# 2 Essential Ray Tracing Algorithms

ERIC HAINES

## 1 INTRODUCTION

The heart of any ray tracing package is the set of ray intersection routines. No matter what lighting models, texture mappings, space subdivision techniques, anti-aliasing schemes, or other elaborations of the ray tracing algorithm are desired, there is always the need to find the intersection point of a ray and an object.

When a ray is sent out into the modelled environment there are a few different kinds of questions to answer about the ray. What information is needed depends on the ray's purpose. For a ray spawned from the eye, an object intersector must return (at least) the closest intersection point and the surface's normal at this point. For a ray sent towards a light (a.k.a. a shadow feeler), all that is needed is whether the intersection point is closer than the light—if so, it blocks or filters the light. Further information may be desired for filtering, depending on the shading model. For any ray tested against a bounding volume, a simple hit/not hit determination is sometimes all that is required. However, more efficient ray tracers will take advantage of the distance along the ray (e.g. [9]).

Another piece of information that is useful in ray tracing is the intersection point's location relative to some reference frame for the surface. This location is typically used in texture mapping to find the surface properties at that point. See [7] for a good overview of texture mapping.

For anyone wishing to write a ray tracer, the ray/object algorithms are usually derived and coded from scratch. As educational as this process can be, many programmers simply do not care to go through it. Also, making these algorithms efficient is often an evolutionary procedure, as mathematically elegant solutions often make for slow algorithms. This document outlines the

basic algorithms used to perform a variety of ray intersection tests and retrieve the essential data. Rather then present abstract equations, derivations are shown in a nuts and bolts fashion, with an example of use following each algorithm. The overarching philosophy is to present an efficient algorithmic approach.

This document covers only objects whose ray/object intersection can be found by using simple algebra. This effectively limits the discussion to quadric surfaces, of which the plane and the sphere are special cases. As the sphere is one of the simplest and most popular primitive objects, it will be discussed first. Planes are then covered, along with the additional algorithms needed for polygons. Bounding box intersection is then presented. Finally, intersection of quadric surfaces is explained. Interspersed are relevant inverse mapping techniques and other topics.

Note that the focus of this presentation is the study of algorithms used in ray tracing, though there are also uses for these methods in other rendering schemes and in other interactive graphical processes, such as hit-testing (a.k.a. picking).

Ray tracing shadows of transmitters, CSG (constructive solid geometry) trees, and some other applications [13] requires that all intersection points for the ray be found. To extend the algorithms explained in this document is fairly straightforward, and so normally will not be discussed. Another subset of ray tracing which is not addressed is ray tracing finite length rays. Such rays have uses for activities such as shadow testing, where the ray's length cannot exceed the distance to the light.

## 1.1 Notes on Notation

The following conventions will be used:

* means 'multiply'
$\cdot$ means 'dot product'
$\otimes$ means 'cross product'
$\equiv$ means 'is equivalent to', and is used to show notation equivalences
$\pm$ means 'plus/minus,' signifying that two values are produced

abs($y$) means 'absolute value of y'
arccos ($y$) means 'inverse cosine of $y$'
$x$ mod $y$ means 'the remainder of $x/y$'
sin($y$) means 'sine of $y$'
sqrt($y$) means 'square root of $y$'

$\pi$ stands for 3.1415926...
All angle calculations are in radians

**V** denotes a vector
**M** denotes a matrix
Capital letters normally denote parameters; lower case, variables

### Examples

$T_a$, where $T$ is a scalar with subscript 'a.'
$\mathbf{B}_0 = [1\ 2\ 4\ 8]$, where **B** is a four element vector with subscript '0.'
$\mathbf{Q}_{r1}$ is a matrix **Q** with subscript 'r1.'

## 2 RAY/SPHERE INTERSECTION AND MAPPING

The sphere is one of the mostly commonly used primitives in ray tracing. Also, its ease of testing for intersection with a ray makes it useful as a bounding volume. As such, an in-depth look at the solutions to this problem is made. First the straightforward algebraic solution is derived. Then the special conditions of the problem are examined and a more efficient geometric solution is presented. A comparison of the results of the analysis shows the underlying equivalence of the two algorithms.

A study of a common bug found in ray tracing is made and some solutions are presented.

The algorithm for the most common inverse mapping of a sphere concludes the section.

### 2.1 Intersection of the Sphere—Algebraic Solution

Define a ray as:

$$\mathbf{R}_{origin} \equiv \mathbf{R}_0 \equiv [X_0\ Y_0\ Z_0]$$
$$\mathbf{R}_{direction} \equiv \mathbf{R}_d \equiv [X_d\ Y_d\ Z_d]$$
where $X_d^2 + Y_d^2 + Z_d^2 = 1$ (i.e. normalized)

which defines a ray as:

$$\text{set of points on line } \mathbf{R}(t) = \mathbf{R}_0 + \mathbf{R}_d * t, \text{ where } t > 0. \qquad \text{(A1)}$$

Points on the line where $t < 0$ are behind the ray's origin. Why $t = 0$ is not included as a point on the ray is explained in the 'Precision Problems' section. Note that the ray direction does not have to be normalized for these calculations. However, such normalization is recommended, otherwise $t$ will represent the distance in terms of the length of the direction vector. Normalizing the direction vector once for the ray before intersection testing

ensures that $t$ will equal the distance from the ray's origin in terms of world coordinates.

Equation (A1) is the *parametric* or *explicit* form of the ray equation. This means that all the points on the ray can be generated directly by varying the value of $t$.

The sphere is defined by:

$$\begin{aligned}&\text{Sphere's center} \equiv \mathbf{S_c} \equiv [X_c\ Y_c\ Z_c]\\&\text{Sphere's radius} \equiv S_r\\&\text{Sphere's surface is the set of points } [X_s\ Y_s\ Z_s]\\&\text{where } (X_s - X_c)^2 + (Y_s - Y_c)^2 + (Z_s - Z_c)^2 = S_r^2.\end{aligned} \tag{A2}$$

The sphere's surface is expressed as an *implicit* equation. In this form points on the surface cannot be directly generated. Instead, each point $[X_s\ Y_s\ Z_s]$ can be tested by the implicit equation; if it fulfills the equation's conditions, the point is on the surface.

To solve the intersection problem, the ray equation is substituted into the sphere equation and the result is solved for $t$. This is done by expressing the ray equation (A1) as a set of equations for the set of points $[X\ Y\ Z]$ in terms of $t$:

$$\begin{aligned}X &= X_0 + X_d * t\\Y &= Y_0 + Y_d * t\\Z &= Z_0 + Z_d * t.\end{aligned} \tag{A3}$$

Substituting this set of equations into the sphere equation's variables $[X_s\ Y_s\ Z_s]$, we obtain:

$$\begin{aligned}&(X_0 + X_d * t - X_c)^2 +\\&(Y_0 + Y_d * t - Y_c)^2 +\\&(Z_0 + Z_d * t - Z_c)^2 = S_r^2.\end{aligned} \tag{A4}$$

In terms of $t$, this simplifies to:

$$A * t^2 + B * t + C = 0 \tag{A5}$$

where

$$\begin{aligned}A &= X_d^2 + Y_d^2 + Z_d^2 = 1\\B &= 2 * (X_d * (X_0 - X_c) + Y_d * (Y_0 - Y_c) + Z_d * (Z_0 - Z_c))\\C &= (X_0 - X_c)^2 + (Y_0 - Y_c)^2 + (Z_0 - Z_c)^2 - S_r^2.\end{aligned}$$

Note that coefficient $A$ is always equal to 1, as the ray direction is normalized.

Also note that $S_r^2$ could be pre-computed for the sphere. This equation is quadratic, and the solution for $t$ is (with $A = 1$):

$$t_0 = \frac{-B - \sqrt{(B^2 - 4 * C)}}{2}$$
$$t_1 = \frac{-B + \sqrt{(B^2 - 4 * C)}}{2}. \tag{A6}$$

When the discriminant (the part in the sqrt() function) is negative, the line misses the sphere. A more accurate formulation for the solution of $t_0$ and $t_1$ is found in Section 5.5 of [11].

Since $t > 0$ is part of the ray definition, the roots $t_0$ and $t_1$ are examined. The smaller, positive real root is the closest intersection point on the ray. If no such point exists, then the ray misses the sphere. Some calculation can be avoided by calculating $t_0$, checking if it is greater than 0, then calculating $t_1$ if it is not.

Once the distance $t$ is found, the actual intersection point is:

$$\mathbf{r}_{intersect} \equiv \mathbf{r}_i = [x_i\ y_i\ z_i] = [X_0 + X_d * t\ \ Y_0 + Y_d * t\ Z_0 + Z_d * t]. \tag{A7}$$

The unit vector normal at the surface is then simply:

$$\mathbf{r}_{normal} \equiv \mathbf{r}_n = \left[ \frac{(x_i - X_c)}{S_r} \frac{(y_i - Y_c)}{S_r} \frac{(z_i - Z_c)}{S_r} \right]. \tag{A8}$$

If the ray originates inside the sphere (and so hits the inside), $\mathbf{r}_n$ should be negated so that it points back towards the ray.

Note that it may be more profitable to pre-calculate the multiplicative inverse of the radius and multiply by this in (A8), since division often takes a fair bit longer than multiplication.

To summarize, the steps in the algorithm are:

Step 1: calculation of $A$, $B$, and $C$ of the quadratic.
Step 2: calculation of discriminant.
Step 3: calculation of $t_0$ and comparison.
Step 4: possible calculation of $t_1$ and comparison.
Step 5: intersection point calculation.
Step 6: calculation of normal at point.

Assuming the most is made out of pre-calculated constants (such as $S_r^2$) and intermediate results, the calculations associated with each step are:

Step 1: 8 additions/subtractions and 7 multiplies.
Step 2: 1 subtraction, 2 multiplies, and 1 compare.
Step 3: 1 subtraction, 1 multiply, 1 square root, and 1 compare.

Step 4: 1 subtraction, 1 multiply, and 1 compare.
Step 5: 3 additions, 3 multiplies.
Step 6: 3 subtractions, 3 multiplies.

For the worst case this gives a total of 17 additions/subtractions, 17 multiplies, 1 square root, and 3 compares.

## Example

Given a ray with an origin at [ 1 −2 −1] and a direction vector of [1 2 4], find the nearest intersection point with a sphere of radius $S_r = 3$ centered at [3 0 5].

First normalize the direction vector, which yields:

$$\text{direction vector magnitude} = \sqrt{(1*1 + 2*2 + 4*4)} = \sqrt{21}$$
$$\mathbf{R}_d = [1/\sqrt{21} \quad 2/\sqrt{21} \quad 4/\sqrt{21}$$
$$= [0.218\ 0.436\ 0.873].$$

Now find $A$, $B$, and $C$, using equation (A5):

$$A = 1 \text{ (because the ray direction is normalized)}$$
$$B = 2*(0.218*(1-3) + 0.436*(-2-0) + 0.873*(-1-5))$$
$$= -13.092$$
$$C = (1-3)^2 + (-2-0)^2 + (-1-5)^2 - 3^2$$
$$= 35.$$

We now check if the discriminant is positive (A6):

Is $B^2 - 4*C > 0$?
Substituting: is $-13.092^2 - 4*35 > 0$?
Yes, $31.400 > 0$.

This means the ray intersects the sphere. From this we can calculate $t_0$ from (A6):

$$t_0 = \frac{-B - \sqrt{(B^2 - 4*C)}}{2}$$

$$= \frac{13.092 - \sqrt{(31.400)}}{2}$$

$$= 3.744$$

Since $t_0$ is positive, we don't have to calculate $t_1$. The actual intersection point

is, by (A7):

$$\begin{aligned}\mathbf{r}_i &= [X_0 + X_d * t \;\; Y_0 + Y_d * t \;\; Z_0 + Z_d * t] \\ &= [1 + 0.218 * 3.744 \quad -2 + 0.436 * 3.744 \quad -1 + 0.873 * 3.744] \\ &= [1.816 \quad -0.368 \quad 2.269].\end{aligned}$$

The unit vector normal is, by (A8):

$$\begin{aligned}\mathbf{r}_n &= \left[\frac{(x_i - X_c)}{S_r} \quad \frac{(y_i - Y_c)}{S_r} \quad \frac{(z_i - Z_c)}{S_r}\right] \\ &= [\,(1.816 - 3)/3 \quad (-0.368 - 0)/3 \quad (2.269 - 5)/3] \\ &= [\,-0.395 \quad -0.123 \quad -0.910].\end{aligned}$$

## 2.2 Intersection of the Sphere—Geometric Solution

Now that a simple sphere intersection routine has been outlined, the next question is, "How can we make it run faster?" Some basic ideas about computing efficiency are useful here.

One observation which generally holds is that using the square root function should be avoided when possible. Check timings on the machine used: often the sqrt() function takes 15–30 times as long as a multiply. Similarly, divisions usually take longer than multiplications, so it is often worthwhile to use the multiplicative inverse to avoid division. For clarity these substitutions are not made within this text, and most should be obvious to the implementer.

Another observation is that calculations can often be cut short. In the case of a sphere, there are a number of tests which can be made to check whether an intersection takes place. The purpose of these tests is to avoid calculations until they are needed.

By studying the geometry of the situation, other properties of the problem become apparent. For example, often the ray points away from the sphere and so does not intersect it. By studying such possibilities, another strategy for testing the intersection of the ray and the sphere was discovered:

(1) Find if the ray's origin is outside the sphere.
(2) Find the closest approach of the ray to the sphere's center.
(3) If the ray is outside and points away from the sphere, the ray must miss the sphere.
(4) Else, find the squared distance from the closest approach to the sphere surface.
(5) If the value is negative, the ray misses the sphere.

(6) Else, from the above, find the ray/surface distance.
(7) Calculate the $[x_i\ y_i\ z_i]$ intersection coordinates.
(8) Calculate the normal at the intersection point.

This strategy essentially breaks up the equations (A5) and (A6) into shorter expressions, which are evaluated as needed. Conditions (3) and (5) detect when the ray misses the sphere, allowing an early halt to calculations.

The above strategy will now be fleshed out and explained. Begin with the original ray (A1) and sphere (A2) equations. To start, find whether the ray's origin is inside the sphere by calculating:

$$\begin{aligned} &\text{origin to center vector} \equiv \mathbf{OC} = \mathbf{S_c} - \mathbf{R_0} \\ &\text{length squared of } \mathbf{OC} \equiv L_{2oc} = \mathbf{OC} \cdot \mathbf{OC}. \end{aligned} \tag{A9}$$

If $L_{2oc} < S_r^2$ then the ray origin is inside the sphere. If $L_{2oc} \geq S_r^2$ then the origin is on or outside the sphere, and the ray may not hit the sphere. Examples for these two cases are shown in *Figure 1*. For the sake of efficiency, $S_r^2$ could be pre-computed and stored.

Note that a ray originating on a sphere is considered not to hit the sphere at the ray's origin. This is standard in ray tracing, where reflected and refracted rays originate on a surface previously intersected. The problem of avoiding these intersections at the origin is discussed in the 'Precision Problems' section.

In either case, the next step is to calculate the distance from the origin to the point on the ray closest to the sphere's center. This is equivalent to finding the intersection of the ray with the plane perpendicular to it which passes through

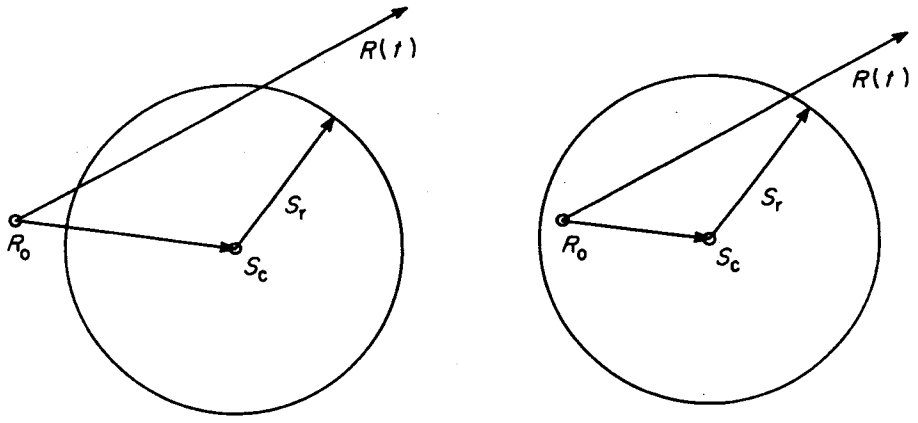


Fig. 1. The ray origin with respect to sphere location.

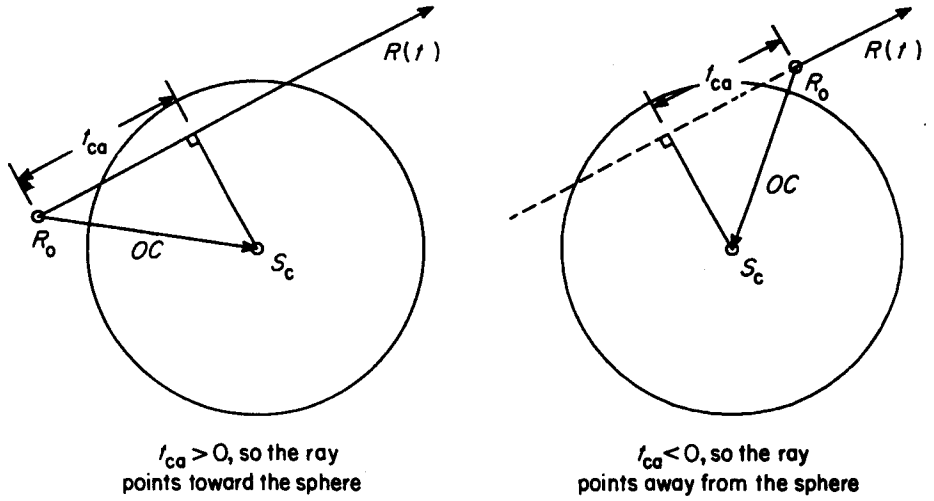


Fig. 2. Ray/sphere pointing directions.

the center of the sphere. This calculation is:

$$\text{closest approach along ray} \equiv t_{ca} = \mathbf{OC} \cdot \mathbf{R_d}. \tag{A10}$$

If $t_{ca} < 0$ then the center of the sphere lies behind the origin. This is not all that important for rays originating inside the sphere, since these must intersect. For rays originating outside this means that the ray cannot hit the sphere and testing is completed. Another way of saying this is that if $t_{ca} < 0$, the ray points away from the center of the sphere. Examples of these cases are shown in *Figure 2*.

Once the closest approach distance is calculated, the distance from this point to the sphere's surface is determined. This distance is:

$$\text{half chord distance squared} \equiv t_{hc}^2 \equiv t_{2hc} = S_r^2 - D^2 \tag{A11}$$

where $D$ is the distance from the ray's closest approach to the sphere's center. Calculate $D$ by the Pythagorean theorem:

$$D^2 = L_{oc}^2 - t_{ca}^2. \tag{A12}$$

Substituting this into (A11):

$$t_{2hc} = S_r^2 - L_{2oc} + t_{ca}^2. \tag{A13}$$

The geometric meaning of these equations is shown in *Figure 3*. This calculation leads to another test as to whether the ray hits the sphere. If

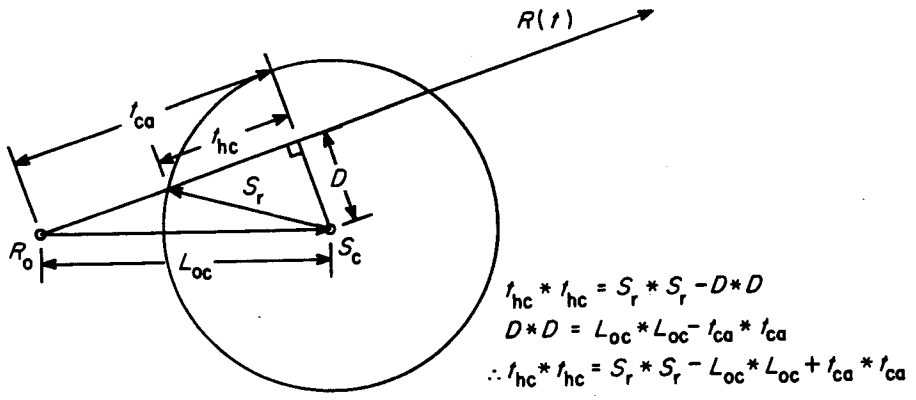


Fig. 3. Geometry of sphere intersection.

$t_{2hc} < 0$, then the ray misses the sphere. This can happen, of course, only when the ray originates outside the sphere.

At this point all factors have been calculated to determine the actual intersection point's distance along the ray. It is:

$$t = t_{ca} - \sqrt{t_{2hc}} \text{ for rays originating outside the sphere,}$$
$$t = t_{ca} + \sqrt{t_{2hc}} \text{ for rays originating inside or on the sphere.} \qquad \text{(A14)}$$

The difference in these formulae is simply that different intersection points along the rays' lines are needed in different cases. Rays which hit (and are not tangent) have two distinct intersection points along the ray's line. When the ray originates outside the sphere, the smaller distance along the ray is desired. If inside, the smaller distance is negative (behind the ray), so the larger distance is used.

Use equations (A7) and (A8) as before to calculate the intersection point and normal.

To summarize, the steps in the algorithm are:

Step 1: find distance squared between ray origin and center.
Step 2: calculate ray distance which is closest to center.
Step 3: test if ray is outside and points away from sphere.
Step 4: find square of half chord intersection distance.
Step 5: test if square is negative.
Step 6: calculate intersection distance.
Step 7: find intersection point.
Step 8: calculate normal at point.

Assuming the most is made out of pre-calculated constants and intermediate results, the calculations associated with each step are:

Step 1: 5 additions/subtractions and 3 multiplies.
Step 2: 2 additions and 3 multiplies.
Step 3: 2 compares (1 if origin inside sphere).
Step 4: 2 additions/subtractions and 1 multiply.
Step 5: 1 compare (none if origin inside sphere).
Step 6: 1 addition/subtraction and 1 square root.
Step 7: 3 additions, 3 multiplies.
Step 8: 3 subtractions, 3 multiplies.

At worst this gives a total of 16 additions/subtractions, 13 multiplies, 1 square root, and 3 compares. Note that this is less than our original method, and that a determination of when the ray misses the sphere can take place after fewer calculations.

## Example

Given a ray with an origin at [1 −2 −1] and a direction vector of [1 2 4], find the intersection point with a sphere of radius $S_r = 3$ centered at [3 0 5].

As before, first normalize the direction vector, which yields:

$$\begin{aligned}&\text{direction vector magnitude} = \sqrt{(1*1 + 2*2 + 4*4)} = \sqrt{21}\\ &\mathbf{R_d} = [1/\sqrt{21} \quad 2/\sqrt{21} \quad 4/\sqrt{21}]\\ &\quad = [0.218\ 0.436\ 0.873].\end{aligned}$$

First find the ray to the center and its length squared (A9):

$$\begin{aligned}\mathbf{OC} &= [3\ 0\ 5] - [1\ -2\ -1]\\ &= [2\ 2\ 6]\\ L_{2oc} &= [2\ 2\ 6] \cdot [2\ 2\ 6]\\ &= 44.\end{aligned}$$

Checking if $L_{2oc} \geq S_r^2$, it is found that the ray originates outside the sphere. Now calculate the closest approach along the ray to the sphere's center (A10):

$$\begin{aligned}t_{ca} &= [2\ 2\ 6] \cdot [0.218\ 0.436\ 0.873]\\ &= 6.546.\end{aligned}$$

Checking if $t_{ca} < 0$, it is found that the center of the sphere lies in front of the origin, so calculation must continue. Calculate the half chord distance squared (A13):

$$\begin{aligned}t_{2hc} &= 3*3 - 44 + 6.546*6.546\\ &= 7.850.\end{aligned}$$

$t_{2hc} > 0$, so the ray must hit the sphere. The intersection distance is then, by (A14):

$$\begin{aligned} t &= 6.546 - \sqrt{7.850} \\ &= 3.744 \end{aligned}$$

This is the same answer calculated for $t_0$ in the earlier algebraic example. As before, the intersection point is, by (A7):

$$\begin{aligned} \mathbf{r}_i &= [X_0 + X_d * t \;\; Y_0 + Y_d * t \;\; Z_0 + Z_d * t] \\ &= [1 + 0.218 * 3.744 \quad -2 + 0.436 * 3.744 \quad -1 + 0.873 * 3.744] \\ &= [1.816 \quad -0.368 \quad 2.269] \end{aligned}$$

The unit vector normal is, by (A8):

$$\begin{aligned} \mathbf{r}_n &= \left[\frac{(x_i - X_c)}{S_r} \; \frac{(y_i - Y_c)}{S_r} \; \frac{(z_i - Z_c)}{S_r}\right] \\ &= [(1.816 - 3)/3 \quad (-0.368 - 0)/3 \quad (2.269 - 5)/3] \\ &= [-0.395 \quad -0.123 \quad -0.910] \end{aligned}$$

## 2.3 Comparison of Algebraic and Geometric Solutions

The algebraic solution is certainly valid, and is fairly close to the geometric solution in number of operations. The strength of the geometric solution lies in its timely use of comparisons. The first geometric test is to find whether the ray is outside and pointing away from the sphere. This test is pretty worthwhile, considering that a randomly placed ray will face away from a sphere half of the time. The original algebraic algorithm does not include this test.

The question arises of why these two solutions should be different at all. The explanation is simple enough: the algebraic algorithm is just inefficient. Compare the operations to calculate $A$, $B$, and $C$ in equation (A5) with the geometric calculations of (A9) through (A13). The following relationships can be identified:

$$\begin{aligned} B &= -2 * t_{ca} \\ C &= L_{2oc} - S_r^2. \end{aligned} \tag{A15}$$

With these in hand, equation (A14) becomes:

$$\begin{aligned} t &= t_{ca} \pm \sqrt{t_{2hc}} \\ &= -B/2 \pm \sqrt{((-B/2)^2 - C)} \\ &= \frac{-B \pm \sqrt{(B^2 - 4 * C)}}{2}. \end{aligned}$$

This is the algebraic solution (A6). However, the algebraic solution is still more complicated than the geometric, as there is still a negation, a multiplication by 4 and a division by 2. Looking at equation (A5), we see that $B$ is calculated by a multiplication by 2. Instead, calculate $NB$, which is set to $-B/2$. Substituting $-2 * NB$ for $B$ in (A6) and simplifying:

$$\begin{aligned} t &= \frac{-(-2 * NB) \pm \sqrt{((-2 * NB)^2 - 4 * C)}}{2} \\ &= \frac{-(-2 * NB) \pm 2 * \sqrt{(NB^2 - C)}}{2} \\ &= NB \pm \sqrt{(NB^2 - C)}. \end{aligned} \tag{A16}$$

These equations are almost as clean as geometric equation (A14), except for the ' $\pm$ ' operation. From substituting $-2 * NB$ for $B$ in (A5), $NB$ is simply:

$$NB = X_d * (X_c - X_0) + Y_d * (Y_c - Y_0) + Z_d * (Z_c - Z_0).$$

Note that $NB$ is equivalent to $t_{ca}$ (equation (A10)).

To eliminate the ' $\pm$ ' problem, where two values $t_0$ and $t_1$ must be calculated for the sphere and the smaller positive value accepted, we need to look deeper. A flaw in the algebraic solution was a lack of a way to cut the processing short. By the equivalences in (A15) we can tell the ray origin lies outside the sphere only if $C > 0$. This eliminates the need for calculating $t_0$ and $t_1$, as the criterion of (A14) can be used to know whether to subtract or add the discriminant from $NB$.

Similarly, the ray must point away from the sphere's center if $NB < 0$. This fact gives a complete equivalence of the two algorithms. The algebraic solution originally did not have a number of useful features. Using insight into the geometry of the situation, a better algorithm was found. Looking back on the algebraic solution, the efficiencies inherent in the situation became clear.

The point here is that studying the nature of the problem can yield algorithmic speed-ups. The algebraic solution was straightforward, but it was aimed at solving the general problem of finding the intersection points of a line

and a sphere. The geometric approach homed in on the special characteristics of the ray (i.e. that a ray defines only part of a line) and the requirements of the problem (i.e. that only the closest intersection point is required).

## 2.4 Precision Problems

Doing floating-point calculations is like moving piles of sand around. Every time you move a pile you lose a little sand and pick up a little dirt [5]. Imprecision can cause a number of errors which must be addressed. A discussion of general numerical problems in computer graphics appears in [5]. What follows is a brief discussion of a common problem to all ray tracing intersection routines.

In ray tracing often the origin of the ray $\mathbf{R}_0$ is a point on the sphere itself. Theoretically, $t = 0$ for these points, which are ignored by testing for this condition. However, in practice, calculational imprecision will creep in and throw these tests off. This imprecision will cause rays shot from the surface to hit the surface itself. Computationally what occurs is that $t$'s are found which are very close to, but not necessarily equal to, zero. If uncorrected, those larger than zero will be considered valid intersections. The result is the nonsensical situation in which a small surface area is shadowed by itself. This problem is shown in *Figure 4*. The practical effect of this imprecision is a case of 'surface acne.' The surface will sometimes shadow itself, causing blotches and spots to appear. Some method of coping with this imprecision is necessary to clean up this problem. The discussion below also applies to any other primitive intersected, as all surfaces have this potential problem.

One method to avoid imprecision is to pass a flag telling whether the origin

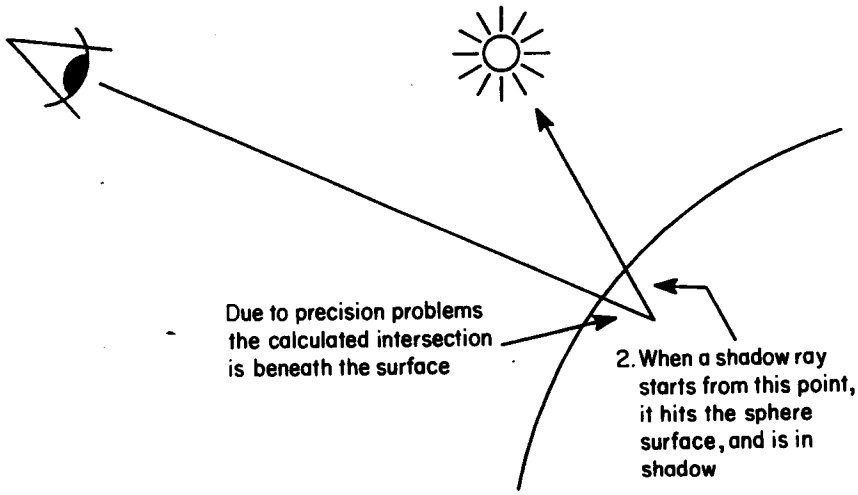


Fig. 4. Problem in surface intersection.

is actually on the sphere. In ray tracing, the last intersection point is known, so the procedure can be informed that the ray starts on the surface. However, if the sphere is a transmitter some testing must be done to allow refraction rays to pass through the sphere and hit its other side. The same problem arises with reflections from the inside of the sphere. In these cases the $t_1$ solution is the valid answer.

A simple solution is to check if $t$ is within some tolerance. For example, if $\text{abs}(t) < 0.00001$, then that $t$ describes the origin as being on the sphere. Scaling this tolerance to the size of the environment is advisable. For example, if the spheres were atoms and the radii were expressed in meters, 0.00001 meters would be much larger than any atom. Choosing these tolerances can be done empirically or, more accurately, by numerical methods for error analysis. For example, the tolerance could also be based on the radius of the sphere intersected.

Root polishing methods may also be useful in solving imprecision problems. For example, say a ray is traced and the $t$ of the closest object (i.e. the object the ray first hits) is found. Find this intersection point (equation (A7)) and use this as the origin of a new ray which uses the same direction. By intersecting the sphere with this new ray and accepting the solution for $t$ closest to zero (even if $t$ is negative), a more accurate intersection point can be found. While $t$ is greater than some given tolerance this procedure is repeated. This method does not eliminate the need for a tolerance factor, but it does allow the programmer to be confident that the intersection point is within a certain distance of the surface.

A fourth solution is to move the intersection point outside or inside the sphere as needed. That is, when the intersection point is found and new rays are spawned, assure that the new origins are on the proper sides of the surface. This can be done by moving each new ray's origin along the normal until it is found to be on the proper side of the sphere. This involves testing if the point is inside or outside by substituting the intersection point into the sphere equation and checking on which side of the surface the point lies (which is done by checking the sign of the surface expression). If not on the desired side, the point is moved by some tolerance along the normal, then tested again. Note that reflection and shadow rays will always move positively along the normal, refraction rays negatively. This method assures that the origin of the spawned ray will be on the correct side of the sphere, so that the ray will not intersect the sphere.

All of the above methods will work to varying degrees. If possible, the first method should be implemented as it is practically foolproof (almost tangent rays can sometimes have problems; however, these are rare). For spheres and other quadrics this is possible. If not, then some design decisions have to be made to choose the solution proper to the application.

## 2.5 Spherical Inverse Mapping

Once an intersection point and normal are found on a sphere, further operations may be desired. A common shading trick is texture mapping, in which the position of the intersection point on the sphere's surface is used to vary the surface characteristics [2]. For example, say a globe is to be rendered, and there is a map of the world stored in the computer. Each time the sphere is intersected the proper color is found on the map and used to color that pixel.

The problem is simply to convert the intersection point into a longitude and latitude. The derivation is fairly straightforward, though it involves some time-consuming trigonometric operations.

The input to this process is the normal $\mathbf{S}_n$ (A8) at the point of intersection $\mathbf{R}_i$ (A7) and the following description of the sphere and its axes:

$$\begin{aligned} &\mathbf{S}_{pole} \equiv \mathbf{S}_p \equiv [X_p\ Y_p\ Z_p] \\ &\mathbf{S}_{equator} \equiv \mathbf{S}_e \equiv [X_e\ Y_e\ Z_e] \\ &\text{by definition, } \mathbf{S}_p \cdot \mathbf{S}_e = 0 \text{ (i.e. are perpendicular).} \end{aligned} \tag{B1}$$

$\mathbf{S}_p$ is a unit vector which points from the sphere's center to the north pole of the sphere. $\mathbf{S}_e$ is a unit vector which points to a reference point on the equator. The parameter $u$ varies along the equator from zero to one. It is traced in the standard direction of the coordinate system used (e.g. if the right hand coordinate system is used, then it varies counterclockwise around the equator). At the poles, define $u$ to be zero. The parameter $v$ varies from zero to one from the south pole to the north (technically speaking, $-\mathbf{S}_p$ to $+\mathbf{S}_p$). This mapping is shown in *Figure 5*.

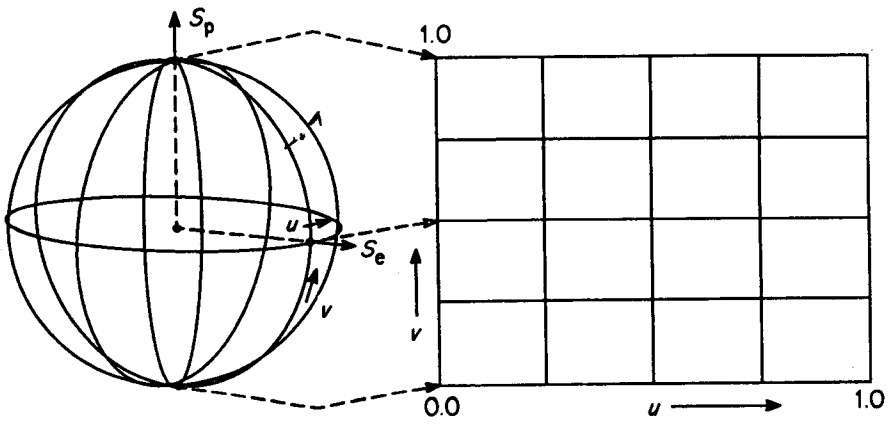


Fig. 5. Inverse mapping for a sphere.

The point of intersection's normal $\mathbf{r}_n$ is the same as the unit vector formed by the center and the intersection point.

From these definitions, first obtain the latitudinal parameter. This is equal to the arccosine of the dot product between the intersection's normal and the north pole:

$$\begin{aligned} \phi &= \arccos\left(-\mathbf{S}_n \cdot \mathbf{S}_p\right) \\ v &= \phi/\pi. \end{aligned} \tag{B2}$$

Note the division by $\pi$ can be changed into a multiplication for extra speed. If $v$ is equal to zero or one, then $u$ is defined to be equal to zero. Otherwise calculate the longitudinal parameter:

$$\theta = \frac{\arccos\left((\mathbf{S}_e \cdot \mathbf{S}_n)/\sin(\phi)\right)}{2 * \pi}. \tag{B3}$$

Now take the cross product of the two sphere axes defining angles and compare this direction with the direction of the normal:

$$\begin{aligned} &\text{if } ((\mathbf{S}_p \otimes \mathbf{S}_e) \cdot \mathbf{S}_n) > 0 \\ &\quad \text{then } u = \theta; \\ &\quad \text{else } u = 1 - \theta. \end{aligned} \tag{B4}$$

Note that the cross product can be pre-calculated once in advance. The effect of this test is to determine which side of the $\mathbf{S}_e$ vector the intersection point lies upon.

## Example

Begin with an intersection point normal $\mathbf{S}_n = [0.577 \;\; -0.577 \;\; 0.577]$ on a sphere whose axes are:

$$\begin{aligned} \mathbf{S}_p &= [0\ 0\ 1] \\ \mathbf{S}_e &= [1\ 0\ 0]. \end{aligned}$$

From these first find the latitudinal parameter (B2):

$$\begin{aligned} \phi &= \arccos\left(-[0\ 0\ 1] \cdot [0.577 \;\; -0.577 \;\; 0.577]\right) = 2.186 \\ v &= 2.186/3.14159 = 0.696. \end{aligned}$$

The longitudinal parameter calculations are (B3):

$$\theta = \frac{\arccos\left([1\ 0\ 0] \cdot [0.577\ \ -0.577\ 0.577] / \sin(2.186)\right)}{2 * 3.14159}$$

$$= 0.125.$$

Now test which side of the axis $\mathbf{S_e}$ the point is on (B4):

$$([0\ 0\ 1] \otimes [1\ 0\ 0]) \cdot [0.577\ \ -0.577\ 0.577]) = \ -0.577.$$

This value is less than 0, so:

$$u = 1 - 0.125 = 0.875.$$

The final answer is then $(u, v) = (0.875, 0.696)$.

# 3 RAY/PLANE ALGORITHMS

This section consists of algorithms which deal with intersecting a ray with a polygon. First the ray/plane intersection itself is presented. Next is an algorithm for testing whether the intersection point is inside a polygon on the plane. Mapping onto polygons is also discussed.

## 3.1 Ray/Plane Intersection

Define a ray in terms of its origin and a direction vector:

$$\mathbf{R}_{\text{origin}} \equiv \mathbf{R}_0 \equiv [X_0\ Y_0\ Z_0]$$
$$\mathbf{R}_{\text{direction}} \equiv \mathbf{R}_d \equiv [X_d\ Y_d\ Z_d]$$
$$\text{where } X_d^2 + Y_d^2 + Z_d^2 = 1 \text{ (i.e. normalized)}$$

which defines a ray as:

$$\text{set of points on ray } \mathbf{R}(t) = \mathbf{R}_0 + \mathbf{R}_d * t, \text{ where } t > 0. \qquad \text{(C1)}$$

The ray direction does not need to be normalized for these calculations. However, such normalization is recommended, otherwise $t$ will represent the distance in terms of the length of the direction vector.

Define the plane in terms of $[A\ B\ C\ D]$, which defines the plane as:

$$\begin{aligned} &\text{Plane} \equiv A * x + B * y + C * z + D = 0 \\ &\text{where } A^2 + B^2 + C^2 = 1. \end{aligned} \tag{C2}$$

The unit vector normal of the plane is defined as:

$$\mathbf{P}_{\text{normal}} \equiv \mathbf{P}_n = [A\ B\ C]$$

and the distance from the coordinate system origin [0 0 0] to the plane is simply $D$. The sign of $D$ determines which side of the plane the system origin is located. This is the implicit formulation of the plane.

The distance from the ray's origin to the intersection with the plane $P$ is derived by simply substituting the expansion of equation (C1) into the plane equation (C2):

$$A * (X_0 + X_d * t) + B * (Y_0 + Y_d * t) + C * (Z_0 + Z_d * t) + D = 0$$

and solving for $t$:

$$t = \frac{-(A * X_0 + B * Y_0 + C * Z_0 + D)}{A * X_d + B * Y_d + C * Z_d}. \tag{C3}$$

In vector notation, this equation is:

$$t = \frac{-(\mathbf{P}_n \cdot \mathbf{R}_0 + D).}{\mathbf{P}_n \cdot \mathbf{R}_d} \tag{C4}$$

To use (C3) more efficiently, first calculate the dot product:

$$v_d = \mathbf{P}_n \cdot \mathbf{R}_d = A * X_d + B * Y_d + C * Z_d.$$

If $v_d = 0$, then the ray is parallel to the plane and no intersection occurs. Admittedly, a ray could be in the same plane, but this case is irrelevant in practice; hitting a polygon edge-on has no effect on rendering. Also, if $v_d \geq 0$, the normal of the plane is pointing away from the ray. If the modelling system uses one-sided planar objects, testing could end here, as the plane is culled. If the ray passes these tests, calculate the second dot product:

$$v_0 = -(\mathbf{P}_n \cdot \mathbf{R}_0 + D) = -(A * X_0 + B * Y_0 + C * Z_0 + D). \tag{C6}$$

Now calculate the ratio of the dot products:

$$t = v_0/v_d \tag{C7}$$

If $t < 0$, then the line defined by the ray intersects the plane behind the ray's origin and so no actual intersection occurs. Else, calculate the intersection point:

$$\mathbf{r}_i = [x_i\ y_i\ z_i] = [X_0 + X_d * t \quad Y_0 + Y_d * t \quad Z_0 + Z_d * t]. \tag{C8}$$

Usually, the surface normal desired is for the surface facing the ray, and so the sign of the normal vector $\mathbf{P}_n$ may be adjusted depending on its relationship with the direction vector $\mathbf{R}_d$. The sign of the normal should be reversed in order to point back toward the ray origin.

$$\begin{aligned} &\text{If } \mathbf{P}_n \cdot \mathbf{R}_d < 0 \\ &\text{(in other words, if } v_d > 0) \\ &\quad \text{then } \mathbf{r}_n = \mathbf{P}_n; \\ &\quad \text{else } \mathbf{r}_n = -\mathbf{P}_n. \end{aligned} \tag{C9}$$

For those with memory to burn, the reversed normal could be pre-computed and saved.

To summarize, the steps in the algorithm are:

Step 1: calculate $v_d$ and compare it to zero.
Step 2: calculate $v_0$ and $t$ and compare $t$ to zero.
Step 3: compute intersection point.
Step 4: compare $v_0$ to zero and reverse normal.

Assuming the most is made out of pre-calculated constants and intermediate results, the calculations associated with each step are:

Step 1: 2 additions, 3 multiplies, and 1 compare.
Step 2: 3 additions, 3 multiplies, and 1 compare
Step 3: 3 additions and 3 multiplies.
Step 4: 1 compare.

This gives a total of 8 additions/subtractions, 9 multiplies, and 3 compares for the worst case.

## Example

Given a plane [1 0 0 −7] (which describes a plane where $x = 7$) and a ray with an origin of [2 3 4] and a direction of [0.577 0.577 0.577], find the intersection with a plane. Assume the plane is two-sided.

First calculate $v_d$ by (C5):

$$v_d = 1 * 0.577 + 0 * 0.577 + 0 * 0.577 = 0.577.$$

In this case, $v_d > 0$, so the plane points away from the ray. For this example the plane has two sides, so in this case there is no early termination. Calculate $v_0$:

$$v_0 = -(1 * 2 + 0 * 3 + 0 * 4 + (-7)) = 5.$$

Now calculate $t$:

$$t = 5/0.577 = 8.66.$$

Distance $t$ is positive, so the point is not behind the ray. This value represents the distance from the ray's origin to the intersection point. The intersection point components are:

$$\begin{aligned} x_i &= 2 + 0.577 * 8.66 = 7 \\ y_i &= 3 + 0.577 * 8.66 = 8 \\ z_i &= 4 + 0.577 * 8.66 = 9. \end{aligned}$$

So $\mathbf{R}_i = [7\ 8\ 9]$. To determine whether the plane's normal points in a direction towards the ray's origin, check if $v_d > 0$. It is, which means that the plane faces away from the ray. Simply negating the normal will give a normal which faces towards the ray, i.e. $[-1\ 0\ 0]$.

## 3.2 Polygon Intersection

This section deals with finding if a point on a plane is inside a polygon on that plane. The polygon is assumed to be entirely within the plane. The plane equation is assumed to be known. If the plane equation is not given, it must be derived. See Rogers' excellent book [12] for methods of deriving the normal.

### Point/polygon inside/outside testing

Once the plane equation is derived, ray/polygon intersection can be performed. After calculating the ray/plane intersection, the next step is to determine if the intersection point is inside the polygon.

A number of different methods are available to solve this problem. Berlin [1] gives a good overview of some techniques. The method presented here is a modified version of the 'ray intersection' algorithm presented in [14]. This

algorithm works by shooting a ray in an arbitrary direction from the intersection point and counts the number of line segments crossed. If the number of crossings is odd, the point is inside the polygon; else it is outside. This is known as the Jordan curve theorem. *Figure 6* depicts the use of this theorem. The modified algorithm presented below elegantly handles the special cases where the test ray intersects a vertex in the polygon. It is my own invention, and appears to be an optimal solution.

Define the polygon as a set of $N$ points:

$$\text{polygon} \equiv \text{set of } \mathbf{G}_n = [X_n\ Y_n\ Z_n], \text{ where } n = (0, 1, \ldots, N-1).$$

The plane defined by these points is:

$$\text{plane} \equiv A * X + B * Y + C * Z + D = 0. \tag{D1}$$

The (not necessarily normalized) normal of the plane is defined as:

$$\mathbf{P}_{\text{normal}} \equiv \mathbf{P}_{\text{n}} = [A\ B\ C]$$

Begin with an intersection point:

$$\mathbf{R}_{\text{i}} \equiv [X_{\text{i}}\ Y_{\text{i}}\ Z_{\text{i}}]$$

which is given as being on the plane $[A\ B\ C\ D]$.

The first step is to project the polygon onto a two-dimensional plane. In this plane all points are specified by a pair $(U, V)$. So, all that is desired is a $(U, V)$

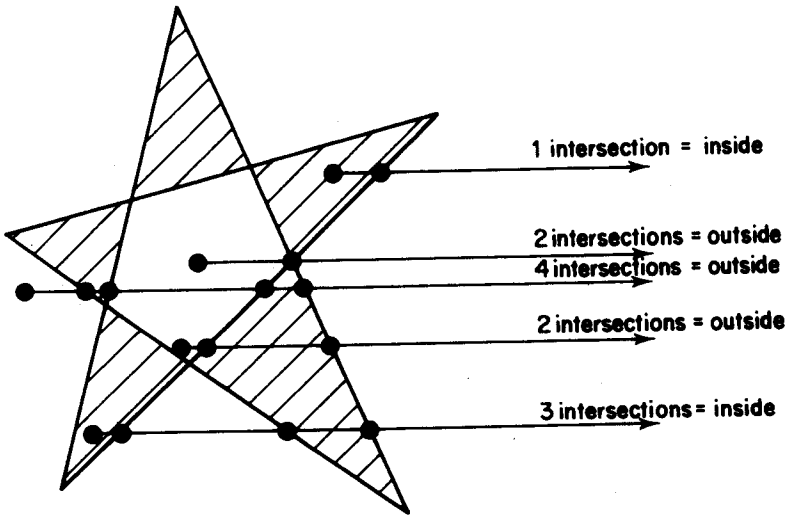


Fig. 6. Jordan curve theorem.

pair for each $[X\ Y\ Z]$ coordinate, such that the topology of the situation is unchanged.

One method would be to rotate around some axis until the normal became parallel to some other axis (say $Z$). After this is done, the two remaining axes ($X$ and $Y$, in this case) could be used to generate the $(U, V)$ pairs. The drawback of this scheme is that a rotation matrix must be generated and stored for each polygon, and that a matrix multiply must be performed for each coordinate.

These costs can be eliminated by simply throwing away one of the $[X\ Y\ Z]$ coordinates and using the other two. This action projects the polygon onto the plane defined by the two chosen coordinates. The area of the polygon is not preserved, but the topology stays the same. Choosing which coordinate to throw away is defined as follows: throw away the coordinate whose corresponding plane equation value is of the greatest magnitude. For example, for a polygon with a $\mathbf{P_n} = [0\ -5\ 3]$ the $Y$ coordinates would be thrown away, with $X$ and $Z$ assigned to $U$ and $V$ (which is $U$ and which is $V$ is arbitrary). We'll refer to the coordinate with largest magnitude as the dominant coordinate.

Once the polygon has been projected upon a plane, the inside–outside test is fairly simple. Translate the polygon so that the intersection point is at the origin, i.e. subtract the intersection point's coordinates $(U_i, V_i)$ from each vertex. Label these new vertices as $(U', V')$. Now imagine a ray starting from this origin and proceeding along the $+U'$ axis. Each edge of the polygon is tested against the ray. If the edge crosses the ray, note this fact. If the total count of crossings is odd, the point is inside the polygon. This operation is shown in *Figure 7*.

As Berlin [1] points out, vertices exactly on the ray must be dealt with as special cases. These special cases can be avoided by defining them away. The

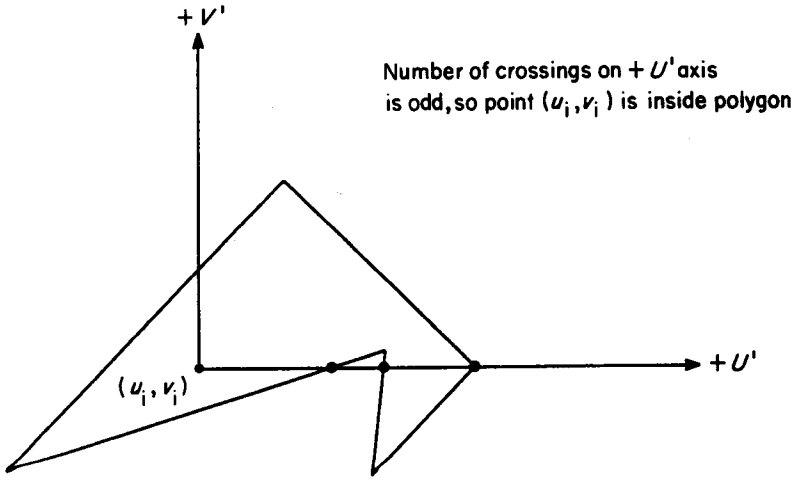


Fig. 7. Polygon inside/outside test.

ray extending along the $+U'$ axis splits the plane into two parts. However, there are also points which are on the $U'$ axis itself. The definition which must be added is to declare that vertices which lie on the ray (i.e. where $V' = 0$) are to be considered on the $+V'$ side of the plane. In this way no points actually lie on the ray, and the special cases disappear. The ray itself has to be redefined to be infinitesimally close to the original ray, but not to pass through any points. It is now a dividing line, instead of a family of points.

The algorithm is then:

For the $NV$ vertices $[X_n\ Y_n\ Z_n]$, where $n = 0$ to $NV - 1$, project these onto the dominant coordinate's plane, creating a list of vertices $(U_n, V_n)$.

Translate the $(U, V)$ polygon so that the intersection point is the origin. Call these points

$$(U_n', V_n').$$

Set the number of crossings $NC$ to zero.

Set the sign holder $SH$ as a function of $V_0'$, the $V'$ value of the first vertex of the first edge: (D2)

Set to $-1$ if $V_0'$ is negative.
Set to $+1$ if $V_0'$ is zero or positive.

For each edge of the polygon formed by points $(U_a', V_a')$ and $(U_b', V_b')$, where $a = 0$ to $NV - 1$, $b = (a + 1)$ mod $NV$:

Set the next sign holder $NSH$: (D3)

Set to $-1$ if $V_b'$ is negative.
Set to $+1$ if $V_b'$ is zero or positive.

If $SH$ is not equal to $NSH$: (D4)

If $U_a'$ is positive and $U_b'$ is positive then (D5)
the line must cross $+U'$, so $NC = NC + 1$.

Else if either $U_a'$ is positive or $U_b'$ is positive then (D6)
the line might cross, so compute intersection on $U'$ axis:

If $U_a' - V_a' * (U_b' - U_a')/(V_b' - V_a') > 0$ then (D7)
the line must cross $+U'$, so $NC = NC + 1$.

Set $SH = NSH$. (D8)

Next edge

If $NC$ is odd, the point is inside the polygon, else it is outside. (D9)

The algorithm's first test (D4) checks whether the edge crosses the $U'$ axis. If it does not, the edge can be ignored. For those edges that do cross, the

vertices are checked (D5) to see if both endpoints are on the $+U'$ part of the plane. If so, the $+U'$ axis must be crossed. Else, if either of the endpoints are in the $+U'$ part (D6), then the exact $U'$ location of where the edge hits the $U'$ axis must be found. If (D7) this $U'$ location is positive (i.e. on the $+U'$ axis), then the edge indeed crosses $+U'$.

This method is highly efficient because most edges can be trivially rejected or accepted. Only when the edge extends from diagonally opposite quadrants does any serious calculation have to be performed.

A minor problem with this and other inside–outside test algorithms is that intersection points exactly on an edge are arbitrarily determined to be inside or outside. There are solutions to this problem, but in practice intersection points on the edges are mostly irrelevant. This is because if an intersection point falls on an edge between two polygons, and both polygons are projected onto the same plane, the algorithm determines that the point is inside one and only one of these polygons (regardless of precision error).

## Example

Given a triangle:

$$\mathbf{G}_0 = [-3\ -3\ 7]$$
$$\mathbf{G}_1 = [\ 3\ -4\ 3]$$
$$\mathbf{G}_2 = [\ 4\ -5\ 4]$$

and an intersection point $\mathbf{R}_i = [-2\ -2\ 4]$, find if the point lies within the triangle. The plane equation is:

$$\mathbf{P} = [1\ 2\ 1\ -2]$$

The dominant coordinate in the plane equation is $Y$, so these coordinates are discarded leaving:

$$\mathbf{G}_{uv0} = [-3\ 7]$$
$$\mathbf{G}_{uv1} = [\ 3\ 3]$$
$$\mathbf{G}_{uv2} = [\ 4\ 4]$$
$$\mathbf{R}_{uvi} = [-2\ 4]$$

The situation at this point is shown in *Figure 8*.

Translating the intersection point $[-2\ 4]$ to the coordinate system origin, the triangle is now:

$$\mathbf{G}'_{uv0} = [-1\ \ 3]$$
$$\mathbf{G}'_{uv1} = [\ 5\ -1]$$
$$\mathbf{G}'_{uv2} = [\ 6\ \ 0].$$

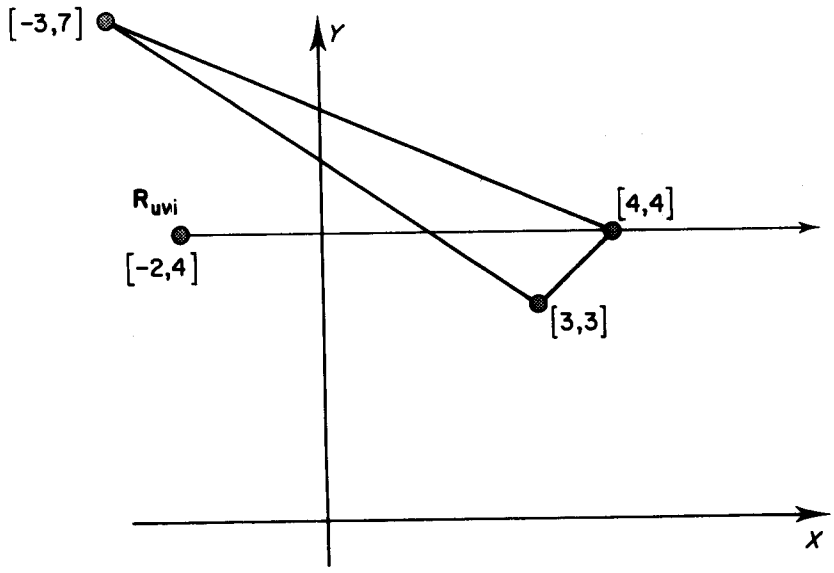


Fig. 8. Inside/outside test example.

The first edge is defined by $(U'_a, V'_a) = (-1, 3)$, $(U'_b, V'_b) = (5, -1)$. The sign holder *SH* is +1 since $V'_a$ is positive.

*NSH* is −1 since $V'_b$ is negative. The first edge passes (D4), since *SH* and *NSH* don't match. Since $U'_a$ is positive and $U'_b$ is not, (D5) fails and (D6) passes, so the true intersection point must be calculated (D7):

$$U'_a - V'_a * (U'_b - U'_a)/(V'_b - V'_a) \equiv -1 - 3 * (5 - (-1))/(-1 - 3) = 3.5.$$

This means that the intersection point is on the $+U'$ axis at 3.5. By the (D7) test, this is considered to be a crossing, and so *NC* is incremented to 1. At the end of testing *SH* is changed to −1 by being set to *NSH* (D8).

The second edge is defined by $(U'_a, V'_a) = (5, -1)$, $(U'_b, V'_b) = (6,0)$. By (D3), *NSH* is +1, and since *SH* doesn't match *NSH*, (D4) is passed, so the line segment must intersect the $U'$ axis. $U'_a$ and $U'_b$ are positive, so by (D5) a crossing takes place, and *NC* is incremented to 2. *SH* is set to +1 by *NSH* (D8).

The third edge is defined by $(U'_a, V'_a) = (6, 0)$, $(U'_b, V'_b) = (-1, 3)$. *NSH* is +1, and since *SH* matches *NSH*, no crossing takes place.

*NC* ends as 2, which is an even number, so the point is decided to be outside the polygon. Note how the vertex $(U', V') = (6, 0)$ lay on the $+U'$ axis, and how it was dealt with by considering the vertex to be consistently above the $+U'$ axis ray.

## Winding number testing

In *Figure 6* the center pentagon of the star is not considered inside the star, as

the number of crossings is even. An alternate definition of the polygon is to consider these points to be inside the polygon.

To perform the inside–outside test for this class of polygons requires a simple change to the previous algorithm. The change is to increment *NC* when the edge crossing the $+U'$ axis passes from $+V'$ to $-V'$, and decrement *NC* when it passes from $-V'$ to $+V'$. If *NC* is 0, the point is outside the polygon, else it's inside.

The number *NC* is called the *winding number*. Imagine the polygon is made of string, and a pencil point is put on the intersection point. If the string is pulled taut, the winding number is how many times the string goes around the point. The sign of *NC* is the direction of the rotation: ' + ' is clockwise, ' – ' counterclockwise.

## 3.3 Convex Quadrilateral Inverse Mapping

Once an intersection point has been found within a polygon, a number of other operations can be performed. If the polygon has been assigned a color pattern, the color at the intersection point must be retrieved. Similar operations must be performed for other texture mapping procedures, such as bump maps. If the polygon is a patch on a curved surface, the exact normal must be derived from the differing normals of the vertices.

This section will present the algorithm for obtaining the location of a point within a convex quadrilateral, since this shape is frequently used in a variety of applications. The parametric values $(u, v)$ are calculated by the algorithm. This coordinate pair represents the location of the point with respect to the four edges, taken as pairs of coordinate axes ranging from 0 to 1. The problem is shown in *Figure 9*.

Note that the mapping itself can also be used as an inside–outside test. If the point is outside the quadrilateral, the $(u, v)$ pair(s) calculated will fall outside the range $(0..1, 0..1)$.

Begin with an intersection point on the quadrilateral's plane:

$$\mathbf{R}_i = [X_i \; Y_i \; Z_i]$$

and a convex quadrilateral defined by four points:

$$\text{Quadrilateral} \equiv \text{set of } \mathbf{P}_{uv}, \text{ where } u = 0, 1 \text{ and } v = 0, 1, \text{ with} \quad \mathbf{P}_{uv} = [X_{uv} \; Y_{uv} \; Z_{uv}] \tag{E1}$$

The points define the axes of $u$ and $v$, e.g. $(\mathbf{P}_{00}, \mathbf{P}_{10})$ defines the $u$ axis at $v = 0$. The normal of the plane (which does not have to be normalized) is called $\mathbf{P}_n$.

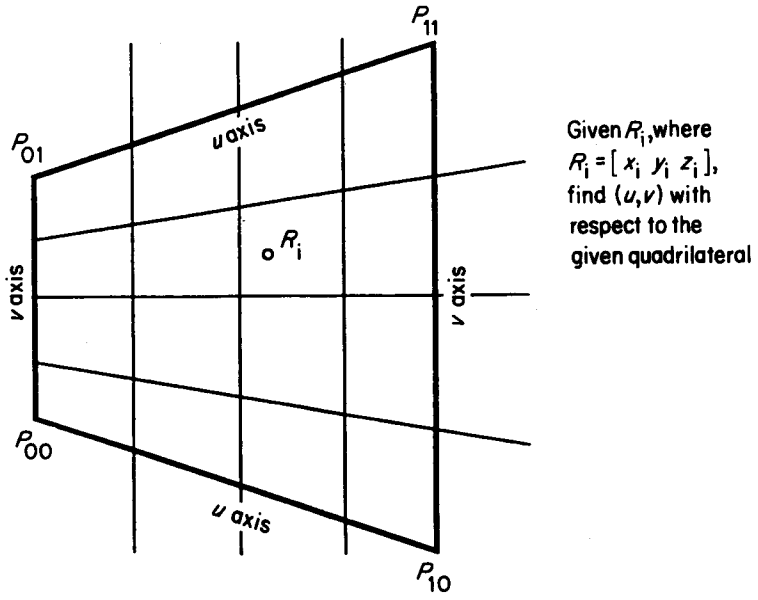


Fig. 9. Quadrilateral inverse mapping.

The derivation is rather involved, so will not be included in this discussion. It is fully covered in [15]. A number of factors must be calculated for the interpolation. These factors divide into two classes: point-plane dependent and plane dependent. Those which are plane dependent can (and should, unless there are other limiting factors such as memory space) be calculated in advance and passed to the algorithm. These plane-dependent factors are:

$$\begin{aligned} D_{u0} &= \mathbf{N}_c \cdot \mathbf{P}_d \\ D_{u1} &= \mathbf{N}_a \cdot \mathbf{P}_d + \mathbf{N}_c \cdot \mathbf{P}_b \\ D_{u2} &= \mathbf{N}_d \cdot \mathbf{P}_b \\ \mathbf{N}_a &= \mathbf{P}_a \otimes \mathbf{P}_n \\ \mathbf{N}_c &= \mathbf{P}_c \otimes \mathbf{P}_n \end{aligned} \tag{E2}$$

where:

$$\begin{aligned} \mathbf{P}_a &= \mathbf{P}_{00} - \mathbf{P}_{10} + \mathbf{P}_{11} - \mathbf{P}_{01} \\ \mathbf{P}_b &= \mathbf{P}_{10} - \mathbf{P}_{00} \\ \mathbf{P}_c &= \mathbf{P}_{01} - \mathbf{P}_{00} \\ \mathbf{P}_d &= \mathbf{P}_{00}. \end{aligned}$$

The basic idea is to define a function for $u$ describing the distance of the perpendicular plane (defined by that $u$ and the quadrilateral's axes) from the

coordinate system origin:

$$D(u) = (\mathbf{N}_c + \mathbf{N}_a * u) \cdot (\mathbf{P}_d + \mathbf{P}_b * u). \tag{E3}$$

The factors computed in (E2) are used to represent this plane-dependent equation. Given $\mathbf{r}_i$, the distance of the perpendicular plane containing this point is:

$$D_r(u) = (\mathbf{N}_c + \mathbf{N}_a * u) \cdot \mathbf{R}_i. \tag{E4}$$

Setting $D(u)$ equal to $D_r(u)$, solving for $u$ and simplifying:

$$A * u^2 + B * u + C = 0, \tag{E5}$$

where

$$\begin{aligned} A &= D_{u2} \\ B &= D_{u1} - (\mathbf{R}_i \cdot \mathbf{N}_a) \\ C &= D_{u0} - (\mathbf{R}_i \cdot \mathbf{N}_c). \end{aligned}$$

This is simply a quadratic equation, the solution of which is straightforward and so will not be shown. To gain further efficiency, some other factors are worth computing once for each quadrilateral and storing. Note that these can be calculated only when $D_{u2} \neq 0$. These factors are:

$$\begin{aligned} \mathbf{Q}_{ux} &= \mathbf{N}_a/(2 * D_{u2}) \\ D_{ux} &= -D_{u1}/(2 * D_{u2}) \\ \mathbf{Q}_{uy} &= -\mathbf{N}_c/D_{u2} \\ D_{uy} &= D_{u0}/D_{u2}. \end{aligned} \tag{E6}$$

With the nine factors from (E2) and (E6) the value of $u$ can be calculated. The solution takes two forms, dependent upon whether the $u$ axes are parallel. Determine whether the axes are parallel by the condition:

$$\text{If } D_{u2} = 0, \text{ then the '}u\text{' axes are parallel.} \tag{E7}$$

If the axes are parallel, the solution is:

$$u_p = -C/B = (\mathbf{N}_c \cdot \mathbf{R}_i - D_{u0})/(D_{u1} - \mathbf{N}_a \cdot \mathbf{R}_i). \tag{E8}$$

If not parallel, calculate the following:

$$\begin{aligned} K_a &= D_{ux} + (\mathbf{Q}_{ux} \cdot \mathbf{R}_i) \\ K_b &= D_{uy} + (\mathbf{Q}_{uy} \cdot \mathbf{R}_i). \end{aligned}$$

There are two answers:

$$\begin{aligned} u_0 &= K_a - \sqrt{(K_a^2 - K_b)} \\ u_1 &= K_a + \sqrt{(K_a^2 - K_b)}. \end{aligned} \tag{E9}$$

At most one value of these two will lie in the range (0..1), so it is the useful value. If the final $u$ value does not lie in (0..1), then the point is outside of the quadrilateral. One quick test is to test if $K_a$ is less than the discriminant. If it is not, calculate $u_0$, else $u_1$. Then check the final $u$ value to see if it is less than 1. Note that if both $u_0$ and $u_1$ are in the valid range, then the quadrilateral is not convex.

The value $v$ can be calculated in a similar fashion. The corresponding factors of (E2) are:

$$\begin{aligned} D_{v0} &= \mathbf{N}_b \cdot \mathbf{P}_d \\ D_{v1} &= \mathbf{N}_a \cdot \mathbf{P}_d + \mathbf{N}_b \cdot \mathbf{P}_c \\ D_{v2} &= \mathbf{N}_a \cdot \mathbf{P}_c \\ \mathbf{N}_a &= \mathbf{P}_a \otimes \mathbf{P}_n \\ \mathbf{N}_b &= \mathbf{P}_b \otimes \mathbf{P}_n \end{aligned} \tag{E10}$$

and of (E6) are:

$$\begin{aligned} \mathbf{Q}_{vx} &= \mathbf{N}_a/(2 * D_{v2}) \\ D_{vx} &= -D_{v1}/(2 * D_{v2}) \\ \mathbf{Q}_{vy} &= -\mathbf{N}_b/D_{v2} \\ D_{vy} &= D_{v0}/D_{v2}. \end{aligned} \tag{E11}$$

The corresponding equations for (E7) to (E9) are formed by substituting $v$ for $u$.

The calculations needed for the point-plane-dependent process itself are, per $u$ or $v$ value, 8 additions/subtractions, 7 multiplies, 1 square root, and 4 compares.

## Example

Given a quadrilateral:

$$\begin{aligned} \mathbf{P}_{00} &= [-5 \quad 1 \quad 2] \\ \mathbf{P}_{10} &= [-2 \quad -3 \quad 6] \\ \mathbf{P}_{11} &= [\ 2 \quad -1 \quad 4] \\ \mathbf{P}_{01} &= [\ 1 \quad 4 \quad -1] \end{aligned}$$

and an intersection point [−2 −1 4], find the $(u, v)$ inverse mapping. The

plane equation is:

$$B + C - 3 = 0, \text{ so}$$
$$\mathbf{P}_n = [0\ 1\ 1].$$

The factors can be calculated from (E2):

$$\mathbf{P}_a = [-2\ -1\ \ 1]$$
$$\mathbf{P}_b = [\ 3\ -4\ \ 4]$$
$$\mathbf{P}_c = [\ 6\ \ 3\ -3]$$
$$\mathbf{P}_d = [-5\ \ 1\ \ 2]$$

so:

$$\mathbf{N}_a = [-2\ -1\ 1] \otimes [0\ 1\ 1] = [-2\ 2\ -2]$$
$$\mathbf{N}_c = [6\ 3\ -3] \otimes [0\ 1\ 1] = [6\ -6\ 6]$$
$$D_{u0} = [6\ -6\ 6] \cdot [-5\ 1\ 2] = -24$$
$$D_{u1} = [-2\ 2\ -2] \cdot [-5\ 1\ 2] + [6\ -6\ 6] \cdot [3\ -4\ 4] = 74$$
$$D_{u2} = [-2\ 2\ -2] \cdot [3\ -4\ 4] = -22.$$

The other factors are, from (E6):

$$\mathbf{Q}_{ux} = [0.0455\ \ -0.0455\ 0.0455]$$
$$D_{ux} = 1.68$$
$$\mathbf{Q}_{uy} = [0.272\ \ -0.272\ 0.272]$$
$$D_{uy} = 1.09.$$

Because $D_{u2} < 0$, the $u$ axes are not parallel. This leads to solving (E9):

$$K_a = 1.68 + [0.0455\ \ -0.0455\ 0.0455] \cdot [-2\ \ -1\ 4] = 1.82$$
$$K_b = 1.09 + [0.272\ \ -0.272\ 0.272] \cdot [-2\ \ -1\ 4] = 1.91$$

so:

$$u_0 = 1.82 - \sqrt{(1.82 * 1.82 - 1.91)} = 0.636.$$

$v$ is calculated by:

$$\mathbf{N}_a = [-2\ 2\ -2] \text{ (from before)}$$
$$\mathbf{N}_b = [3\ -4\ 4] \otimes [0\ 1\ 1] = [-8\ -3\ 3]$$
$$D_{v0} = [-8\ -3\ 3] \cdot [-5\ 1\ 2] = 43$$
$$D_{v1} = [-2\ 2\ -2] \cdot [-5\ 1\ 2] + [-8\ -3\ 3] \cdot [6\ 3\ -3] = -58$$
$$D_{v2} = [-2\ 2\ -2] \cdot [6\ 3\ -3] = 0.$$

Since $D_{v2} = 0$, the $v$ axes must be parallel. By analog of (E8):

$$v_p = ([-8\ -3\ 3] \cdot [-2\ -1\ 4] - 43)/(-58 - [-2\ 2\ -2] \cdot [-2\ -1\ 4])$$
$$= 0.231.$$

The solution is then that the points lie at $(u, v) = (0.636, 0.231)$ within the quadrilateral.

## Triangle inverse mapping

Inverse mapping can also be applied to triangles. One technique is to pass the triangle to the algorithm, doubling the last vertex in order to give the routine four points to work with. For example, if the standard routine accepts the quadrilateral's points in the order $\mathbf{P}_{00}, \mathbf{P}_{10}, \mathbf{P}_{11}, \mathbf{P}_{01}$, then the triangle's last point $\mathbf{P}_{11}$ is sent again for $\mathbf{P}_{01}$. In this case the mapping of $(u, v)$ would appear as in *Figure 10*. Note that the $u$ axes are defined as being parallel.

A special case occurs when the point to be mapped is at the doubled vertex. At this vertex, all values of one parameter converge. In the example, at $\mathbf{P}_{11}$ in *Figure 10* all $u$ values are correct. Since this singularity has no valid answer, we can choose to either consider it as an invalid point which is outside the polygon, or can assign it an arbitrary value (zero is a likely candidate). Test for this special case by checking if the divisor in equation (E8) is equal to zero.

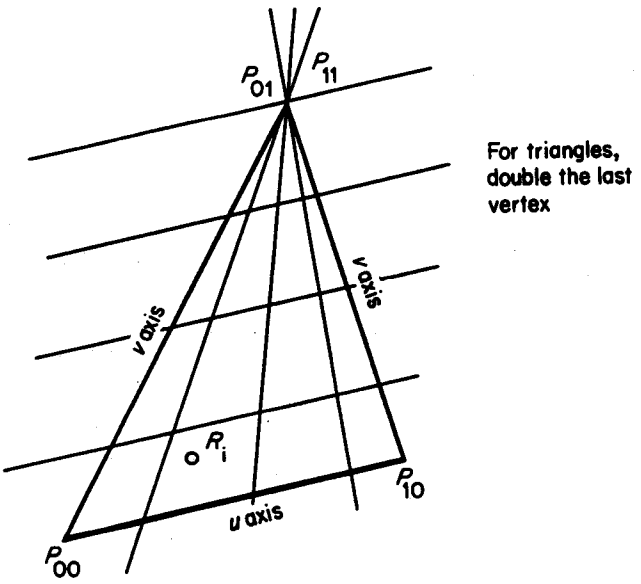


Fig. 10. Inverse mapping for a triangle.

# 4 RAY/BOX INTERSECTION

A common form used within ray tracing is the rectangular box. This primitive object is used both for objects which are visible and for bounding volumes, which are used to speed the intersection testing of complex objects.

Kay and Kajiya presented a method of handling these objects based on slabs [9]. A slab is simply the space between two parallel planes. The intersection of a set of slabs defines the bounding volume. The method relies on intersection of each pair of slabs by the ray, keeping track of the near and far intersection distances. If the largest near value is greater than the smallest far value, then the ray misses the bounding volume; otherwise, it hits.

One of the simplest finite bounding volumes is the intersection of two parallel planes each aligned so that their normals are in the same direction as the $X$, $Y$, and $Z$ axes. This configuration has a number of properties which make it efficient to test for intersection. The following algorithm uses these properties to allow quick testing of a bounding box. It is written so as to return a boolean value: TRUE if the box has hit, FALSE otherwise.

Define the orthogonal box by two coordinates:

$$\begin{aligned} &\text{box's minimum extent} \equiv \mathbf{B}_l = [X_l \; Y_l \; Z_l] \\ &\text{box's maximum extent} \equiv \mathbf{B}_h = [X_h \; Y_h \; Z_h]. \end{aligned} \tag{F1}$$

Define a ray in terms of its origin and a direction vector:

$$\begin{aligned} \mathbf{R}_{origin} &\equiv \mathbf{R}_0 = [X_0 \; Y_0 \; Z_0] \\ \mathbf{R}_{direction} &\equiv \mathbf{R}_d = [X_d \; Y_d \; Z_d] \end{aligned}$$

which defines a ray as:

$$\text{set of points on ray} \equiv \mathbf{R}(t) = \mathbf{R}_0 + \mathbf{R}_d * t \tag{F2}$$

where $t > 0$. We do not require the ray direction to be normalized for these calculations, though this normalization is desirable if the intersection distance is needed.

The algorithm is as follows, returning TRUE if the box is hit:

Set $t_{near} = -\infty$ and $t_{far} = \infty$ (i.e. arbitrarily large).

For each pair of planes $PP$ associated with $X$, $Y$, and $Z$ (shown here for the set of $X$ planes):

If the direction $X_d$ is equal to zero, then the ray is parallel to the planes, so:

If the origin $X_0$ is not between the slabs, i.e. $X_0 < X_l$ or $X_0 > X_h$, then return FALSE.

Else, if the ray is not parallel to the planes, then

begin:
Calculate intersection distances of planes:
$t_1 = (X_1 - X_0)/X_d$
$t_2 = (X_h - X_0)/X_d$

If $t_1 > t_2$, swap $t_1$ and $t_2$.
If $t_1 > t_{near}$, set $t_1 = t_{near}$.
If $t_2 < t_{far}$, set $t_2 = t_{far}$.

If $t_{near} > t_{far}$, box is missed so return FALSE.
If $t_{far} < 0$, box is behind ray so return FALSE.
end.
end of for loop.

Since the box survived all tests, return TRUE.

If the box is hit, the intersection distance is equal to $t_{near}$, and the ray's exit point is $t_{far}$. The intersection point can be calculated as shown in the 'Ray/Plane Intersection' section, equation (C8). *Figure 11* shows two cases for the intersection test. For a more efficient algorithm, unwrap the loop, expand the swap of $t_1$ and $t_2$ into two branches, and change the calculations to multiply by the inverse of the ray's direction to avoid divisions. Unwrapping the loop allows elimination of comparing $t_1$ and $t_2$ to $t_{near}$ and $t_{far}$ for the $X$ planes, as $t_{near}$ will always be set to the smaller and $t_{far}$ the larger of $t_1$ and $t_2$.

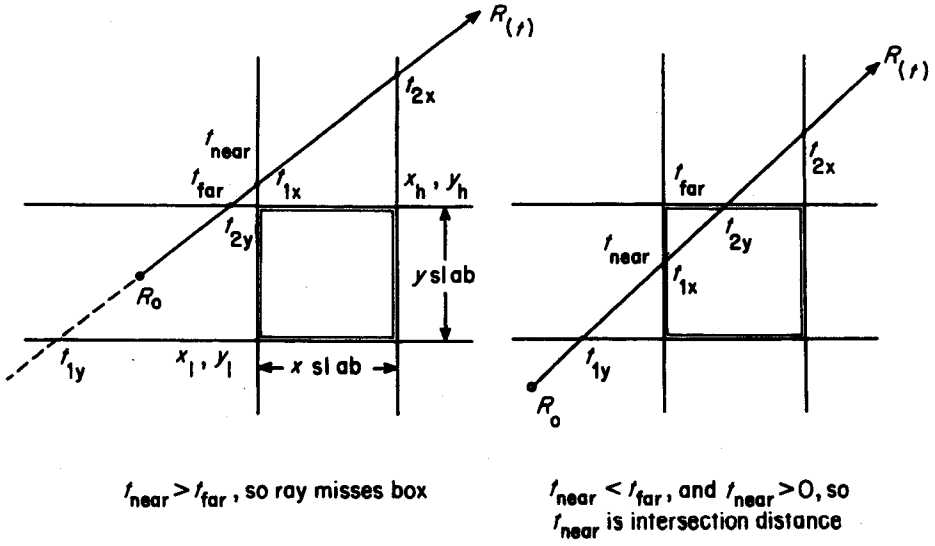


Fig. 11. Ray/box intersection testing.

### Example

Given a ray with origin [0 4 2] and direction [0.218 − 0.436 0.873] and a box with corners:

$$\mathbf{B}_l = [-1\ 2\ 1]$$
$$\mathbf{B}_h = [3\ 3\ 3]$$

find if the ray hits the box. The algorithm begins by looking at the $X$ slab, defined by $X = -1$ and $X = 3$. The distances to these are:

$$t_{1x} = (-1 - 0)/0.218 = -4.59$$
$$t_{2x} = (3 - 0)/0.218 = 13.8$$

and so set $t_{near} = -4.59$ and $t_{far} = 13.8$. Neither $t_{near} > t_{far}$ (impossible for the first slabs test) nor $t_{far} < 0$, so the $Y$ slab is examined:

$$t_{1y} = (2 - 4)/-0.436 = 4.59$$
$$t_{2y} = (3 - 4)/-0.436 = 2.29.$$

Since $t_{1y} > t_{2y}$, swap these values. Update $t_{near} = 2.29$ and $t_{far} = 4.59$. Again, neither test was failed, so check the $Z$ slab:

$$t_{1z} = (1 - 2)/0.873 = -1.15$$
$$t_{2z} = (3 - 2)/0.873 = 1.15.$$

$t_{near}$ is not updated and so is still 2.29, and $t_{far} = 1.15$. $t_{near} > t_{far}$ at this stage, so the ray must miss the box.

## 5 RAY/QUADRIC INTERSECTION AND MAPPING

A general class of objects which are relatively simple to intersect with a ray are the quadrics: cylinders, cones, ellipsoids, paraboloids, hyperboloids, etc. Spheres and planes are special subclasses of this family of objects. For reasons of efficiency, such simple objects are often given their own intersection routines. For example, see [13] for a quicker cylinder intersection method. This section will cover the generalized intersection of these objects. Again, a parametric ray formulation and an implicit surface equation are used to solve the intersection problem. Standard mappings are discussed at the section's end.

## 5.1 Ray/Quadric Intersection

The technique for intersection is to use the ray equation:

$$\mathbf{R}_{\text{origin}} \equiv \mathbf{R}_0 \equiv [X_0\ Y_0\ Z_0]$$
$$\mathbf{R}_{\text{direction}} \equiv \mathbf{R}_d \equiv [X_d\ Y_d\ Z_d]$$
$$\text{where } X_d^2 + Y_d^2 + Z_d^2 = 1 \text{ (i.e. normalized)}$$

which defines a ray as:

$$\text{set of points on line } \mathbf{R}(t) = \mathbf{R}_0 + \mathbf{R}_d * t, \text{ where } t > 0. \tag{G1}$$

Using the formulation in [4], the quadric surface equation is:

$$[X\ Y\ Z\ 1] * \begin{bmatrix} A & B & C & D \\ B & E & F & G \\ C & F & H & I \\ D & G & I & J \end{bmatrix} * \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} = 0 \tag{G2}$$

The matrix is labelled **Q** and is useful for performing transformations and other operations on the quadric. See [6] and [4] for further discussion of these operations. This equation is equivalent to where the function $F(X, Y, Z) = 0$:

$$F(X, Y, Z) \equiv A^*X^2 + 2^*B^*X^*Y + 2^*C^*X^*Z + 2^*D^*X + E^*Y^2 + 2^*F^*Y^*Z + 2^*G^*Y + H^*Z^2 + 2^*I^*Z + J$$

Substituting (G1) into (G2) and solving for $t$ yields coefficients for the quadratic formula:

$$A_q = A^*X_d^2 + 2^*B^*X_d{}^*Y_d + 2^*C^*X_d{}^*Z_d + E^*Y_d^2 + 2^*F^*Y_d{}^*Z_d + H^*Z_d^2$$

$$B_q = 2^*(A^*X_0{}^*X_d + B^*(X_0{}^*Y_d + X_d{}^*Y_0) + C^*(X_0{}^*Z_d + X_d{}^*Z_0) + D^*X_d + E^*Y_0{}^*Y_d + F^*(Y_0{}^*Z_d + Y_d{}^*Z_0) + G^*Y_d + H^*Z_0{}^*Z_d + I^*Z_d)$$

$$C_q = A^*X_0^2 + 2^*B^*X_0{}^*Y_0 + 2^*C^*X_0{}^*Z_0 + 2^*D^*X_0 + E^*Y_0^2 + 2^*F^*Y_0{}^*Z_0 + 2^*G^*Y_0 + H^*Z_0^2 + 2^*I^*Z_0 + J.$$

If $A_q \neq 0$, then check the squared discriminant. If $B_q^2 - 4^* A_q{}^* C_q < 0$, no intersection takes place. Otherwise calculate $t_0$ and possibly $t_1$, if needed. The smallest positive value of $t$ is used to calculate the closest intersection point.

$$t_0 = \frac{-B_q - \sqrt{(B_q^2 - 4^* A_q{}^* C_q)}}{2^* A_q}$$
$$t_1 = \frac{-B_q + \sqrt{(B_q^2 - 4^* A_q{}^* C_q)}}{2^* A_q} \tag{G3}$$

If $A_q = 0$, then the equation to be solved is simply:

$$t = -C_q/B_q \tag{G4}$$

Once $t$ has been computed, the intersection point $\mathbf{r}_i$ is calculated using equation (C8). The normal of a quadric surface is formed by taking partial derivatives of the function $F$ with respect to $X$, $Y$, and $Z$:

$$\mathbf{r}_n \equiv [x_n\ y_n\ z_n] = [\mathrm{d}F/\mathrm{d}X \quad \mathrm{d}F/\mathrm{d}Y \quad \mathrm{d}F/\mathrm{d}Z] \tag{G5}$$

$$\begin{aligned} x_n &= 2^*(A^* x_i + B^* y_i + C^* z_i + D) \\ y_n &= 2^*(B^* x_i + E^* y_i + F^* z_i + G) \\ z_n &= 2^*(C^* x_i + F^* y_i + H^* z_i + I). \end{aligned}$$

Note that $\mathbf{r}_n$ is not normalized. The multiplication by 2 can be factored out, since the length of the normal is unimportant at this point. Also, the normal should be for the surface facing the ray, so the direction of this vector must be reversed depending on its relationship with the direction vector $\mathbf{R}_d$. If $\mathbf{r}_n \cdot \mathbf{R}_d > 0$, then the normal should be reversed.

## Example

Given a ray with an origin at [4 5 −3] and a direction vector of [0.577 0.577 −0.577], find the intersection point with an ellipsoid at [6 9 −2] with the axes lengths $X_a = 12$, $Y_a = 24$, $Z_a = 8$.

From basic analytic geometry, the ellipsoid's equation is:

$$\frac{(X-6)^2}{12^2} + \frac{(Y-9)^2}{24^2} + \frac{(Z-(-2))^2}{8^2} = 1$$

Simplifying, the quadric function is then:

$$F(X, Y, Z) \equiv 4^* X^2 - 48^* X + Y^2 - 18^* Y + 9^* Z^2 + 36^* Z - 315 = 0.$$

The equivalent matrix (G2) is formed by finding equivalences to the parameters $A$ through $J$, and is:

$$[X\ Y\ Z\ 1] * \begin{bmatrix} 4 & 0 & 0 & -24 \\ 0 & 1 & 0 & -9 \\ 0 & 0 & 9 & 18 \\ -24 & -9 & 18 & 315 \end{bmatrix} * \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} = 0 \qquad \text{(G2)}$$

The coefficients for $t$ are:

$$\begin{aligned} A_q = {} & 4^*0.577^*0.577 + 2^*0^*0.577^*0.577 + 2^*0^*0.577^*(-0.577) + \\ & 1^*0.577^*0.577 + 2^*0^*0.577^*(-0.577) + \\ & 9^*(-0.577)^*(-0.577) \\ = {} & 4.67 \end{aligned}$$

$$\begin{aligned} B_q = {} & 2^*(4^*4^*0.577 + 0^*(4^*0.577 + 0.577^*5) + 0^*(4^*(-0.577) + \\ & 0.577^*(-3)) - 24^*0.577 + \\ & 1^*5^*0.577 + 0^*(5^*(-0.577) + 0.577^*(-3)) - 9^*0.577 + \\ & 9^*(-3)^*(-0.577) + 18^*(-0.577)) \\ = {} & -3.46 \end{aligned}$$

$$\begin{aligned} C_q = {} & 4^*4^*4 + 2^*0^*4^*5 + 2^*0^*4^*(-3) + 2^*(-24)^*4 + \\ & 1^*5^*5 + 2^*0^*5^*(-3) + 2^*(-9)^*5 + \\ & 9^*(-3)^*(-3) + 2^*18^*(-3) - 315 \\ = {} & -535. \end{aligned}$$

The expression $B_q^2 - 4^* A_q{}^* C_q$ is positive, so an intersection point exists. The distance $t$ is then either $t_0$ or $t_1$. First check $t_0$ by (G3):

$$t_0 = (-(-3.46) - \frac{\sqrt{((-3.46)^2 - 4^*4.67^*(-535))}}{2^*4.67}$$

$$= -10.3.$$

$t_0$ is negative (behind the ray), so check $t_1$:

$$t_1 = (-(-3.46) + \frac{\sqrt{((-3.46)^2 - 4^*4.67^*(-535))}}{2^*4.67}$$

$$= 11.1.$$

$t_0$ is positive, so this is the intersection point distance $t$. Note that the origin is

inside the ellipsoid because only $t_1$ is positive. The intersection point is then (A8):

$$\begin{aligned}\mathbf{r}_i &= [4 + 0.577^*11.1 \quad 5 + 0.577^*11.1 \quad -3 + (-0.577)^*11.1] \\ &= [10.4 \quad 11.4 \quad -9.4].\end{aligned}$$

Calculate the normal at the surface (G5):

$$\begin{aligned}x_n &= 4^*10.4 + 0^*11.4 + 0^*(-9.4) - 24 = 17.6 \\ y_n &= 0^*10.4 + 1^*11.4 + 0^*(-9.4) - 9 = 2.4 \\ z_n &= 0^*10.4 + 0^*11.4 + 9^*(-9.4) + 18 = -66.6.\end{aligned}$$

Normalizing, we get:

$$\mathbf{r}_n = [0.255 \quad 0.0348 \quad -0.966].$$

This is a vector whose dot product with $\mathbf{R}_d$:

$$[0.255 \; 0.0348 \; -0.966] \cdot [0.577 \; 0.577 \; -0.577]$$

is 0.725, which means that the surface normal faces in the direction of the ray. This means that the direction of the normal should be reversed so as to point toward the ray's origin.

## Efficiency concerns

There are quite a few techniques which can be applied to this algorithm to make it more computationally efficient. One important idea is factoring out common values in an equation. This makes for less elegant-looking formulae, but for efficiency buffs this is unimportant. For example, the formula for calculating $A_q$ in (G3) could be rewritten:

$$\begin{aligned}A_q = X_d^*(Z^*X_d + 2^*B^*Y_d + 2^*C^*Z_d) + \\ Y_d^*(E^*Y_d + 2^*F^*Z_d) + \\ H^*Z_d^2\end{aligned}$$

thereby getting rid of 3 multiplies. Another simple change is to factor all constant multiplications (i.e. '2*...') into the factors given, creating new factors as needed. This is recommended only if memory constraints are not a problem. Finally, modifying the quadratic equation in a manner similar to (A16) will save a few more operations. In essence, substitute $NB_q = B_q/2$ into the equation (G3) and solve.

Kernighan and Plauger's [10] basic programming rule is "Write clearly—don't be too clever." This should be balanced against Press' comment [11], "Come the (computer) revolution, all persons found guilty of such criminal behavior [of not factoring] will be summarily executed, and their programs won't be!" A good route is to carefully comment any confusing formulae that are created for efficiency reasons.

Incorporating all of these changes leads to a modified (G3):

$$t_0 = K_a - \sqrt{(K_a^2 - K_b)}$$
$$t_1 = K_a + \sqrt{(K_a^2 - K_b)}$$

where:

$$K_a = -NB_q / A_q$$
$$K_b = C_q / A_q$$

$$A_q = X_d{}^*(A^* X_d + NB^* Y_d + NC^* Z_d) + Y_d{}^*(E^* Y_d + NF^* Z_d) + H^* Z_d^2$$

$$NB_q = X_d{}^*(A^* X_0 + B^* Y_0 + C^* Z_0 + D) + Y_d{}^*(B^* X_0 + E^* Y_0 + F^* Z_0 + G) + Z_d{}^*(C^* X_0 + F^* Y_0 + H^* Z_0 + I)$$

$$C_q = X_0{}^*(A^* X_0 + NB^* Y_0 + NC^* Z_0 + ND) + Y_0{}^*(E^* Y_0 + NF^* Z_0 + NG) + Z_0{}^*(H^* Z_0 + NI) + J$$

where:

$$NB = 2^* B,\ NC = 2^* C,\ ND = 2^* D,\ NF = 2^* F,\ NG = 2^* G,\ NI = 2^* I.$$

For reasons of efficiency, the normal calculation could be separate from the intersection routine [16]. Of all the surfaces tested, only one will actually be closest to the ray's origin, which means that this object would be the only one where the normal was relevant. For calculations such as shadow testing the normal is never needed. After all calculations, the normal could be computed if desired.

The problems of floating point arithmetic imprecision must again be addressed. This imprecision affects the tests for $A_q$ and $B_q$ almost equal to 0. The case where the origin of the ray begins on the quadric surface must also be addressed. Refer to 'Precision Problems' in the 'Ray/Sphere Intersection' section to find a discussion of the problem and its possible solutions. The

quadratic formula calculation as given in section 5.5 of [11] is recommended to help avoid precision problems.

The steps of the algorithm are:

Step 1: calculate coefficients.
Step 2: if $A_q$ is not zero, compute $K_a$ and $K_b$.
Step 3: if $K_a^2 - K_b$ is less than zero, no solution exists.
Step 4: compute the intersection distance $t_0$ or $t_1$.
Step 5: compute the intersection point.
Step 6: compute the normal, without normalizing or sign change.
Step 7: redirect normal.
Step 8: normalize normal.

Assuming precomputation and following the worst case, the calculations for each step are:

Step 1: 25 additions and 30 multiplies.
Step 2: 1 subtraction, 2 divides, 1 compare.
Step 3: 1 subtraction, 1 multiply and 1 compare.
Step 4: 1 subtraction, 1 multiply, 1 square root and 1 compare.
Step 5: 3 additions and 3 multiplies.
Step 6: 9 additions and 9 multiplies.
Step 7: 2 additions, 3 multiplies, 1 compare.
Step 8: 2 additions, 6 multiplies, 1 division, 1 square root.

The total is 44 additions/subtractions, 53 multiplies, 3 divisions, 2 square roots, and 4 compares.

## 5.2 Standard Inverse Mappings

How to perform inverse mappings from a quadric intersection point to $(u, v)$ parametric space is mostly a matter of choice. This is especially true for the less used quadrics, such as the hyperboloid sheets. However, there are objects used in solid modelling and other computer graphics-related fields which have standard mapping definitions. These algorithms are included here, as they can aid both graphical functions such as texture mapping and also a number of non-graphical applications. Mapping parametric coordinates to world coordinates is not covered, as this mapping is not normally needed within most ray tracing applications.

### Inverse mapping for a circle

The inverse mapping of a circle is mostly just a problem of converting from Cartesian to polar coordinates. Define a circle laying on the $XY$ plane with its

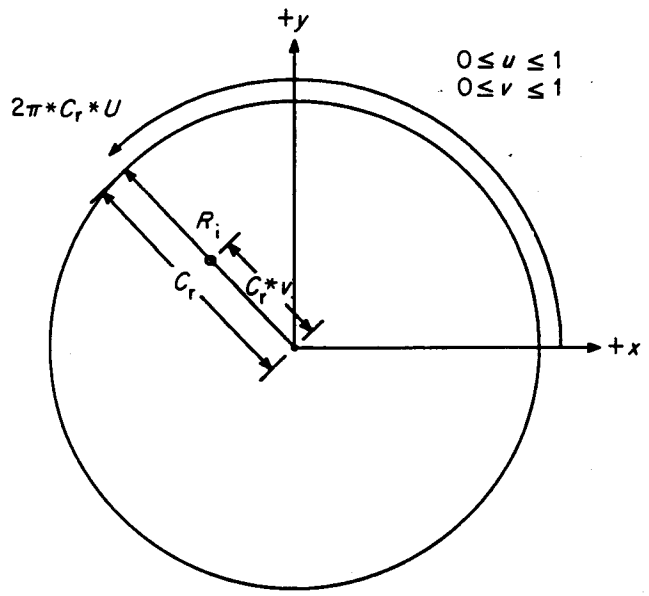


Fig. 12. Circle inverse mapping.

center at the origin and a radius $C_r$:

$$X_c^2 + Y_c^2 = C_r^2 \tag{H1}$$

Obviously, in an environment a circle will have a different orientation and location than this simple definition. Assume that some transformation matrix is associated with the circle, so that the circle and related data can be made to coincide with the definition.

Also given is an intersection point:

$$\mathbf{R}_i = [X_i \; Y_i \; Z_i]$$

which lies on the $XY$ plane (i.e. $Z_i = 0$). The $(u, v)$ coordinates are defined as $u$ ranging from (0..1) starting at the $+X$ axis moving towards the $+Y$ axis, and $v$ ranging from (0..1) from the origin to the edge of the circle. This mapping is shown in *Figure 12*. These parameters are calculated from $\mathbf{R}_i$ as follows:

$$v = \sqrt{((X_i^2 + Y_i^2)/C_r^2)}$$

$$u' = \frac{\arccos\,(X_i/\sqrt{(X_i^2 + Y_i^2)})}{2 * \pi} \tag{H2}$$

if $Y_i < 0$ then set $u = 1 - u'$, else set $u = u'$.

Note that we could eliminate a multiply and a division by setting $C_r = 1$. This

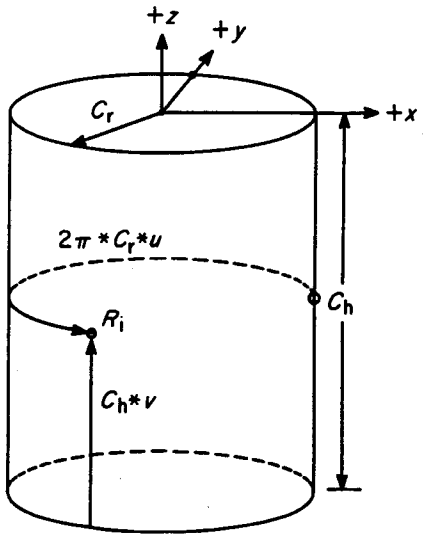


Fig. 13. Cylindrical inverse mapping.

could be performed by concatenating a scaling matrix into the earlier transformation matrix for the object so that the circle is a unit circle.

## Inverse mapping for a cylinder

Define a cylinder of radius $C_r$ and height $C_h$ as:

$$X_c^2 + Y_c^2 = C_r^2, \text{ with } 0 \leqslant Z_c \leqslant C_h \tag{H3}$$

and again have an intersection point $\mathbf{R}_i$ vec $|i$. The $(u, v)$ coordinates are defined as $u$ ranging from (0..1) starting at the $+X$ axis moving towards the $+Y$ axis, and $v$ ranging from (0..1) from the base to the top of the cylinder. This mapping is shown in *Figure 13*. These parameters are calculated as follows:

$$v = Z_i / C_h \tag{H4}$$

$$u' = \frac{\arccos(X_i / C_r)}{2 * \pi}$$

if $Y_i < 0$ then set $u = 1 - u'$, else $u = u'$.

## Inverse mapping for a cone

Define a cone of height $C_h$ with radius $C_{r0}$ at $Z = 0$ and $C_{rh}$ at $Z = C_h$ as:

$$\sqrt{(X_c^2 + Y_c^2)} = C_{r0} + (C_{rh} - C_{r0}) * Z_c / C_h, \tag{H5}$$
$$\text{with } 0 \leqslant Z_c \leqslant C_h$$

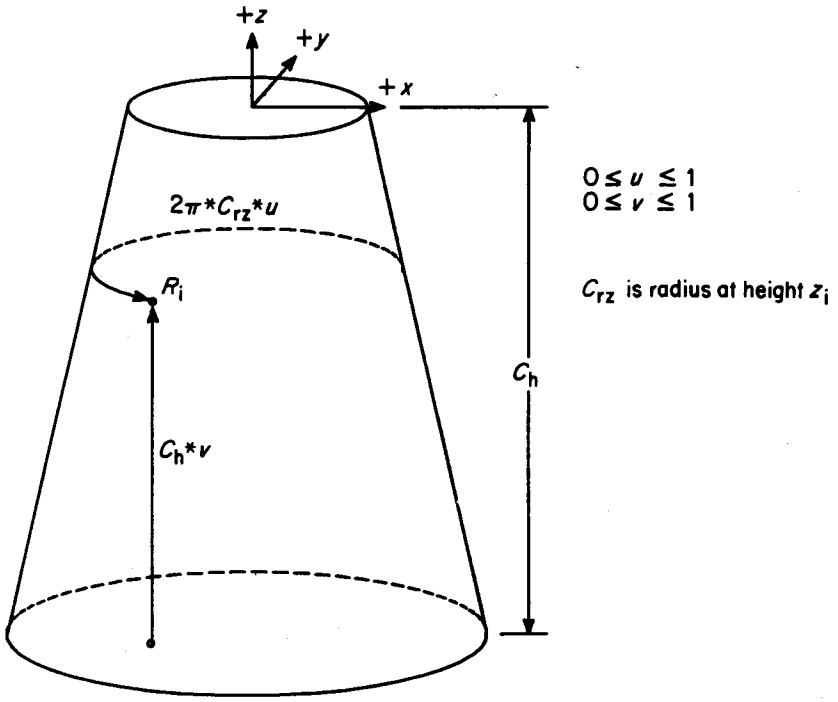


Fig. 14. Conic inverse mapping.

and an intersection point $\mathbf{R}_i$. The $(u, v)$ coordinates are defined as $u$ ranging from (0..1) starting at the $+X$ axis moving towards the $+Y$ axis, and $v$ ranging from (0..1) from the base to the top of the cylinder. This mapping is shown in *Figure 14*. These parameters are calculated as follows:

$$v = Z_i / C_h \tag{H6}$$

$$u' = \frac{\arccos\left(X_i / (C_{r0} + (C_{rh} - C_{r0}) * Z_i / C_h)\right)}{2 * \pi}$$

if $Y_i < 0$ then set $u = 1 - u'$, else $u = u'$.

Alternatively $u$ could be calculated as for the circle. Note that a number of the divisions could be done once for the cone and re-used. Also, note that when $C_{r0} = 0$ and $Z_i = 0$ (or $C_{rh} = 0$ and $Z_i = C_h$), division by zero will result. At this point $u$ is undefined, and can arbitrarily be assigned any value from (0..1).

## BIBLIOGRAPHY AND REFERENCES

1. Berlin, E.P. Jr., 'Efficiency Considerations in Image Synthesis.' Siggraph Course Notes, Vol. 11, July 1985.

2. Blinn, J.F. and Newell, M.E., Texture and reflection in computer generated images. *Commun. ACM* **19**(10), 542–547, October 1976.
3. Blinn, J.F., A homogeneous formulation for lines in 3 space, *Comput. Graph.* **11** (2) Summer 1977.
4. Blinn, J.F., 'The Algebraic Properties of Homogenous Second Order Surface.' Siggraph Course Notes, Vol. 12, July 1984.
5. Duff, T., 'Numerical Methods for Computer Graphics. 'Siggraph Course Notes, Vol. 15, July 1984.
6. Goldman, R.N., Two approaches to a computer model for quadric surfaces. *IEEE Comput. Graph. Appl.* **3**(6), 21–24, September 1983.
7. Heckbert, P.S., Survey of texture mapping *IEEE Comput. Graph Appl.* **6**(11), 56–67, November 1986.
8. Kajiya, J.T., 'Siggraph '83 Tutorial on Ray Tracing.' Siggraph '83 State of the Art in Image Synthesis Course Notes, July 1983.
9. Kay, T.L. and Kajiya, J.T., 'Ray Tracing Complex Scenes.' Siggraph 269–278.
10. Kernighan, B.W. and Plauger, P.J., *The Elements of Programming Style*, McGraw-Hill, New York, 1978.
11. Press, W.H. *et al.*, *Numerical Recipes*, Cambridge University Press, Cambridge, England, 1986.
12. Rogers, D.F., *Procedural Elements for Computer Graphics*, McGraw-Hill, New York, 1985.
13. Roth, S.D., 'Ray casting for modeling solids.' *Comput. Graph. Image Process.* **18**(2), 109–144, Feb. 1982.
14. Sedgewick, R., *Algorithms*, Addison-Wesley, Reading, Mass., pp. 315–317.
15. Ullner, M.K., Parallel Machines for Computer Graphics. PhD Thesis, California Institute of Technology, Computer Science Technical Report 5112, 1983.
16. Whitted, T., 'The Hacker's Guide to Making Pretty Pictures.' Siggraph '85 Image Rendering Tricks Course Notes, July 1985.