A Fast Voxel Traversal Algorithm for Ray Tracing

John Amanatides

Andrew Woo

Dept. of Computer Science
University of Toronto
Toronto, Ontario, Canada M5S 1A4

ABSTRACT

A fast and simple voxel traversal algorithm through a 3D space partition is introduced. Going from one voxel to its neighbour requires only two floating point comparisons and one floating point addition. Also, multiple ray intersections with objects that are in more than one voxel are eliminated.

Introduction

In recent years, ray tracing has become the algorithm of choice for generating high fidelity images. Its simplicity and elegance allows one to easily model reflection, refraction and shadows.[1] Unfortunately, it has a major drawback: computational expense. The prime reason for this is that the heart of ray tracing, intersecting an object with a ray, is expensive and can easily take up to 95% of the rendering time. Unless some sort of intersection culling is performed, each ray must intersect all the objects in the scene, a very expensive proposition.

There are two general strategies for intersection culling: hierarchical bounding volumes[1, 2, 3, 4] and space partitioning.[5, 6, 7, 8]

The general idea of the first approach is to envelop complicated objects that take a long time to intersect with simpler bounding volumes that are much easier to intersect, such as spheres or boxes. Before intersecting the complicated object, the bounding volume is first intersected. (Actually, it is not a full intersection test; all we care about is *if* the ray hits the bounding volume, not where). If there is no intersection with the bounding volume, there is no need to intersect the complicated object, thus saving time. For a complicated scene made up of many objects, a bounding volume is placed around the entire scene with each object also containing a bounding volume. If an object is made up of several parts each of these parts can also have a bounding volume. We thus can built a tree of bounding volumes, with each node containing a bounding volume that envelops its children. Objects within a subtree are intersected only if their parent node bounding volume is intersected by the ray. In this manner, the amount of actual intersections are significantly reduced. Of course, we now have to spent time intersecting bounding volumes but this is more than offset by the reduced total intersections.

The second approach of reducing intersections is to partition space itself into regions or voxels. Each voxel has a list of objects that are in that voxel. If an object spans several voxels it is in more than one list. When a ray is shot, we first look into the voxel in which it originates. If it hits any objects in the starting voxel's list, the intersections are sorted and the closest one is retained. If the intersection is in the

current voxel there is no need to intersect any other objects as we have found the closest intersection. If no intersection is found in the current voxel or the object list is empty, we follow the ray into a neighbouring voxel and check its object list. We continue until either we find an intersection or we completely traverse the space partition. Since we intersect objects roughly in the order as they occur along the ray and trivially reject objects far from the ray, the number of intersections that need to be performed is vastly reduced. There are two popular space partition schemes: octrees by Glassner,[6] where voxels are of different sizes, and constant size voxel partitioning (hereafter called a grid partition) by Fujimoto et. al.[7, 8] The first conserves space but makes traversal difficult while the latter allows for simpler traversal at the expense of more voxels.

In this paper, we introduce a fast and simple incremental grid traversal algorithm. Like Fujimoto et. al.,[7, 8] it is a variant of the DDA line algorithm. However, instead of basing it on the simple DDA (Fujimoto et. al.), in which an unconditional step along one axis is required, ours has no preferred axis. This considerably simplifies the inner loop and allows for easy testing of an intersection point to see if it is in the current voxel. Along with the the new traversal algorithm, we introduce a technique to eliminate multiple intersections when an object spans several voxels. This technique can be used with all space subdivision algorithms with minimum modifications.

The New Traversal Algorithm

Let us derive the new traversal algorithm. We consider the two dimensional case first; the extension to three dimensions is straightforward. Consider figure 1:

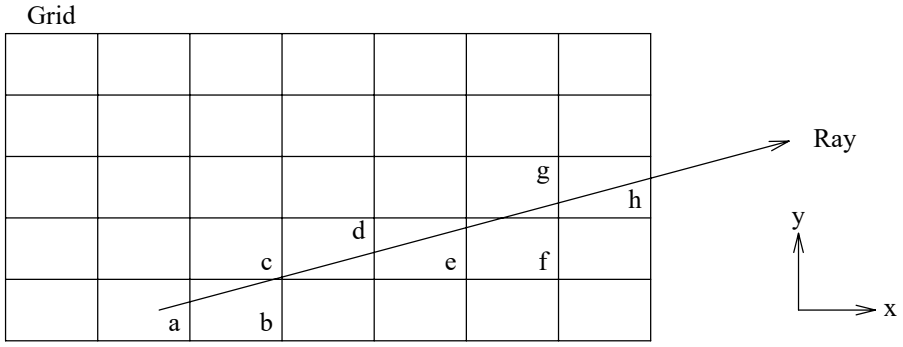

Figure 1

To correctly traverse the grid, a traversal algorithm must visit voxels a, b, c, d, e, f, g and h in that order. The equation of the ray is $\vec{u} + t\vec{v}$ for $t \geq 0$. The new traversal algorithm breaks down the ray into intervals of t, each of which spans one voxel. We start at the ray origin and visit each of these voxels in interval order.

The traversal algorithm consists of two phases: initialization and incremental traversal. The initialization phase begins by identifying the voxel in which the ray origin, $\vec{u}$, is found. If the ray origin is outside the grid, we find the point in which the ray enters the grid and take the adjacent voxel. The integer variables X and Y are initialized to the starting voxel coordinates. In addition, the variables stepX and stepY are initialized to either 1 or -1 indicating whether X and Y are incremented or decremented as the ray crosses voxel boundaries (this is determined by the sign of the x and y components of $\vec{v}$).

Next, we determine the value of t at which the ray crosses the first vertical voxel boundary and store it in variable tMaxX. We perform a similar computation in y and store the result in tMaxY. The minimum of these two values will indicate how much we can travel along the ray and still remain in the current voxel.

Finally, we compute tDeltaX and tDeltaY. TDeltaX indicates how far along the ray we must move (in units of t) for the horizontal component of such a movement to equal the width of a voxel. Similarly,

we store in tDeltaY the amount of movement along the ray which has a vertical component equal to the height of a voxel.

The incremental phase of the traversal algorithm is very simple. The basic loop is outlined below:

```
loop{
    if(tMaxX < tMaxY) {
        tMaxX= tMaxX + tDeltaX;
        X= X + stepX;
    } else  {
        tMaxY= tMaxY + tDeltaY;
        Y= Y + stepY;
    }
    NextVoxel(X,Y);
}
```

We loop until either we find a voxel with a non-empty object list or we fall out of the end of the grid. Extending the algorithm to three dimensions simply requires that we add the appropriate z variables and find the minimum of tMaxX, tMaxY and tMaxZ during each iteration. This results in:

```
list= NIL;
do  {
    if(tMaxX < tMaxY) {
        if(tMaxX < tMaxZ) {
            X= X + stepX;
            if(X == justOutX)
                return(NIL); /* outside grid */
            tMaxX= tMaxX + tDeltaX;
        } else  {
            Z= Z + stepZ;
            if(Z == justOutZ)
                return(NIL);
            tMaxZ= tMaxZ + tDeltaZ;
        }
    } else  {
        if(tMaxY < tMaxZ) {
            Y= Y + stepY;
            if(Y == justOutY)
                return(NIL);
            tMaxY= tMaxY + tDeltaY;
        } else  {
            Z= Z + stepZ;
            if(Z == justOutZ)
                return(NIL);
            tMaxZ= tMaxZ + tDeltaZ;
        }
    }
    list= ObjectList[X][Y][Z];
} while(list == NIL);
return(list);
```

The loop above requires two floating point comparisons, one floating point addition, two integer comparisons and one integer addition per iteration. The initialization phase requires 33 floating point operations (including comparisons) if the origin of the ray is inside the grid and up to 40 floating point operations otherwise.

To correctly determine visibility, we must make sure that the "closest" intersection point is in the current voxel. This is best illustrated by the figure 2. The first voxel that has an object in its list that can be intersected is b. But the actual intersection of that object (B) is in voxel d with a closer object (A) in voxel c.

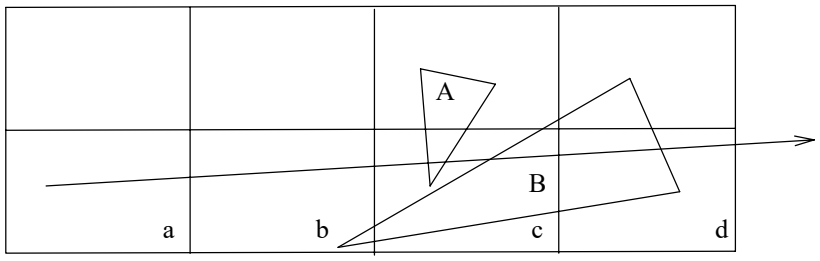

Figure 2

At first, checking to make sure that the intersection point is within the voxel would seem to require six floating point comparisons. But we can reduce these to just one: the value of t at the intersection point compared to the maximum value of t allowed in the current voxel. If it is less than or equal to the maximum allowed, it is in the current voxel; otherwise, we must continue traversal until either we reach the voxel in which the intersection occurs or find a closer object. The easiest way to perform this comparison is to include it into the incremental traversal code just after the minimum of tMaxX, tMaxY and tMaxZ is determined. Of course, we don't want to do this comparison until we have hit a surface so we have two traversal functions, one without the extra comparison and the other with. After an intersection is found we call the incremental traversal function again; this time we call the version with the extra comparison. If the intersection is in the current voxel, the function returns NIL and we stop. Otherwise, we continue traversal until either we find a non-empty voxel or the voxel in which the intersection occurred. Thus, for minimal cost, we can decide if the intersection is within the current voxel.

Avoiding Multiple Intersections

A major drawback[4] with current space subdivision schemes is that since objects may reside in more than one voxel, a ray may intersect the same object many times. We introduce a technique that blunts this criticism. But first we will outline what happens when we perform an intersection test with an object.

We shall assume that the ray, along with $\vec{u}$ and $\vec{v}$, has stored with it a pointer to information regarding the current visibility "winner"; that is, the object that is closest to the ray origin of all the objects that have been intersected so far. In particular, we store the t value of the intersection point. As we intersect a new candidate we compare the t value of the intersection point (if any) with that of the current winner. If the candidate is closer, it becomes the new winner and updates the winner data. Otherwise, it is rejected.

To solve the multiple intersection problem we add to each ray an integer variable, called rayID. Every ray shot through the grid will have it's own unique value for rayID. Also, each object will have stored with it the rayID of the ray that most recently performed an intersection test with it (this variable is initialized to 0, a value that no ray is ever allowed to have). Before an intersection test, the rayID of the ray and the object are compared. If they are equal, the object has previously been intersected by the same ray and the current intersection is not necessary. Otherwise, the intersection test is performed and the object's rayID is set to that of the ray's. In this manner, a ray intersection test need only be performed once for any object. Of course, this approach requires an extra integer comparison; this extra comparison

is well worth it as a full intersection test is typically several orders of magnitude more expensive.

The fact that multiple intersections are eliminated means that we do not have to spend as much time in determining the minimal number of voxels that an object occupies. Glassner indicated that he would include an object in a voxel's list only if the object's *surface* existed within the voxel. Computing these voxels is non-trivial, especially for complicated objects. Faster, though slightly less tight algorithms can be used as we no longer perform multiple intersections on the same object.

Results

The above traversal algorithm, with the indicated correction, has been implemented in C on a Sun 3/75 running Unix. The results of four test scenes are revealed in the table below. In all cases, the images were computed at 512 by 512 resolution with one sample per pixel and with one light source casting a shadow.

Figure	Time (min)	Grid Subdivision	Objects	Objects/ Voxel	Intersections/ Ray	Intersections/ Ray (with rayID)
4	49.5	20	4	0.3	4.6	1.7
5	44.8	20	31	0.9	5.3	2.0
6	21.7	30	62	0.2	1.9	1.4
7	32.6	40	3540	1.1	6.6	3.8

We see that even for very large numbers of objects, the number of objects that must actually be intersected with stays small. Also, the rayID optimization significantly reduces the number of objects that must be intersected. As the level of grid subdivision increases, the rayID optimization will become more significant as objects cover more voxels. Unfortunately, as we increase grid subdivision, voxel initialization time, traversal time as well as memory usage also become significant thus ultimately limiting the subdivision rate.

Figure 3 plots the execution time of rendering the scene in figure 7 for several different levels of subdivision. We see that for low levels of subdivision, rendering the scene is very expensive but the cost plummets with only moderate increases in the subdivision rate.

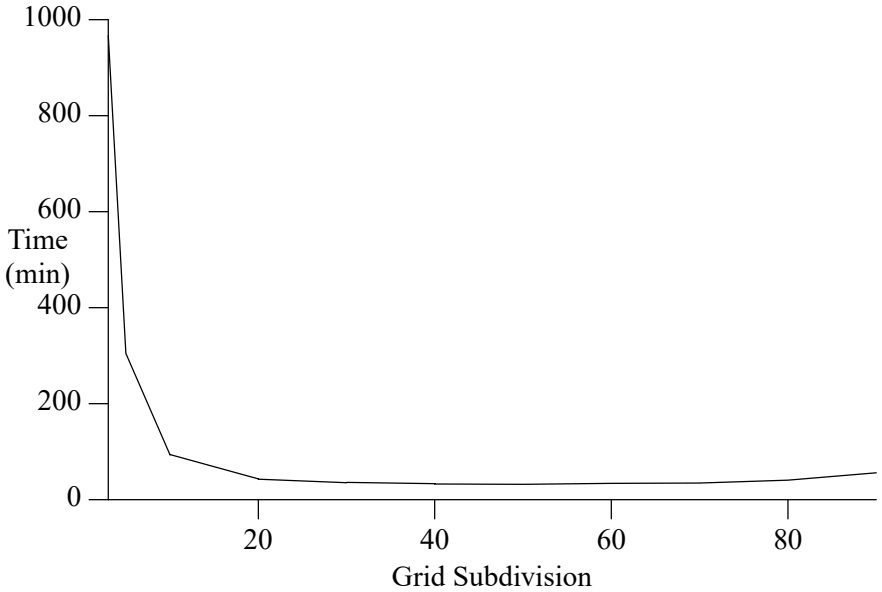

Figure 3

Conclusions and Future Research

We have introduced a space partition algorithm that requires very few floating point operations. Multiple intersection tests for objects are eliminated.

We are currently studying the feasibility of reducing the algorithm to just integer operations to further reduce operating time. Also, we are looking to see how easily the algorithm can be modified to traverse octree space subdivision schemes.

The quick traversal times through a grid suggest its use for more than just simple intersection culling. For example, we could store in the grid the seed points of iterative processes. A good candidate is ray tracing bicubic parametric patches using Newton's method for finding roots.[9, 10]

Another possibility is using grids to reduce shadow rays. If there are many light sources, shadow rays are very important.[11] At the cost of two bits per light source per voxel (one indicating that the voxel is completely in shadow, the other indicating that nothing blocks the voxel from seeing the light source) a great number of these rays no longer are necessary.

References

1. T. Whitted, "An Improved Illumination Model for Shaded Display," *Comm. of the ACM,* 23(6), pp. 343-349 (June 1980).
2. S.M. Rubin and T. Whitted, "A 3-Dimensional Representation for Fast Rendering of Complex Scenes," *Computer Graphics,* 14(3), pp. 110-116 (July 1980).
3. H. Weghorst, G. Hooper, and D.P. Greenberg, "Improved Computational Methods for Ray Tracing," *ACM Trans. on Graphics,* 3(1), pp. 52-69 (January 1984).
4. T.L. Kay and J.T. Kajiya, "Ray Tracing Complex Scenes," *Computer Graphics,* 20(4), pp. 269-278 (August 1986).
5. J.G. Cleary, B. Wyvill, G.M. Birtwistle, and R. Vatti, "Multiprocessor Ray Tracing," *Research Report No. 83/128/7 Dept. of Computer Science University of Calgary* (1983).
6. A.S. Glassner, "Space Subdivision for Fast Ray Tracing," *IEEE Computer Graphics and Applications,* 4(10), pp. 15-22 (October 1984).
7. A. Fujimoto and K. Iwata, "Accelerated Ray Tracing," *Proc. CG Tokyo '85,* pp. 41-65.
8. A. Fujimoto, T. Tanaka, and K. Iwata, "ARTS: Accelerated Ray-Tracing System," *IEEE Computer Graphics and Applications,* 6(4), pp. 16-26 (April 1986).
9. D.L. Toth, "On Ray Tracing Parametric Surfaces," *Computer Graphics,* 19(3), pp. 171-179 (July 1985).
10. M.A.J. Sweeney and R.H. Bartels, "Ray Tracing Free-Form B-Spline Surfaces," *IEEE Computer Graphics and Applications,* 6(2), pp. 41-49 (February 1986).
11. E.A. Haines and D.P. Greenberg, "The Light Buffer: A Shadow-Testing Accelerator," *IEEE Computer Graphics and Applications,* 6(9), pp. 6-16 (September 1986).