (8) describes a plane passing through a point, b. If b is the origin, then this reduces to:

$$\mathbf{n}.\mathbf{p} = 0 \tag{19}$$

or in matrix form:

$$\mathbf{n}^{\tau}\,\mathbf{p} = 0 \tag{20}$$

Suppose this is an equation in primitive space. Let **q** be the point corresponding to **p** in world space.

$$\mathbf{q} = S\,\mathbf{p} \;\text{ and }\; \mathbf{p} = S^{-1}\,\mathbf{q} \tag{21}$$

so

$$\mathbf{n}^{\tau}\,S^{-1}\,\mathbf{q} = 0 \tag{22}$$

This describes a plane in world space whose normal vector is $\mathbf{n}^{\tau}\,S^{-1}$ but the normal of the transformed plane is the correct transformation of the normal, $\mathbf{n}_W$.

$$\mathbf{n}_W = \mathbf{n}^{\tau} S^{-1} = S^{-1\tau} \mathbf{n} \tag{23}$$

Thus the normal is put into world space using the transpose of the inverse matrix.

This result makes it easy to ray trace any transformation of a model for which we can ray trace the original. Fig. 11 shows a collection of ladders arranged in a logarithmic spiral. Each ladder is a different transformation of a single original ladder. Indeed each ladder consists of differently transformed cylinders. The ray tracer that produced this image works exactly as described above. The cylinder routine works only for a standard cylinder in its primitive space.

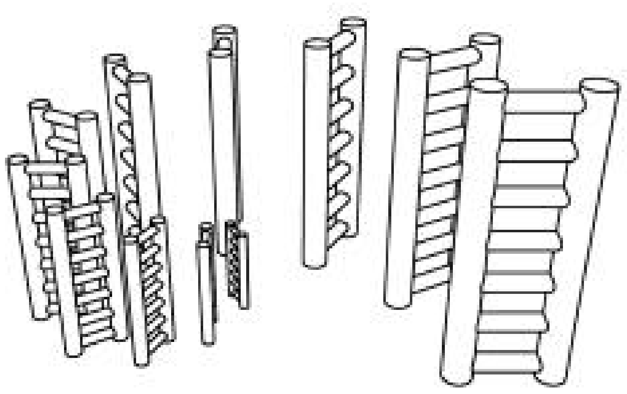


Fig. 11 Hierarchical example: Ladders

7. Efficiency

7.1 Bounding boxes.

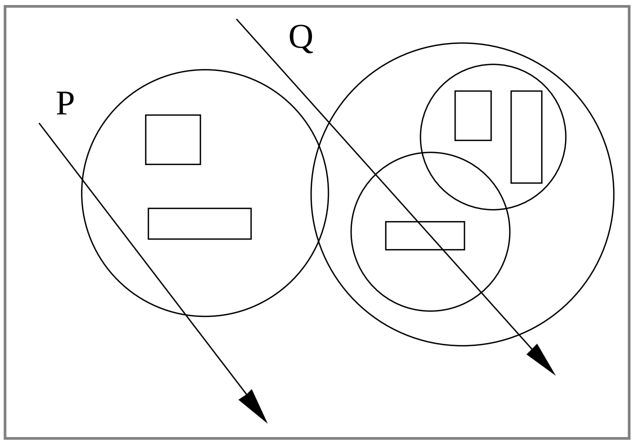
In principle, a ray tracer must test every object in the scene for intersection with every ray cast. That includes the light-source rays used to find shadows and all the rays cast to identify reflections and refraction. The original rays from the eyepoint usually constitute less than a third of the total. In quite a modest picture a million rays may be cast.

All the strategies for reducing the number of intersection tests depend on the idea of doing some kind of preliminary sorting of objects in space. This means that some objects can be eliminated without actually performing the intersection tests.

The simplest way to do this is to use bounding spheres. Suppose some of the objects in a scene are completely contained by some sphere, S. Then a ray that does not intersect S cannot intersect any of those objects. If we set up the quadratic equation (6) for sphere S we need only perform step one of the solution: if $B^2 \geq 4\ A\ C$ there are no real roots and the ray misses S and everything inside it. This is a very cheap test. Also, the method can be extended very simply to a hierarchy of spheres within spheres. In Fig. 12, the ray, P, gets tested against one sphere and two objects (rectangles). Ray, Q, gets tested against three spheres and one object.

If the objects in a scene can be sparated by spherical bounds in this way, then we can expect to make only log(N) tests where we have N objects. Unfortunately, there seems to be no simple, automatic way to place the bounding spheres. So this method needs a lot of hand work. Even so there are times when it is justified. If you wanted to make an animation of a walk through a complex but unchanging scene. The spheres would be placed only once.

Other bounding strategies use axis-aligned cubes in suitable places, or parallelepipeds (boxes with parallel opposite faces not necessarily at right angles). In general, the tests for intesection with these bounds are more expensive than with spheres and there is still no good automatic way to place them.



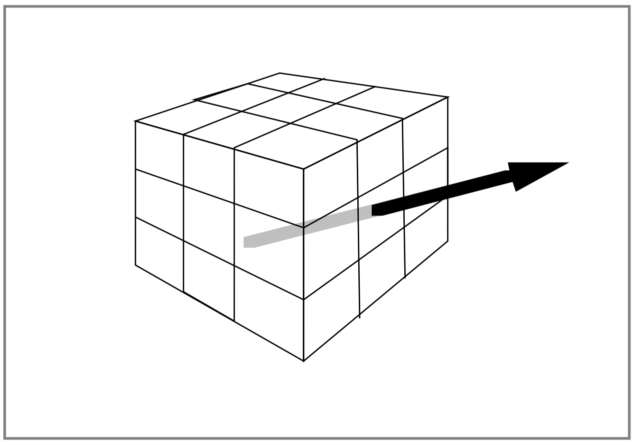


Fig. 12 Spherical bounding boxes Fig. 13. Ray classified by exit window

7.2 Ray sorting

Another technique altogether is to sort the objects by position and angle [Ohta 1987]. The objects are divided into clusters and each cluster is enclosed in a cube with windows, Fig. 13. A table is then made up to record which clusters are visible from which other clusters through which windows. Any given ray has to start from inside a cluster and it leaves the cluster through exactly one window. The table entry for that window is a list of clusters that contain objects that might be hit by that ray. This method and variations are often described as very efficient but the selection of clusters is still done by hand.

7.3 Space division

The best method of reducing intersection tests for most scenes seems to be spatial sorting and the easiest to understand is the method described by Cleary [1988]. Here the scene is divided into a 3D grid of identical cubic boxes. With each box is associated a list of objects, of which some parts are in that box. If the amount of work done in following the ray through the grid is less than that for the intersection tests avoided, we get a speed up. Fig. 14 illustrates this effect. The ray shown, gets intersected with only two of the objects. This is because the ray tracer only checks intersection with objects in the current space division cell. The line of the ray passes many other objects but it doesn't enter a cell where these are present until after the intersection has been found.

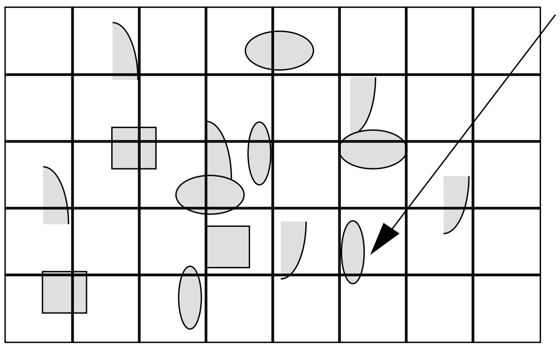


Fig. 14 Space division: Each object occupies a small subset of the cells.

An efficient procedure for the ray to skip cells requires a little cunning. Fig. 15 shows the geometry. A ray starts from **p** and makes an angle θ with the x-axis. The distances dx and dy, measured along the ray tell us how far away is the next intersection with the y-axis and x-axis. If dx < dy then our next move into a new cell will be in the x-direction. As soon as we make this move, we can find a new value for dx:

$$dx := dx + s / \cos(\theta) \tag{24}$$

where s is the size of the grid cell. This means that we can again compare dx with dy to find whether our next move is horizontal or vertical. The important thing to notice is that s / cos(θ) is a constant calculated only once for each ray.

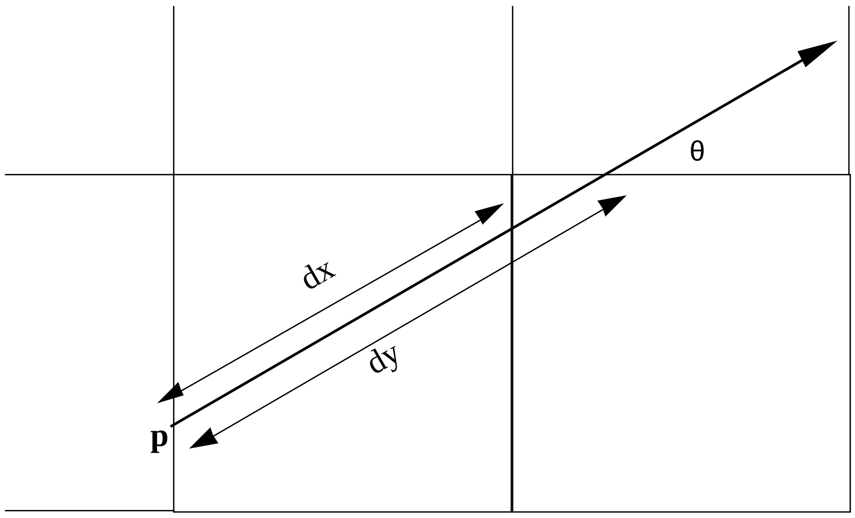


Fig. 15. Cell skipping algorithm.

This idea readily transfers to the 3D case where we define dx, dy, dz in the same way. Measured along the ray, dz is the distance from p to the next intersection with the x/y-plane. With a few refinements, we can turn this into an algorithm that uses only half a dozen integer operations to identify the next cell.

We set up a data structure that represents a 3D array of cells where each array entry contains a list of objects that need to be tested in that cell. With the very high speed of the cell-skipping algorithm it is possible to use a fine grid, say 100×100×100 cells. This can reduce the average number of intersection tests per ray to two or three even if there are thousands of objects.

8. Antialiasing

Casting one ray per pixel as suggested in Section 1 produces jagged lines, moiré fringes and similar aliasing artefacts in the pictures. The underlying reason for these is that there is more detail in the picture than our pixels can display. The best solution, in general is to blur the picture to eliminate this detail and then turn it into pixels. But with ray tracing you don’t have a picture to blur. The only information you have is from the rays.

A partial solution is supersampling. We throw more than one ray per pixel and average the result. This is very expensive. To eliminate jagged lines effectively you need about twenty five rays per pixel. It is very common for authors making claims about fast ray tracers to ignore this problem and quote times for pictures with only one ray per pixel.

There is a cheap and effective way to deal with jagged edges, but it doesn’t help much with other aliasing artefacts. The idea is to locate the edges and then estimate the area of each pixel that is covered by each different colour. Suppose we start by casting rays at the corners of the pixels. If the colours of two adjacent corners are different then we can cast an extra ray halfway between them. We repeat this process of casting an extra ray between differently coloured points until we have located the edge to some fraction of the pixel side, Fig. 16.

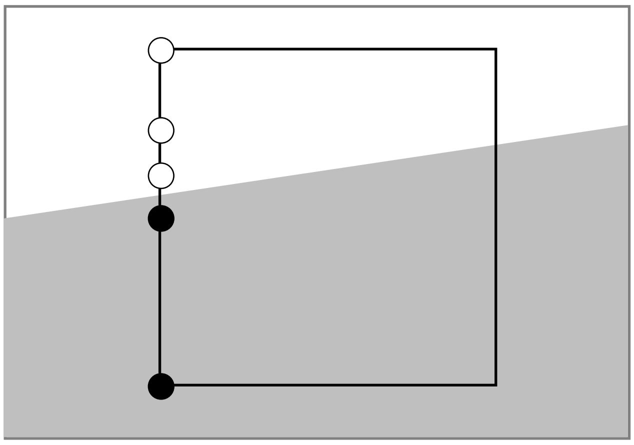


Fig. 16. Binary search to locate edges

Where the pixel is intersected with only one edge, we assume that the edge is approximately straight within the pixel and the two areas of different colour can be calculated easily. When more edges meet in one pixel it is a little more complicated. This is shown in Figure 17. From the binary search we get a number of points on the pixel boundary where the colour changes. Between each pair of adjacent points is a boundary segment all of one colour. We define the point, m, to be the centroid of these points of colour change. That is, for n points, p_1, p_2 .. p_n:

$$m_x = \frac{1}{n}\sum_{i=1}^{n} p_{i_x} \quad \text{and} \quad m_y = \frac{1}{n}\sum_{i=1}^{n} p_{i_y} \tag{25}$$

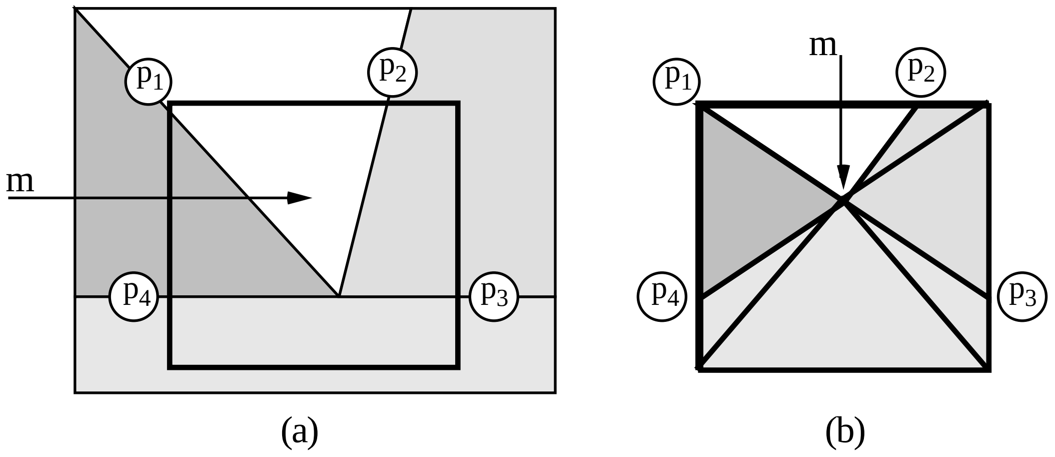


Fig. 17. several edges meeting in one pixel

We then treat the pixel as if the various colour areas met at m. The area of each colour is easily calculated as the sum of up to four triangles. A sample triangulation is shown in Fig. 17. Of course, this is not the true area covered, but it is a useful approximation. Where only one straight edge intersects the pixel, the centroid of the entry and exit points lies on the edge, so the same process reduces to the simpler case.

This method of removing jagged edges is appealing because it is very cheap. Just three extra rays locate the edge to within one eighth of the pixel side. The total number of extra rays cast depends on the picture but a lot of practical experience suggests that images made with this technique use about 50% more rays than unantialiased pictures.

9. Limitations of ray tracing

The underlying idea of ray tracing is imitating what light rays do. But the real behaviour of light is rather complicated. We can cast a ray at a point light source to find shadows, but real light sources are not points. As a result, shadows are soft rather than sharp. There is no known cheap way to get realistic shadows from a ray tracer.

In the real world everything is illuminated by light reflected from surrounding objects. Light gets reflected from shiny surfaces and bent by transparent objects. Sometimes this indirect light is a major component of the total. This is particularly noticeable when the light is focused like the reflections on the inside of a cup or the effect of sunlight on the bottom of a pool of water.

Bibliography

Here are listed the main sources of the material in this course.

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.)

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.

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

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.

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., 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.