Abstract

Ray tracing is becoming popular as the best method of rendering high quality images from three dimensional models. Unfortunately, the computational cost is high. Recently, a number of authors have reported on ways to speed up this process by means of space subdivision which is used to minimize the number of intersection calculations. We describe such an algorithm together with an analysis of the factors which affect its performance. The critical operation of skipping an empty space subdivision can be done very quickly, using only integer addition and comparison. A theoretical analysis of the algorithm is developed. It shows how the space and time requirements vary with the number of objects in the scene.

I Introduction

Ray tracing is a technique for rendering pictures from a three dimensional model by following the paths of simulated light rays through the scene. Whitted (1980) reported that for complex scenes, over 95% of the computer time for ray tracing was taken by the process of finding intersections of rays with surfaces in the scene. The reason for this is that it is necessary to test every ray against every object. Since then, a variety of schemes have been suggested for bounding objects in some way to reduce this cost. If a ray is found to be outside a bound, then a whole group of objects can be eliminated at once. Whitted himself (1980) used bounding spheres for this purpose. Rubin and Whitted (1980) used parallelepipeds in a hierarchical structure of bounding volumes, Kajiya (1983) used a hierarchy of bounds to render fractals efficiently, and Roth (1982) used box enclosures around the components of a solid model. A comparison of bounding volume algorithms can be found in Weghorst (1984). More recently, Glassner (1984), Fujimoto (1985), Kunii and Wyvill (1985, 1986), and Cleary (1986) have independently produced schemes whereby the whole space in which rays are traced is divided into cubical regions called "voxels". Objects, intersecting or contained by a particular voxel, are accessed via the data structure representing that voxel. Meanwhile, Kay and Kajiya (1986) have produced another promising algorithm based on bounding hierarchies, which they have compared favourably with Glassner's algorithm. Ohta (1987) has described a completely new approach. He divides the space of viewing angles into subregions so that, given a ray origin and direction, we can use an indexing operation to find a list of objects potentially intersected.

Comparison of the relative speeds of these algorithms is difficult, because the various ray tracers have been written in different languages on different computers and demonstrated with

different kinds of scene. However, it is becoming clear that the method of uniform space division (Cleary 1986), (Fujimoto 1985) is faster than adaptive space division (Glassner 1984), (Wyvill 1985). Both methods rely on tracing a ray's path through a sequence of empty voxels until a non-empty voxel is encountered. In the adaptive schemes, the voxels are of unequal size, which makes the cost of traversing empty voxels relatively high. In the case of uniform division, the cost of skipping empty voxels is very low, but there are more to be skipped. The advantage of the uniform division is partly due to the fact that the optimum level of space division is surprisingly small. It is usually sufficient to divide the world space into a few hundred voxels. Even if a few voxels still contain a large number of objects, provided the majority are empty or nearly empty, ray tracing is still efficient. Another advantage of uniform subdivision is that it does not require knowledge of how to group objects efficiently. This often requires intervention by the user to describe the scene appropriately.

In the next section, we present an algorithm for fast skipping of empty voxels in sufficient detail that implementation should be straightforward. In the following section, we analyse this algorithm. The analysis describes how the cost will vary with the complexity of the scenes being rendered.

II The Algorithm

-- Insert Figure 1 near here --

The most naive ray tracing algorithms check every ray against all objects in the scene. At the other extreme, there is an irreducible number of intersection calculations to be done — those where a ray actually intersects an object. Algorithms mentioned above, using bounding volumes or space division, make an intermediate number of intersection calculations. The approach taken here is to subdivide the space being traced into equally sized subvolumes or voxels. Each voxel has a list of the objects which intersect with it, and when a ray passes through, only the objects on this list need to be checked. Figure 1 shows the situation for a typical ray. As it passes through the scene, it skips a succession of empty voxels, and intersection tests are necessary only for objects near the path of the ray. The ray propagation can stop at the first voxel in which an intersection is found (any intersections in later voxels will be further from the ray source). This approach can greatly reduce the number of unsuccessful intersection calculations, because only objects which lie close to the ray's path are ever checked. However, an additional burden has been introduced: we must check each of the successive cells along the ray's path, to see whether it contains any objects. In this paper, we show how the step from one cell to the next can be made very quickly. The overall effect of this is to allow many small voxels and a concomitantly small number of unsuccessful intersection calculations. Little attention will be

paid here to the intersection calculations themselves, except to note that usually they are expensive, involving tens of floating point operations. In contrast, it will be shown that the next cell calculations can be done in two or three integer operations. Our formulation and derivation is different from that of Fujimoto (1985) and our algorithm is faster.

There are different ways to implement this algorithm, particularly if a tradeoff must be considered between space and speed. It is important, therefore, to explain the thinking behind the algorithm as well. For this reason, we present the algorithm in a sequence of stages of development. At each stage we introduce an additional refinement.

Figure 2 shows the geometry used in the next cell calculations. For simplicity, only the two dimensional case is shown. The ray in Fig. 2 enters a new cell, either by passing left to right through a vertical wall, or by passing from top to bottom through a horizontal wall. Considering only the passage through vertical walls, the distance along the ray between such crossings is a constant, labeled ∂x. There is a similar constant distance, ∂y, between successive crossings of horizontal walls, and, in the three dimensional case, a third constant, ∂z.

-- insert Figure 2 near here --

The problem is to determine which type of crossing will occur next. Will a horizontal or vertical wall be encountered first? This can be done by keeping two variables, dx and dy, which record the total distance *along the ray* (from some arbitrary origin) to the next crossing of a vertical or horizontal wall respectively. If dx is less than dy, then the next crossing will be of a vertical wall and the next cell will be a horizontal neighbour. Correspondingly, if dy is less than dx, then a horizontal wall will be encountered first and the next cell will be a vertical neighbour. dx and dy can be maintained easily: when a vertical wall is crossed, dx is incremented by ∂x, and when a horizontal wall is crossed, dy is incremented by ∂y. It is also necessary to know which cell is being visited. For the moment this will be determined by two integer coordinates, i and j. Depending on the direction of the ray, it will pass from left to right or right to left through a vertical wall, incrementing i by +1 or -1 respectively. This direction flag will be stored in px (and py and pz for the other two directions). The first simple version of the next cell algorithm in two dimensions is:

```
initialize values of px, py, ∂x, ∂y, dx, dy, i and j for current ray;
repeat
        if dx ≤ dy then
        begin
            i:=i+px;
        dx:=dx+∂x
        end;
        if dx ≥ dy then
begin
            j:=j+py;
        dy:=dy+∂y
        end;
until an intersection is found in cell (i, j)
```

The most expensive calculation in each cycle of this algorithm is the indexing of the array cell(i, j) or, in the three dimensional case, cell(i, j, k). Assuming cell is an n×n×n array, it is usually stored as a linear array with n^3 elements. To access cell(i, j, k), an index, p, to the linear array is calculated using:

p:= i*n*n + j*n + k;

This is expensive, as it introduces multiplications into the loop. However, p can be maintained directly, without complicating the algorithm, by using px, py and pz appropriately. Each time i is incremented, p should be incremented by $\pm n^2$, and each time j is incremented, p should be incremented by ±n. Apart from initializing p, the variables i, j and k can be dispensed with completely. The algorithm (for three dimensions) is:

```
initialize values of px, py, pz, ∂x, ∂y, ∂z, dx, dy, dz, and p for current ray;        repeat
    if (dx ≤ dy) and (dx ≤ dz) then
    begin
        p:=p+px;
    dx:=dx+∂x
    end;
    if (dy ≤ dx) and (dy ≤ dz) then
    begin
    p:=p+py;
dy:=dy+∂y
    end;
    if (dz ≤ dy) and (dz ≤ dx) then
    begin
    p:=p+pz;
dz:=dz+∂z
    end;
until an intersection is found in cell[p]
```

Hashing

The cell array contains n^3 entries. To avoid redundant intersection calculations, n should be large enough that most of the entries are null. (Most cells have no objects intersecting them.) This implies that most of the space in the cell array will be wasted. This situation can be significantly improved by using a hash table to store the entries in cell. Then only non-null entries need consume space. A simple way to do this is to use p **mod** M as the index in a table of length M. A division is not necessary, as p can be checked at the end of each loop to see if it exceeds M. Also, if M is chosen as a power of 2, a simple masking operation can be used (although care must then be taken that n is odd).

This hashing can be speeded up for empty voxels by maintaining two arrays: a full-sized array of one-bit entries to indicate whether or not each voxel is empty, and a smaller hash table. The hash table is only consulted when the current voxel is not empty. At this stage, the cost is small compared with performing intersection calculations. The choice, for best performance, depends on the speed of certain machine operations as well as the value of n. If bit-indexing is slow and space must be saved, then the use of just the simple hash table is preferable.

Termination

One important problem to be solved is when to terminate the propagation of a ray. The simplest way to do this is to detect when it leaves the rectangular volume within which the scene is assumed to lie. Suppose we compute the distances along the ray to the points where it intersects the bounding faces of the containing rectangular volume. Call these distances sx, sy, sz. Then on each cycle, when dx, dy or dz is incremented, it can be tested against its limit. This adds no more than three comparisons per cycle.

If the bit table is being used, then an inexpensive way to handle termination is to increase the size of cell by one layer of cells on all faces. The hash bit is turned on for each of these surface cells, but there is no corresponding entry in the hash table. This can be detected when the hash table is checked and the ray terminated. The main advantage of this is that it does not slow down the usual case in the loop, and the only extra space required is for the bits.

The complete algorithm, including hashing and using sx, sy and sz for termination, is now:

```
initialize values of px, py, pz, ∂x, ∂y, ∂z, dx, dy, dz, sx, sy, sz and p for
current ray;
repeat
   if(dx ≤ dy) and (dx ≤ dz) then
   begin
      if dx ≥ sx then exit;
      p:=p+px;
      dx:=dx+∂x
   end
   else if(dy ≤ dx) and (dy ≤ dz) then
   begin
      if dy ≥ sy then exit;
      p:=p+py;
      dy:=dy+∂y
   end
   else if(dz ≤ dy) and (dz ≤ dx) then
   begin
      if dz ≥ sz then exit;
      p:=p+pz;
      dz:=dz+∂z
   end;
   if p > M then p:=p-M;
until an intersection is found in cell with hash key p
```

It is possible to further optimize this code for speed:

- Some of the tests are redundant.
- If a particular ray leaving the scene is detected by the test dx ≥ sx, the tests dy ≥ sy and dz ≥ sz do not need to be in the loop. It is worth keeping three copies of the code, each with only one termination test. When each ray is set up, it can be decided which of the three pieces of code to invoke for voxel skipping.
- If ∂x is the largest of ∂x, ∂y, ∂z, then, after a cycle in which dx is updated, we know that dx will not be updated in the next cycle. Extending such reasoning by assuming ∂x>∂y>∂z and duplicating the code as often as necessary, allows some loops to be completed without any comparisons.

If the algorithm is carefully coded in assembler using all the techniques above, then each loop requires, at worst, 3 adds, 2 comparisons, a bit fetch from memory, a store and two branches for a total of 8 instructions. At best, no comparison is required, and only 6 instructions are needed. Further, because the values of dx, dy, and dz are bounded by the size of the containing volume, it is easy to arrange that they be represented by integers rather than floating point numbers. Note that no multiplication or division is needed.

Initialization

--insert Figure 3 near here --

For the sake of completeness, the initialization calculation is included here. It is assumed that we start with an origin for the ray at coordinates x, y, z somewhere within the enclosing volume for the ray tracing. This corresponds most directly to a situation where a ray has been reflected from an object in the ray traced area. However, it is easily adapted to the situation where the initial rays from the eye through the pixels are being generated from outside the ray traced volume. Figure 3 shows the two dimensional case when computing the initial value of dx. It is assumed that the coordinates have been scaled so that the voxels are $1 \times 1 \times 1$ cubes. If, as is the case in the diagram, the ray is heading towards the right, then dx is given by the formula ($\lfloor x \rfloor + 1 - x)\partial x$. If it is heading towards the left, $dx = (x - \lfloor x \rfloor)\partial x$. ($\lfloor x \rfloor$ is the greatest integer less than x.)

Figure 4 shows the two dimensional geometry when calculating sx – the distance until the ray leaves the scene. The expression which results is $sx = (n-x)\partial x$.

The direction cosines are given by cx, cy and cz. Figure 5 shows the two dimensional geometric situation when computing ∂x. The formula derived from this is $\partial x = \sqrt{(cx^2+cy^2)}/cx$. This is trivially extensible to the three dimensional case and values for ∂y and ∂z.

In the algorithm above, the only way that the values of dx, dy, dz, sx, sy and sz are used is to be compared with one another. Also, dx, dy and dz are only changed by being incremented by ∂x, ∂y and ∂z. As well, the initial values of dx, sx, dy, sy and dz and sz are proportional to ∂x, ∂y and ∂z respectively. Consequently, it is possible to scale ∂x, ∂y and ∂z by an arbitrary factor. Thus, if they are divided by $\sqrt{(cx^2+cy^2+cz^2)}$ the scaled values are given by the simplified expressions $\partial x = 1/cx$ etc. For working in integers, we use $\partial x = q/cx$ where q is a suitable scaling value to provide sufficient precision without causing overflow.

Combining these formulae the code for initializing these values for the x coordinate is given below. The code for the y and z coordinates is essentially identical.

```
mx:= n^2 mod M;
if cx = 0 then
begin
    dx := ∞;
    sx := 0;
end else
begin
    if cx > 0 then
    begin
        ∂x := q/cx;
        px := mx;
        dx := (⌊x⌋+1 - x)*∂x;
        sx := (n-x)*∂x
    end else
    if cx < 0 then
    begin
        ∂x := -q/cx;
        px := M-mx;
        dx := (x- ⌊x⌋)*∂x;
        sx := x*∂x
    end
end;
```

Having completed these calculations for all three coordinates, the only remaining value to be computed is p:

```
p:= ( ⌊x⌋*mx + ⌊y⌋*my + ⌊z⌋*mz) mod M;
```

Intersection calculations

--insert Figure 6 near here --

So far, we have considered intersection calculations to be single indivisible operations unaffected by how we decide when to apply them. But for many objects, such as polygons, these calculations can be significantly accelerated by the space division. A naive way of processing a polygon is first to find the intersection of the ray with the plane of the polygon. Then this point is tested against each edge of the polygon to determine whether it lies inside or outside. The time to do this is proportional to the number of edges the polygon has. If only part of a polygon lies within a subvolume, then only those edges also intersecting the subvolume need to be tested. Figure 6 shows an example of this for a concave polygon with four bounding edges labeled A, B, C, and D. As a ray approaches the polygon, it encounters three subvolumes each containing part of the polygon and one or more of the bounding lines. In the first subvolume, an intersection point has to be tested against edge A only, in the second against edge B, and finally, in the third, against edges A, B and C. Any particular ray intersects the polygon plane at only one point, and we need to test against only those edges which intersect the current

voxel. If there is a large polygon with many edges, a conventional ray tracer will have to check all the edges for every intersection. But with space division, there will often be many whole voxels which are intersected by the polygon plane but lie wholly inside its edges. In this case, many rays which strike the polygon will not require any checking against the edges. This idea can also be extended to other types of object. For example, a cylinder can be treated as an infinite cylinder with two bounding planes at either end. Only subvolumes, including parts of the ends, need do tests against them. This is very much the approach taken by the CSG system (Wyvill 1985) which handles a polygon as an infinite plane with pieces subtracted off by the bounding edges. Presumably, more complex objects such as quadric surfaces or swept volumes can be simplified in the same way.

Because some objects occupy many voxels, it is possible that a particular ray can be tested more than once for intersection with a given object (Glassner 1984), (Wyvill 1985). The simplest way to avoid this, is to associate a record with each object in the system to contain the results of the last intersection calculation with that object. Each new ray can be given a unique integer *signature* which is also written into this record at the time the calculation is done. The first operation in any intersection calculation is to inspect this record to see if the signature is that of the current ray. If it is, then the record contains either a flag saying that the ray does not intersect, or it contains the intersection point and other relevant data. Because the signature is unique, these records do not need to be initialized for each ray.

Let us see how this is applied in the case of Fig. 6.

- In the first cell, the intersection of the ray and the plane is computed and the resulting point is stored. The intersection is outside the current voxel, so nothing further is done.
- In the second cell, the plane is checked and found to have a precomputed result which is not in the current voxel. Nothing further is done.
- In the third cell, the plane is checked and found to have an intersection in the current cell. The edges A, B and C are then checked.

To summarize, then, the algorithm analyzed below assumes that: surfaces such as planes, cylinders and spheres are checked only in voxels which they intersect; the intersection calculation is done only once for each ray with such a surface; and checks against bounds on surfaces, such as the edges of a polygon or the ends of a cylinder, are done only in the voxels which contain these bounds.

III Space and time requirements

The time required to process a given scene will depend on many factors, including n^3, the number of voxels, and the geometric properties of the picture itself. The crucial technical problem to be solved in doing this analysis is how to compute the number of voxels which will contain a reference to a particular object. The technique used to solve this problem is the calculation of the augmented volume of an object. The first subsection below gives the basic results using this idea, and the following subsections apply it to analyzing the algorithm. A previous use of augmented volumes in analyzing algorithms can be found in (Cleary 1979). The analysis and results here are similar to those for the analysis of a multiprocessor ray tracing algorithm in (Cleary 1986).

Augmented Volume

-- insert Figure 7 near here --

Figure 7 shows two examples of the number of voxels intersected by a line in two dimensions where voxels become squares with sides of length x. The figure shows that the number of squares intersecting the line depends on the position of the squares relative to the line, as well as to the length and orientation of the line. However, it is possible to calculate the number of squares averaged over all possible positions of the tiling relative to the line. As the tiling is by identical squares, all such relative positions can be specified by two numbers ranging from 0 to x, which denote displacement of the grid along the axes. The orientation of the squares is assumed to remain constant. The position of a square can be determined uniquely by choosing some point, say its midpoint, and specifying where it lies. For some such positions the square will intersect the line and for others it will not. Consider the set of all points near the line whose associated square intersects the line. This set constitutes the *augmented line.*

-- insert Figure 8 near here --

Figure 8 shows such an augmented line and how the calculation of its area, A(x), can be carried out. The definition of the augmented area can be reversed and a square can be positioned so that its centre lies on the line. Every point covered by that square will be a member of the augmented line. Also, a point will be a member of the augmented line only if there is such a square which covers it. So, the augmented line can be obtained by positioning the midpoint of a square at one end and then sweeping it along the line. The area of the resulting figure can be split into three components: a square of area x^2 and two parallelograms with an area proportional to Lx where L is the length of the line. Thus the total area can be expressed as

$$A(x) = 2gLx + x^2 \quad (1a)$$

for some geometric constant, g. (The factor 2 is to preserve compatibility with equation (1b) below.)

-- insert Figure 9 near here --

This result is a special case of that for a convex polygon which is shown in Fig. 9 where

$$A(x) = A + Cgx + x^2 \quad (1b)$$

given that A is the area of the polygon, C is the total length of its circumference and g is a constant dependent only on the orientation and shape of the polygon. This result can be obtained by slightly rearranging the augmented area so that it forms a parallelogram against each face, plus a patch at each vertex. The area of each parallelogram is proportional to the length of its side, and by symmetry will average to half the augmented area of the line. The patches at the vertices sum to a single square.

These results can be extended to a convex two dimensional polygon embedded in three dimensions. The tiling squares now become tiling cubes (voxels). However, similar reasoning leads to the result that the augmented volume is

$$V(x) = Ag_1x + Cg_2 x^2 + x^3 \quad (2a)$$

where A is the area of the polygon, C is the sum of the lengths of the edges and g_1 and g_2 are geometric constants determined by the shape and orientation of the polygon. The special case for a line is given by

$$V(x) = 2Lg_2 x^2 + x^3 \quad (2b)$$

where L is the length of the line.

The connection of the augmented volume to our analysis is obtained by noting that if the augmented volume of an object is V(x) then the average number of voxels which intersect the object is $V(x)/x^3$.

Any convex object can be approximated in the limit by a sequence of polygons whose area and circumference converge to the area and circumference of the object. So (1) and (2) hold for *any* convex shape. For concave or disjoint objects, these results provide upper bounds to their areas and volumes. The problem with a concave object is that the volumes swept out when following different edges may overlap, causing the volume to be overestimated. It seems likely that the qualitative results obtained for convex surfaces extend to concave objects.

It is possible to go one step further and to compute the actual values of the constants in equations (1) and (2) above, by averaging over all possible orientations of the objects. g can be

obtained by noting that the area of each of the parallelograms in Fig. 8 is $\frac{Lx}{2}\sin\theta$ where θ is the orientation with respect to the axes. Integrating with respect to θ gives $g = \frac{2}{\pi}$ and

$$A(x) = \frac{4}{\pi}Lx + x^2 \quad (3a)$$

Applying this result to the general case gives:

$$A(x) = A + \frac{2}{\pi}Cx + x^2 \quad (3b)$$

g_1 in equation (2) can be calculated by considering the limit of large A. That is, an infinite plane coplanar with the polygon. The mean number of voxels intersected by it will be inversely proportional to the mean area of the intersection of the plane with an individual voxel. Let this be M. In other words, $g_1 = 1/M$. M can be computed by sweeping the plane, parallel to its normal, across a single voxel. Taking a unit sized voxel, the mean area is given by the integral of the area of intersection at each point along the normal, divided by the total distance the plane is moved. That is

$$M = \int_0^D \text{area}(x)\,dx \Big/ D$$

where D is the diameter of the cube along the normal. The top integral is just the volume of the cube, so $M = {}^1/_D$ and $g_1 = D$. D can be computed by noting that it is just the sum of the lengths of three of the sides of the cube projected onto the normal. The mean length of a unit line projected against another line in three dimensions is given by the integral

$$\frac{\int_0^{\frac{\pi}{2}} 2\pi \sin\theta \cos\theta \, d\theta}{\int_0^{\frac{\pi}{2}} 2\pi \sin\theta \, d\theta} = \frac{1}{2} \quad (4)$$

So finally, $g_1 = {}^3/_2$.

Calculation of g_2 can be done by using (2b) and arguing in a similar way to that used for a line in two dimensions (see Fig. 8). In this case, the volume will consist of three parallelepipeds which are obtained by sweeping one face of the voxel parallel to the line. The volume of one of

these parallelepipeds is given by the area of the face times the length of the line projected against the normal to the face. So g_2 equals 3/2 times the mean projection of a unit line against another line. Curiously, this is given by the integral in (4) giving the final result $g_2 = {}^3/_4$. The result of all this is:

$$V(x) = {}^3/_2Ax + {}^3/_4Cx^2 + x^3 \quad (5a)$$

For the special case of a line in three dimensions, this gives:

$$V(x) = {}^3/_2Lx^2 + x^3 \quad (5b)$$

Time

The execution time per ray can be divided into three components. First, there is a constant composed of the calculations in initiating the ray and doing any processing after a successful intersection calculation (if any). The second component is doing intersection calculations whether successful or unsuccessful. The third component is the time spent moving between voxels. Estimating the times for the latter two requires use of the results for augmented volumes presented above. In what follows, for simplicity, we treat the ray traced volume as a unit cube, so that $x = {}^1/_n$. (In the algorithm earlier, voxels were considered to be unit cubes.)

Next voxel

Consider one ray passing through the scene. The number of next voxel calculations will be equal to the number of voxels which the ray intersects (less one allowing for the first voxel). Using (5b) and $x = {}^1/_n$ this is given by

$$({}^3/_2Lx^2 + x^3)/x^3 - 1 = {}^3/_2Ln$$

Averaging over all rays and letting ρ be the mean length of a ray and t_x be the time taken to do a next voxel calculation, the average time taken for next voxel calculations per ray is

$${}^3/_2\, t_x\rho n \quad (6)$$

Intersections

An object needs to be checked for an intersection any time that a ray enters a voxel containing a reference to the object, and any time that a ray starts inside such a voxel. Because care is taken to do a full intersection calculation only when a ray encounters and object for the first time, it is necessary to calculate both the total number of checks and the number of different objects encountered by a ray. First we consider the total number of intersection checks.

If it is assumed that the flux of rays is uniform near the object, then the total number of rays entering the voxels will be proportional to the surface area of the voxels which intersect the object times the flux of rays. The number of voxels is given by the augmented volume of the object

divided by the volume of an individual voxel. The mean flux of rays can be calculated by considering a plane extending across the whole scene. If the rays are all projected onto the normal of the plane, then the expected number of rays crossing the plane will be equal to the number of rays which lie on any point along the normal. Using (4) the expected average length of the rays after projection is $\rho/2$ and allowing for the fact that only rays passing *into* the voxels are of interest, the mean flux per unit area is $\rho/4$. Using this result, (5a) and $x=1/n$, the number of repeated calculations per ray is given by:

$$6x^2\ \rho/4(\,{}^3/_2Ax + {}^3/_4Cx^2 + x^3)/x^3 = {}^3/_2\rho(\,{}^3/_2A + {}^3/_4Cn^{-1} + n^{-2}) \qquad (7a)$$

where $6x^2$ is the total surface area of a voxel.

On average the number of rays starting in a voxel is proportional to the volume of the voxel. So the total number of new rays which require a repeated check is proportional to the augmented volume of the object. Using (5a) the number of repeated checks from this source is

$${}^3/_2An^{-1} + {}^3/_4Cn^{-2} + n^{-3} \qquad (7b)$$

Initial Intersections

It is also necessary to know the number of different objects encountered by a ray or, conversely, the number of rays which do at least one check against an object. Consider the volume formed by all voxels which intersect some object. The number of initial checks will be proportional to the number of rays which enter this volume or which start within it. For this last see (7b).

Arguing as above, the number of rays which enter the volume will be proportional to the surface area of the volume times the flux of rays. Note that this is smaller than the sum of the surface areas of the voxels because some faces will be shared by adjacent voxels. For a particular set of voxels, this surface area can be obtained by projecting the surface and its voxels along each of the three axes in turn. Such a projection reduces the problem to that of finding the augmented *area* of the projected polygon. If the area of a polygon is A, then the mean area of its projection along one of the axes is A/2. (Consider the surface area of a hemisphere projected down onto the area of a circle giving the ratio $\pi/2\pi$.) If the original circumference is C, then the mean length of a line after projection is given by the integral

$$\frac{\int_0^{\frac{\pi}{2}} 2\pi \sin\theta \sin\theta \, d\theta}{\int_0^{\frac{\pi}{2}} 2\pi \sin\theta \, d\theta} = \frac{\pi}{4}$$

Combining these results about projections, (3b), and the result for the flux F above, and summing over all six possible projections (two directions along each of three axes) the number of initial checks against a particular object by rays entering from outside is:

$$3/2\rho(A/2 + C/2\, n^{-1} + n^{-2}) \quad (8)$$

Let t_i be the time to do an initial intersection calculation, t_r be the time to do a *repeated* intersection check, N be the total number of objects, α be the mean area of objects, and χ be the mean circumference of objects, then, the total time spent in doing intersection calculations is:

$$\frac{3}{2}N\{t_i\,[\frac{\rho\alpha}{2} + (\frac{\rho\chi}{2}+\alpha)n^{-1} + (\rho+\frac{\chi}{2})n^{-2} + \frac{2}{3}n^{-3}] + t_r\,[\rho\alpha + \frac{\rho\chi}{4}n^{-1}]\,\} \quad (9)$$

Note that here the time to do an initial intersection check is assumed constant. That is, the process of optimizing checks against bounding edges is ignored. A rather more difficult analysis shows that this does not significantly alter the forms of the equations above. They still contain only terms in n^{-1}, n^{-2} and n^{-3}.

Total Time

After combining (6) and (9), letting t_0 be the constant overhead for initializing a ray and rearranging, the total time per ray is:

$$t = t_0 + \frac{3}{2}t_x\rho n \; +$$

$$\frac{3}{2}N\,\{\,(t_r+\frac{t_i}{2})\rho\alpha + [\,\rho\chi\frac{t_i}{2} + \alpha t_i + \rho\chi\frac{t_r}{4}\,]\,n^{-1} + t_i(\rho + \frac{\chi}{2})n^{-2} + \frac{2}{3}t_i\,n^{-3}\,\} \quad (10)$$

For a fixed scene, all these values except n are fixed. Later, this equation will be used to investigate changes with N, the number of objects. But to do this, the changes in t_0, ρ, α and χ with N need to be accounted for.

Equation (10) is quite complex and it pays to remind ourselves of the meaning of the various terms. The term $^3/_2 t_x\rho n$ is solely contributed by the next cell calculations. This

increases linearly with n. As it is the only term which increases with n, it is critical in determining such things as the optimum value of n.

The constant terms which are independent of n come from three sources. t_0 is the constant overhead for each ray. It is an irreducible overhead in any ray tracing scheme and is composed of the times to initialize a ray (either when it is projected through a pixel or after a reflection or refraction) and the time to do any shading/lighting/background/texture calculations after the ray terminates. This will obviously depend greatly on the sophistication and complexity of the various techniques being used. $^3/_2 N t_i \rho^{\alpha}/_2$ is the time for the irreducible number of intersection checks which actually succeed. $^3/_2 N t_r \rho \alpha$ is the time for repeated checks resulting from rays which actually intersect a surface, but do so at an acute angle. As these rays approach the surface, they encounter a number of voxels containing references to the surface before entering the final voxel where the intersection occurs. Taking the ratio of these two terms, we see that, on average, a ray that finally strikes an object encounters the object in three different voxels (including the final one). Kay and Kajiya (1986) suggest that these repeated intersection checks are a major drawback of space partitioning algorithms. This result confirms that the problem will occur frequently, but the use of ray signatures avoids repeating the intersection calculation itself and the cost is not too great.

The remaining terms in n^{-1}, n^{-2} and n^{-3} are caused by rays which pass near objects but which do not actually strike them. These can obviously be reduced by increasing n. The optimum value of n, giving the smallest value of t, will be found when the linear increase in n offsets the decrease in these terms. It turns out that, in practice, this minimum lies in the region from n=50-1000. This implies, in turn, that the terms in n^{-2} and n^{-3} will be small compared to n^{-1}. Figure 10 shows a graph of t against n for some "reasonable" terms (the actual values used will be discussed later). This shows that the term in n^{-3} is so small as to be invisible, while the term in n^{-2} is about 5% of the total.

A rough estimate of where the minimum will lie can be obtained by considering the terms in n and n^{-1} and taking the derivative. This gives the following expression for the minimum point:

$$n_{min} = \sqrt{N \frac{\rho\chi \, (t_i + \frac{t_r}{2}) + \alpha t_i}{2\rho t_x}} \qquad (11a)$$

Substituting this value back into (9) and neglecting terms less than $O(^1/_n)$ gives the following expression for the total time:

$$t_{min} = t_0 + \frac{3}{2}N(t_r + \frac{t_i}{2})\rho\alpha + \sqrt[3]{\rho N t_x}\left(\rho\chi\frac{t_i}{2} + \alpha t_i + \rho\chi\frac{t_i}{4}\right) \quad (11b)$$

One interesting exception to this approximation is a scene consisting of very small objects whose area and circumference are effectively 0. This might be a scene consisting entirely of a scattering of small dots. Letting α=0 and χ=0 in (10) gives:

$$t = t_0 + {^3/_2} t_x \rho n + {^3/_2} N \{t_i \rho n^{-2} + {^2/_3} t_i n^{-3}\}$$

The term in n^{-1} disappears from this equation. However, the term in n^{-3} can still be neglected, giving the following expressions for the optimum value of n and the resulting value of t.

$$n_{min} = \sqrt[3]{(2Nt_i/t_x)} \quad (12a)$$

$$t_{min} = t_0 + 3\rho\sqrt[3]{2Nt_x{}^2 t_i} \quad (12b)$$

Space

The total space required for the algorithm is readily calculated. The major requirements for space are the bit table, the hash table, the list of references to objects attached to each voxel and the description of the objects themselves. Let the space required for each entry in the bit table be s_b, then the total space is $s_b n^3$. Let R be the total number of references to objects in voxels. The number of references for an object is given by its augmented volume divided by the volume of a voxel, so, using (5a) and $x={^1/_n}$, $R = 3N({^3/_2}\alpha n^2 + {^3/_4}\chi n + 1)$. If it is assumed that the hash table is sized so as to maintain a constant load factor λ (fraction of non-null entries) as N is varied, and s_h is the size of each entry in the hash table, then the hash table uses $s_h{^R/_\lambda}$. Each non-empty entry in the hash table will have a chain of references to the objects intersecting that voxel. Probably the simplest and most economical form for these entries is that they contain a unique identifier for the voxel they represent, a pointer to the object that they refer to and a pointer to the next reference in the chain. The identifier is necessary because more than one voxel may hash to the same location. If the space for each such reference is s_r then the total space required for all such references is $s_r R$. The space required to describe all the objects is given by $s_o N$ where s_o is the space used by each object. Combining these expressions, the total space required per object is:

$$s = \frac{s_b n^3}{N} + \left(\frac{s_h}{\lambda} + s_r\right)\left(\frac{3}{2}\alpha n^2 + \frac{3}{4}\chi n + 1\right) + s_o \quad (13)$$

Example

-- insert Figure 10 near here --

To give some further feel for the interpretation of these equations, Fig. 10 shows graphs for time and space. The scene used contains 10, 000 randomly oriented squares. The times used for the various operations are taken from an unoptimized implementation of the algorithm and consequently should not be taken too seriously. They are given in milliseconds (on a VAX 11/780).

Figure 10a shows a graph of time against n built up from its components. The two horizontal lines at the bottom show the time to check for rays which actually intersect a square. The next line up shows the time needed to check for repeated intersections (that is, objects that are encountered in more than one voxel). The next area shows the time to do next cell calculations. These increase linearly with n. The next two areas show the contributions from terms of order n^{-1} and n^{-2}. The terms of order n^{-3} are so small as to be invisible.

Figure 10b shows a graph of space against n built up from its components. The space is expressed as bytes per object and can be split into three components. The first (set at 100 bytes per object) is the constant space required for storing the object itself. The next component which has terms of order n^2 and n, is the space required for the references to objects from voxels etc. It contributed most of the increase in space requirements. The third component is the space for the bit table. It is small over the range of the example shown, but is increasing quickly. At the point of minimum time, the extra space required is just a little less than 3 times that required for storing the objects themselves.

IV Varying number of objects

It is of interest to know the behaviour of the algorithm as the number of objects N is varied. This poses some additional problems, as it has been assumed until now that the scene being examined is static and that other values such as the number of subdivisions, n, have been varied. However, if N is to be changed the *picture* must be changed in some way. Another difficulty is that many of the values which are constants in equations (10) and (13) for a fixed scene will now vary with N. For simplicity, it is assumed that there is a fixed number of rays. That is, all rays terminate at the first object they encounter and there is no transparency or specular reflection.

Two ways of changing the scene will be used. In the first, which we will refer to as scaling, the average geometric properties of the scene are assumed to remain constant, except that the size of the traced scene is scaled isometrically. This will be the case, for example, if the

original scene is a small part of a much bigger scene, and a larger part is selected for rendering. If the bounding box for the scene is arbitrarily assumed to be a unit cube, then the average areas and circumferences of the polygons in the scene will change with N. It is easily seen that the relationship is:

$$\alpha = \alpha_0 N^{-2/3} \quad (14a)$$

$$\chi = \chi_0 N^{-1/3} \quad (14b)$$

The second way of varying N leaves the average area, α, and circumference, χ, unchanged. This effectively stuffs more objects into the same space, gradually filling the scene. We will refer to this as the unscaled case. The paragraphs below will investigate the equations for time, (10), and space, (13), term by term and express them as functions of N for each of the two scaling techniques.

Time

Equation (10) gives the expression for the total time per ray. For convenience it is reproduced below:

$$t = t_0 + \frac{3}{2} t_x \rho n + \frac{3}{2} N \left\{ \left(t_r + \frac{t_i}{2}\right)\rho\alpha + \left[\rho\chi \frac{t_i}{2} + \alpha t_i + \rho\chi \frac{t_r}{4} \right] n^{-1} + t_i\left(\rho + \frac{\chi}{2}\right) n^{-2} + \frac{2}{3} t_i n^{-3} \right\} \quad (10)$$

The following paragraphs will take the terms of the equation one by one and express them as a function of N.

t_0

This term is the constant overhead for every ray. It consists of two components: the time to initialize and start a ray and the time to complete the ray. The initialization will take a constant time, say t_s. When a ray completes, two things can happen taking potentially quite different times. The ray can exit from the scene without encountering any objects. This will result in the intensity being computed from some background; for example: a uniform "sky" colour or some background scene. Alternatively, the ray will encounter an object with a concomitant amount of computing depending on the lighting model being used. As N is varied, the fraction of rays which exit the scene or encounter an object will vary. The effect of this can be approximated as follows.

Assume initially all rays in the scene are of equal length, say ρ_0. The actual length of the rays will decrease as N increases and more of the rays encounter objects. Consider a line along

the direction of some ray. It will encounter the first (if any) object that the ray encounters and then pass on through any remaining objects in that direction. The expected number of surfaces which the line will pass through is given by the mean area of the objects in the scene projected along the direction of the ray. This projected area is $\frac{\alpha N}{2}$ and the expected number of intersections for an initial ray of length ρ_0 is $\delta = \frac{\alpha N}{2}\rho_0$. This assumes that the scene being rendered is a unit cube.

The fraction of rays that encounter no objects can be computed from the Poisson distribution to be $e^{-\delta}$. So, if t_s is the time to start a ray, t_e the time to complete the execution of a ray that exits from a scene, and t_l the time to do the lighting calculations when an intersection occurs, then

$$t_0 = t_s + t_e e^{-\delta} + t_l (1-e^{-\delta}) \qquad (15)$$

At small values of N, $t_0 = t_s + t_e$ because all rays exit from the scene without encountering any objects. For large values of N, $t_0 = t_s + t_l$ and every ray encounters an object. In between, the time will rise steadily from $t_s + t_e$ to $t_s + t_l$ in the usual sigmoid curve. This result assumes that the initial rays are all of equal length. This would be true, for example, if the viewpoint were at infinity when all initial rays would be parallel and of unit length. In general, however, some rays will pass through longer or shorter parts of the scene. The effect of this is to smear out the transition between the two times, but not to alter the qualitative form of the curve.

ρ

This is the length of an average ray and enters into the expression for time in a number of places. Using the reasoning from above, it can be expressed in terms of some initial ray length in an empty scene, ρ_0. If the line along the direction of a ray intersects i objects, then the mean length of the ray will be $\frac{\rho_0}{i+1}$. Using the Poisson distribution, the average length for a ray is given by:

$$\rho = \frac{\rho_0}{\delta}(1 - e^{-\delta}) \qquad (16)$$

When N is large, this simplifies to $\rho = \frac{2}{\alpha N}$ and when N is small, to $\rho = \rho_0$. That is, when N is large, the rays very soon hit an object and the expected length is independent of the initial length, and when N is small, most rays do not hit any object.

n

The value for n needs to be chosen anew for each value of N. Making a best case assumption, this can be done by finding the minimum time over n using (10). In general, this involves solving a fourth order polynomial in n whose analytical solution is not very enlightening. However, some feel for what is happening can be obtained by noting that, in most cases, n lies between $N^{1/3}$ and $N^{1/2}$. This is borne out both by the special cases which have analytical solutions and by numerical solutions of the general case.

Example 1

-- insert Figure 11 near here --

-- insert Figure 12 near here --

-- insert Table I near here --

To give some feel for the meaning of these results, three simple example systems will now be examined. The first is a scene consisting of randomly placed squares with no reflections and with the viewpoint at infinity. Figures 11 and 12 show the results of equations (10), (13), (14), (15) and (16) applied to this scene. Table I shows the parameters used in the calculations. The time and space constants were taken from an unoptimized implementation of the algorithm written in C on a VAX 11/780. The time values are in milliseconds per ray and the space values in bytes. Values for n were calculated by numerically minimizing (10). Figure 10 was calculated using the same scene, with N held at 10,000.

Figure 11 shows time (t), space (s) and the optimum value of n plotted against the log of N for both ways of varying the scene. The constants were chosen in such a way that the two ways of varying the scene coincide at N = 10,000. For a scaled scene, the time increases steadily, eventually reaching an asymptote for large N. For the unscaled scene, a peak in the execution time occurs a little above N = 10,000 and thereafter, the execution time decreases. This appears to happen because the scene becomes very crowded for large N and the rays penetrate only a very short distance. In fact, the unscaled assumption is not very realistic for values of N much above 10,000, as there will be a very crowded scene with many overlapping and interpenetrating polygons.

For a scaled scene, the space required per object stays almost constant. This is because n scales at almost exactly the same rate as the size of the polygons, so that each object is recorded in the same number of voxels. This observation is verified below for some cases which have analytic solutions. The space requirements for the unscaled scene explode exponentially for large N. This is because many polygons will overlap and interpenetrate so that every polygon is

recorded in many voxels. However, as noted above, this type of scene is unlikely to occur in practice.

The graph for the optimum value of n shows both the scaled and unscaled scene as well as two lines proportional to $N^{1/3}$ and $N^{1/2}$. The n value in the scaled scene is parallel to $N^{1/3}$. That is, $n = n_0 N^{1/3}$. In the range of reasonable values for the unscaled scene, n is initially parallel to $N^{1/3}$ and later increases to $N^{1/2}$. As N increases further, n increases even more rapidly, but again this is not likely to occur in practice.

Figure 12 gives a further breakdown of the time into that irreducible part which cannot be avoided in any ray tracing scheme, and the part caused by checking rays against objects with which no intersection actually occurs. The irreducible part consists of the time to initialize the ray and to perform any terminal lighting or other calculations. It would be miraculous for a ray tracing algorithm to reach this minimum, as it would have to avoid any checks against objects other than the one which it finally hit. However, the ratio of the two times does give some idea of how much can be gained by further improving space division or bounding. The answer seems to be that not much improvement is possible. In almost all cases, the irreducible minimum is more than half of the total execution time. The one exception is for small values of N in the unscaled scene where there will be a small number of small objects scattered about the scene. That is, only a small number of pixels will actually be coloured by an object. Most will just be background *sky colour*. Here, the ray tracing times are small anyway, but an algorithm which visited each object in turn and only drew rays through those pixels which would eventually be coloured, would be a significant improvement on the current algorithm.

One interesting special case can be derived from these results. Consider the case where N is large and the scene is scaled. Using the expressions for ρ, α and χ given above, assuming that $n = n_0 N^{1/3}$ and eliminating terms of $O(N^{-1/3})$ and less, the following simplified expression for t is obtained:

$$t = t_s + t_l + 3\left(t_r + \frac{t_i}{2}\right) + \frac{3\alpha_0 t_i}{2n_0} + \frac{3\chi_0 t_i}{4n_0^2} + \frac{t_i}{n_0^3}$$

That is, for large N and a scaled scene, the execution time will be a constant independent of N. This is borne out by the result in Fig. 11.

Example 2

The next two examples give analytic results and are in some sense a worst case for the algorithm. The first examines a hypothetical scene which we call the "spider's web". This contains many long thin lines. This means that very few actual intersections occur, although many of the long thin objects have to be checked. This scene can be modeled by assuming that the area of each polygon is 0 and the the circumference is finite. One consequence of this, is that the average length of the rays remains constant, independent of the number of objects. This gives the following simplified expression for the total time:

$$t = t_s + t_e + \frac{3}{2} t_x \rho_0 n + \frac{3}{2} N \{ \rho_0 \chi [\frac{t_i}{2} + \frac{t_r}{4}] n^{-1} + t_i(\rho_0 + \frac{\chi}{2}) n^{-2} + \frac{2}{3} t_i n^{-3} \}$$

It turns out that, in practice, the terms in n^{-2} and n^{-3} are small and can be neglected. This further simplifies the equation to:

$$t = t_s + t_e + \frac{3}{2} t_x \rho_0 n + \frac{3}{2} N \rho_0 \chi \left(\frac{t_i}{2} + \frac{t_r}{4} \right) n^{-1}$$

Taking the derivative of this in n and computing the minimum gives

$$n_{min} = \sqrt{N \frac{\chi}{t_x} \left(\frac{t_i}{2} + \frac{t_r}{4} \right)}$$

Note that n is thus proportional to $N^{1/2}$.

Substituting this back into the expression for t gives

$$t = t_s + t_e + 3\rho_0 \sqrt{N \chi t_x \left(\frac{t_i}{2} + \frac{t_r}{4} \right)}$$

In an unscaled scene, χ is a constant and t will be proportional to $N^{1/2}$, unlike the first example where it eventually reached a constant maximum. In a scaled scene, $\chi = \chi_0 N^{-1/3}$ so t will be proportional to $N^{1/3}$.

Example 3

The third example is a hypothetical scene we call "stellar dots". It consists of dots scattered at random throughout the scene. This can be modeled by setting both the area and circumference to 0. This gives the following simplified expression for t:

$$t = t_s + t_e + \frac{3}{2} t_x \rho n + N t_i n^{-3}$$

Solving for the optimum value of n gives:

$$n_{min} = \sqrt[4]{\frac{2Nt_i}{t_x \rho}}$$

Substituting back into the expression for t gives

$$t = t_s + t_e + 4(2t_x\rho)^{-3/4}(Nt_i)^{1/4}$$

That is, t is proportional to $N^{1/4}$ for both scaled and unscaled scenes.

Space

It is somewhat easier to provide a general analysis of the expression for space. Equation (13) is reproduced here for reference.

$$s = \frac{s_b n^3}{N} + \left(\frac{s_h}{\lambda} + s_r\right)\left(\frac{3}{2}\alpha n^2 + \frac{3}{4}\chi n + 1\right) + s_o \qquad (13)$$

For a scaled scene, n is proportional to $N^{1/3}$ and after substituting the correct expressions for area and circumference, s can be re-expressed as:

$$s = s_b n_0^3 + \left(\frac{s_h}{\lambda} + s_r\right)\left(\frac{3}{2}\alpha_0 n_0^2 + \frac{3}{4}\chi_0 n_0 + 1\right) + s_o$$

This is independent of N as shown in Figure 11.

For a scaled scene, the situation is somewhat more complex as n does not vary so simply with N. If N is small and n is proportional to $N^{1/3}$ then s becomes:

$$s = s_b n_0^3 + \left(\frac{s_h}{\lambda} + s_r\right)\left(\frac{3}{2}\alpha n_0^2 N^{2/3} + \frac{3}{4}\chi n_0 N^{1/3} + 1\right) + s_o$$

It is thus dominated by terms of order $N^{2/3}$ and $N^{1/3}$.
Alternatively, if n is proportional to $N^{1/2}$ then s becomes:

$$s = s_b n_0^3 N^{1/2} + \left(\frac{s_h}{\lambda} + s_r\right)\left(\frac{3}{2}\alpha n_0^2 N + \frac{3}{4}\chi n_0 N^{1/2} + 1\right) + s_o$$

In this instance, it is dominated by terms of the order of N and $N^{1/2}$.

IV Conclusion

The major accomplishment of this paper has been to show that it is possible to analyze a ray tracing algorithm in some detail. The full value of this analysis will be realized when it is possible to analyze other competitive techniques and compare them.

One very important point that does emerge is that, in many cases, the irreducible overhead of any ray tracing algorithm (initializing the ray, doing lighting calculations etc.) is a large fraction of the total execution time — usually more than 50%. This implies that dramatic gains cannot be expected from incremental improvements to ray tracing itself. It may imply that other techniques, such as beam casting, which exploit more of a scene's coherence, will be necessary if further improvements are to be made in execution time.

In some cases, the storage requirements of the algorithm can increase dramatically with the number of objects in the scene. It is not clear whether the cases where this occurs (unscaled scenes with large number of objects) are likely in practice, but it does seem to be a possible weakness of the algorithm. One possibility not explored here is to optimize the scene subdivision for space rather than time, or to seek a compromise between the two.

Acknowledgements

We would like to thank Andrew Pearce for providing time and space measurements from his implementation. This work was partially funded by the Natural Sciences and Engineering Research Council of Canada.

References

Cleary JG (1979) An Analysis of an Algorithm for Finding Nearest Neighbours in Euclidean Space. ACM Trans on Mathematical Software 5(2): 183-192

Cleary JG, Wyvill BLM, Birtwistle G, Vatti R (1986) Multi Processor Ray Tracing. Comput Graph Forum 5(1): 3-12

Fujimoto A, Iwata K (1985) Accelerated Ray Tracing. Proc CG Tokyo '85: 41-65

Fujimoto A, Perrott CG, Iwata K (1986a) Environment for Fast Elaboration of Constructive Solid Geometry. Proc. CG Tokyo '86: 20-33

Fujimoto A, Tanaka T, Iwata K (1986b) ARTS: Accelerated Ray-Tracing System. IEEE Comput Graph Appl 6(4): 16-26

Glassner AS (1984) Space subdivision for fast ray tracing. IEEE Comput Graph Appl 4(10): 15-22

Heckbert PS, Hanrahan P (1984) Beam Tracing Polygonal Objects. Comput Graphics 18(3) (Proc SIGGRAPH '84): 119-127

Kajiya JT (1983) New Techniques for Ray Tracing Procedurally Defined Objects. Comput Graphics 17(3) (Proc SIGGRAPH '83): 91-102

Kay TL, Kajiya JT (1986) Ray Tracing Complex Scenes. Comput Graphics 20(4) (Proc SIGGRAPH '86): 269-278

Kunii TL, Wyvill G (1985) CSG and Ray Tracing Using Functional Primitives. Computer Generated Images The State of The Art. Springer Verlag: 137-152

Ohta M, Maekawa M (1987) Ray Coherence Theorem and Constant Time Ray Tracing Algorithm. Computer Graphics 1987 (Proc CG International '87): 303-314

Roth SD (1982) Ray Casting for Modeling Solids. Computer Graphics and Image Processing 18: 109-144

Rubin SM, Whitted T (1980) A 3-Dimensional Representation for Fast Rendering of Complex Scenes. Comput Graphics 14(3) (Proc. SIGGRAPH '80): 110-116

Weghorst H, Hooper G, Greenberg DP (1984) Improved Computational Methods for Ray Tracing. ACM Trans Graphics 3(1): 51-69

Whitted T (1980) An Improved Illumination Model for Shaded Display. Commun ACM 23(6): 343-349

Wyvill G, Kunii TL (1985) A functional model for constructive solid geometry. The Visual Computer 1(1): 3-14

Wyvill G, Kunii TL, Shirai Y (1986) Space Division for Ray Tracing in CSG. IEEE Comput Graph Appl 6(4): 28-34

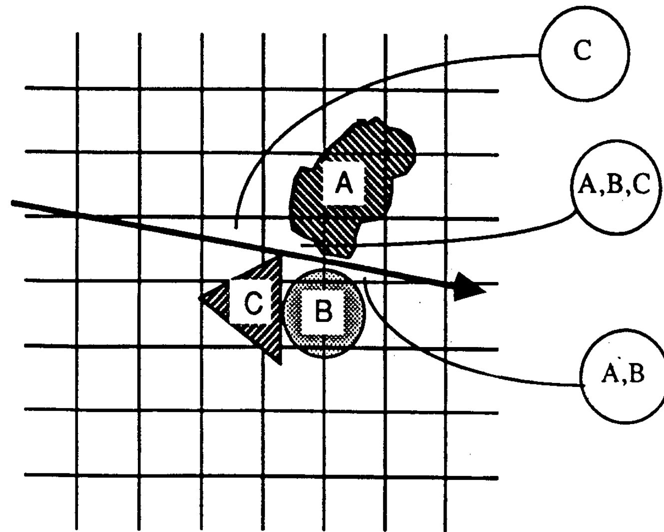
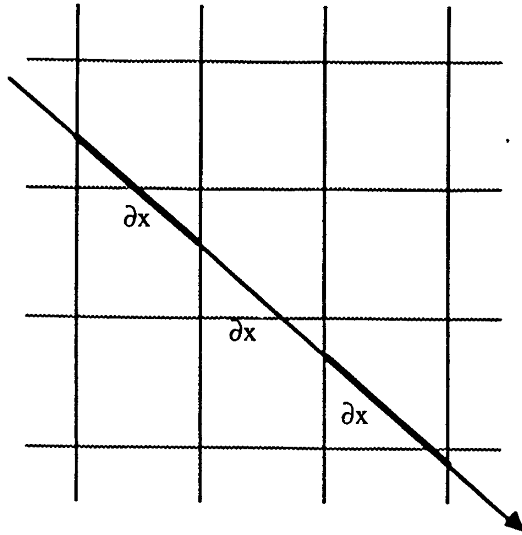
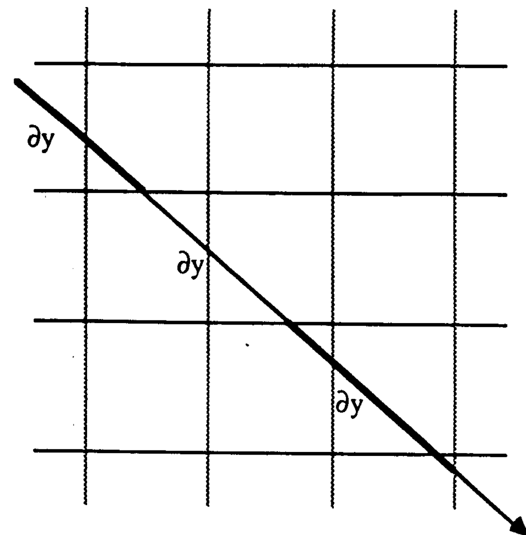
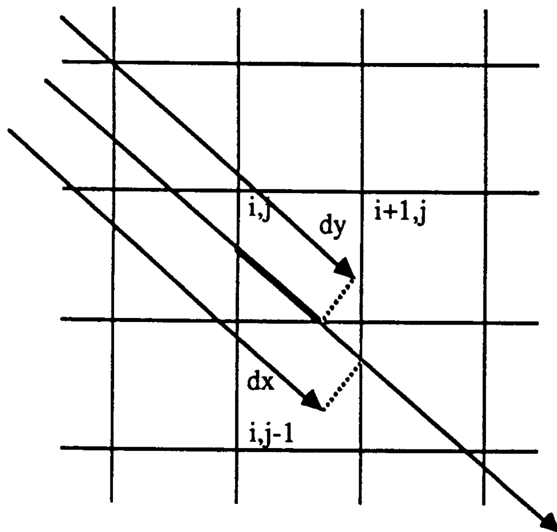


Figure 1. Passage of a ray through voxels and the resulting intersection checks.







Simplified 2D algorithm

```
px=+1
py= -1
initialize ∂x,∂y,dx,dy,i,j
repeat
    if dx ≤ dy then
     begin
       i := i+px;
       dx:=dx+∂x;
    end;
    if dx ≥ dy then
    begin
       j:=j+py;
       dy:=dy+∂y;
    end;
until intersection in cell i,j;
```

Figure 2. Next cell calculation.

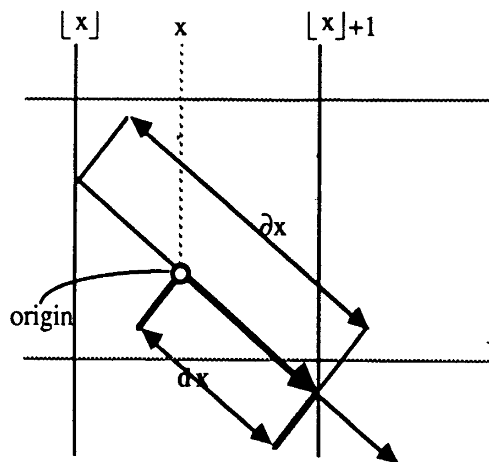


Figure 3. Initial calculation of dx.

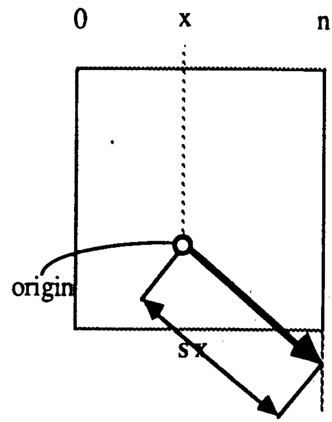


Figure 4. Initial calculation of sx.

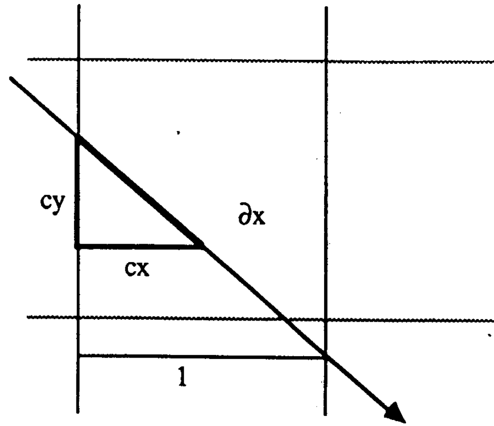


Figure 5. Initial calculation of ∂x.

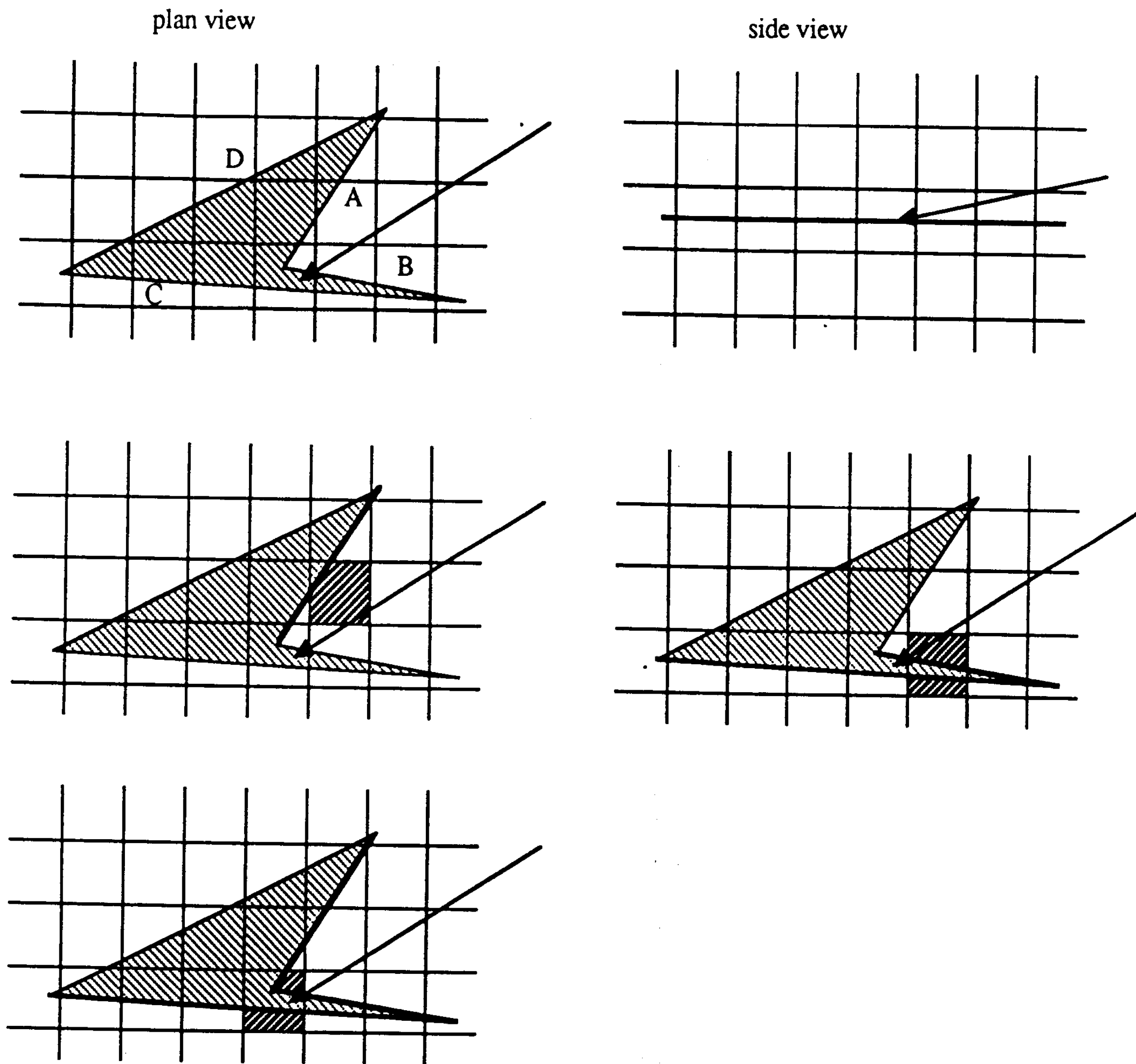


Figure 6. Objects considered in successive cells as a ray approaches intersection.

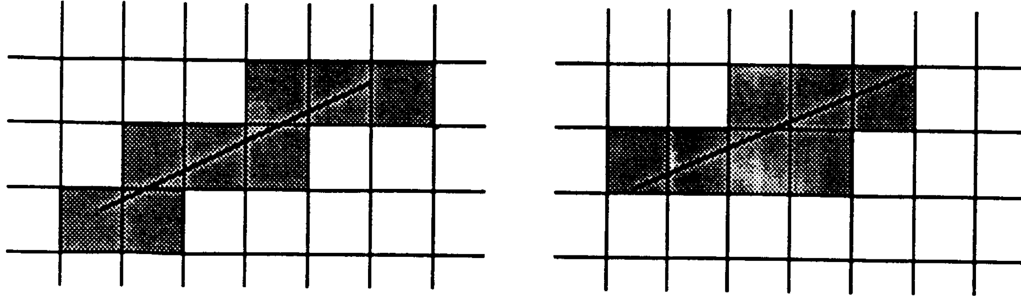


Figure 7. Intersecting voxels for two different positions of tiling squares.

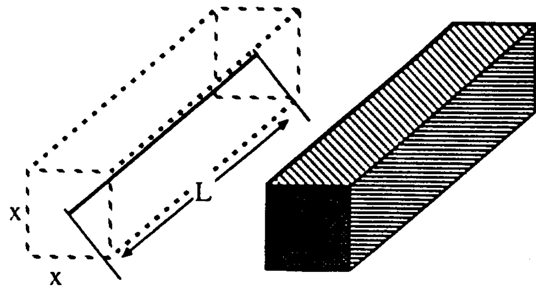


Figure 8. Augmented area of a line.

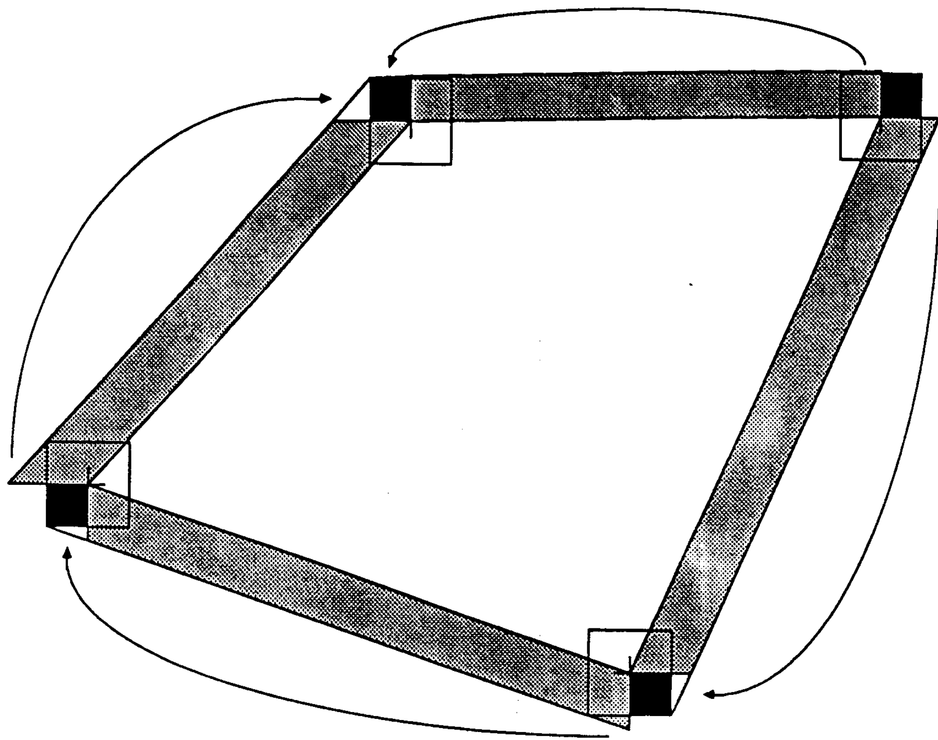


Figure 9. Augmented area of convex polygon.

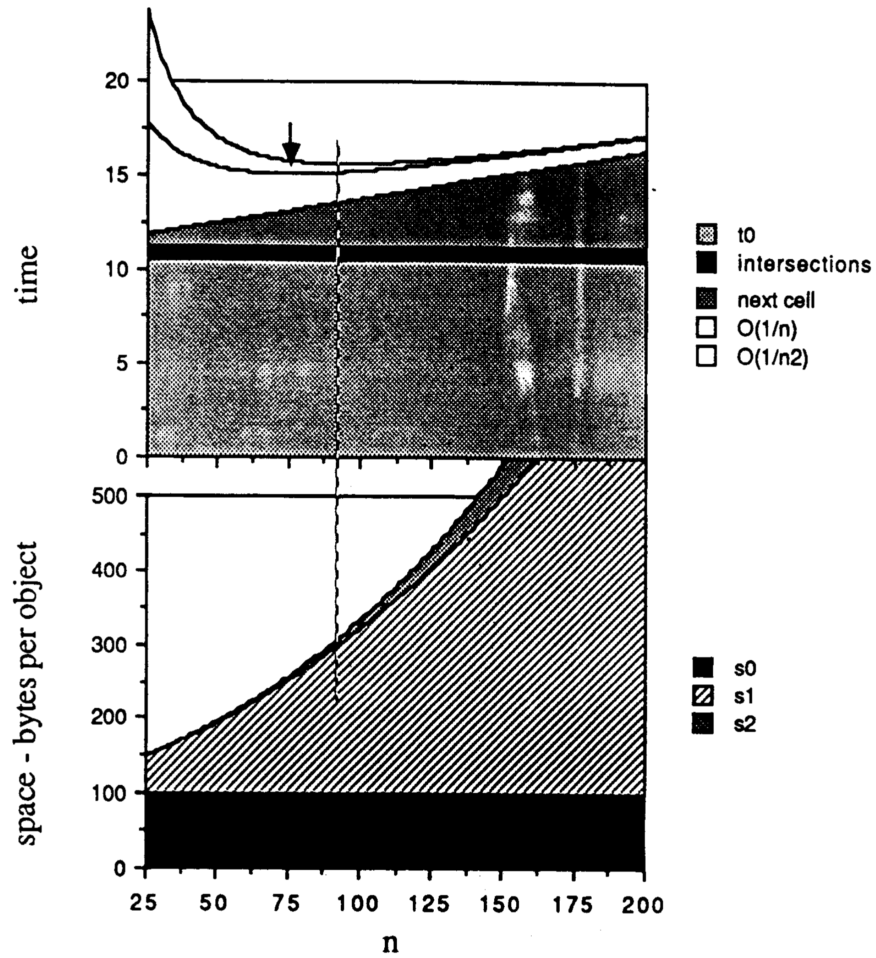


Figure 10. Time and space plotted against n.

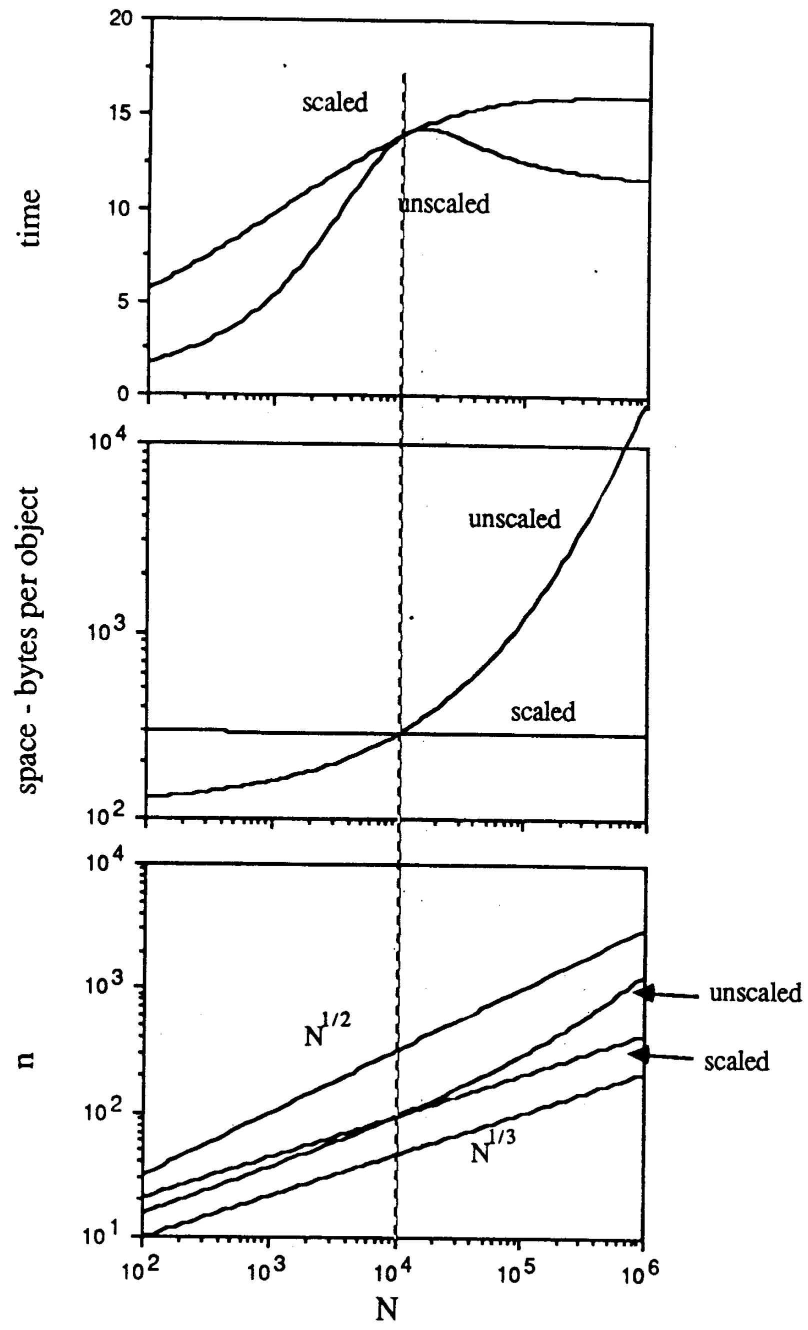


Figure 11. Changes in time, space and optimum n against number of objects.

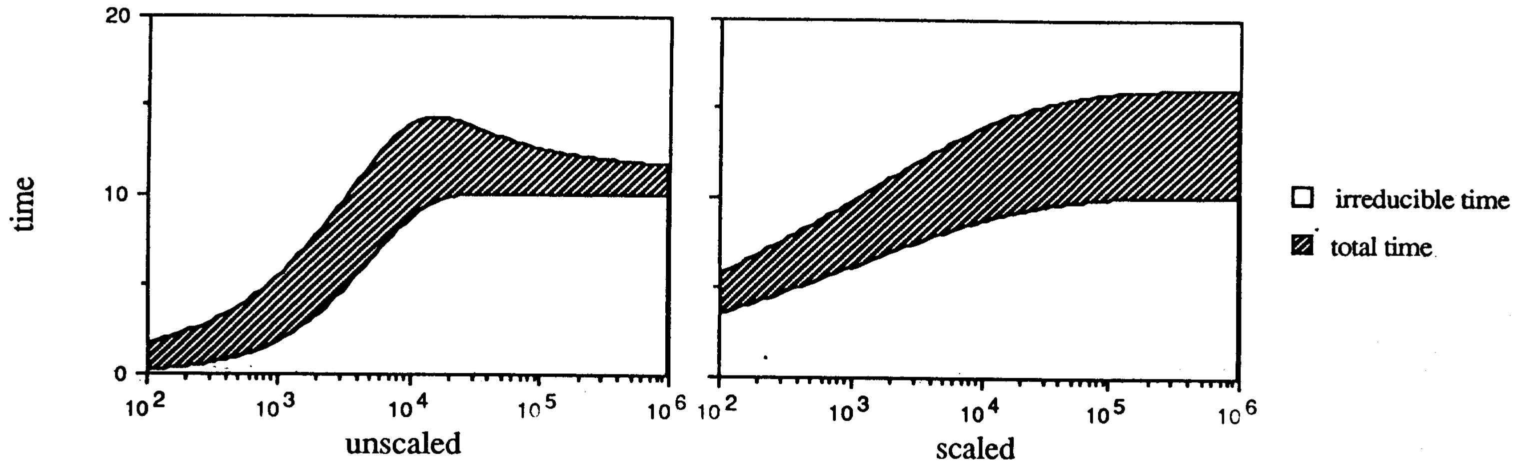


Figure 12. Breakdown of time into overhead of ray-tracing and irreducible time.