Simple space subdivision for faster ray tracing

Patrick S. Várilly* and Francisco J. Torres-Rojas
varilly@mit.edu, torres@ic-itcr.ac.cr
Centro Nacional de Alta Tecnología (CENAT)
Costa Rica

Abstract

Traditional ray tracing is extremely slow: images take many minutes, even hours to render. Several methods have been developed over the last 15 years to speed up ray tracing, but remain unused from either lack of awareness or apparent complexity of implementation. In this paper, we examine one such technique: uniform spatial subdivision, in which space is divided into a regular grid of voxels which can be traversed efficiently. We give a simple method for the preprocessing stage, and then examine a method of traversal not much more complicated than Bresenham's line-drawing algorithm. We conclude with some empirical evidence of the vast speed increase spatial subdivision can afford. We hope this exposition will allow a more widespread use of spatial subdivision, resulting in shorter rendering times and more fruitful experimentation.

Keywords: computer graphics, ray tracing, optimization, spatial subdivision, octree, voxel

*Patrick S. Várilly was an invited researcher at CENAT, currently he is a student at M.I.T.

1 Introduction

Ray-tracing, initially used for hidden surface removal by Appel [1], is a 3D rendering technique pioneered by Kay [2] and Whitted [3]. It determines a pixel's color by tracing rays light from the viewer towards the objects in a scene and performing illumination calculations on the visible surfaces, frequently spawning new rays. This method, capable of handling reflections, refractions and shadows in an elegant manner, has produced some of the most realistic computer-generated images to date.

One of the main problems of traditional ray-tracing is the immense computational overhead that is involved. It's not unusual for images to take many minutes, or even hours to render at very modest resolutions, making it impractical for real-time 3D applications (such as gaming or scientific visualization) and animation. This effect is worsened by some more advanced techniques, such as distributed ray-tracing [4], where each pixel requires 16 times as many rays to render than with traditional methods. The main bottleneck is the number of ray-object intersection calculations. Whitted reports that as much as 95% of rendering time is spent on these calculations.

Previous work (especially during the mid-80s) has managed to improve performance substantially by performing space subdivision. Glassner [5] used a technique where space is hierarchically subdivided into an octree (the three-dimensional analogue of the binary tree); the leaves of the octree are cuboidal regions of space. Then each ray is tracked through the leaves of the octree, and intersection calculations are performed only on the objects inside each leaf. This technique makes tracing as much as 35 times faster, but other methods provide even greater speed boosts.

Fujimoto, Tanaka and Iwata [6] take a slightly different approach which results in an order-of-magnitude improvement. They divide space into a uniform grid of voxels they call SEADS. In the preprocessing stage, one builds a list of objects which cross each voxel. For tracking the ray, they develop an algorithm called a 3D-DDA which traverses the SEADS in an extremely efficient manner, much like the way Bresenham's algorithm finds the pixels closest to a 2D line. Their experiments reveal an improvement of over 400 times in rendering speed. But, once again, coding the 3D-DDA is very error-prone, and not enough details are provided in [6] to help the implementer.

This has created a situation where, although relatively simple methods exist to make ray-tracing many times faster, very few people in the computer graphics community actually use them. Our purpose in this paper is to give a simple exposition of standard space subdivision techniques, with enough detail to make it easy to implement. We also provide some examples of the drastic improvement which the technique affords.

In Section 2, we give a method for initializing the voxel grid which spatial subdivision creates. In Section 3, we explore how an arbitrary ray is tracked through this voxel grid efficiently. We implement this method and show the results we obtained in Section 4. In Section 5, we examine some possible enhancements to the techniques given in the paper. Finally, Section 6 summarizes our conclusions.

2 Initializing the voxel grid

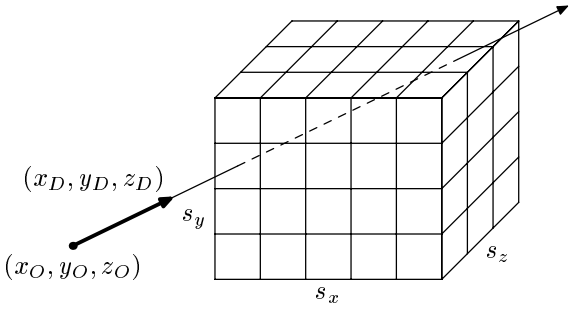


Figure 1: A ray traced through the voxel grid

We begin by establishing a suitable region of space (e.g., the union of all objects' bounding boxes) and subdividing it at uniform intervals to create a voxel grid in space with $s_x \times s_y \times s_z$ voxels (see Fig. 1).

Our first task is to determine which voxels intersect a given object. Clearly, given a particular object, we need only consider those voxels which intersect its bounding box. We thus test each such voxels for intersection with the object.

For this test, we propose a general method to identify which voxels are crossed by a surface given in implicit form (some extra voxels may also be flagged).

In general, a surface is defined implicitly by the formula

$$F(x, y, z) = 0. \tag{1}$$

Let (x_0, y_0, z_0) and (x_1, y_1, z_1) be the coordinates in world space of the current voxel's corners. The point (X, Y, Z) is in this voxel if

$$\begin{aligned} x_0 \le X \le x_1, \\ y_0 \le Y \le y_1, \\ z_0 \le Z \le z_1. \end{aligned} \tag{2}$$

We want to find whether $F(X, Y, Z) = 0$, given (2). This can only happen if

$$\min(F) \le 0 \le \max(F), \tag{3}$$

because F is continuous. If this isn't true, F can never be 0 in the voxel, so the object never crosses it.

Our method provides simple estimates for $\min(F)$ and $\max(F)$ which we can insert into (3). We will use capital letters to distinguish between the estimate (Min) and the true value (min) of the minimum of F within the voxel.

Calculating Min and Max is relatively easy. We will use E_1 and E_2 as arbitrary expressions, and k an arbitrary constant. Then we can apply the following simple rules:

$$\begin{aligned} \mathrm{Min}(k) &= k, \\ \mathrm{Min}(E_1 + E_2) &= \mathrm{Min}(E_1) + \mathrm{Min}(E_2), \\ \mathrm{Min}(E_1 - E_2) &= \mathrm{Min}(E_1) - \mathrm{Max}(E_2), \\ \mathrm{Min}(kE_1) &= \begin{cases} k\,\mathrm{Min}(E_1), & \text{for } k \text{ positive}, \\ k\,\mathrm{Max}(E_1), & \text{for } k \text{ negative}, \end{cases} \\ \mathrm{Min}(E_1^2) &= \begin{cases} 0, & \text{if } \mathrm{Min}(E_1) \le 0 \le \mathrm{Max}(E_1), \\ \min[\mathrm{Min}(E_1)^2, \mathrm{Max}(E_1)^2], & \text{otherwise.} \end{cases} \end{aligned}$$

The rules for Max are analogous.

Our method can be easily extended to handle inequalities such as $F(X, Y, Z) \le 0$ as conditions for intersections. This can be used to implement cutting planes, where an object consists of an implicit surface (e.g., a sphere), with the portions on the back sides of the cutting planes removed. The same principle can be used to intersect CSG (Constructive Solid Geometry) objects with voxels.

We now apply this method to test for plane, sphere, disk and polygon-voxel intersections.

2.1 Plane-voxel intersection

A plane is defined by

$$P(x, y, z) := Ax + By + Cz + D = 0.$$

We wish to calculate $\mathrm{Min}(P)$ and $\mathrm{Max}(P)$. We proceed as follows:

$$\mathrm{Min}(P) = \mathrm{Min}(Ax) + \mathrm{Min}(By) + \mathrm{Min}(Cz) + D.$$

Then,

$$\mathrm{Min}(Ax) = \begin{cases} Ax_0 & \text{for } A \text{ positive}, \\ Ax_1 & \text{for } A \text{ negative}. \end{cases}$$

The other terms are similar. A code listing for this intersection test is given in Fig. 2.

2.2 Sphere-voxel intersection

A sphere centered at (a, b, c) with radius r is defined by

$$S(x, y, z) := (x - a)^2 + (y - b)^2 + (z - c)^2 - r^2 = 0. \tag{4}$$

We need $\mathrm{Min}(S) \le 0 \le \mathrm{Max}(S)$. Let us calculate $\mathrm{Min}(S)$; $\mathrm{Max}(S)$ is similar:

$$\mathrm{Min}(S) = \mathrm{Min}[(x - a)^2] + \mathrm{Min}[(y - b)^2] + \mathrm{Min}[(z - c)^2] - r^2.$$

```
if( A > 0 ) minX = A*x0, maxX = A*x1;
else        minX = A*x1, maxX = A*x0;
if( B > 0 ) minY = B*y0, maxY = B*y1;
else        minY = B*y1, maxY = B*y0;
if( C > 0 ) minZ = C*z0, maxZ = C*z1;
else        minZ = C*z1, maxZ = C*z0;

minP = minX + minY + minZ + D;
maxP = maxX + maxY + maxZ + D;

return (minP <= 0) && (0 <= maxP);
```

Figure 2: Intersecting a voxel with a plane

From the rules above, we see that

$$\mathrm{Min}[(x-a)^2] = \begin{cases} 0, & \text{if } \mathrm{Min}(x-a) \le 0 \le \mathrm{Max}(x-a), \\ \min[\mathrm{Min}(x-a)^2, \mathrm{Max}(x-a)^2], & \text{otherwise.} \end{cases}$$

and so,

$$\mathrm{Min}[(x-a)^2] = \begin{cases} 0, & \text{if } x_0 \le a \le x_1, \\ \min[(x_0-a)^2, (x_1-a)^2], & \text{otherwise.} \end{cases}$$

The other two terms are calculated in the same way. A partial code listing for testing whether a sphere intersects with a voxel is given in Fig 3. It is easy to see how this method can be extended for a general quadric.

```
// X coordinate
u = (x0-a)*(x0-a);
v = (x1-a)*(x1-a);
maxX = max( u, v );
if( (x0 <= a) && (a <= x1) ) minX = 0;
else minX = min( u, v );

// Y coordinate
...

minS = minX + minY + minZ;
maxS = maxX + maxY + maxZ;

return (minS <= 0) && (0 <= maxS);
```

Figure 3: Intersecting a voxel with a sphere

2.3 Disk-voxel intersection

A disk is simply the intersection of a plane and the interior of a sphere. We have already shown how perform a plane-voxel intersection. The interior of the sphere is simply (4) modified slightly:

$$S(x, y, z) \le 0.$$

This requires only that $\mathrm{Min}(S) \le 0$. This can be done as in the case of a sphere.

2.4 Polygon-voxel intersection

Glassner suggests clipping the polygon to the voxel's faces using a Sutherland-Hodgman clipper [7]. While this solves the problem exactly, we advise against it for two reasons:

- Its implementation is complicated and error-prone, and its execution is moderately expensive.

- Most polygons are quite small, and so intersect very few voxels.

This last condition is especially true of large polygon meshes. Hence, we have found that simply intersecting the voxels inside the polygon's bounding box with the plane of the polygon works extremely well, and it's very simple to implement.

3 Tracking a ray through space

A ray has an origin (x_O, y_O, z_O) and a direction vector (x_D, y_D, z_D). We wish to find which object are hit by this ray. We can do this by finding out which voxels the ray goes through, and intersecting it with those objects inside these voxels. A ray in general will start outside the voxel grid, enter the grid, and then exit the grid. Any objects not completely surrounded by the voxel grid are kept in a list of *outside objects*. Initially, we build a sorted list of intersections between the ray and the outside objects. If the ray doesn't pass through the voxel grid, then intersection calculation is the same as in ordinary ray-tracing.

Otherwise, we calculate the entry and exit coordinates of the ray into the grid, using a standard ray-box intersection test (see, for example, [8]). We then convert these into voxel coordinates (where the corners of each voxel are located at integer coordinates) and proceed to track the ray along this segment. The subray stretches from (x_0, y_0, z_0) to (x_1, y_1, z_1) and has direction vector (A, B, C).

To track it through the voxel grid, we use the "6-line tripod algorithm," due to Cohen-Or and Kaufman [9], which we now describe briefly. We'll assume $A, B, C > 0$; the generalization is straightforward.

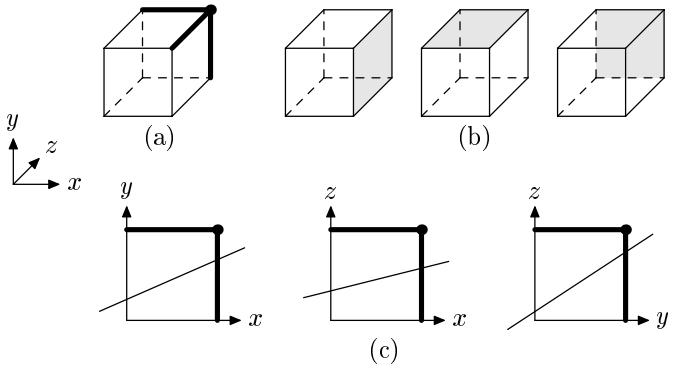


Figure 4: The 6-line tripod algorithm (after Cohen-Or and Kaufman). (a) The tripod placed on a voxel's corner. (b) The x, y and z faces of the voxel. (c) The projection of the ray on the coordinate planes; here, clearly the ray doesn't exit through the y or z face, so it must leave through the x face.

Suppose we've already identified one voxel through which the ray passes. The algorithm works by placing a "tripod" on the corner of this voxel and determining which face the ray exits through by a method of elimination (see Fig. 4). It uses the projections of the ray on the three coordinate planes, which are:

- On the xy plane, $e_{xy} := Ay - Bx - D_1 = 0$.
- On the yz plane, $e_{yz} := Bz - Cy - D_1 = 0$.
- On the xz plane, $e_{xz} := Az - Cx - D_1 = 0$.

For an arbitrary (x, y, z), the sign of e_{xy}, e_{yz} and e_{xz} tells us whether the projected point is above or below the projected line. In particular, if we choose the corner of the tripod, the sign tells us which face the ray does *not* cross (Fig. 4(c)):

- if $e_{xy} < 0$, the ray doesn't cross the x face;
- if $e_{yz} < 0$, the ray doesn't cross the y face;
- if $e_{xz} < 0$, the ray doesn't cross the x face.[1]

[1] Actually, these sign tests are misstated in [9].

```
// Initialization
exy = A*(floor(y0)+1-y0) - B*(floor(x0)+1-x0);
eyz = B*(floor(z0)+1-z0) - C*(floor(y0)+1-y0);
exz = A*(floor(z0)+1-z0) - C*(floor(x0)+1-x0);
ptr = &grid[floor(x0)][floor(y0)][floor(z0)];

voxelptr trackray( ... );
...
while( n-- ) {
    voxel = ptr;
    if( exy < 0 )      // Not x
        if( eyz < 0 )  // Not y
            ptr += offsetz, eyz += B, exz += A;
        else           // Not z
            ptr += offsety, exy += A, eyz -= C;
    else               // Not y
        if( exz < 0 )  // Not x
            ptr += offsetz, eyz += B, exz += A;
        else           // Not z
            ptr += offsetx, exy -= B, exz -= C;
    return voxel;
}
return NULL;
```

Figure 5: Tracking a ray through the voxel grid

Two of these comparisons are enough to establish which face the ray exits through. The tripod is moved accordingly, suitable adjustments are made to e_{xy}, e_{yz} and e_{xz}, and the algorithm begins again.

The first voxel crossed by the ray is $(\lfloor x_0 \rfloor, \lfloor y_0 \rfloor, \lfloor z_0 \rfloor)$, so the tripod's corner begins at $(\lfloor x_0 \rfloor + 1, \lfloor y_0 \rfloor + 1, \lfloor z_0 \rfloor + 1)$. We can calculate e_{xy} at this point by knowing that (x_0, y_0, z_0) is *exactly* on the line. Hence,

$$0 = Ay_0 - Bx_0 - D_1. \tag{5}$$

Since we wish to determine,

$$e_{xy} = A(\lfloor y_0 \rfloor + 1) - B(\lfloor x_0 \rfloor + 1) - D_1, \tag{6}$$

we subtract (5) from (6) to get,

$$e_{xy} = A(\lfloor y_0 \rfloor + 1 - y_0) - B(\lfloor x_0 \rfloor + 1 - x_0). \tag{7}$$

The initial values of e_{yz} and e_{xz} are calculated similarly.

Finally, the last voxel crossed by the ray is $(\lfloor x_1 \rfloor, \lfloor y_1 \rfloor, \lfloor z_1 \rfloor)$. Let $\Delta x = \lfloor x_1 \rfloor - \lfloor x_0 \rfloor$, $\Delta y = \lfloor y_1 \rfloor - \lfloor y_0 \rfloor$ and $\Delta z = \lfloor z_1 \rfloor - \lfloor z_0 \rfloor$. It's clear that the ray crosses $|\Delta x| + |\Delta y| + |\Delta z| + 1$ voxels (the final +1 accounts for the initial voxel). This gives us a suitable stopping condition for the algorithm's main loop.

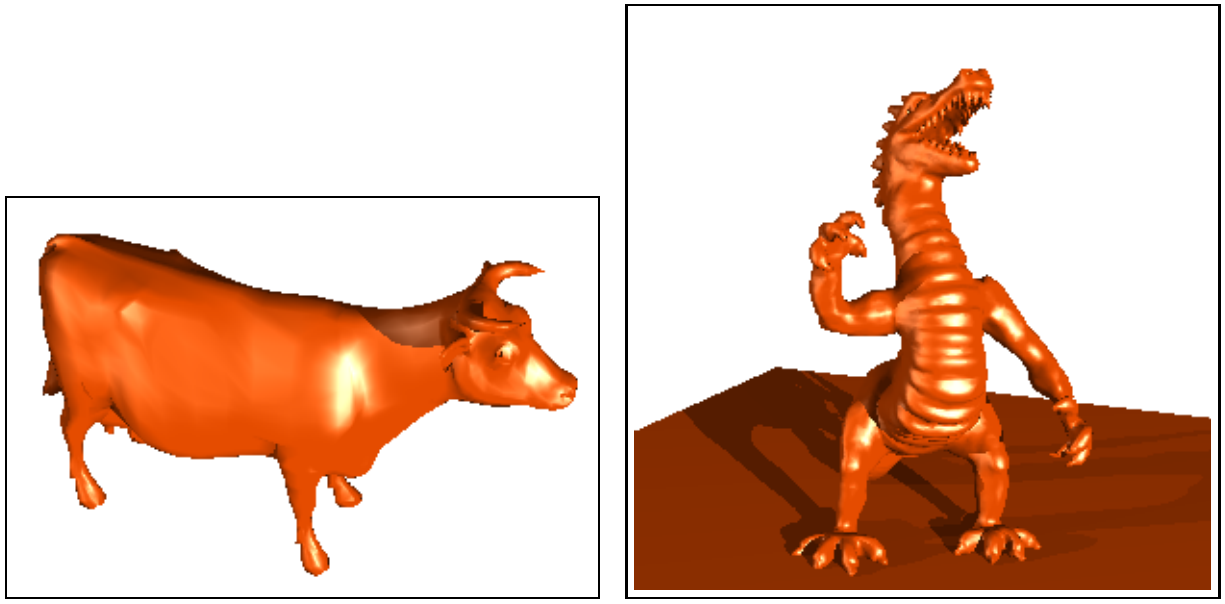
Generalizing the algorithm can be done in two steps. First, the tripod's initial position depends on the signs of A, B and C (this affects the +1 in the initial values of e_{xy}, e_{yz} and e_{xz}). Secondly, the sign tests are reversed if the ray has a negative slope in the projected plane. Thus, if $A, C > 0$ and $B < 0$, the tripod would start at $(\lfloor x_0 \rfloor + 1, \lfloor y_0 \rfloor, \lfloor z_0 \rfloor + 1)$ and the sign tests for e_{xy} and e_{yz} would be reversed.

A code listing, similar to that in [9], is given in Fig. 5. Compared to Fujimoto et al.'s 3D-DDA, the tripod algorithm is both faster and simpler to implement. The 3D-DDA can output up to 3 voxels per loop, which requires some bookkeeping to handle.

4 Results

The algorithms described here were coded in C and tested on an Apple iMac with a 600 MHz G3 processor.

We used three complex models to test the effectiveness of space subdivision. Figure 6(a) shows a symmetrical cow (with 5,804 triangles) commonly used in computer graphics tests. Figure 6(b) shows a more complex dragon (with 50,761 triangles). Figure 7(a) shows the Stanford Bunny (with 69,451 triangles; available at [10]), while Figure 7(b) shows a metallic version of this bunny, reflecting a menacing sky. All models are rendered at 300×300 resolution (except the cow, which is rendered at 300×200). The cow and dragon are illuminated by two light sources; the Stanford bunny has an additional light source behind it.



(a) A Cow (5,804 triangles)

(b) A Dragon (50,761 triangles)

Figure 6: Some dense models

The techniques outlined in this paper show an order of magnitude increase in rendering speeds; the exact timing results are given in Table 1 (based on Table 1 of [5]).

The resolution of the voxel grid used is important, both for memory considerations and even more so for speed. Table 2 shows timing results and voxel occupancy statistics for the Stanford bunny at different grid resolutions (we used the same number of voxels in the x, y and z axis for the voxel grid, but this is not necessary in general). Table 3 shows more detailed voxel content information for different models at different resolutions. Memory requirements for the voxel grid are quite modest by modern standards. In the $100 \times 100 \times 100$ Stanford Bunny test, for example, only 6.5Mb went to maintaining the grid. In contrast, nearly 6 times as much memory was used by the mesh (this is, in general, not the case; it is simply that our implementation of a polygon mesh was not optimized for memory).

It is interesting to see where the greatest computational effort is being exerted to render each image. To that end, we've used density plots. These show the numbers of intersection calculations made per ray for every pixel. Fig. 8 shows one such density plot for the cow model. It's deliberately rendered at a coarse voxel resolution to display the individual voxels. Fig. 9(a) shows the same plot for the Stanford Bunny. Here, we see the most complex parts of the model are those near silhouettes. This is because the rays here cross many heavily populated voxels without touching any of the objects inside them; hence, many more

Model Grid Resolution	Standard Cow $50 \times 50 \times 50$	Dragon $100 \times 100 \times 100$	Stanford Bunny $50 \times 50 \times 50$	Stanford Bunny $100 \times 100 \times 100$
Number of objects	5,804	50,761	69,451	69,451
Number of rays traced	106,758	170,330	204,609	204,609
Avg. objects per voxel	0.435	0.136	1.355	0.313
Number of voxels	125,000	1,000,000	125,000	1,000,000
Old intersections	619,623,432	8,646,121,130	14,210,229,659	14,210,229,659
New intersections	1,422,388	7,555,069	9,643,617	4,756,303
Avg. intersections/ray	13.3	44.4	47.1	23.2
Old Time	13m 02.37s	3h 30m 45s (estimate)	4h 57m 00s (estimate)	4h 57m 00s (estimate)
Voxel Grid Build Time	0.09s	0.64s	0.46s	0.96s
New Rendering Time	1.79s	7.83s	8.51s	5.85s
Speed-up factor	×435	×1615	×2092	×3044

Table 1: Timing statistics using uniform space subdivision

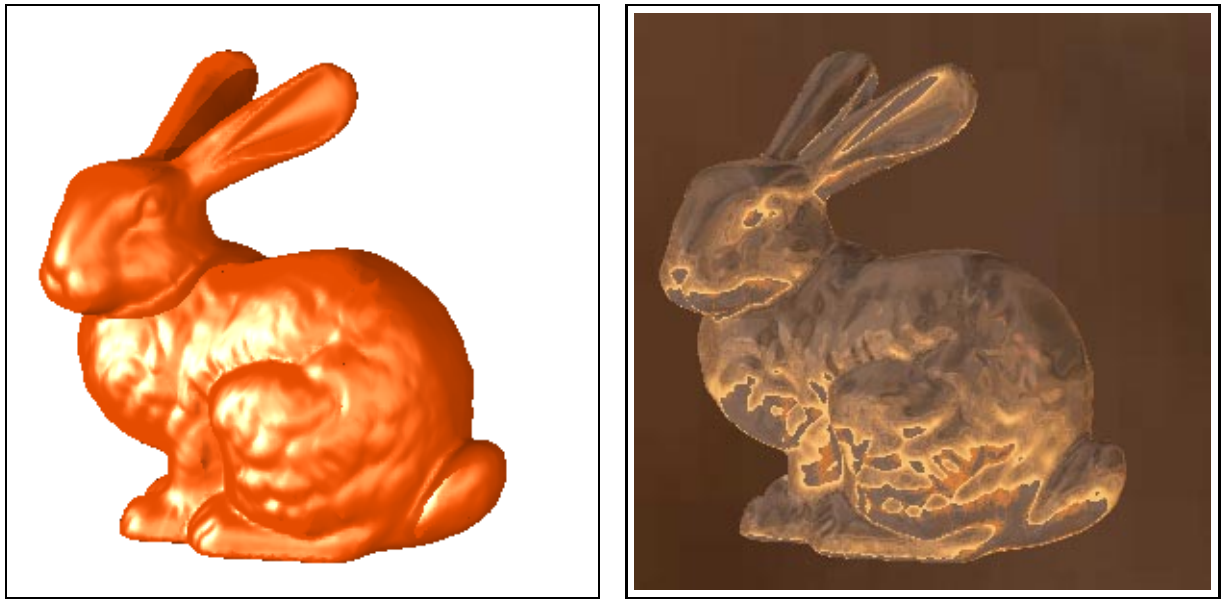
Voxel Grid Resolution	Grid Init Time(s)	Rendering Time(s)	Total Time(s)[a]	Average objects per voxel	Average voxels per object	Average intersections/ray
10	0.31	130.53	130.84	84.943	1.22	509.1
20	0.32	31.02	31.34	12.900	1.48	165.9
30	0.36	15.58	15.94	4.571	1.78	91.2
40	0.40	10.63	11.03	2.269	2.09	61.7
50	0.44	8.52	8.96	1.355	2.44	47.1
60	0.50	7.33	7.83	0.900	2.80	38.2
70	0.58	6.65	7.23	0.646	3.19	32.5
80	0.69	6.18	6.87	0.488	3.60	28.3
90	0.79	5.98	6.77	0.385	4.04	25.4
100	0.95	5.80	6.75	0.313	4.51	23.2

[a] Estimated traditional rendering time: 4hr 57min $\approx$ 17820s

Table 2: Timing statistics for the Stanford Bunny, using different voxel grid resolutions

Objects in voxel	Stanford Bunny $100 \times 100 \times 100$		Stanford Bunny $50 \times 50 \times 50$		Dragon $100 \times 100 \times 100$		Dragon $50 \times 50 \times 50$		Cow $30 \times 30 \times 30$	
	#	%	#	%	#	%	#	%	#	%
0	958296	95.83%	114662	91.73%	985405	98.54%	121422	97.14%	23103	85.57%
1	1549	0.15%	251	0.20%	100	0.01%	13	0.01%	262	0.97%
2	961	0.10%	133	0.11%	6902	0.69%	1123	0.90%	241	0.89%
3	2133	0.21%	267	0.21%	248	0.02%	64	0.05%	284	1.05%
4	3568	0.36%	266	0.21%	2630	0.26%	973	0.78%	399	1.48%
5	3369	0.34%	269	0.22%	170	0.02%	49	0.04%	383	1.42%
6	3036	0.30%	287	0.23%	106	0.01%	10	0.01%	359	1.33%
7	4357	0.44%	284	0.23%	130	0.01%	35	0.03%	366	1.36%
8	7624	0.76%	356	0.28%	372	0.04%	258	0.21%	406	1.50%
9	3997	0.40%	323	0.26%	115	0.01%	18	0.01%	242	0.90%
10	3311	0.33%	296	0.24%	127	0.01%	19	0.02%	224	0.83%
11	3206	0.32%	313	0.25%	112	0.01%	9	0.01%	142	0.53%
12	3391	0.34%	344	0.28%	130	0.01%	17	0.01%	128	0.47%
13	482	0.05%	335	0.27%	114	0.01%	11	0.01%	74	0.27%
14	262	0.03%	315	0.25%	116	0.01%	10	0.01%	70	0.26%
15	185	0.02%	355	0.28%	130	0.01%	11	0.01%	44	0.16%
16	82	0.01%	467	0.37%	138	0.01%	8	0.01%	44	0.16%
17	94	0.01%	407	0.33%	132	0.01%	8	0.01%	32	0.12%
18	96	0.01%	632	0.51%	139	0.01%	13	0.01%	25	0.09%
19	0	0.00%	393	0.31%	125	0.01%	11	0.01%	22	0.08%
20+	1	0.00%	4045	3.24%	2559	0.26%	918	0.73%	150	0.56%

Table 3: Voxel occupancy for various models at different grid resolutions



(a) Plastic

(b) With a twist

Figure 7: The Stanford Bunny (69,451 triangles)

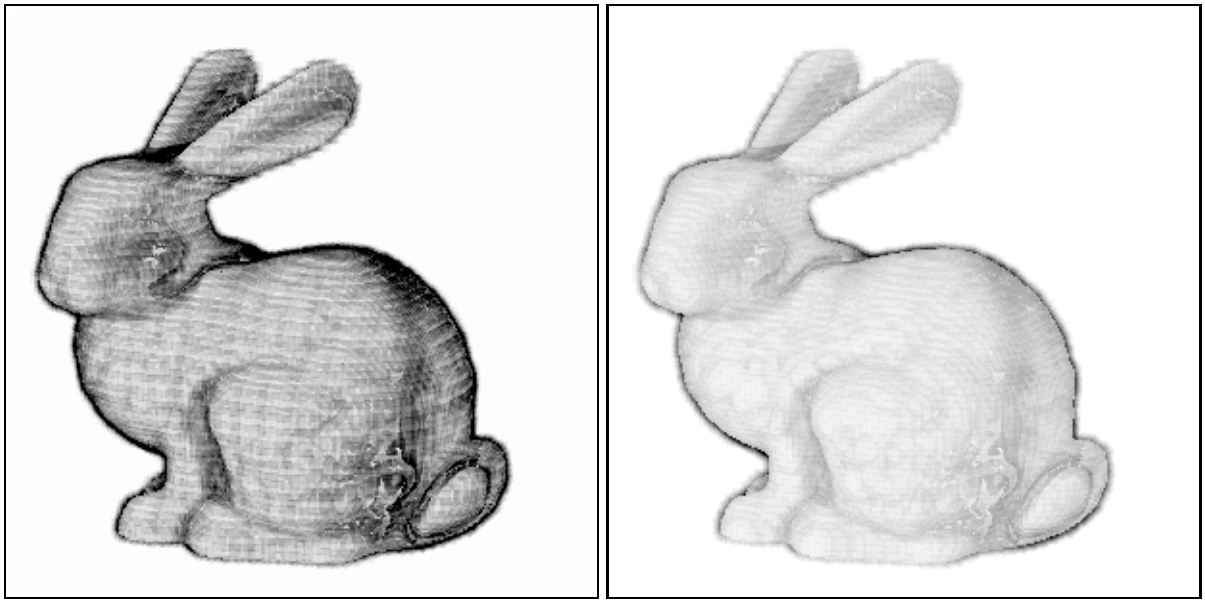
non-empty voxels need to be scanned per ray. This is shown in Fig. 9(b), where the number of non-empty voxels crossed per ray traced is plotted for each pixel.

5 Evaluation and Future Work

The most important obstacle to the general use of spatial subdivision is its memory overhead, which becomes significant for voxel grid resolutions over $250 \times 250 \times 250$ (where simply storing an empty grid takes up just under 60 Mb). We are currently experimenting on ways to remedy this problem. While many researchers use octrees (or an octree-spatial subdivision hybrid algorithm) to reduce the amount of space wasted by large regions of empty voxels, they are hard to implement efficiently, and the amount of overhead they entail is non-trivial. More importantly, though, the results of Fujimoto et al suggest that by using octrees, one incurs a significant speed penalty. However, we estimate the amount of space saved in the $100 \times 100 \times 100$ Stanford Bunny test to be around 50%, and this should increase with the voxel grid resolution.

A very simple approach to save memory is to store each voxel's object list in a compact manner. In our tests, we used a simple linked list. Each node contained a pointer to an object and a pointer to the next node

Figure 8: Density plot of intersections per ray for the Standard Cow, using a $30 \times 30 \times 30$ voxel grid. Scale: white = 0 inters/ray; black = 60 or more inters/ray. Here, a coarse grid and a tight scale has been used to show the underlying voxel grid structure.



(a) Density plot of intersections per ray needed for the Stanford Bunny, using a $70 \times 70 \times 70$ voxel grid. Scale: white = 0 inters/ray; black = 100 or more inters/ray. The most complex parts are the edges, since each ray there intersects many populated voxels before hitting a polygon or the background.

(b) Density plot of non-empty voxels crossed per ray for the Stanford Bunny, using a $70 \times 70 \times 70$ voxel grid. Scale: white = 0 voxels; black = 20 or more voxels

Figure 9: Density plots for the Stanford Bunny

on the list; hence, half the memory for non-empty voxels (about a third of the total voxel grid memory) is wasted. We suggest linking blocks of three or four nodes at a time. That is, each node holds pointers to the next 3 or 4 objects on the list. In this way, we save space for voxels with many objects inside them, while adding only a few lines of code and very small amounts of wasted space for voxels with just 1 or 2 objects. Using this approach on the Stanford Bunny, we saved 9.4% of voxel grid memory.

Since spatial subdivision encourages coherent memory access, managing the processor cache efficiently becomes extremely important. The above method for storing voxel content lists allows more objects in the lists to be in the cache simultaneously, thus improving performance. Another way to exploit coherent accesses is to trace 5×5 square blocks of the image at each point, since the rays traced for these pixels will tend to travel through the same voxels.

The techniques described here fail for very large models, such as those freely available from Georgia Tech [10]. These models, produced by automatic scanning of real-life objects, tend to have hundreds of thousands, if not millions, of polygons. If we were to try and render them with spatial subdivision as described here, the memory requirements would be so large so as to make it impossible on current hardware. More ambitious models, such as the David from the Digital Michelangelo Project [11] (with over 2 billion polygons), make this approach impossible in most foreseeable hardware. One possible solution is to divide the voxel grid into "chunks" (e.g. of size $10 \times 10 \times 10$) which are stored on disk. Then, as a ray is traced, old chunks are deleted and new chunks are read in to find all the relevant voxel data. We have yet to test this approach.

An interesting possibility which we are actively investigating is using the voxel grid to help speed up animation. The voxel grid would first be extended to 4-dimensional spacetime (see Glassner [12] for details). Suppose further that, for each object, we can find out how long that object will remain stationary. Then each voxel is augmented with information as to how long the entire voxel's contents will remain stationary. For each traced ray, we would record its "maximum lifetime" by finding the minimum lifetime of all the voxels the ray goes through. Thus, if the camera is not moving, and most of the background is stationary, subsequent frames can directly reuse the ray's resulting color with no tracing as long as the ray's maximum lifetime is greater than the time of the frame.

6 Conclusions

We have found uniform space subdivision to be an extremely useful technique to implement in any ray-tracer. It provides enormous speed boosts for relatively little effort. The implementation of the ray tracker is not much harder than that of Bresenham's line drawing algorithm: our entire implementation has around 300 lines of code. Furthermore, the method we outline for intersecting objects with voxels is also very straightforward to code. We hope this simplicity allows the use of spatial subdivision techniques to become more common.

References

[1] A. Appel, "Some Techniques for Machine Renderings of Solids", *AFIPS Conference Proceedings*, Vol. 32, 1968, 37–45.

[2] D. S. Kay, "Transparency, Refraction and Ray Tracing for Computer Synthesized Images", master's thesis, Cornell University, Ithaca, N.Y., Jan. 1979.

[3] T. Whitted, "An Improved Illumination Model for Shaded Display," *Comm. ACM*, Vol. 23, No. 6, Jun. 1980, pp. 343–349.

[4] R. L. Cook, T. Porter, and L. Carpenter, "Distributed ray tracing," *Computer Graphics*, Vol. 18, No. 3, 1984, pp. 137–145.

[5] A. Glassner, "Space subdivision for fast ray tracing," *IEEE CG&A*, Vol. 4, No. 10, Oct. 1984, pp. 15–22.

[6] A. Fujimoto, T. Tanaka, and K. Iwata, "ARTS: Accelerated Ray-Tracing System," *IEEE CG&A*, Vol. 6, No. 4, Apr. 1986, pp. 16–26.

[7] I. E. Sutherland and G. W. Hodgman, "Reentrant Polygon Clipping," *Comm. ACM*, Vol. 17, No. 1, Jan. 1974, pp. 32–42.

[8] A. Watt, *3D Computer Graphics*, Addison-Wesley, Harlow, England, 1993; p. 283.

[9] D. Cohen-Or and A. Kaufman, "3D Line Voxelization and Connectivity Control," *IEEE CG&A*, Vol. 17, No. 6, Nov.-Dec. 1997, pp. 80–87.

[10] G. Turk and B. Mullins, "Large Geometric Models Archive," `http://www.cc.gatech.edu/projects/large_models/`.

[11] "The Digital Michelangelo Project," `http://www-graphics.stanford.edu/projects/mich/`.

[12] A. Glassner, "Spacetime ray tracing for animation," *IEEE CG&A*, Vol. 8, No. 2, Mar. 1988, pp. 60–70.