# Comparison of some Ray Generators for Ray Tracing Volumetric Data

Miloš Šrámek *
Institute of Measurement Science
Slovak Academy of Sciences,
Dúbravská cesta 9, 842 19 Bratislava, Slovak Republic

**Abstract**

Ray tracing is a popular technique for rendering not only analytical objects, but also volumetric data, either synthetized or measured. Different techniques for the ray definition by traversal of the voxel space were designed. The first part of the paper is an overview of ray tracing or ray casting algorithms for the volume data. Stress is laid upon exploiting various kinds of speed up techniques. In the secon part, we compare some of these approaches in a computer experiment. Further we show that an appropriate choice of background traversal algorithm can significantly shorten computation time of the exact ray-surface intersection point.

# 1 Introduction

Together with the fast development of various 3D imaging modalities (CT, MRI, PET, SPECT, confocal microscopy etc.) and increasing performance of contemporary computing machinery, demands on high quality data visualization, more complex illumination models and, of course high speed of processing come into foreground. Due to its 3D nature, each software designer strucks with the problem of tens or even hundreds of megabytes, which should be processed in order to render a single image within an acceptable, if not even real time. And the problems encrease, when global illumination models, as e.g. recursive ray tracing, should be incorporated.

One way, which could solve much of these problems is the hardware progress. However, the experience shows that the demands are always a step ahead of the hardware capabilities, or, if the appropriate hardware exists, it is often not affordable due to limited financial resources.

In this paper we are going to deal with the software solution of the problem for ray tracing of volumetric data. Ray tracing is a well established computer graphics methodology for high quality rendering of analytical objects with shiny surfaces, interreflections and transparency. However, its high computational demands are further increased by various kinds of possible objects types which can be incorporated into the system. Recently techniques were proposed, known as *volume graphics*, which offer an alternative way for the standard computer graphics[KCY93]. Its main idea is to represent all kinds of objects by one metaobject, voxel, by filtered scan conversion of the analytical object description[WK93]. Since now only one object type is to be processed, the resulting ray tracer code can be much simplier and thus more effective. To add a new type of object into the rendering system, say some new kind of parametric surface, it is necessary only to write a special scan conversion routine, which is usually much easier than to incorporate the new object type into the ray tracer.

*The research reported in the paper was supported partially by the Slovak Grant Agency for Science(grant No.2/999003/92) and by the grant P8189-MED of the FWF (Austria)

This voxel representation is similar to the volumetric representation of real world structures obtained by various 3D scanners and as such can be also used for simultaneous rendering of scenes composed of measured and synthetic objects. Although various attributes of the object as well as of the scene and illumination conditions can be precomputed and stored (surface normals, textures, light visibility) [YCK92] in order to speed up the computations, the voxel is usually characterized only by *occupancy function* $g$, characterizing a fraction of the voxel size occupied by the object. It can be modeled as a convolution $g = p_s \otimes f$, where $f$ represents the real or synthetic object and $p_s$ is the point spread function of the scanner or scan conversion filter.

# 2 Volume visualization by ray tracing

The term *surface rendering* denotes a set of 3D data visualization techniques where only object surfaces contribute to the rendered image. One possibility is to build up a *surface model* i.e. to define a set of patches approximating the surface. These patches can be then rendered by some standard technique, usually with a hardware support.

The *binary surface rendering* techniques represent an alternative. Although still only surfaces contribute to the image, no explicit surface model is defined. Instead, a trivariate implicit function $f^i = f^i(\mathcal{P}, \sigma)$ is given, depending on voxel position $\mathcal{P}$ and some neighborhood $\sigma$, which locally interpolates the discrete occupancy function. A continuous surface description can be then obtained by thresholding this function at some threshold T.

In order to have a possibility to specify various surface properties (color, reflectivity etc.) for different objects, an *object identifier* can be assigned to a voxel, either directly during the scan conversion of analytical description or as a result of *segmentation* in the case of scanned data. Voxels with no identifier assigned belong to *background* and can be ignored during the processing, since they cannot contribute to the rendered image.

The task of ray tracing volumetric data can be then defined as follows:

> Given a segmented scalar scene, represented by 3D occupancy and object identifier arrays, and an interpolating function $f^i$ with threshold T, render objects in the scene by recursive ray tracing and, if necessary, with supersampling

Due to the fact that the scene is defined within a 3D discrete raster the ray should be represented as a *discrete ray*, i.e. as an ordered sequence of voxels pierced by the given ray with the following properties:

1. To enable supersampling and recursivity, the ray should be able to start at any point outside of the scene, or inside, and with arbitrary direction.

2. To get correct images, no object voxels along the ray should be missed. Therefore, the ray should, at least in the vicinity of an object, fulfill the demands of 6-connectivity[KS86].

Traversal of the ray voxels has more phases. Background voxels, surrounding the objects, are usually found first. Their traversal stops either when the ray leaves the scene or when the first object voxel is found. In the second case a *hit-miss test* should be performed in order to know, if the ray should continue further (Fig. 1, Ray B), by the following object or background voxel, or if a *ray-surface intersection* should be searched for (Fig. 1, Ray A). The hit-miss test can be performed by evaluation of the interpolating function $f^i$ at one or some points lying in the voxel and comparing the results with the threshold value. In order to detect the ray-surface intersection point exactly, a system defined by the ray and surface ($f^i(\mathcal{P}, \sigma) = T$) equations[Š94b] should be solved either analytically or numerically.

The *volume rendering* techniques represent an alternative to the surface approaches. Rather than segmenting the scene into objects and background, an *opacity* and *color* are assigned to each voxel, based on local properties of the occupancy function. The opacity reflects a measure by which

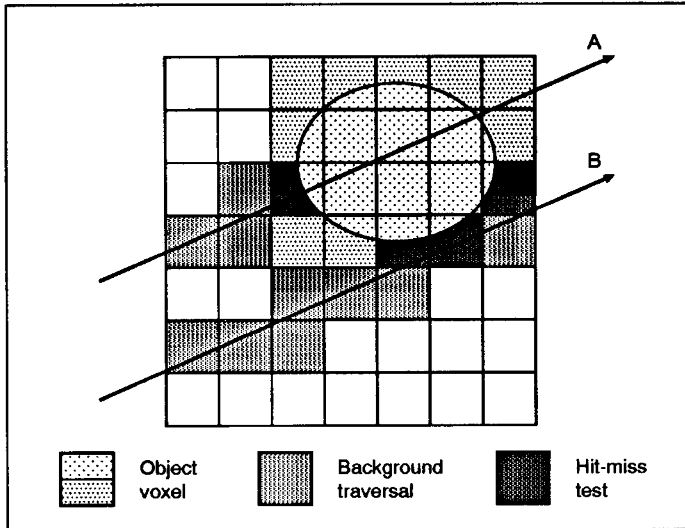


Fig. 1: Background traversal, hit-miss test and surface detection

the given voxel can contribute to the rendered image. A *culling* function can be defined, identifying the voxels, which cannot contribute to rendition and which can be discarded from consideration alike the background voxels in the surface rendering. Among others, techniques tracing primary rays (*ray casting*) through the scene were proposed, accumulating color and opacity of voxels or data samples obtained by interpolation along the ray.

We can see that visualization of volumetric data by ray tracing is a task, which is algorithmically similar to standard ray tracing of analytical objects if some space subdivision speed up method is used. In this case the object space is subdivided, either hierarchically or uniformly, again into voxels[FTI86], which can be empty or can contain a list of contributing objects. The primary goal of the subdivision is to minimize a number of ray-object intersection tests, which is itself a costly operation, to tests only with objects belonging to voxels pierced by the ray.

Ray traversal algorithms designed for the subdivision speed up techniques can therefore be used also for ray tracing volumetric data. However, one difference still exists, namely the voxel scene size. While the optimal subdivision rate for the speed up techniques was found to be low (only hundreds of voxels are sufficient [AW87]), data orders of magnitude larger are processed in visualization tasks. Therefore applicability of these algorithms is only moderate and special techniques for ray tracing volumetric data were developed.

# 3 Voxel traversal algorithms

The discrete ray generators known from volume visualization and computer graphic literature can be categorized as

**continuous ray generators** which define the ray as a sequence of points lying on an analytical line and the corresponding voxel coordinates are derived secondarily and

**discrete ray generators,** which generate the sequence of voxels pierced by a ray directly.

In the latter category we can further distinguish *integer* based algorithms, which assume that the ray has start and end points with integer coordinates [KS86] and algorithms enabling arbitrary start point and direction [AW87, CW88].

Usually only a fraction of the scene space is occupied by object voxels. The remaining background voxels are empty, do not contribute to the rendered image and can be skipped during the

processing. Therefore various algorithms were developed to speed up or even avoid traversal of the background voxels by gathering them into macro regions or by adding distance-from-the-object information.

In the following sections we shall describe some approaches for the voxel traversal published in the literature, compare them from the point of view of speed and then mention the consequences, which does the choice of the traversal algorithm has onto the exact surface-ray intersection point computation.

## 3.1 Single voxel step algorithms

The most straightforward definition of a ray in 3 dimensions is via its parametric equation

$$\mathcal{X} = \mathcal{A} + t\vec{u}$$

where $\mathcal{A}$ is one point lying on the ray, usually in the virtual image plane and $\vec{u}$ is its direction vector. This so called floating point 3D DDA algorithm defines the ray as a sequence of equidistant samples and due to its algorithmic simplicity, it was used by more authors [Lev88, RB88]. Parameter $t$ defines the distance between subsequent samples along the ray. However, it is not possible to apply any of the ray connectivity criteria [CK91] to the sequence of voxels thus defined, and the ray can skip some important voxels. This is usually overcome by increasing the point density, which degrades the algorithm performance. This equation enables to define rays for parallel as well as for the perspective projection.

Although algorithmically very simple, this ray representation is computationally expensive. It can be overcome by the discrete ray generators, directly defining the sequence of voxels rather than samples.

Integer based discrete ray generators are usually modifications of the well known Bresenham's DDA algorithm, extended to 3 dimensions. E.g. Kaufman and coworkers [KS86] developed a set of algorithms for generating 3D lines with various types of voxel connectivity. An observation that 6- and 26-connected lines have different lengths:

$$\begin{aligned} L_6(p,q) &= |x_2 - x_1| + |y_2 - y_1| + |z_2 - z_1| \\ L_{26}(p,q) &= MAX(|x_2 - x_1|, |y_2 - y_1|, |z_2 - z_1|) \end{aligned}$$

led them to development of an algorithm adaptively alternating between 6- and 26-connected rays, based on proximity sensing. The 6-connected line is defined by all voxels pierced by a ray; therefore each pair of adjacent voxels has a common face. The adjacency condition is weaker for 26-rays: only a common vertex is sufficient. Since the 26-rays are shorter, they can be generated faster, but they can skip some important voxels or even penetrate through a thin surface. Therefore they are used only far enough from the object. In its vicinity, which is indicated by a flag bit, the line type switches to 6-connected version, which is slower, but no voxels are skipped.

This algorithm was used for recursive ray tracing of scanned as well as voxelized volumetric objects. Due to its integer precision, it did not enable higher order interpolations and therefore high resolution data sets (e.g. $320^3$) were to be used. Further, the surface normals for shading and secondary ray spawning had to be precomputed and stored, which increased the size of the data set into tens of megabytes.

Various authors investigated algorithms for generating 6-connected rays overcoming the drawback of limited precision of the previously mentioned approaches [AW87, CW88]. Their effort has been primarily oriented on uniformly subdivided scenes, speeding up ray tracing of analytically defined objects. Their work has been extended to **n**-dimensional space and parallelepipedal voxels in [Sla92].

The algorithm for fast ray tracing (FRT) [CW88] is symmetric in all three axes. The decision, as to which voxel will be the next in the sequence, is controlled by three decision variables $dx$, $dy$ and $dz$, which record the total distance along the ray from some common point to its last crossing

with X, Y and Z type voxel faces. If, say, $dx$ is the smallest one, than the next crossing will be with the X type face and therefore the next voxel will have the $x$ coordinate incremented. This approach results in an efficient incremental algorithm that has only a few operations for each loop and might be implemented in integer arithmetic.

This algorithm was used for rendering of volumetric data in [TPR+91]. Typical size of volumetric data sets is usually much bigger than the optimal size for ray tracing subdivision speed up techniques. Therefore, in order to increase the speed and thus to shorten the time necessary for empty background traversal, an auxiliary supervoxel data structure was build. The supervoxel describes a volume of $N^3$ voxels, with $N$ in the range from 2 to 4. This auxiliary structure enables to define a course approximation of the surface very fast.

## 3.2 Macro region based voxel traversal algorithms

Not all voxels along the ray contribute to the rendered image with the same weight. Only some of them belong to the interresting objects, while the others can be traversed rapidly or even totally skipped. This capability is called *space-leaping* [YS93] and exploits some kind of *coherency* inherent to the object and/or image space as well as to a sequence of consecutive images.

The macro region based voxel traversal algorithms exploit the object space coherency, i.e. a tendency of object (background) voxels to occupy connected regions of the space. In this case, background voxels are gathered into cubic, parallelepipedal or spherical macro regions, which can be skipped at one step thus minimizing the total rendering time. Various schemes for the macro region definition are possible. Some of them are based on hierarchical encoding of the scene space, others define the macro regions directly for the original voxel scene.

### 3.2.1 Hierarchical approaches

The first attempt to make use of hierarchical encoding of a volumetric scene for visualization purposes was done by Meagher [Mea82]. The scene was binarized and encoded into an *octree*, which was subsequently projected in back-to-front order onto a screen, resulting into depth shaded image of the object.

Levoy [Lev90] uses *complete octree* for his visualization approach. His algorithm renders data in image order, by tracing viewing rays from the observer position through the octree. For the data set measuring $N = 2^M + 1$ voxels on a side the octree is represented as a pyramid of $M + 1$ binary cell volumes. A *cell* represents a space with 8 voxels in its corners. Volume $V_0$ is the original set, with $N - 1$ cells on a side, volume $V_M$ is a single cell.

A cell in the $V_0$ is assigned 0 if all corresponding voxels have opacity equal to zero; otherwise it is assigned 1. A cell in the volume $V_m$ contains zero, if all 8 corresponding cells in the volume $V_{m-1}$ contain zero. When the ray enters a cell, its value is tested. If it contains a zero, the cell is skipped to the next one. If their parents are different, we move up to the parent of the second cell, in order to perform a larger step, if it is empty. If the cell is occupied, we move down, until we find a nonempty cell at the lowest level. Within this cell, we sample the data at the corresponding evenly spaced location and add its value to the resulting ray color.

The Spackman's and Willis's SMART (Spatial Measure for Accelerated Ray Tracing) scene traversal algorithm [SW91] works on the octree represented as a breadth first list encoding only the occupied object voxels. Two decision vectors control the progress of the ray through the octree:

- $H_{SMART}$ vector, which navigates iterative horizontal steps from sibling to sibling by examination of its component signs, and
- $V_{SMART}$ vector, which controls recursive vertical steps from parent to child by midpoint comparison of its components and a comparison variable.

Both decision vectors are maintained in an incremental way and may be implemented in integer arithmetic.

A different kind of hierarchical representation was proposed by Subramanian and Fussell [SF90]. A *median cut* space partitioning scheme, combined with *bounding volumes* was used to store the voxels of interrest in a *k-d* tree. First, a *culling function* is defined, identifying voxels not contributing to the rendition. In the second step, the interresting voxels are encoded into a *k-d* tree, storing, if necessary, a bounding volume of the voxels corresponding to the given node. If the voxels are not equally distributed across the node space, the bounding box enables to skip the node for some rays without inspecting its successors. During the encoding, for each internal (non-leaf) node of the tree a partitioning plane is defined, dividing the voxels within the node into two parts with equal size (median-cut).

The *k-d* tree structure can be ray traced efficiently, with no limitations on ray direction and origin. If the ray reaches a leaf node, then the voxel at this node is sampled and its contribution to the color of the ray is computed. If it is an internal one, two cases are possible: the ray traverses the node entirely on one side of the partitioning plane, or it crosses it. In the second case both regions should be examined, unless the accumulated opacity reaches unity or the surface is found in the first one.

### 3.2.2 Distance based approaches

The idea to exploit distance of the actual background voxel to the nearest object was introduced in [ZKV92]. The proposed Ray Acceleration by Distance Coding (RADC) scheme works in two phases:

**Preprocessing:** The volume is segmented and the distance information is added to background voxels by a subsequent 3D distance transform.

**Rendering:** The floating point 3D DDA is used to calculate sample points along the ray, exploiting the distance information for skipping the empty regions rapidly.

Since objects in volumetric data sets tend to be centered in the middle of the volume, rays usually skip rapidly the off-center parts and slow down until they hit an object. For parallel projection, if the ray totally misses the object, the minimal distance along its path can be utilized for further speed up. In such a case, this distance defines a circle in the image plane, where it is not necessary to fire new rays, since they all miss the object.

The RADC algorithm works with various digital approximations of the ideal Euclidean distance [Bor86]. Of course, higher speed up can be reached with its better approximation, as e.g. with the chamfer distance. A different method exploiting the distance information, proposed in [Š94a], relies exclusively on the chessboard distance (CD). The CD equal to $n$ defines a cubic macro region with side size $2n + 1$ centered in the actual background voxel. Since we know that no object voxels are inside, we can skip the whole macro region and proceed with the first voxel outside of it. The cubic geometry enables to design a hybrid algorithm with features of both continuous and discrete ray generators. The voxel coordinates are generated directly, as well as a series of points lying exactly on the ray due to the fact that the coordinates of ray entry point into the voxels are used as a parameter which controls the traversal.

The CD algorithm thus defines a sequence of nonuniform samples along the ray, lying exactly on voxel faces. In the object vicinity, where the background voxels have CD equal 0, 6-connected line is defined. As will be shown in the following sections, for the binary surface rendering both these properties enable to perform hit-miss test and exact ray-surface intersection computation rapidly.

## 4 Speed comparison of various algorithms

A performance comparison of various algorithms from the data published by their authors is impossible, since different test scenes as well as hardware platforms and operating systems are

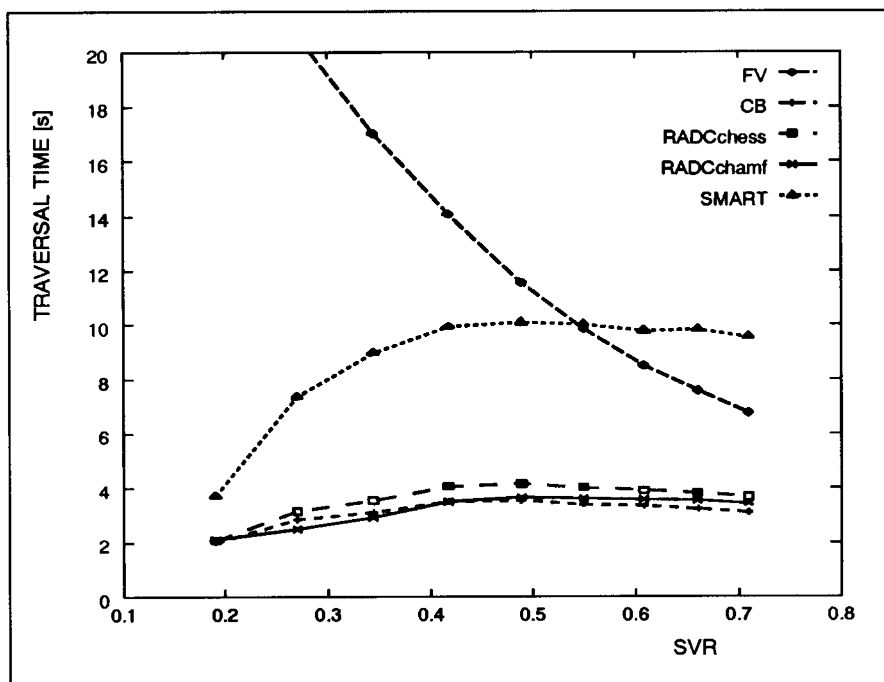


Fig. 2: Comparison of various voxel traversal algorithms

usually used for their implementation. Therefore a computer experiment was set up, based on rendering of phantom scenes with randomly positioned spheres of various size and number.

Each scene built up of 128x128x128 voxels was subdivided into $N^3$ subregions ($N = 1, \ldots, 10$). Within each subregion, a voxelized sphere was randomly placed (1–1024 spheres), such that total volume of all spheres was identical for all N. Thus we obtained scenes with equal number of object voxels but different surface to volume ratio (SVR).

In the experiment, only parallel primary rays were traced until the first object voxel was found, with no subsequent shading. The results of the experiment are depicted in Figure 2. The $y$ axis values represent the pure traversal time necessary for rendering of a 250x250 image. Values on the left side of the graph with lower SVR correspond to the simple scenes with low number of spheres, while those on the right side belong to the more complex scenes.

The comparison of four algorithms fulfilling the conditions from the Section 2 was done:

1. FRT algorithm (Section 3.1) representing the single voxel step approaches,
2. SMART (Section 3.2.1) representing hierarchical speed up methods,
3. RADC (Section 3.2.2) with two different distance types used: chamfer (RADCchamf) and chessboard (RADCchess) and
4. CD (Section 3.2.2), both representing the distance techniques.

The necessary chessboard and chamfer distance initialization took from 18 to 21 seconds for each scene.

The rendering time for the single voxel step FRT algorithm was shorter for more complex scenes than for simple scenes. This behavior is typical, since in the first case the object is find earlier in the complex scene.

All other algorithms show opposite dependence, since their speed is influenced also by the mean step length, which is shorter for more populated scenes and which overweights the gain of earlier

object detection. The worst results were obtained by the hierarchical approach, due to the complex octree traversal in both horizontal and vertical directions.

Both distance methods show very similar behavior. As it was expected, the RADC performance was worser with the chessboard than with the chamfer distance. Since the CD algorithm, with chessboard distance shows nearly the same results as the RADC with the chamfer distance, which is a better approximation of the Euclidean distance, we can conclude that it can utilize its macro region structure more effectively.

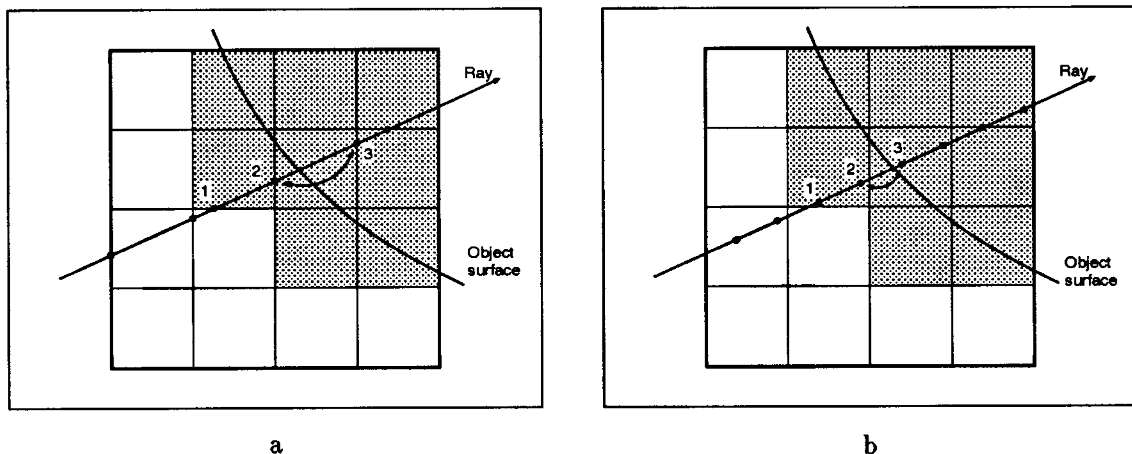


Fig. 3: Exact surface point detection, (a) CD algorithm, (b) RADC

# 5 Hit-miss test and surface detection

As we have already mentioned in the previous sections, choice of the background traversal algorithm can also influence subsequent steps of hit-miss test and exact ray-surface intersection computation. From this point of view, we compared two ways of the continuous line definition, based on

1. nonuniform samples, lying on faces of pierced voxels (represented by the CD approach) and

2. equidistant samples along the ray (as e.g. RADC).

For the comparison purposes a scene with a voxelized sphere was generated and rendered with such magnification that all rays hit the surface.

The first case is depicted in Fig. 3(a). Once an object voxel is reached, it is necessary to evaluate the function $f^i$ at the sample points (Fig. 3(a), points 2, 3) until its value exceeds the threshold T (point 3). In the final step, exact position of the intersection should be searched for between the last two samples (points 2 and 3).

It should be pointed out that

1. it is necessary to evaluate $f^i$ only at samples on faces shared by two object voxels, since those lying between an object and background voxel will always give value below the threshold T,

2. for the samples lying on voxel faces, the trilinear interpolation function degenerates to bilinear, which can be computed faster and

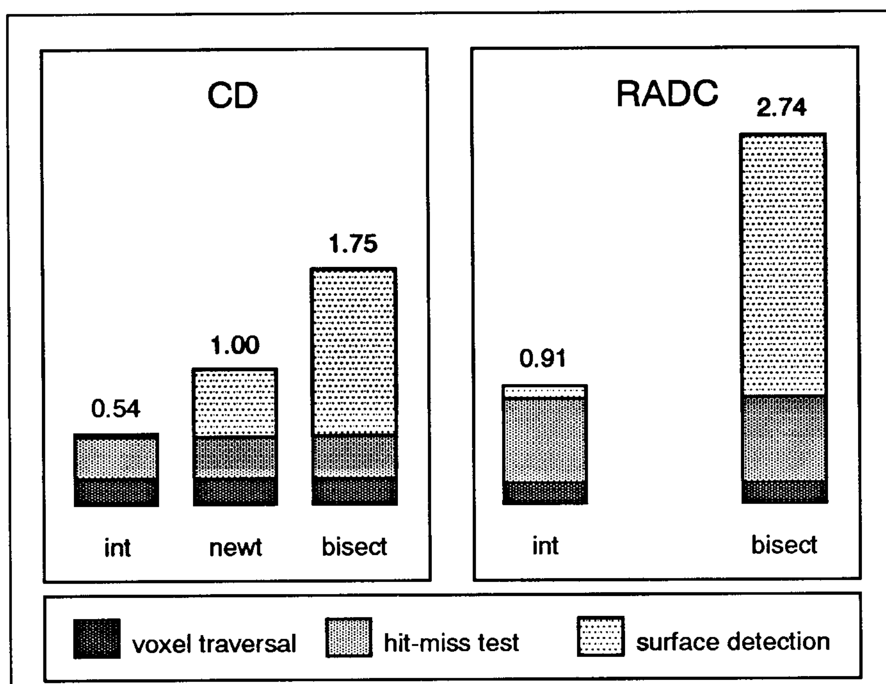


Fig. 4: Background traversal and surface detection results for two different ray generators

3. since $f^i$ is continuous within the voxel, a numerical root finding method based on derivatives can be involved (e.g. Newton-Raphson).

However, the situation is quite different with the uniform samples. The complete trilinear function should be evaluated often for more points ((Fig. 3(b), points 1, 2, 3) and bisection should be used due to discontinuity of its derivatives at voxel faces. Some overhead is also added, since the search process works within more than one voxel.

Results of this experiment are summarized in Fig. 4. In addition to the two already mentioned iterative root finding schemes (`newt`, `bisect`) a simple noniterative intersection estimation by linear interpolation between the two $f^i$ values above and under the threshold T was added. The rendering times are expressed with respect to the best result reached for the iterative approach.

The background traversal, although with 3D complexity, represents the shortest time interval. Much more time is spent for the hit-miss test and the surface position detection. The `bisect` surface detection is slower for the RADC than for CD (more voxels involved), which is further slower than `newt` surface detection (derivatives). The RADC hit-miss test also needs more time, due to more samples and trilinear interpolations.

We can see that the approach with uniform ray sampling needs nearly 3 times more time than that with samples on voxel faces. The intersection estimation (`int`) is of course the fastest, although not so precise. However its precision is high enough for applications with only primary rays, as is usually the case of visualization of scanned objects.

# References

[AW87] John Amanatides and Andrew Woo. A fast voxel traversal algorithm for ray tracing. In G. Marechal, editor, *Proc. EUROGRAPHICS '87*, pages 3–10. North-Holland, 1987.

[Bor86] Gunilla Borgefors. Distance transformations in digital images. *Computer Vision, Graphics, and Image Processing*, 34(3):344–371, 1986.

[CK91] Daniel Cohen and Arie Kaufnan. Scan-Conversion Algorithms for Linear and Quadric Objects. In Arie Kaufman, editor, *Volume Visualization*, pages 280–300. IEEE Computer Society Press, 1991.

[CW88] John C. Cleary and Geoff Wyvill. Analysis of an algorithm for fast ray tracing using uniform space subdivision. *The Visual Computer*, 4(2):65–83, July 1988.

[FTI86] Akira Fujimoto, Takayuki Tanaka, and Kansei Iwata. Arts: Accelerated ray-tracing system. *IEEE Computer Graphics and Applications*, 6(4):16–26, 1986.

[KCY93] Arie Kaufman, Daniel Cohen, and Roni Yagel. Volume graphics. *IEEE Computer*, 26(7):51–64, July 1993.

[KS86] Arie Kaufman and Eyal Shimony. 3D scan-conversion algorithms for voxel-based graphics. In Frank Crow and Stephen M. Pizer, editors, *Proceedings of 1986 Workshop on Interactive 3D Graphics*, pages 45–75, Chapel Hill, North Carolina, October 1986.

[Lev88] Marc Levoy. Display of surfaces from volume data. *IEEE Computer Graphics and Applications*, 8(3):29–37, May 1988.

[Lev90] Marc Levoy. Efficient ray tracing of volume data. *ACM Transactions on Computer Graphics*, 9(3):245–261, 1990.

[Mea82] Donald Meagher. Geometric modeling using octree encoding. *Computer Graphics and Image Processing*, 19:129–147, 1982.

[RB88] R. A. Robb and C. Barillot. Interactive 3-D image display and analysis. In *Proceedings SPIE on Hybrid Image and Signal Processing*, volume 939, pages 173–195, Bellingham, WA, 1988.

[SF90] K. R. Subramanian and D. Fussell. Applying space subdivision techniques to volume rendering. In A. Kaufman, editor, *Visualization '90*, pages 150–159, IEEE Computer Science Society Press, 1990.

[Sla92] Mel Slater. Tracing a ray through uniformly subdivided n-dimensional space. *The Visual Computer*, 8(9):39–46, 1992.

[SW91] John Spackman and Philip Willis. The SMART navigation of a ray through an oct-tree. *Comput.&Graphics*, 15(2):185–194, 1991.

[TPR+91] Reinhard A. Thaller, Hellmuth Petsche, Peter Rappelsberger, Helmut Pockberger, Klaus Lindner, and Herwig Imhof. An approach to a synopsis of EEG parameters, morphology of brain convolutions and mental activities. *Brain Topography*, 4(1):65–73, 1991.

[Š94a] Miloš Šrámek. Cubic macro-regions for fast voxel traversal. *Machine Graphics & Vision*, 3(1/2):171–179, 1994.

[Š94b] Miloš Šrámek. Fast surface rendering from raster data by voxel traversal using chessboard distance. In R. Daniel Bergeron and Arie E. Kaufman, editors, *Visualization'94*, pages 188–195, Washington, D.C., October 17–21, 1994. IEEE Computer Society Press.

[WK93] Sidney W. Wang and Arie Kaufman. Volume sampled voxelization of geometric primitives. In *Visualization '93*, pages 78–84, San Jose, CA, October 1993.

[YCK92] Roni Yagel, Daniel Cohen, and Arie Kaufman. Discrete ray tracing. *IEEE Computer Graphics and Applications*, 12(5):19–28, September 1992.

[YS93] Roni Yagel and Zhouhong Shi. Accelerating volume animation by space-leaping. In *Visualization '93*, pages 62–84, San Jose, CA, October 1993.

[ZKV92] Karel J. Zuiderveld, Anton H. J. Koning, and Max A. Viergever. Acceleration of ray-casting using 3D distance transforms. In R. A. Robb, editor, *Visualization in Biomedical Computing II, Proc. SPIE 1808*, pages 324–335, Chapel Hill, NC, 1992.