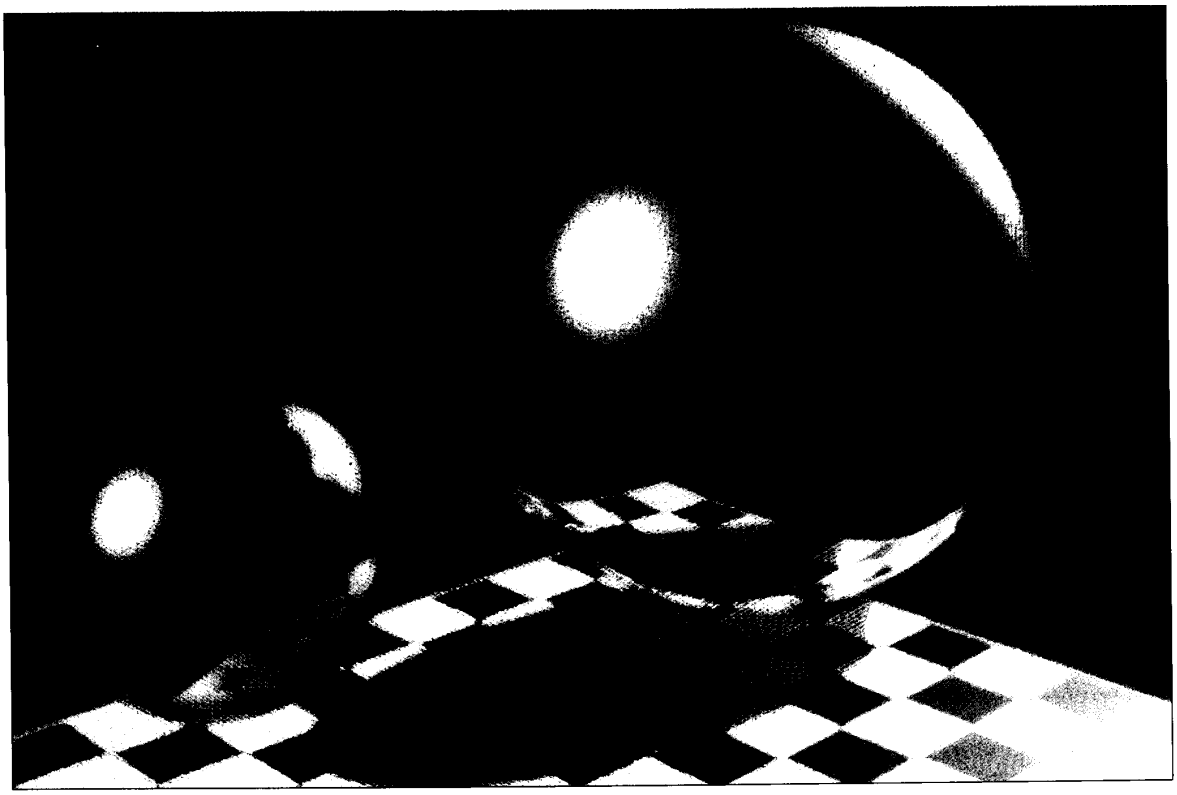


Discrete Ray Tracing

Roni Yagel
Ohio State University

Daniel Cohen
Tel-Aviv University

Arie Kaufman
State University of New York at Stony Brook

This ray tracing method, called 3D raster ray tracing, is insensitive to a scene's complexity and thus substantially improves computational speed over existing algorithms.

Ray tracing, despite its superior image quality, popularity, power, and simplicity, has the reputation of being computationally very expensive. The basic ray-tracing operation calculates the intersection points between rays and objects. This operation reportedly consumes at least 95 percent of the computation time in rendering complex scenes. A major strategy for reducing the cost of ray tracing involves reducing the average cost of ray-scene intersection, either by devising efficient algorithms for intersecting rays with specific objects or by reducing the number of objects tested against the ray. We can achieve the latter mainly by two methods: hierarchical bounding volumes and space subdivision.

The space subdivision method divides the world space into a set of rectangular subvolumes called *cells* or *voxels*. We refer to subvolumes as cells and reserve the label voxel for the atomic volume element, which is the

0272-17-16/92/0900-0019$03.00 © 1992 IEEE

smallest space enumeration entity. A cell consists of a list of all objects residing in that region of the world, while a voxel maintains information on one object only.

Space subdivision can be implemented in two variations: nonuniform and uniform. Nonuniform subdivision partitions space into portions of varying sizes to suit the features of the scene, using either octrees or BSP (binary space partition) trees. Uniform subdivision partitions space into uniform size cells organized in a 3D lattice.[1-3] ARTS (Accelerated Ray Tracing System[1]) employs a 3D digital differential analyzer (3DDDA) for incrementally computing all cells in a uniform partition pierced by a ray. Only the lists of objects residing in the pierced cells are candidates for ordered ray-object intersection calculation.

When the scene to be rendered consists of volumetric data sets such as those arising in biomedical imaging and scientific visualization, practitioners[4-6] have applied *ray casting* methods, in which only primary rays are traced. Levoy[4] used the term "ray tracing of volume data" to refer to ray casting with compositing of evenly spaced samples along the primary viewing rays. Since ray casting follows only primary rays, itdoes not directly support the simulation of light phenomena such as reflection, shadows, and refraction. On the other hand, studies of rendering volume densities by ray casting include, among others, a probabilistic simulation of light passing through and being reflected by clouds of small particles[5] and the light-scattering equations of densities within a volume grid, such as clouds, fog, flames, dust, and particle systems.[6]

To represent a 3D scene, discrete ray tracing uses a 3D raster of voxels in the same way a 2D raster of pixels represents a 2D image.

We call our approach here *discrete ray tracing* or *3D raster ray tracing* (RRT). Unlike existing ray tracing methods that used geometric representation for the 3D scene, RRT employs a 3D discrete raster of voxels for representing the 3D scene in the same way a 2D raster of pixels represents a 2D image. Each voxel is a small quantum unit of volume that has numeric values associated with it representing some measurable properties or attributes of the real object or phenomenon at that voxel. Like a pixel, the voxel is very small and thus assumed to be homogeneous, that is, it contains information regarding a single object only. The aggregate of voxels tessellates the 3D grid of voxels, termed the *3D raster* or the *volume buffer*.

RRT operates in two phases: preprocessing voxelization and discrete ray tracing. In the voxelization phase (discussed in the next section), the geometric model is digitized using 3D scan-conversion algorithms that convert the continuous representation of the model into a discrete representation within the 3D raster. For already digitized data sets, as in 3D medical imaging and 3D computational visualization, the discretization step is of course unnecessary. In the second phase, RRT employs a discrete variation of the conventional recursive ray tracer. Unlike conventional algorithms, which intersect analytical rays with the object list to find the closest intersection, RRT employs 3D discrete rays traversed through the 3D raster to find the first surface voxel. Encountering a nontransparent voxel indicates a ray-surface hit.

In conventional ray tracing, computation time grows with the number of objects.[1] This occurs because in crowded scenes a ray might pierce a substantial number of objects and because a cell might well contain more than one object. Moreover, conventional ray tracers are extremely sensitive to the type of objects comprising the scene; intersection calculation between a ray and a parametric surface is significantly more time consuming than intersecting the ray with a sphere or a polygon. In contrast, RRT eliminates the computationally expensive ray-object intersections calculation. Instead it relies solely on a fast discrete ray traversal mechanism and a single simple type of object—the voxel. Consequently, RRT is in practice independent of the number of objects in the scene or the object's complexity or type. However, RRT performance is sensitive to the 3D raster's resolution. For a given resolution, therefore, ray tracing time is nearly constant and can even decrease as the number of objects in the scene increases, since less stepping is needed before encountering an object.

Any change in the view-dependent parameters forces a traditional ray tracer to execute a full-fledged rendering. This consists of recomputing many view-independent attributes such as surface normal, texture color, and light source visibility and illumination. In contrast, RRT precomputes the view-independent attributes during the voxelization phase and stores them within each voxel. These attributes are readily accessible for multiple rendering of the fixed scene, in which viewing, lighting, and shading parameters change.

Traditional ray tracers can only render objects represented by geometric surfaces. RRT, on the other hand, can also render 3D sampled data sets (such as 3D magnetic resonance imaging) and computed data sets (such as fluid dynamics simulations), as well as hybrid models in which such data sets are intermixed with geometric models (for example, a scalpel superimposed on a CT image, radiation beams superimposed on a scanned tumor, or a plane fuselage superimposed on a computed air pressure data).[7] Unlike nonrecursive ray casting techniques, RRT—which recursively considers both primary and secondary rays—can model shadows and reflections for photorealistic imaging. RRT can also improve visualization of sampled and computed volumetric data sets.[8]

In ray tracing constructive solid geometry models, traditionally one would convert the CSG tree into a boundary

representation (B-rep) in an extremely time-consuming process. Alternatively, the direct CSG-rendering approach delays the computation of the Boolean operations until the rendering stage. However, the fact that processing time increases more rapidly than the complexity of the model remains a major hurdle confronting this approach. In contrast, because the spatial enumeration of the 3D raster lends itself to voxel-by-voxel Boolean operations, RRT can easily support ray tracing of CSG models that are synthesized and constructed efficiently during the voxelization phase.

Although the voxelization and the ray traversal phases introduce some aliasing, RRT uses the true normal stored with each voxel to spawn secondary rays. This smoothes out the aliasing of the surface. We can further reduce the aliasing artifacts by generating antialiased nonbinary objects during the voxelization phase, by supersampling during the ray tracing phase, and by consulting an object table for the exact geometric structure at the hit point. (See the section "Reducing aliasing" for more details.)

We generated the images shown in this article on 80-Mbyte and 128-Mbyte machines from 256^3 through 320^3 spatial resolution images. Our images are comparable in quality to images of equivalent resolution created using conventional ray tracing. The RRT approach assumes the geometric model or the sampled data set are discretized to the desired resolution of the pixel image. In other words, the voxel footprint approximately equals that of the pixel. Therefore, rendering a higher resolution image requires a higher resolution volume. However, as large memories prevail on graphics workstations, we need no longer question whether the voxel representation is a feasible alternative for realistic high-resolution rendering with ray tracing.

Voxelization phase

Raster ray tracing of geometric scenes commences with a preprocessing voxelization phase that precedes the actual ray tracing. In voxelization, 3D scan-converting (voxelizing) each of the geometric objects comprising the scene converts the scene into a discrete voxel representation. A voxelization algorithm for a given geometric object generates the set of voxels that "best" approximates the continuous specification of that object. The algorithm stores the discrete representation as a 3D array of unit voxels.

RRT's voxelization algorithms generate color, opacity, texture, and normal vector for each surface voxel, by sampling those fields at the integral grid point located at the voxel center (although other sampling paradigms are also possible). In addition, two bits per light source are provided for each voxel to denote whether that light source is visible, occluded, or visible through a translucent material.[9] Actually, one can also precompute and add to the voxel color the view-independent parts of the illumination equation, that is, the ambient illumination and the sum of the attenuated diffuse illumination of all the visible light sources. All the view-independent attributes precomputed during voxelization and stored with the voxel are then readily accessible for speeding up the ray tracing phase (see the next section).

Solids are commonly digitized by spatial enumeration algorithms that employ point or cell classification methods in an exhaustive fashion, or preferably by recursive subdivision. However, subdivision techniques for model decomposition into rectangular subspaces are computationally expensive and thus inappropriate for medium- or high-resolution grids. The voxelization algorithms we employ, on the other hand, follow the same paradigm as the 2D scan-conversion algorithms commonly used in 2D raster graphics. They are incremental, use simple arithmetic (preferably integer based), and have a complexity not more than linear with the number of voxels generated. (See Table 2 for voxelization times of scenes with an increased population of spheres.)

The literature on 3D scan conversion is relatively small. Mokrzycki[10] employed a 3D curve algorithm where the curve is defined by the intersection of two implicit surfaces. We introduced elsewhere[11] voxelization algorithms for 3D lines, 3D circles, and a variety of surfaces and solids, including polygons, polyhedra, and quadric objects. We also described efficient algorithms for voxelizing polygons using an integer-based decision mechanism embedded within a scan-line filling algorithm[12]; parametric curves, surfaces, and volumes using an integer-based forward differencing technique[13]; and quadric objects such as cylinders, spheres, cones, and tori using "weaving" algorithms, by which a discrete circle or line sweeps along another discrete circle or line.[14] All these algorithms preserve the topology of the voxelized objects and allow some control over their connectivity and thickness.

Assume a voxel to be a tiny cube centered at the 3D grid point. Two voxels are *26-adjacent* if they share a vertex, an edge, or a face. Every voxel has 26 such adjacent voxels. Eight share a vertex (corner) with the center voxel, twelve share an edge, and six share a face. Every two consecutive voxels along a *26-connected* curve or line are 26-adjacent. Similarly, face-sharing voxels are defined as *6-adjacent*, and every two consecutive voxels in a *6-connected* curve or line are 6-adjacent. Our voxelization algorithms can generate curves and lines that are either 6-connected or 26-connected.

A voxelized surface approximating a connected continuous surface has to be connected (for example, 26-connected). However, connectivity does not fully characterize the surface because the voxelized surface may contain local discrete holes, termed *tunnels*, not present in the continuous surface. A tunnel is a passage by a discrete line through a voxel-based surface; the line crosses from one side of the surface to the other. A requirement of our surface voxelization algorithms is that the volumetric representation of a surface must be "thick enough" not to allow the discrete rays (3D lines) to penetrate it. A voxelized surface through which 6-connected rays do not penetrate is a *6-tunnel-free* surface, while a thicker surface through which even 26-connected rays cannot pass is a *26-tunnel-free* surface.[14] Our voxelization algorithms

Figure 1. A 256^3 resolution reconstructed MRI head reflected in a circular mirror. A portion of the head was removed by a CSG subtraction. RRT time is 28.1 seconds.

can generate any of these types of surfaces. The decision of which one to use depends primarily on the connectivity of the discrete rays employed during the discrete ray tracing phase.

Rasters, in general, lend themselves to block operations such as bitblt or their 3D counterparts, voxblt operations.[15] In particular, the 3D raster lends itself to Boolean operations that can be performed during voxelization. This makes ray tracing of CSG models trivial. Voxel-by-voxel block operations during voxelization add a variety of modeling capabilities that aid in the task of image synthesis by supporting the manipulation of true solids (that is, interior information as well as exterior surfaces). Figure 1 shows an RRT of a 256^3 reconstructed MRI head reflected in a mirror. We generated the mirror and the head during voxelization. We generated the cut in the head by the CSG operation of subtracting a box from the MRI data set.

Discrete ray tracing phase

Once the discrete representation of the scene is available, we employ a discrete variation of the conventional ray tracing algorithm. Rays are recursively traced through the scene by stepping along the 3D discrete line representation of the ray through the 3D raster of voxels. The discrete ray traversal algorithm is thus central to the discrete ray tracing stage. In the next section we present an efficient algorithm for 3D discrete ray traversal.

For each screen pixel, our ray tracer computes the intersections of the ray emanating from that pixel with the boundary of the 3D raster, defining the two endpoints of the discrete ray. The discrete ray traversal commences at the closer endpoint—the discrete ray origin. The first nontransparent voxel encountered by the discrete ray traversal indicates a ray-surface hit. We use the term *hit* for the point where the discrete ray and the discrete surface meet and the term *intersection* for the traditional continuous ray-object crossing. When a hit occurs, the voxel encountered by the ray is known and its attributes are readily available. RRT thus eliminates the need to compute and search for an intersection among several objects that might be present in the cell. Furthermore, unlike traditional ray tracing techniques, RRT does not compute the normal at the intersection point. Instead, the true normal vector to the surface (stored in the hit voxel) is retrieved and used for spawning reflective and transmitted rays.

Each surface voxel also stores the color, illumination, and texture color for that voxel. Since texture mapping is view independent, it is precomputed during the voxelization algorithm from a 2D texture/photo map or a 3D texture. We do not need, for example, to recompute for every different viewpoint the inverse mapping needed for texture mapping.

Since the visibility of the light sources from each voxel is also view independent, shadow rays are fired from each surface voxel to all light sources as part of the voxelization preprocessing stage. The light visibility information is stored in two bits per light source per voxel, denoting whether that light source is visible, occluded, or visible through a translucent material. A shadow ray will be fired at rendering time only for the third case, to determine the actual light intensity. This provides substantial savings, since the vast majority of shadow (or illumination) rays will not be fired at all during the rendering phase. Furthermore, as mentioned in the previous section, the view-independent ambient and attenuated diffuse illumination of all the visible light sources can be precomputed during the voxelization stage and stored with each voxel value. The ray tracing stage has to compute only the attenuated specular illumination for each light source. (In fact,

Figure 2. Newell's teapot over a textured floor, mirrored in two mirror walls. RRT time for this 256^3 resolution image is 36.8 seconds.

the attenuation percentage can also be precomputed and saved with each voxel.) The sum of all the illumination values plus the recursive reflected and transmitted components are added during the discrete ray traversal phase to the illumination value stored at the voxel, producing the total illumination of that voxel.

The only geometric computations performed by the discrete ray tracing phase are for generating discrete rays, spawning secondary rays, and compositing colors. Since all objects have been converted into one object type—the unit voxel—we can view the traversal of discrete rays as performing an efficient ray-voxel intersection calculation. Having only one object type frees the ray tracer from any dependency on the type of objects composing the image. Figure 2 shows an image consisting of a Newell's teapot ray traced in 36.8 seconds, which is approximately the same time as ray tracing one sphere in the same environment.

The RRT rendering time is sensitive to the total distance traversed by the rays, which depends primarily on the constant 3D raster resolution and the portion of the 3D raster occupied by the objects. In practice, the RRT rendering time is insensitive to the complexity of the scene. It thus does not surprise us to achieve improved performance when tracing scenes of higher complexity. See the "Results" section for examples.

The RRT approach provides true recursive ray tracing (rather than ray casting) of sampled or computed data sets, as well as hybrid scenes where geometric and sampled or computed data intermix. Figure 1 shows ray tracing of a medical data set in which additional information becomes available to the observer when using reflection. The normal vector to the sampled or computed surfaces can be estimated, either during a preprocessing stage when it is stored with the voxel or during the ray tracing stage. The normal estimation can employ one of many volume shading approaches: contextual shading, gray-level gradient, context-sensitive normal estimation, or biquadratic local surface interpolation. We chose to employ a variation of the gray-level gradient approach[16] that examines all values in a $3 \times 3 \times 3$ neighborhood in order to enhance normal integrity.

Efficient discrete ray traversal

In RRT, we eliminated the computationally expensive intersection calculation. Instead, the load falls almost exclusively on the discrete ray traversal algorithm. We have observed in our experiments with RRT that the discrete ray traversal algorithm consumes up to 90 percent of the ray tracer execution time. An efficient discrete ray traversal algorithm is thus imperative for an efficient RRT. Such an algorithm uses a 3D discrete line voxelization mechanism to generate the sequence of voxels along the ray.

Researchers have previously devised such line algorithms for uniform space-subdivision ray tracing. Fujimoto et al.[1] presented a 3DDDA algorithm that generates a 6-connected line, which includes all of the cells pierced by the continuous line. A more efficient algorithm was developed[2] and later elegantly presented,[3] with suggestions for further optimization. Basically, all these algorithms generate 6-connected lines with floating-point calculations of a parametric line, using fixed point arithmetic. We presented an integer-based 26-connected line algorithm[11] later extended to handle other types of connectivities.[14] Specifically, we devised a 6-connected line algorithm that employs only integer arithmetic and requires less operations per voxel.

As mentioned above, every two consecutive voxels in a 6-ray (6-connected line) are face-adjacent, which guarantees that the set of voxels along the discrete line includes all voxels pierced by the continuous line. A 6-ray from (x, y, z) to $(x+\Delta x, y+\Delta y, z+\Delta z)$ (assuming integer line endpoints) has the total length of n_6 voxels to traverse, where

$$n_6 = |\Delta x| + |\Delta y| + |\Delta z| \quad \textbf{(1)}$$

In a 26-ray (26-connected line), each two consecutive voxels can be either face-adjacent, edge-adjacent, or corner-adjacent. The line has a length of

$$n_{26} = \max(|\Delta x|, |\Delta y|, |\Delta z|) \quad \textbf{(2)}$$

Clearly, a 26-ray is shorter than a 6-ray:

$$1 \le n_6 / n_{26} \le 3 \quad \textbf{(3)}$$

When the line is parallel to one of the primary axes, $n_6 = n_{26}$, while approaching the main diagonal increases the ratio n_6/n_{26} up to 3. We observed in our experiments a traversal speedup of 26-rays compared to 6-rays of up to 1.84, which is the expected speedup deducible from the metrics of the two connectivities (Equations 1 and 2).

Since the performance of RRT depends almost solely on the total number of voxels traversed by the discrete rays, our algorithm traverses 26-rays, which have significantly fewer voxels than 6-rays. We based our ray traversal on a fast version of the 26-connected line algorithm that employs a tree of condition statements that performs one increment, per iteration, of a pointer into the 3D array (raster). The algorithm is actually an enhanced double Bresenham's 2D line algorithm. It takes unit steps along the direction with the largest extent and uses a decision mechanism for the other two axes.

While 26-rays have the clear speed advantage, their use implies that the surfaces voxelized during the preprocessing stage are 26-tunnel-free, that is, they must be thick enough to eliminate possible penetration by a 26-ray. In addition, a 26-ray does not traverse all voxels pierced by the continuous ray and thus may skip a voxel in which the ray hits the object. Consequently, ray-surface hits might be missed at the object

silhouette (see point C in Figure 3a) and occasionally hit a voxel underneath the true surface.

Although we found from experience that the percentage of 26-rays that missed was usually less than 1.5 percent, an image-dependent artifact could force us to abandon the 26-ray speed advantage for the 6-ray accuracy. In this case, we can also save some preprocessing time by voxelizing only 6-tunnel-free surfaces.

Our solution to the speed versus accuracy dilemma is to use 26-rays to rapidly traverse the empty space and employ the 6-rays only when approaching the scene objects. However, this approach requires both a line algorithm that can efficiently switch between connectivities and a mechanism to sense object proximity. We observe that since both the 26- and 6-ray algorithms use the same variables and only the decision mechanism differs, it is possible to adaptively alternate between 26- and 6-rays with no extra cost. To implement the feature of proximity sensing, we must add (during the voxelization stage) a proximity flag for all the voxels around the object surface to indicate that we are in the vicinity of an object.

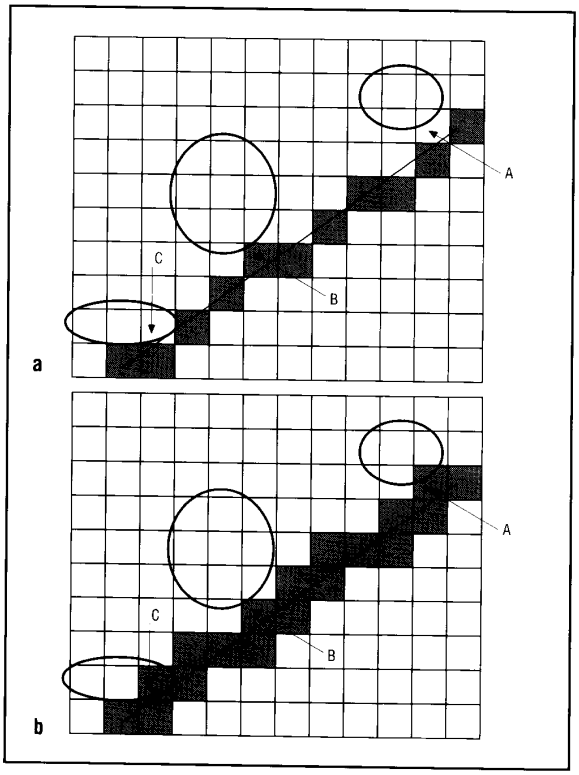
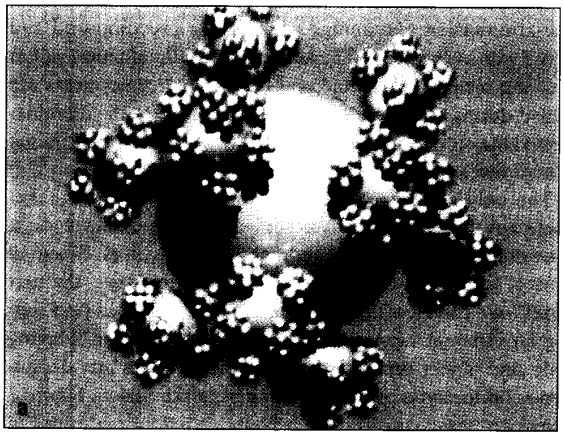


Figure 3. (a) A 26-ray passing through a volume consisting of three elliptical objects. At point C the ray has a miss hit and at point B it has a false hit. (b) The passage of a 6-ray in the same scene. The 6-ray has a false hit at A and B, but did not miss the hit at C.



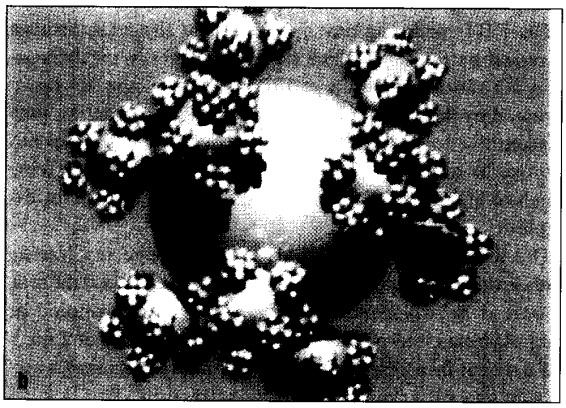


Figure 4. (a) A 820-sphereflake of 320^3 resolution generated with 6-rays RRT in 74.9 seconds. (b) The same scene ray traced with true intersection verification in 85.0 seconds.

The traversal starts as a 26-ray that rapidly skips over the transparent regions. Whenever a proximity flag is encountered, the 26-line algorithm adaptively "slows down" and takes 6-connected steps to avoid misses at the object surface. The adaptive line algorithm performs the switching between connectivities without noticeable time penalty. The extra proximity flags in the adaptive algorithm add only one test per iteration to the overall execution time of the algorithm. Employing 26-rays has an overall speedup compared to 6-rays in all views and not only from the diagonal directions (where $n_6 = 3n_{26}$), since primary rays are not parallel to each other and secondary rays are spawned in arbitrary directions.

Reducing aliasing

Spatial aliasing in traditional ray tracing results from point sampling in screen space. A common cure for it is supersampling in screen space. RRT also supports screen supersam-

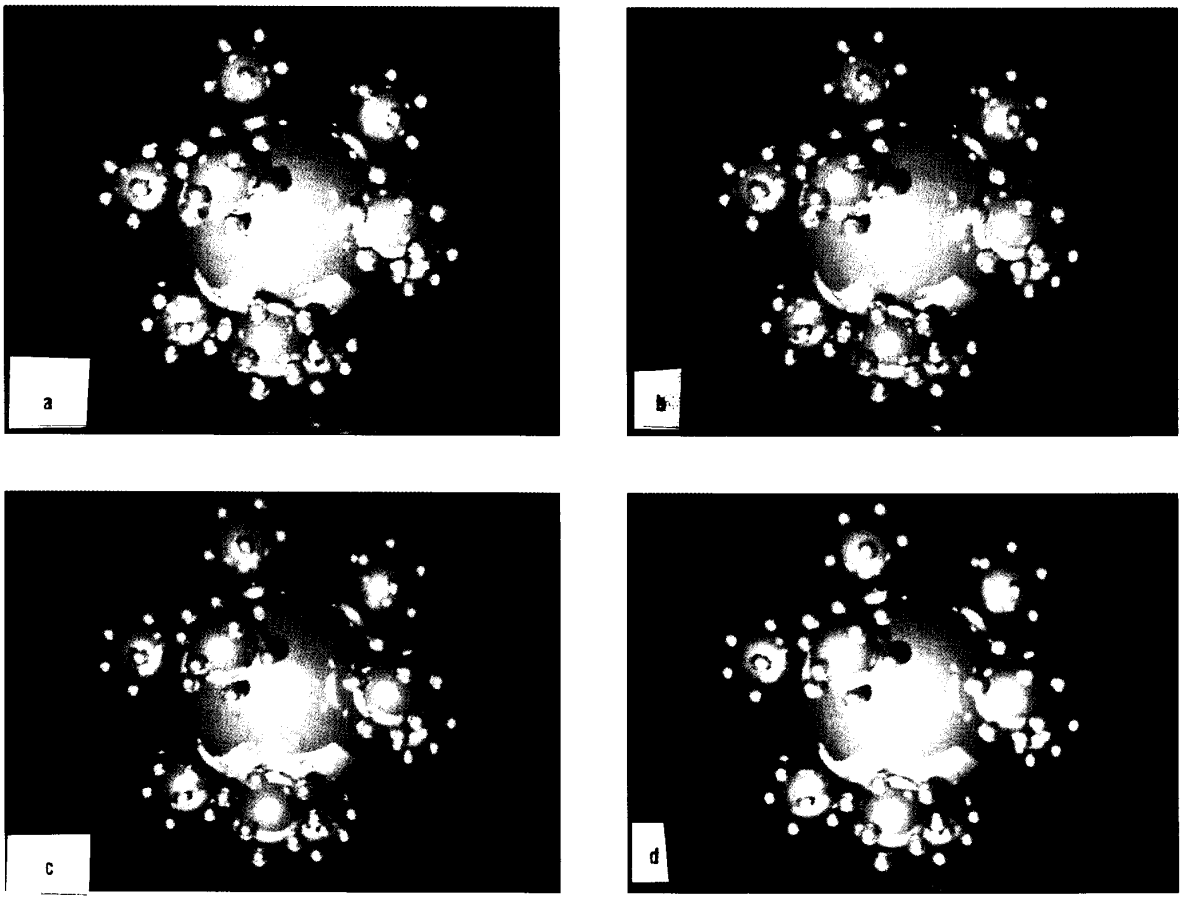


Figure 5. A comparative ray tracing of 91-sphereflake in 320^3 resolution. (a) Ray traced in 72 seconds with the basic RRT algorithm. (b) RRT with supersampling yielded this view in 342 seconds. (c) Produced in 75 seconds by RRT with intersection verification. (d) Produced in 352 seconds by RRT with intersection verification and supersampling.

pling by adopting a 3D line algorithm that can handle subpixel addressing of the ray origin.

RRT also suffers from another type of aliasing caused by point sampling in object space, as the values of the voxel attributes are determined by sampling the object at integral coordinates (voxel centers). Another manifestation of object space aliasing is the false hit problem encountered in both 6- and 26-rays (see Figure 3, point *B*). It stems from the fact that a discrete hit point between the discrete ray and the discrete surface does not necessarily indicate an intersection between their continuous counterparts. That is, the continuous ray and the continuous surface can pass through the same voxel and still not meet each other. From our experience, however, we found that only 8 percent or fewer of the rays experience a false hit.

The remedy to object-space aliasing is based on storing in each voxel the information regarding the geometric definition of the object occupying that voxel. Specifically, during voxelization each voxel belonging to an object is assigned an object-id instead of the normal vector information, as in the basic version of RRT. The object-id serves as an index into an object table storing of the object's geometric definition. Whenever a discrete ray hits a voxel, the hit is verified by computing the true intersection between the continuous ray and the geometric object definition. If no intersection occurs, the ray resumes its discrete traversal of the 3D raster. Figure 4a shows the 320^3 820-sphereflake ray traced with 6-rays and no intersection verification, while Figure 4b shows the same image ray traced with true intersection verification. The two images are hardly distinguishable in this high volume resolution, and their ray tracing time is about the same (74.9 seconds and 85.0 seconds, respectively).

The slight computation overhead involved in intersection verification can also provide us with the true intersection point and the normal vector, solving the problem of object space aliasing in terms of surface attribute sampling. Figure 5

compares the four different variations of RRT: the basic RRT (described above), RRT with screen supersampling, RRT with intersection verification, and RRT with both screen supersampling and intersection verification. Figure 6 shows an image of Whitted's classical spheres and checkerboard. We used the true intersection point and the true normal vector for image generation with supersampling.

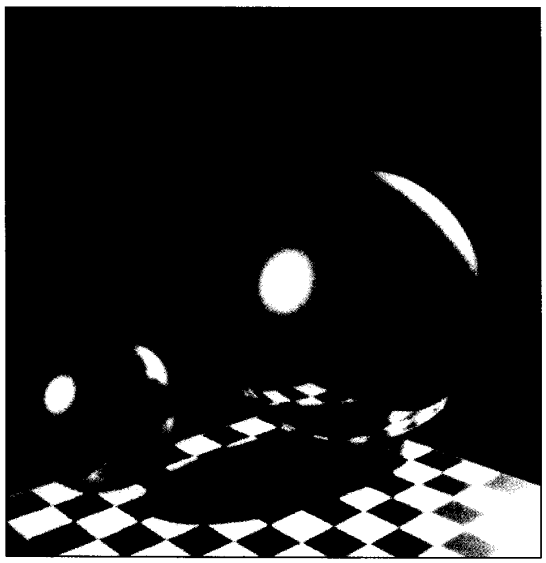


Figure 6. Turner Whitted's spheres and a checkerboard floor in 320^3 raster resolution ray traced in 377 seconds with intersection verification and supersampling.

Although we now maintain the objects' geometric information, we remain loyal to the 3D raster approach, in which a voxel is assumed to be atomic in the sense that it contains information only about one object. This assumption is similar to that of 2D rasters, in which a pixel maintains only one color entity. In contrast, space-subdivision methods (such as ARTS) assume that each cell maintains a list of geometrically defined objects. This distinction poses some limitations on scene density in RRT in the sense that image quality degrades as the scene becomes too crowded relative to the voxel size, and many objects tend to occupy the same voxel.

In summary, comparing RRT to existing ray tracing methods, RRT suffers from large memory requirements or some limitation on scene density. On the other hand, the primary advantages of RRT, even when it employs intersection verification, include its practical independence of scene complexity (that is, the number of objects), the ability to ray trace sampled and computed data sets, the support of voxblt and CSG operations, and the ability to precompute all the view-independent attributes. When RRT does not perform intersection verification, its performance is also insensitive to surface complexity (that is, the type of surface).

Results

We obtained the results described here by running our RRT software on one 20-MIPS (25 MHz) processor of a Silicon Graphics 4D/240GTX. We report the results in CPU time. We obtained similar results on Hewlett-Packard 400s and Sun Microsystems Sparcstation-1 workstations. We generated images on 80-Mbyte and 128-Mbyte machines from 256^3 and 320^3 spatial resolution 3D rasters. We did all ray tracings with two generations of secondary rays and shadow rays to two light sources. We did not precompute shadow rays; that could have further improved the reported results by about 20 percent for two light sources.

Table 1 shows rendering times of various 256^3 resolution test scenes using the basic RRT algorithm. The table indicates that RRT is in practice insensitive to the number or type of objects. Our 256^3 resolution test images were ray traced in less than 40 seconds, with almost no degradation in performance even when image complexity increased or when surface type changed. For example, despite an order of magnitude increase in the number of objects ray traced for the 91-, 820-, and 7,381-sphereflake images, the performance degraded very slowly (26.5, 29.3, and 38.4 seconds, respectively). This slight slowdown in speed was *not* caused by the increase in the number of objects but by the inflated number of rays bouncing between the many tiny structures of this fractal-like scene instead of terminating by reaching the 3D raster boundary.

Table 1. Rendering time of the basic RRT of various 256^3 resolution scenes.

Scene	Seconds
91-sphereflake	26.5
820-sphereflake	29.3
7,381-sphereflake	38.4
MRI head	28.1
Newell's teapot	36.8

As discussed above, the lengths of the 26-rays and 6-rays differ significantly. We ray traced a test image with 26-rays in 32.3 seconds, 23.5 of which (73 percent) were spent in the 26-ray traversal algorithm. The same image took 56.2 seconds in 6-connected tracing, which in turn spent 47.3 seconds (85 percent) in the 6-ray traversal algorithm. We observed similar results from various viewpoints, with a speedup factor of up to 1.84 for the traversal of 26-rays. We also discovered that less than 1.5 percent of the 26-rays missed a surface voxel pierced by a 6-ray. A performance comparison between the 26-ray algorithm and the adaptive algorithm based on proximity bit showed a time penalty of less than 2.9 percent, while completely eliminating the "miss" problem. Voxelization with proximity flags does not require additional computations, only more memory-write operations.

The performance of the RRT algorithm when employing intersection verification does depend on the type of surfaces comprising the scene. In the case where surfaces are easy to intersect (such as spheres), we observed no significant performance penalty; ray tracing a 320^3 volume consisting of a 91-sphereflake with and without intersection verification took 75 and 72 seconds, respectively. In terms of image quality, the number of rays that benefit from intersection verification by eliminating false hits depends on the scene and on the complexity of its silhouettes. For example, in the 91-, 820-, and 7,381-sphereflake images, 1.72 percent, 5.06 percent, and 7.94 percent of the rays, respectively, experienced a false hit that we remedied by intersection verification.

For purposes of comparison, we implemented and tested the performance of the uniform space subdivision algorithm ARTS[1] for tracing several scenes. We generated these scenes by randomly distributing an increasing number of spheres inside the 3D raster. The top row of Table 2 shows that indeed voxelization time grows linearly to the scene complexity. The results, shown in the bottom row of Table 2 and by the solid line on the graph in Figure 7, indicate that once the scene gets relatively crowded, RRT's ray tracing time might even decrease because the total space traversed by the rays gets shorter. ARTS, on the other hand, still performs intersection calculation in each cell and therefore remains sensitive to the scene and object complexity. When ARTS subdivision resolution equals that of RRT 3D raster, ARTS performs the costly intersection calculation just for the sake of distinguishing between multiple objects in a single voxel—an infrequent phenomenon in high-resolution rasters and common scenes. On average, when comparing both methods under the same parameters (for example, when both operate in the same space resolution and render the same number of spheres), the RRT method is 9.1 times faster than ARTS. In reality, ARTS does not require that its cell footprint be larger than a pixel and therefore can more efficiently operate in lower space resolutions. The upper two dashed lines in Figure 7 show

Table 2. Voxelization and discrete ray tracing times (in seconds) of a 256^3 resolution scene with increased population of spheres.

Number of spheres	10	100	200	400	1,000	5,000
Voxelization	0.07	0.74	1.49	2.98	7.46	36.01
Ray tracing	20.2	23.9	30.1	30.4	28.9	21.5

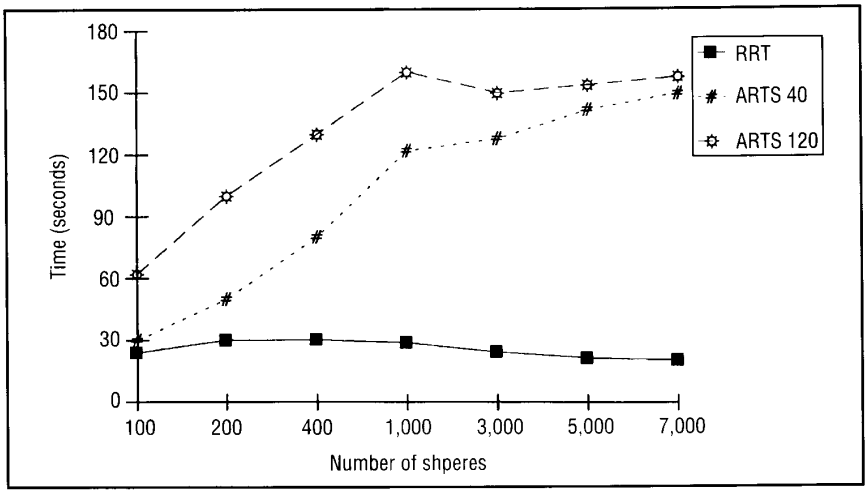


Figure 7. A comparison between RRT and ARTS. The dashed lines show ARTS ray tracing time employing 40^3 (ARTS40) and 120^3 (ARTS120) uniform space subdivision. The scene was composed of randomly distributed spheres of radius 10 in a 256^3 world.

the degradation in ARTS' performance for 40^3 and 120^3 space subdivisions. We should note that this example favors ARTS because ray-sphere intersection calculation is simple. If we used teapots or fractals instead, for example, the difference would have been much larger.

Conclusions

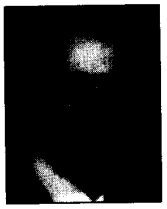
Our new ray tracing approach, RRT, in its basic version ray traces voxelized scenes in record speeds, at the price of negligible degradation in image quality. One can progressively refine the image by employing the more sophisticated variations of the algorithm, which employ supersampling and intersection verification. Even the processing time of the algorithm's most elaborate variation proves attractive compared to existing techniques, with comparable image quality. RRT especially suits complex scenes and constructive solid models, exploiting voxel representation for ray tracing geometry. It also offers ray tracing for photorealism of sampled and computed voxel data sets, or mixtures thereof with synthetic models. ❑

Acknowledgments

This project was partially supported by the National Science Foundation under grants MIP-8805130 and IRI-9008109, and by a grant from Hewlett-Packard. We would like to thank Dan Gordon, Pat Hanrahan, Marc Levoy, and Nelson Max for reviewing the manuscript and for their helpful suggestions, and Qiang Zhang for her help in implementing parts of RRT.

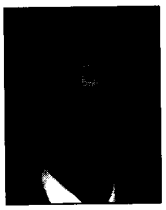
References

1. A. Fujimoto, T. Tanata, and K. Iwata, "ARTS: Accelerated Ray-Tracing System," *IEEE Computer Graphics and Applications*, Vol. 6, No. 4, April 1986, pp. 16-26.
2. J. Amanatides and A. Woo, "A Fast Voxel Traversal Algorithm for Ray Tracing," *Proc. Eurographics 87*, North Holland, Amsterdam, Aug. 1987, pp. 3-9.
3. J.G. Cleary and G. Wyvill, "Analysis of an Algorithm for Fast Ray Tracing using Uniform Space Subdivision," *Visual Computer*, Vol. 4, 1988, pp. 65-83.
4. M. Levoy, "Display of Surfaces from Volume Data," *IEEE Computer Graphics and Applications*, Vol. 8, No. 5, May 1988, pp. 29-37.
5. J.F. Blinn, "Light Reflection Functions for Simulation for Clouds and Dusty Surfaces," *Computer Graphics* (Proc. Siggraph), Vol. 16, No. 3, July 1982, pp. 21-29.
6. J.T. Kajiya and B.P. Von Herzen, "Ray Tracing Volume Densities," *Computer Graphics* (Proc. Siggraph), Vol. 18, No. 3, July 1984, pp. 165-174.
7. A. Kaufman, R. Yagel, and D. Cohen, "Intermixing Surface and Volume Rendering," in *3D Imaging in Medicine: Algorithms, Systems, Applications*, K.H. Höhne, H. Fuchs, and S.M. Pizer, eds., Springer-Verlag, Berlin, 1990, pp. 217-227.
8. R. Yagel, A. Kaufman, and Q. Zhang, "Realistic Volume Imaging," *Proc. Visualization 91*, IEEE Computer Society Press, Los Alamitos, Calif., Oct. 1991, pp. 226-231.
9. A. Woo and J. Amanatides, "Voxel Occlusion Testing: A Shadow Determination Accelerator for Ray Tracing," *Proc. Graphics Interface 90*, Canadian Information Processing Society, Toronto, 1990, pp. 223-230.
10. W. Mokrzycki, "Algorithms of Discretization of Algebraic Spatial Curves on Homogeneous Cubical Grids," *Computers and Graphics*, Vol. 12, No. 3/4, 1988, pp. 477-487.
11. A. Kaufman and E. Shimony, "3D Scan-Conversion Algorithms for Voxel-Based Graphics," *Proc. 1986 Workshop on Interactive 3D Graphics*, University of North Carolina, Chapel Hill, Oct. 1986, pp. 45-75.
12. A. Kaufman, "An Algorithm for 3D Scan-Conversion of Polygons," *Proc. Eurographics 87*, North Holland, Amsterdam, Aug. 1987, pp. 197-208.
13. A. Kaufman, "Efficient Algorithms for 3D Scan-Conversion of Parametric Curves, Surfaces, and Volumes," *Computer Graphics* (Proc. Siggraph), Vol. 21, No. 4, July 1987, pp. 171-179.
14. D. Cohen and A. Kaufman, "Scan-Conversion Algorithms for Linear and Quadratic Objects," in *Volume Visualization*, A. Kaufman, ed., IEEE Computer Society Press, Los Alamitos, Calif., 1990, pp. 280-301.
15. A. Kaufman, "The Voxblt Engine: A Voxel Frame Buffer Processor," in *Advances in Graphics Hardware III*, A.A.M. Kuijk and W. Strasser, eds., Springer-Verlag, Berlin, 1992.
16. U. Tiede et al., "Investigation of Medical 3D-Rendering Algorithms," *IEEE Computer Graphics and Applications*, Vol. 10, No. 3, March 1990, pp. 41-53.



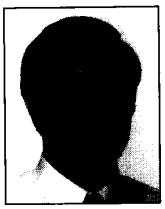
Roni Yagel is an assistant professor in the Department of Computer and Information Science at The Ohio State University. Previously he was a researcher in the Department of Anatomy and the Department of Physiology and Biophysics in the State University of New York at Stony Brook. His research interests include algorithms for voxel-based graphics, imaging, and animation, 3D user interfaces, hardware for volume viewing, and visualization tools for biomedical applications.

Yagel received his PhD in 1991 from the State University of New York at Stony Brook. He received his BS Cum Laude and MS Cum Laude from the Department of Mathematics and Computer Science at Ben-Gurion University of the Negev, Israel, in 1986 and 1987, respectively.



Daniel Cohen is an assistant professor in the Department of Computer Science at Ben-Gurion University and a lecturer in the School of Mathematics at Tel-Aviv University. He is also developing a real-time ray tracer of terrain systems at Milikon. His research interests include rendering techniques, volume visualization, architectures, and algorithms for voxel-based graphics.

Cohen received a BS Cum Laude in both mathematics and computer science in 1985, an MS Cum Laude in computer science in 1986 from Ben-Gurion University, and a PhD from the Department of Computer Science in 1991 at State University of New York Stony Brook.



Arie E. Kaufman is a professor of computer science and the director of the Cube project for volume visualization at the State University of New York at Stony Brook. He has conducted research and consulted in computer graphics for 18 years, specializing in volume visualization; graphics architectures, algorithms, and languages; user interfaces; and scientific visualization.

Kaufman received a BS in mathematics and physics from the Hebrew University of Jerusalem in 1969, an MS in computer science from the Weizmann Institute of Science, Rehovot, in 1973, and a PhD in computer science from Ben-Gurion University, Israel, in 1977. He is a member of ACM, Siggraph, IEEE Computer Society, IEEE EMBS, and Eurographics. Currently, he is the chair of the IEEE Computer Society Technical Committee on Computer Graphics.

Readers may contact Yagel at the Department of Computer and Information Science, The Ohio State University, 228 Bolz Hall, 2036 Neil Ave., Columbus, Ohio 43210-1277, email yagel@cis.ohio-state.edu.