Fast ray-tracing of rectilinear volume data

Miloš Šrámek

Institute of Measurement Science, Slovak Academy of Sciences,
Dúbravská cesta 9, 842 19 Bratislava, Slovak Republic
e-mail: miloss@umhp.savba.sk

Abstract. Tomographic devices often produce data with directionally and spatially dependent resolution. Resampling to cubic voxels is possible at the cost of a significant increase of data volume and rendering time. We present an algorithm for direct ray tracing of rectilinear grids, which enables the implementation of surface rendering with subvoxel surface detection based on local interpolation, as well as different volume rendering techniques (color compositing, reprojection, maximum intensity projection). Further we present a faster version of the basic algorithm, based on cubic macro-regions assigned to each background voxel. Each macro-region is defined by its chessboard distance to the nearest foreground voxel and can be skipped during the scene traversal. The speed-up is thus gained by increasing the step along the ray, maintaining 6-connectivity of the ray in the object vicinity, which is necessary for correct surface detection.

1 Volume visualization by ray tracing

Ray tracing is now common in the field of computer graphics, and the technique has also gained popularity in volumetric visualization. This is due to its ability to enhance spatial perception of the scene using such effects as transparency, mirroring and shadow casting. Correct understanding of the nature of the processed data further necessitates usage of various visualization techniques. In the framework of its basic scheme, ray tracing enables the implementation of various surface as well as volume rendering techniques (color compositing, reprojection, MIP), which makes it an ideal tool for data exploration.

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 a *surface model*, i.e., to define a set of patches approximating the surface. These patches can then be rendered by some standard technique, usually with a hardware support. The *binary volume rendering* techniques represent an alternative. Although the image is still only contributed to by surfaces, no explicit surface model is defined. Instead, a trivariate implicit function $\mathcal{F} = \mathcal{F}(P, P_\sigma)$ is given, depending on voxel position P and data samples in some neighborhood P_σ. A continuous surface description can be then obtained by thresholding this function at some level T.

In order for there to be 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 an analytical object 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 do not contribute to the rendered image.

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 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 and

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[1].

Traversal of the ray voxels usually 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 by the following object or background voxel, or if a *ray-surface intersection* should be searched for. The hit-miss test can be performed by evaluation of the interpolating function $\mathcal{F}$ at one or more 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 of equations defined by the ray and the surface $\mathcal{F}(\mathcal{P}, \sigma) = T$) should be solved either analytically or numerically[2].

The *probabilistic 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 data. The opacity reflects a measure by which the given voxel can contribute to the rendered image. A *culling* function can be defined, identifying the voxels which cannot contribute to the rendition and which can be discarded from consideration, alike to 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, which can be empty or can contain a list of contributing objects. The primary goal of the subdivision is to limit the number of ray-object intersection tests, which are themselves costly operations, by only performing tests with objects belonging to voxels pierced by a ray.

Ray traversal algorithms designed for the subdivision speed up techniques can therefore also be 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[3]), data sets which are 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.

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 interesting objects or surfaces, while the others can be traversed rapidly or even totally skipped. This capability is called *space-leaping*[4] and exploits some kind of *coherence* 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 coherence, i.e., the 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 in one step thus reducing the number of steps and therefore also 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 in the original voxel scene.

2.1 Distance based speed up techniques

Distance transforms convert a 2D (3D) binary image into an image, where each background pixel (voxel) is assigned a value corresponding to its distance to the nearest object pixel (voxel). Although computing the distances is in principle a global operation, algorithms were developed for approximating the global distances by propagating distances between neighboring pixels[5].

The idea to exploit the distance transforms to speed up the background traversal was introduced by Zuiderveld *et al.*[6]. 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 3D distance transform.

Rendering: The floating point 3D DDA algorithm defining the ray as a sequence of equidistant samples is used, exploiting the distance information for skipping empty regions.

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 region in the image plane, where it is not necessary to fire new rays, because they all miss the object. The RADC algorithm works with various digital approximations of the ideal Euclidean distance. Since different shapes of thus defined free regions were not taken into account, highest speed has been obtained with the chamfer distance, which is the best approximation of the Euclidean distance.

A similar technique was proposed by Cohen and Sheffer[7]. The authors call the free zones defined by the distance transform *proximity clouds*. Once a ray enters a cloud cell, it can safely skip the distance determined by the cell's value. The algorithm is also based on the floating point 3D DDA algorithm and it differs from the RADC in that it takes the shape of the free zone into account: the step size depends not only on the distance value, but also on the kind of the distance and the ray direction. If the ray is defined by its direction vector $\vec{r} = (r_x, r_y, r_z)$ and the assigned distance is d then the coordinate increment should be

$$\left(\frac{d\,p_x}{D(\vec{r})}, \frac{d\,p_y}{D(\vec{r})}, \frac{d\,p_z}{D(\vec{r})} \right) \tag{1}$$

where $D(\vec{r})$ is size of the projection vector in the corresponding metrics (Euclidean metrics is used in the case of chamfer distance):

$$D(\vec{r}) = \begin{cases} |p_x| + |p_y| + |p_z| & \text{for the city block distance} \\ \sqrt{(p_x^2 + p_y^2 + p_z^2)} & \text{for the Euclidean distance} \\ \max(|p_x|, |p_y|, |p_z|) & \text{for the chessboard distance.} \end{cases} \tag{2}$$

Authors have shown that the average step for the city block distance can even be a few percent longer than for the Euclidean distance. Another advantage of the city block distance is that its computation is easier than computation of the chamfer distance, which is usually used instead of the Euclidean distance.

The proposed technique has two drawbacks:

1. The distances are calculated for the cell centers, while the current location along the ray is not necessarily in the center. Therefore, to avoid skips beyond the free zone, the computed distance d is decreased by 1.

2. The sequence of cells generated by the floating point 3D DDA algorithm does not fulfill the condition of 6-connectivity, which may cause some of the object voxels to be missed. Therefore, in the object's vicinity, the algorithm is switched to the incremental cell traversal algorithm[3] generating the 6-connected sequence.

A different method, the CD voxel traversal algorithm[2], relies exclusively on cubic macro-regions defined by the chessboard distance (CD). In this case, the ray is defined as a sequence of nonuniform samples at its intersections with the macro-region walls. This precision allows the utilization of the full size of the macro-region and overcomes the first drawback of the previous “proximity clouds” technique. In the close vicinity of an object, where CD equals zero, the macro-regions are identical to single voxels. Since the samples lie on their faces, a 6-connected sequence of voxels is defined, overcoming the second drawback.

The algorithm has two control variables: *faceType* (face is X type if it is perpendicular to the x axis) and relative position $\mathcal{P} = (p_x, p_y, p_z)$ of the sample with respect to the voxel vertex. Its main loop is based on the following observation: for each ray direction and each assigned CD there is some threshold $t_y^x(n)$ (assuming X is the actual face type and n is the assigned CD) of the p_x coordinate. If $p_x < t_y^x$, then the macro-region exit face is also X type, otherwise it is Y type. The algorithm therefore distinguishes between two step types, those with equal and those with different entry and exit face types. The fact that the sample coordinate is maintained during the traversal can speed up detection of the ray-surface intersection point[8].

3 Traversal of rectilinear grids by cubic macro-regions

In the previous section we described algorithms speeding up a binary scene traversal by exploiting distance information, assigned to each background voxel. Their common feature was that they assumed cubic voxels. In this section we introduce an algorithm which speeds up the traversal of scenes defined by a rectilinear grid. It is similar to the CD algorithm described above, in that it is also based on cubic macro-regions. However, in this case we assign the distances to background voxels of a secondary scene with cubic voxels and the same dimensions. Thus, parallelepipedal empty regions are defined in the original scene. The exact voxel dimensions are taken into account during the traversal.

Let **G** be a 3D grid of $(N_x + 1) \times (N_y + 1) \times (N_z + 1)$ points p_{ijk}:

$$\mathbf{G} = \{p_{ijk} = (S_x^i, S_y^j, S_z^k) : 0 \le i \le N_x, 0 \le j \le N_y, 0 \le k \le N_z\}, \tag{3}$$

where i, j and k are integers,

$$S_\nu^m = \sum_{i=1}^{m} \Delta_\nu^{i-1}, \quad \nu = x, y \text{ or } z \tag{4}$$

are sample coordinates ($S_\nu^0 = 0$) and $\Delta_x^0, \Delta_x^1, \ldots, \Delta_x^{N_x}, \Delta_y^0, \Delta_x^1, \ldots, \Delta_y^{N_y}$ and Δ_z^0, Δ_z^1, $\ldots, \Delta_z^{N_z}$ is the spacing between samples along each coordinate axis.

Let a voxel be a tuple $V_{ijk} = (v_{ijk}, h_{ijk})$, where $v_{ijk} = (S_x^i, S_x^{i+1}) \times (S_y^j, S_y^{j+1}) \times (S_z^k, S_z^{k+1})$ is voxel volume and $h_{ijk} \in \{0, 1\}$ is its value. Grid points p_{ijk} are defined in voxel vertices, and therefore the voxel scene **V**:

$$\mathbf{V} = \{V_{ijk} : 0 \le i < N_x, 0 \le j < N_y, 0 \le k < N_z\}, \tag{5}$$

has one element less along each axis than **G**. The voxel value $h = 1$ means that the voxel can contribute to the image: either an object surface passes through its volume (binary volume rendering) or it can contribute with nonzero color and opacity in probabilistic volume rendering. The value $h = 0$ means, that the voxel cannot contribute, and therefore it can be skipped during the traversal without processing. We denote voxels with values equal to 0 (resp. 1) 0-voxels (resp. 1-voxels).

Let the secondary voxel scene $\mathbf{V}'$ consist of the same number of unit cubic voxels along each axis and let the corresponding voxels have equal values:

$$h'_{ijk} = h_{ijk} \tag{6}$$

If we assign now to each 0-voxel V_{ijk} its chessboard distance n to the nearest 1-voxel, we define a cubic macro-region in $\mathbf{V}'$:

$$\mathcal{O}'_n(V_{ijk}) = \{h'_{pqr} = 0 : i-n \le p \le i+n, j-n \le q \le j+n, k-n \le r \le k+n\} \tag{7}$$

with its center in V_{ijk} and with side size $2n+1$. A corresponding macro-region is defined in **V**:

$$\mathcal{O}_n(V_{ijk}) = \{h_{pqr} = 0 : S_x^{i-n} \le p \le S_x^{i+n}, S_y^{j-n} \le q \le S_y^{j+n}, S_z^{k-n} \le r \le S_z^{k+n}\}, \tag{8}$$

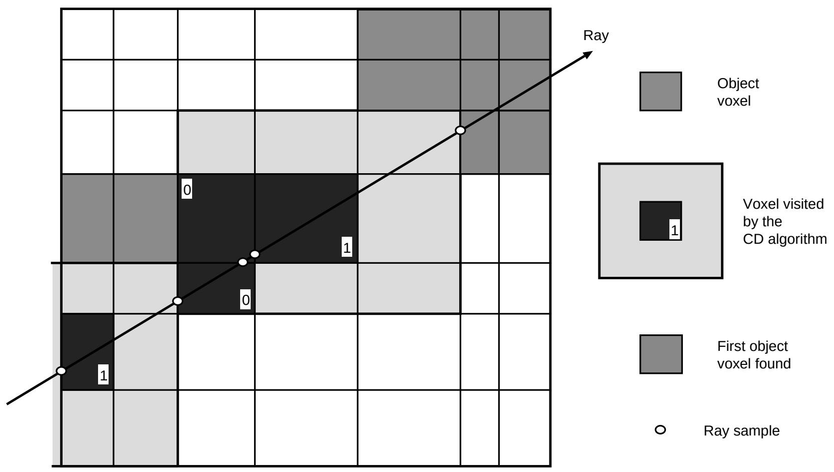


Fig. 1. Rectilinear grid traversal by cubic macro-regions

which is, of course, not cubic and voxel V_{ijk} is not situated in its center.

The voxel scene $\mathbf{V}'$ corresponds to the form of our data as stored in main memory or in a file. We do not care about the grid spacing and therefore the necessary distance transform is done in exactly the same way as for the cartesian scenes. The grid spacing should be taken into account during the ray definition, as will be shown in the next section.

3.1 The algorithm

We know that there are no object voxels within $\mathcal{O}^n(V)$, so we can jump from V directly to the first voxel outside of $\mathcal{O}^n(V)$. The traversal speed up is thus increased by reducing the number of visited voxels (Figure 1). We shall assume that the direction vector has only nonnegative coordinates. Generalization to all possible directions is done by proper initialization of some variables.

Let us imagine that the ray has reached voxel V with coordinates $\mathcal{V} = (v_x, v_y, v_z)$ and assigned chessboard distance n, at an entry point $\mathcal{P} = (p_x, p_y, p_z)$ positioned at one of its walls. It is necessary to find the nearest intersection of the ray $\mathcal{X} = \mathcal{P} + t\vec{r}$ with the planes:

$$\begin{aligned} x &= S_x^{v_x+n} &\Rightarrow\quad t_x &= \frac{S_x^{v_x+n} - p_x}{r_x} \\ y &= S_y^{v_y+n} &\Rightarrow\quad t_y &= \frac{S_y^{v_y+n} - p_y}{r_y} \\ z &= S_z^{v_z+n} &\Rightarrow\quad t_z &= \frac{S_z^{v_z+n} - p_z}{r_z}, \end{aligned} \tag{9}$$

The nearest intersection is defined by $t = \min(t_x, t_y, t_z)$.

Due to symmetry of the algorithm with respect to all three coordinate axes and due to the possibility given by the C language macro preprocessor to manipulate source code

```
#define CDstep( t_x, t_y, t_z, n )
    if( t_x ≤ t_y ∧ t_x ≤ t_z ){
        Step( x, y, z, n )
    else if(t_y ≤ t_z)
        Step( y, z, x, n )
    else
        Step( z, x, y, n )
```

```
#define Step( x, y, z, n )
    MasterStep( x, n )
    SlaveStep( x, y, n )
    SlaveStep( x, z, n )
```

Fig. 2. Traversal of rectilinear grids by cubic macro-regions: the algorithm kernel

symbols, the algorithm can be coded in a compact form, represented by two macros (Figures 2 and 3):

MasterStep updates the sample position and voxel coordinate for that axis ν, for which $t_\nu = \min(t_x, t_y, t_z)$. Versions **MasterStep_1** and **MasterStep_n** for single voxel and macro region traversal (Figure 3) differ only in the step size, which updates the coordinate value. The remaining two lines, which test if the voxel is within the scene bounds and update the point coordinate p_x, are the same.

SlaveStep updates the variables for the remaining two axes. Now, the macros differ significantly. In the single voxel step (**SlaveStep_1**) the new ray point is positioned on a wall of the same voxel as the previous point. Therefore it is sufficient to update only the point coordinate, while the voxel coordinate remains the same. The situation is different for the macro-region step (**SlaveStep_n**):

1. The ray may leave the scene in the y direction, therefore the new point coordinate should be compared with the scene bounding box, and
2. the voxel coordinate should be updated, since the ray may skip several voxels along the y direction. It is not possible to compute its value directly from the point coordinate p_y (as in the case of uniform or cartesian grid). Function **Locate** returns this coordinate by binary search in the array S_y of the grid point coordinates. Values v_y and $v_y + n$ define lower and upper bounds for this search.

The algorithm can be easily extended to arbitrary rays by mirroring the scene along the axes with negative projection vector coordinate:

```
if(r_ν < 0) then
    p_ν ← S_ν^{N_ν} − p_ν
    S_ν^i ← S_ν^{N_ν} − S_ν^i
endif
```

This inversion should also be taken into account when addressing the 3D attribute and data arrays.

A faster version of the algorithm can be obtained by replacing the sample coordinate p_ν by $\frac{p_\nu}{r_\nu}$, by which we remove the divisions in Eq. 9 and multiplications in **SlaveStep** macros (Figure 3). This change should be applied also to the S_ν arrays.

4 Results

We implemented the algorithm on a HP720 workstation equipped with 32 MB of main memory. The test data we used was the well known CT head from the VOLII data set provided by the University of North Carolina. Originally, the data is cubic. Therefore we converted it to rectilinear first, by summing several voxels in the top-bottom direction, to get a typical CT data set: the voxel was $1 \times 1 \times 2$ in the lower facial part of the skull and $1 \times 1 \times 4$ in the cranial part. The size of the data set thus reduced from $175 \times 235 \times 225$ to $175 \times 235 \times 94$. Figure 4(a) shows a distorted image when assuming cubic voxels. The distortion is removed in Figure 4(b), where proper voxel dimension were taken into account.

Figure 5 demonstrates the difference between the single voxel and macro-region versions of the grid traversal algorithm from the point of view of effectiveness. For each ray its cost was computed, given by the number of steps necessary to reach the first surface voxel. This cost is depicted by values of gray; dark pixels represent rays with smaller number of steps. We can see a significant difference between both cost maps, preferentially for rays missing the object. Table 1 summarizes results obtained by measuring rendering times for both cases. The *background traversal time* column represents time spent exclusively by traversal of the empty background, while the second *rendering time* column also includes time necessary for the surface point detection and shading. We see that the macro-region traversal introduces a more than three-fold speed up in the background traversal phase.

5 Conclusion

Direct processing of rectilinear data without resampling to cubic voxels can save much memory space and processing time. We proposed a ray generator defining a 6-connected ray in such rectilinear data grid. Traversal of rays in empty regions of the rectilinear grid can be speeded up by macro-regions, defined by chessboard distance transform in a secondary scene with cubic voxels. Experiment has shown, that the proposed technique can speed up the traversal more than 3 times.

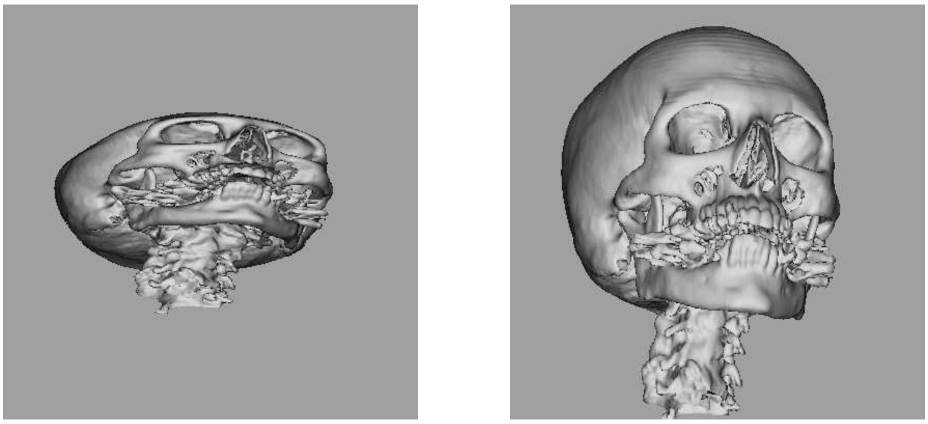
```
#define MasterStep_1( x )
    v_x ← v_x + 1
    if( v_x ≥ N_x )
        return SceneExit
    p_x ← S_x^{v_x}

#define SlaveStep_1( x, y )
    p_y ← t_x * r_y
```

```
#define MasterStep_n( x, n )
    v_x ← v_x + n
    if( v_x ≥ N_x )
        return SceneExit
    p_x ← S_x^{v_x}

#define SlaveStep_n( x, y, n )
    p_y ← t_x * r_y
    if( p_y ≥ S_y^{N_y} )
        return SceneExit
    v_y ← Locate( p_y, S_y, v_y, v_y + n )
```

Fig. 3. Traversal of rectilinear grids by cubic macro-regions: voxel and macro-region steps



(a) (b)

Fig. 4. Data set 1: (a) cubic voxel assumed, (b) voxel dimensions taken into account.

(a) (b)

Fig. 5. Voxel traversal cost map: (a) single voxel traversal, (b) macro-region traversals

version	background traversal time [s]	rendering time [s]
single voxel	56.1	67.2
macro-region	18.5	29.1

Table 1. Comparison of background traversal and rendering times for the single voxel and macro-region versions of the algorithm (data set 1).

6 References

[1] 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.

[2] 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.

[3] 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.

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

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

[6] 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.

[7] Daniel Cohen and Zvi Sheffer. Proximity clouds - an acceleration technique for 3D grid traversal. *The Visual Computer*, 10(11):27–38, November 1994.

[8] Miloš Šrámek. A comparison of some ray generators for volume graphics. In Václav Skala, editor, *The Third International Conference in Central Europe on Computer Graphics and Visualization 95*, pages 466–475, University of West Bohemia, Plzeň, Czech republic, February 14 – 18, 1995.