Proximity Clouds - An Acceleration Technique for 3D Grid Traversal

Daniel Cohen and Zvi Sheffer
Department of Computer Science
Ben Gurion University, Beer-Sheva 84105, Israel

Abstract

This paper presents a new method for the acceleration of ray traversal through a regular 3D grid. A distance transformation is pre-computed and mapped onto the empty grid space. A ray traversing the empty space is assisted by the distance values which permit to perform long skips along the ray direction. We show that the City-Block metric simplifies the preprocessing with no penalty at the traversal phase. Different schemes are discussed and the trade off between the preprocessing time and the speed up is analyzed.

Key Words: Ray tracing - Space subdivision - Distance transformation - Distance map - Grid traversal.

1 Introduction

Ray tracing is one of the most popular techniques for synthesizing realistic images. Many speed-up techniques have been developed to reduce its high computational cost (Arvo and Kirk 1989). The most effective approach, based on object space coherence, is to reduce the number of intersection calculations for each ray by testing the ray only against a fraction of the objects in the scene (Arvo and Kirk 1987; Kay and Kajiya 1986; Rubin and Whitted 1980). One prominent class of methods partitions the 3D space into cubical subspaces, called voxels. Each voxel contains only the objects whose volume is included, even partially, within the voxel boundaries. During ray tracing, a routine incrementally propagates through the voxels pierced by the ray, and tests for intersections only with the objects stored in those voxels. Although it clearly reduces the number of intersection computations it introduces an overhead of the voxel traversal routine.

The subdivision strategies fall into two schemes, *regular* and *adaptive*. The regular subdivision generates only one size voxel, while adaptive subdivision creates an octree data structure whose leaves are the voxels. The voxels which contain one or few objects only, form leaf nodes while others are subdivided further. Thus, the voxels may be of different sizes. Those schemes exhibit a tradeoff between the cost of a single step and the number of steps along the ray (Fujimoto et al. 1986). The regular subdivision approach simplifies the voxel traversal routine, but more traversal steps are usually needed in comparison with the adaptive subdivision.

The traversal of an adaptive voxel grid is composed of *horizontal* and *vertical* moves. A horizontal move is a step between adjacent voxels of the same size which are immediate siblings at the octree, while a step between non immediate siblings is composed of vertical moves up and/or down the tree. The vertical move requires a *next-cell detection* test which is the search for the next adjacent voxel along the ray (Glassner 1984; Jansen 1986; Jevans and Wyvill 1989; Kunii and Wyvill 1985; Samet 1989; Sung 1991).

The traversal of a regular grid of voxels consists of horizontal steps only and avoids the overhead of the vertical moves. It has been shown that a regular space subdivision can speed up ray tracing (Fujimoto et al. 1986). The regularity of the grid structure (voxel space) provides a very simple mechanism for its traversal (Amanatides and Woo 1987; Cleary and Wyvill 1988; Cohen and Kaufman 1991). In (Yagel et al. 1992) it has been shown that when the resolution is high enough and the size of a voxel is about the size of an image pixel, it is possible to generate pleasing images by maintaining a single atomic value in each voxel. It avoids the computationally intensive analytic intersection calculation by using a simple discrete hit to detect the "intersection point" of the ray with the object. The discrete ray tracer provides substantial speed up over traditional ray tracers. However, the discrete ray tracer spends most of the time on traversing the empty voxels, a fact that has motivated us to accelerate the traversal routine. The new method presented in this paper, however, is valid for all schemes which use a regular grid of voxels and not only for a discrete ray tracer.

The main idea of this paper is that the empty voxels space can be used for our advantage by storing in those empty voxels an information that assists the traversal process. In a preprocessing stage the empty voxels are filled with scene dependent information that indicates the proximity to the surrounding objects. The information stored in an empty voxel defines a *free-zone* in which it resides. Thus, it is possible to leap over empty voxels (hereafter: a *skip*) along the ray's direction without missing a possible intersection with an object. This idea is close to the technique of *macro-region* (Devillers 1989). The macro-regions are a set of boxes bounding empty voxels. A list of macro-regions is generated in a preprocessing stage. Every empty voxel of the scene points to one or more macro-regions that surround it. A ray traversing through an empty voxel can jump to the farthest intersection point of the ray with one of the macro-regions pointed by the voxel (Figure 1). The macro-regions are, however, expensive to generate, consuming additional space and there is no optimization method for their best location. The schemes we present in this paper are easy to generate and use no extra space. In Section 7 we analyze the efficiency and speed of the new schemes by evaluating the number of steps and the cost of each step.

In order to simplify the following discussion, without loss of generality, we will consider a 2D "flat" ray tracer. Thus, we will use the term cell instead of voxel, since cell holds for 2D and 3D as well.

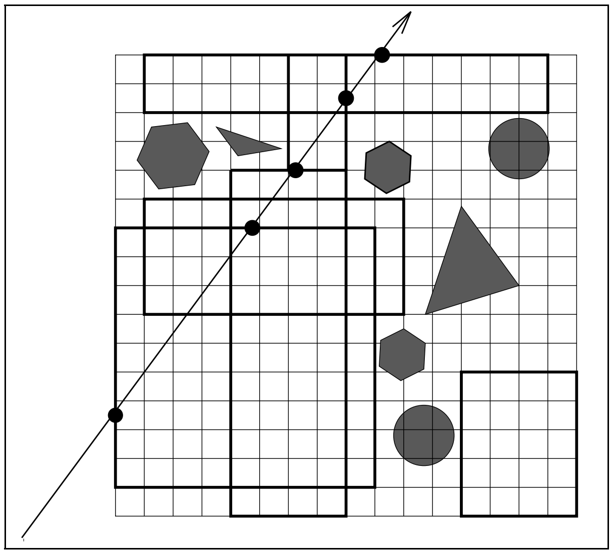


Figure 1: *A ray traversing through macro-regions. The macro-regions borders are in bold lines. For clarity, only part of the macro-regions are included in the figure.*

2 Proximity Clouds

The regular subdivision techniques suggest that it is faster to perform many simple steps rather than fewer steps but more expensive ones. However, this holds only when the ray traced scene is crowded and most of the space is occupied. In many cases large areas of the scene are left empty and the adaptive subdivision schemes become more effective. Here we propose a technique by which large areas can be traversed more effectively than by the adaptive schemes within the framework of the regular subdivision.

A "full" octree defines a pyramid data structure. Because it has a "fix" structure it provides an easy access to the nodes of every level. When a ray traverses the scene it can consult the pyramid upper levels to skip over large empty regions. Unfortunately, the pyramid does not deliver the expected speed up due to the overhead of stepping vertically up and down the pyramid levels. In order to avoid the vertical steps and the costly next-cell detection, we propose to "flatten" the pyramid so that it has no hierarchies. The empty cells of the *flat pyramid* contain indices that point to the highest hierarchy of the pyramid which contains an empty ancestor of the cell (see Figure 2). When traversing a flat pyramid the next-cell detection is avoided since the necessary information is readily available at the current position along the horizontal steps. We can therefore conclude that the flat pyramid scheme is not slower to traverse than any octree/pyramid scheme.

Although our motivation is to eliminate the overhead of vertical steps (next-cell detection), we note that unlike the conventional pyramid, the flat pyramid obviously uses no extra space (actually, we are stealing the most significant bit of the cell word as an empty/non-empty flag).

A cell of a flat pyramid with an index l, implicitly points to a "macro-region" of 2^l by 2^l cells. Unlike the macro-region scheme, the free-zones of the flat pyramid do not overlap, and when the ray traverses the empty space, the end of one free-zone is the beginning of another. Note also that it is very simple to build the flat pyramid. However, the traversal step of a flat pyramid cell, as well as of a macro-region, requires a ray-cell intersection calculation which include divisions (for an efficient algorithm see Kunii and Wyvill 1985). For more details on the flat pyramid the reader is referred to (Cohen and Sheffer 1993).

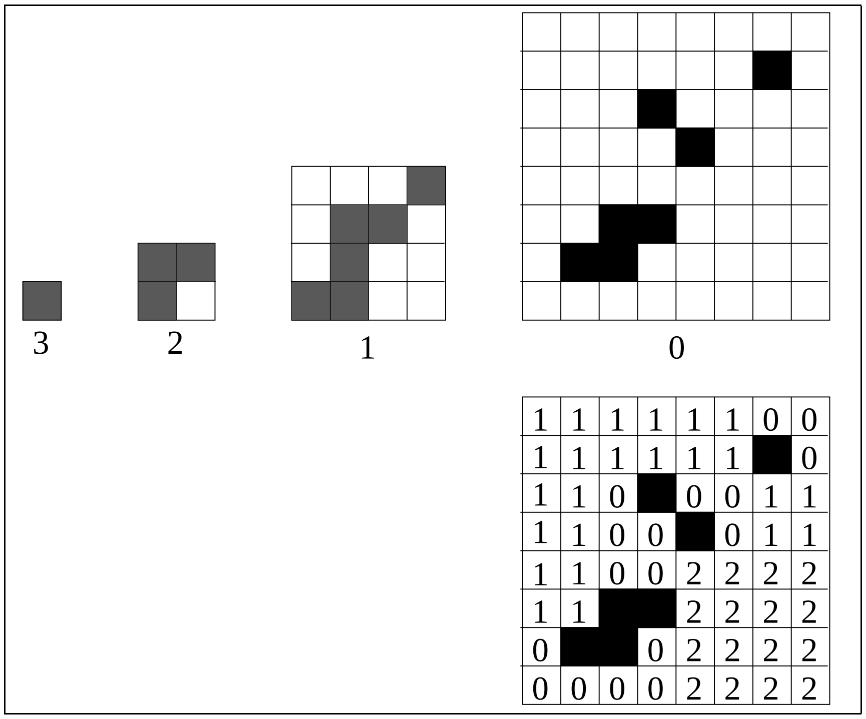


Figure 2: *The empty cells of the flat pyramid contain indices that point to the highest hierarchy which first indicates the cell to be empty.*

Following the flat pyramid concept we propose a new scheme which uses the empty cells in order to reduce the cost of a single step as well as the number of steps. The idea is to use a tight partition to free-zones which adaptively fits the shape of the objects in the scene, instead of the fixed binary partition of the octree. Those zones, due to their flexible shapes and their functionality, have been named *proximity clouds*. Each cloud layer indicates a certain distance from the nearest non-empty cell. A ray that enters a cloud cell can thus safely skip a distance determined by the cell's value.

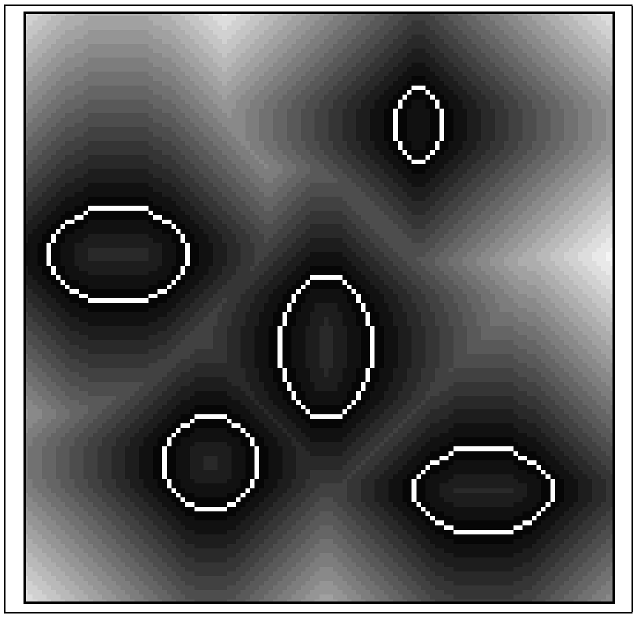


Figure 3: *Proximity Clouds (16 levels) filling the empty cells around the white objects.*

What seems to be a good realization of this idea is the following: in a preprocessing stage constructing a distance map A' where every cell (x, y) contains the value $A'(x, y)$ which is the distance to the nearest non-empty cell (k, l) (see Figure 3):

$$A'(x,y) = min_{\{(k,l),A(k,l)=1\}} D(x-k, y-l) \tag{1}$$

where in the case of the Euclidean metric

$$D(\Delta x, \Delta y) = \sqrt{\Delta x^2 + \Delta y^2} \tag{2}$$

A is a binary image where $A(x, y) = 1$ iff the cell (x, y) is not empty.

In the ray tracing stage, the rays progress along their directions with the assistance of the empty cell values encountered. A ray visiting an empty

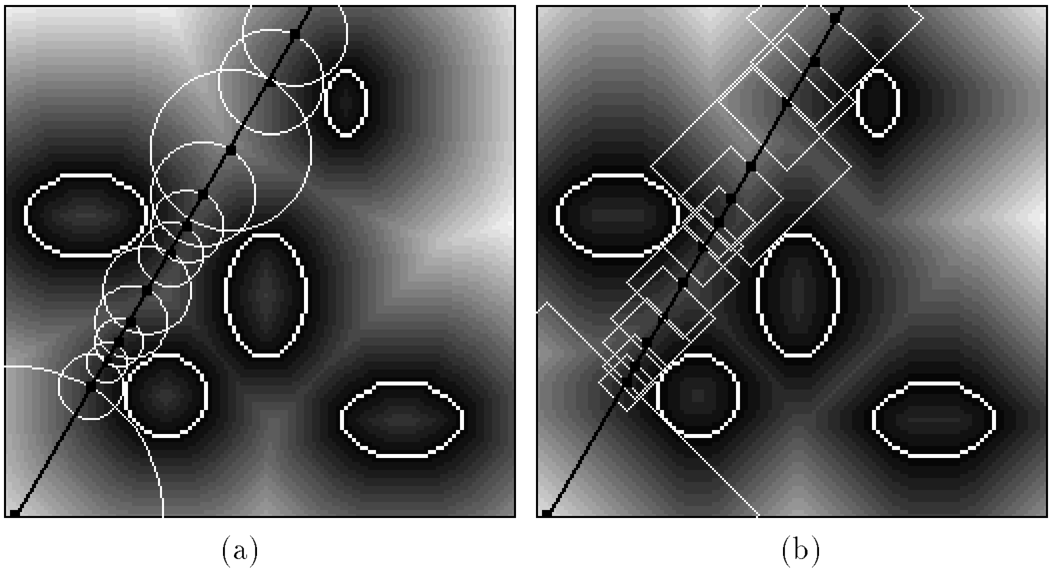


Figure 4: *A ray traversing along proximity clouds: (a) Euclidean metric. (b) City-Block metric.*

cell v_i at (x, y) with value d_i can skip up to the cell v_{i+1} at (x', y'), which contains the intersection of the ray and the sphere of radius d_i centered at v_i (see Figure 4(a)). The number of steps depends upon the scene, and on the average it is slightly less than the number of steps needed to traverse a flat pyramid. However, the cost of one skip must be checked carefully. Since only few steps are taken along one ray, the ray initialization time is substantial. In Section 2.1 we give a short review on the class of L_p metrics. In Section 3 we describe the algorithm for the traversal of a single ray through proximity clouds under an arbitrary L_p metric. We show that the City-Block metric simplifies both the ray initialization calculations (Section 3) and the preprocessing (Section 5) with no performance penalties with respect to the Euclidean metric at the traversal stage (Section 6).

2.1 L_p Metrics

We discuss the class of L_p metrics over R^2. However, the following holds for any dimension.

Let $r = (x_1, y_1)$ and $s = (x_2, y_2)$ be two points in R^2. The distance between r and s under L_p is defined as

$$D_p(r,s) = (|x_2 - x_1|^p + |y_2 - y_1|^p)^{1/p}.$$

For any $p \geq 1$, D_p is a *metric* since it satisfies the following:

1. $D_p(r,r) = 0$
2. $D_p(r,s) > 0, \quad r \neq s$
3. $\forall q \in R^2$, $D_p(r,s) \leq D_p(r,q) + D_p(q,s)$.

Let $\Delta x = x_2 - x_1$ and $\Delta y = y_2 - y_1$. The distance can be expressed as a function of Δx and Δy:

$$D_p(\Delta x, \Delta y) = (|\Delta x|^p + |\Delta y|^p)^{1/p}.$$

Known metrics with simple computation are (see Figure 5):

1. L_1, City-Block, where $D_1(\Delta x, \Delta y) = |\Delta x| + |\Delta y|$.
2. L_2, Euclidean, where $D_2(\Delta x, \Delta y) = \sqrt{\Delta x^2 + \Delta y^2}$.
3. L_∞, Chessboard, where $D_\infty(\Delta x, \Delta y) = max(|\Delta x|, |\Delta y|)$.

Note that $D_\infty \leq D_2 \leq D_1$.

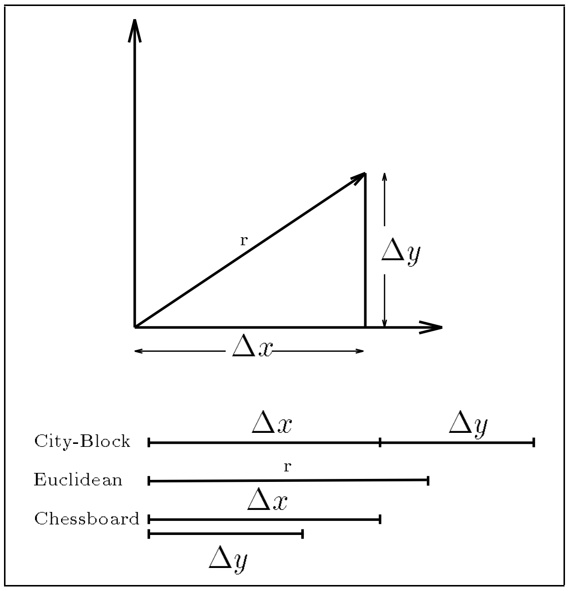


Figure 5: *The City-Block, Euclidean, and Chessboard metrics.*

3 The Traversal of the Proximity Clouds

We first consider a mechanism for skipping between two arbitrary points along a ray direction. Then, we adapt this mechanism to skip between cell .

Assume an arbitrary point $R_1 = (x_1, y_1)$ on a ray R with a direction vector $R_d = [c_x, c_y]$ and an L_p distance, d. We wish to find the coordinates of the point $R_2 = (x_2, y_2)$ which is d units ahead along the ray direction. Observe the parametric representation of the ray:

$$R = R_o + tR_d, \quad t \geq 0 \tag{3}$$

Both R_1 and R_2 satisfy the ray equation:

$$R_1 = R_o + t_1 R_d \tag{4}$$

$$R_2 = R_o + t_2 R_d$$

Subtracting the first from the second yields:

$$R_2 - R_1 = R_d \Delta t \tag{5}$$

where $\Delta t = t_2 - t_1$. Let $\Delta x = x_2 - x_1$ and $\Delta y = y_2 - y_1$. We get

$$\begin{aligned} \Delta x &= c_x \Delta t \\ \Delta y &= c_y \Delta t \end{aligned} \tag{6}$$

The L_p distance between these two points is given by:

$$d = D_p(\Delta x, \Delta y) = (|\Delta x|^p + |\Delta y|^p)^{1/p} \tag{7}$$

From (Eq. 6 and 7) we get

$$d = D_p(c_x \Delta t, c_y \Delta t) = \Delta t \;\; D_p(c_x, c_y) \tag{8}$$

thus,

$$\Delta t = \frac{d}{D_p(c_x, c_y)}. \tag{9}$$

Substitution of Δt in (Eq. 6) yields the skipping equations:

$$\begin{aligned} x_2 &= x_1 + \frac{d}{D_p(c_x, c_y)} c_x \\ y_2 &= y_1 + \frac{d}{D_p(c_x, c_y)} c_y. \end{aligned} \tag{10}$$

We can initialize the ray with the constants:

$$\begin{aligned} C_x &= \frac{c_x}{D_p(c_x, c_y)} \\ C_y &= \frac{c_y}{D_p(c_x, c_y)} \end{aligned} \tag{11}$$

thus, the skip step is:

$$x_2 = x_1 + d \; C_x \qquad (12)$$
$$y_2 = y_1 + d \; C_y$$

Note that using the City-Block metric simplifies the ray initialization since $D_1(c_x, c_y)$ is simple to compute.

At each step, the cell traversal algorithm stores the coordinates of the exact location along the ray. To avoid floating point operations, fixed-point arithmetic is used. Finding the current cell is immediate: let (x_1, y_1) be the current location along the ray, then $(\lfloor x_1 \rfloor, \lfloor y_1 \rfloor)$ are the current cell coordinates. x_1 and y_1 holds scaled integer values which should be divided by the fixed-point factor to yield the cell indices. By using a fixed-point factor of a power of two, the division can be performed using fast shift operations.

Note that the distances were calculated for the cell centers, while the current location along the ray is not necessarily at the center. Thus, to avoid skipping beyond the free-zone we can skip a distance of $d - 1$, precalculated into the distance map.

The algorithm presented in Figure 6 proceeds with such "skips" while still not in the vicinity of an object. When this is no longer the case, we switch to a face-connected cell incremental stepping to avoid a possible miss of an object. Such a miss is, unfortunately, possible, since by skipping even the minimal distance of $d = 1$, we might skip over the corner of an adjacent cell and miss a possible intersection (see Yagel et al. 1992). Moreover, when the ray "grazes" an object, the skips are too small and incremental steps will perform better.

```
**loop** {

    $x_1$ += $d * C_x$;
    $y_1$ += $d * C_y$;
    $v = \text{cell}(\lfloor x_1 \rfloor, \lfloor y_1 \rfloor)$
    **if** $(EMPTY(v))$
        $d = DISTANCE(v)$;
    **else**
        **if** $(INTERSECT(v))$
            **return**(TRUE);
        **else**
            **break**;
}
```

Figure 6: *The main loop of the proximity clouds traversal.*

The incremental stepping (see Figure 7) uses the incremental cell traversal algorithm (Amanatides and Woo 1987; Cleary and Wyvill 1988). Denote by $t_x(i)$ and $t_y(j)$ the times when the ray crosses the $i-th$ vertical and the $j-th$ horizontal grid lines, respectively. The algorithm maintains two variables t_x and t_y indicating the crossing time of the next vertical and horizontal line, respectively. Thus, at a current position the smallest of the two indicates whether the ray will next cross a vertical grid line or a horizontal one. Denote by δx and δy the portion of time for the ray to cross consecutive parallel grid lines. Since those values are constant, t_x and t_y are updated incrementally.

```
loop {

    if ($t_x < t_y$) {
        $x++$;
        $t_x$ += $\delta x$;
    }
    else {
        $y++$;
        $t_y$ += $\delta y$;
    }
    $v = \text{cell}(x, y)$
    if not ($EMPTY(v)$)
        if ($INTERSECT(v)$)
            return(TRUE);
}
```

Figure 7: *The main loop of the incremental traversal.*

When switching from the skipping stage to the incremental stepping stage we need to compute t_x and t_y by:

$$t_x = t_{x_{start}} + (x - x_{start}) * \Delta x; \tag{13}$$
$$t_y = t_{y_{start}} + (y - y_{start}) * \Delta y;$$

where x_{start} and y_{start} are the coordinates of the first cell pierced by the ray, (x, y) is the current cell location, and $t_{x_{start}}$, $t_{y_{start}}$ are the initial time values (not necessarily at the cell center).

The proximity clouds skipping algorithm in Figure 6 involves multiplications in every step. A different realization of proximity clouds uses only the

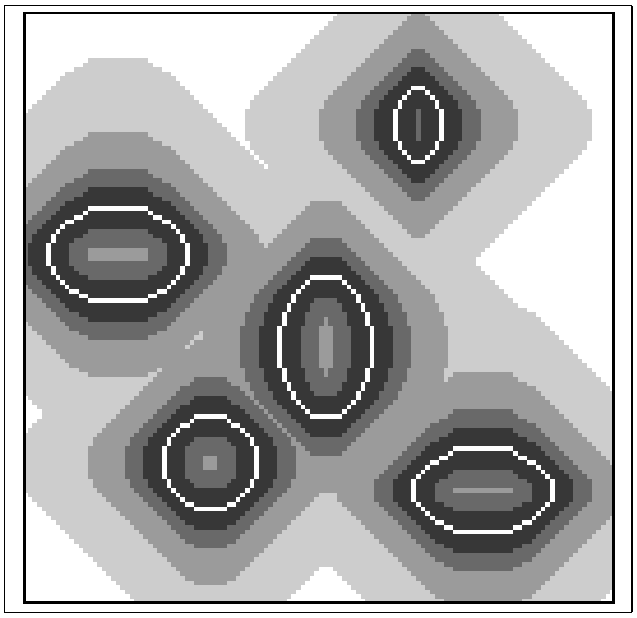


Figure 8: *Proximity Clouds of logarithmic levels.*

integer logarithm k of the distances (see Figure 8):

$$k = \lfloor \log d \rfloor. \tag{14}$$

In this way the multiplications are executed by a shift, thus reducing the cost of a single step to only one shift and one addition for each dimension. The skipping distances are shorter than before since $2^k \leq d$; but when the shift operation is significantly less costly than a multiplication the resulting computation is more efficient.

4 Comparison With Other Works

It is interesting to compare the macro-region and the proximity clouds schemes. Devillers (Devillers 1989) has found that defining multiple boxes per cell causes some overhead during the traversal due to the cost of ray-box intersections. Therefore, he has suggested to assign only one box per cell. Intuitively, every cell would be assigned to the largest box around it. It should be emphasized that using the largest box has not been shown to yield the best results. In any case, computing the intuitive partitioning into boxes as above is not practical because of the space issue: the box macro-region is defined by six values which cannot be compressed to be stored in the empty cells similarly to the proximity clouds scheme. It is possible to use a single value per pixel only if the one (maximal) cubical box is centered at the cell, which actually defines a macro-region which can be stored within the empty cell. Such compact macro-region representation actually coincides with the proximity clouds scheme in which the Chessboard metric is used.

Unlike the proximity clouds, in the macro-region model there is no deterministic strategy for the location and the size of the boxes, and thus it is hard to evaluate its performance. The generation of proximity clouds is much more simple, and it does not require any additional space. Moreover, a single step in the macro-region model consists of a ray-box intersection calculation which is much more costly than the simple step in the proximity clouds scheme.

Recently, Zuiderveld et al (Zuiderveld et al. 1992) have independently developed a similar method in which 3D distance transforms are used to accelerate volume rendering. In their work the 3D distance transform is applied to a separate additional volume, rather than to the empty cells. Moreover, they use a Chamfer metric (see Section 5) to approximate the Euclidean metric and a simple DDA traversal. As we will show, the generation of a Chamfer transform is much more expensive than that of the City-Block transform. The DDA traversal does not guarantee the traversal of all the cells pierced by the ray and thus it is not appropriate for grid traversal applications. It is important to note that in a volume rendering application most of the time is spent on the integration/composition of the non-empty cells rather on than the empty space traversal. Thus, it is not clear whether the use of extra space and preprocessing time is justified. Moreover, in a typical volume rendering application only primary rays are traversed, hence the process can be accelerated using a bounding volume (Avila et al. 1992).

5 The Generation of Distance Maps

As mentioned above, in a preprocessing stage the empty cells have to be initialized with values indicating the proximity to the surrounding objects, namely, there is a need to construct a distance transformation and map it to the empty cells. The generation of distance maps is discussed in details in (Borgefors 1986). The basic idea is to use a *mask* of local distances and to propagate these local distances over the image. The mask is applied to each cell, and each element is summed with the value under it; the value of the cell is the minimum of these sums:

$$V_{i,j} = min_{(k,l)\in M}(V_{i+l,j+k} + M[k,l])$$

where $M[l,k]$ is the local distance at position (k,l), and $V_{i+l,j+k}$ is the current value at position $(i+l, j+k)$.

The mask size and values are determined by the metric that we wish to use. Figure 9 shows the most commonly used masks. Borgefors (Borgefors

1986) has shown that it is possible to use the *Chamfer* metric (i.e., a discrete approximation to the Euclidean distance) and get a good approximation (less than 2% error) of the Euclidean distance with a 5 x 5 mask (Figure 9(d)). However, the City-Block distance map is very easy to generate since it requires only a 3 x 3 mask in which only four values are actually used (see Figure 9(a)). Alternatively, the City-Block distances can be generated by a flooding style algorithm where at each iteration a new distance layer grows out of a previous one.

For sequential computation of the distance map the mask is split into a *forward mask* and a *backward mask*, as shown in Figure 10. The forward mask is passed over the image from left to right, and from top to bottom, and the backward mask from right to left, and from bottom to top.

In the generalization of this algorithm to 3D the mask is extended to a cube (Figure 9(e)). The 3D grid is scanned once from left to right top down and once from right to left bottom up. The 3D distance map construction is therefore a non-expensive linear time operation, preprocessed once for a given scene, and especially fast when using the City-Block metric.

6 City-Block vs. Euclidean

The advantage of using a City-Block metric is three fold. Clearly, generating a City-Block distance transform is relatively simple compared to the generation of an Euclidean distance transform or the Chamfer one. It is also advantageous during the traversal stage since the initialization of each ray is much more simple than of the Euclidean metric. Consider Equation 11;

∞	+1	∞
+1	0	+1
∞	+1	∞

(a)

+1	+1	+1
+1	0	+1
+1	+1	+1

(b)

$\frac{1}{3} \cdot$

+4	+3	+4
+3	0	+3
+4	+3	+4

(c)

$\frac{1}{5} \cdot$

∞	+11	∞	+11	∞
+11	+7	+5	+7	+11
∞	+5	0	+5	∞
+11	+7	+5	+7	+11
∞	+11	∞	+11	∞

(d)

∞	∞	∞
∞	+1	∞
∞	∞	∞

∞	+1	∞
+1	0	+1
∞	+1	∞

∞	∞	∞
∞	+1	∞
∞	∞	∞

(e)

Figure 9: *The most commonly used masks. (a) The* 3 x 3 *City-block. (b) The* 3 x 3 *Chessboard. (c) The* 3-4 Chamfer. *(d) The* 5-7-11 *Chamfer. (e) The* 3D *City-block.*

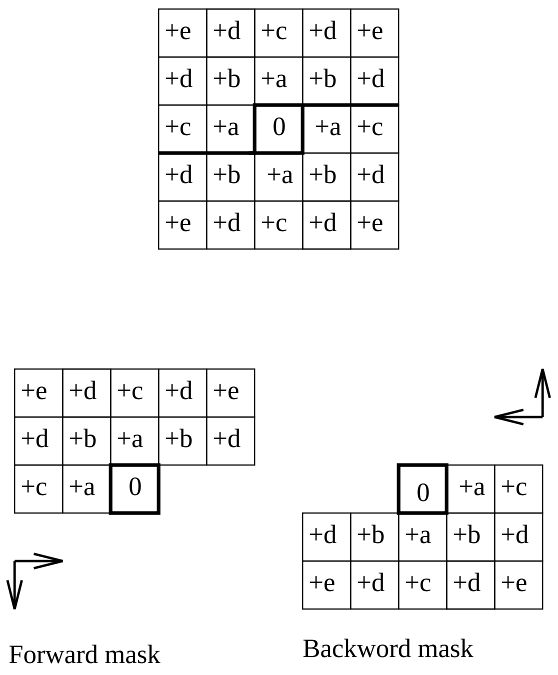


Figure 10: *A 5 x 5 mask split (at the thick line) into a forward mask and a backward mask.*

assigning the Euclidean distance function for $D_p(c_x, c_y)$ yields the expressions

$$C_x = \frac{c_x}{\sqrt{{c_x}^2 + {c_y}^2}}, \quad \text{and} \quad C_y = \frac{c_y}{\sqrt{{c_x}^2 + {c_y}^2}} \tag{15}$$

which are much more computationally expensive than the respective expressions when the City-Block distance function is used:

$$C_x = \frac{c_x}{|c_x| + |c_y|}, \quad \text{and} \quad C_y = \frac{c_y}{|c_x| + |c_y|}. \tag{16}$$

Surprisingly enough, although the City-Block metric is, in general, a poor approximation of the Euclidean one, less steps are required on the average for traversing through a City-Block map than for traversing an Euclidean map. To prove this, we first examine a simple case. Assume a scene with a single pixel obstacle located at $p = (x, y)$. Without loss of generality, assume that the ray emanates from the origin and that distance is scaled such that the Euclidean distance to p is $d_e = x^2 + y^2 = 1$. The straight line connecting the origin to p forms an angle ϑ with the X main axis. The City-Block distance to p is $d_c = x + y$ or in polar coordinates (see Figure 11):

$$d_c = sin\vartheta + cos\vartheta. \tag{17}$$

The distances d_e and d_c determine respectively the sizes s_e and s_c by which it is possible to skip along the ray vector without meeting or skipping over p. In the Euclidean model $s_e = d_e = 1$ independently of the ray direction φ. However, in the City-Block model

$$d_c = s_c sin\varphi + s_c cos\varphi \tag{18}$$

as shown in Figure 11. Thus, the step size in the City-Block map, s_c, is a function of φ given by:

$$s_c = \frac{d_c}{sin\varphi + cos\varphi}. \tag{19}$$

Using (Eq. 17) we obtain

$$s_c = \frac{sin\vartheta + cos\vartheta}{sin\varphi + cos\varphi}. \tag{20}$$

The performance of the City-Block metric with respect to the Euclidean metric can be evaluated by computing the ratio of the sums of all possible

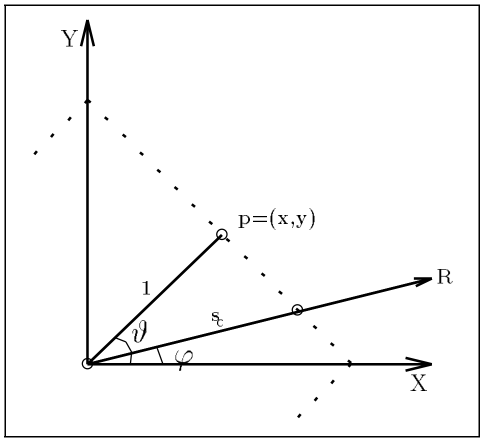


Figure 11: *A single pixel obstacle located at p forming an angle ϑ with X-axis, and a ray R emanating from the origin forming an angle φ with X-axis. The skip size along R is defined by the intersection of R and the free-zone defined by ϑ (dashed line).*

step sizes in the City-Block and Euclidean distances respectively. Let A denote that ratio. Then:

$$A = \frac{I(s_c)}{I(s_e)} \tag{21}$$

where:

$$I(s_c) = \int_0^{\frac{\pi}{2}} \int_0^{\frac{\pi}{2}} s_c d\vartheta d\varphi \tag{22}$$

$$I(s_e) = \int_0^{\frac{\pi}{2}} \int_0^{\frac{\pi}{2}} s_e d\vartheta d\varphi \tag{23}$$

as $\frac{\pi}{2}$ is the upper integral limit (rather than 2π) due to symmetry. Since $s_e = 1$, the sum of Euclidean step sizes is given by

$$I(s_e) = \frac{\pi^2}{4} \tag{24}$$

which yields the following expression for the ratio:

$$\frac{I(s_c)}{I(s_e)} = \frac{4}{\pi^2}\int_0^{\frac{\pi}{2}}\int_0^{\frac{\pi}{2}} \frac{sin\vartheta + cos\vartheta}{sin\varphi + cos\varphi} d\vartheta d\varphi. \tag{25}$$

Evaluating this expression analytically, we get:

$$A = 1.0103347 \tag{26}$$

This shows that in this case, the City-Block metric has 1% longer jumps than with the Euclidean metric (similar calculation for the 3D case has yielded 1.8%).

Consider now the general 3D case with an arbitrary number of obstacles. Assume a scene with m single-pixel obstacles located at p_j where $j = 1, \ldots, m$. Denote by $d_c(p_j)$ and $d_e(p_j)$ the City-Block and Euclidean distances from the origin to p_j, respectively. The origin will contain the minimum size:

$$\begin{aligned} d_c &= min(d_c(p_j)), \quad j = 1, \ldots, m. \\ d_e &= min(d_e(p_j)), \quad j = 1, \ldots, m. \end{aligned} \tag{27}$$

Let ψ and φ be the angles determined by the ray direction. s_c and s_e are given by:

$$s_c = \frac{d_c}{cos\psi + sin\psi(sin\varphi + cos\varphi)} \quad , \tag{28}$$

and

$$s_e = d_e. \tag{29}$$

$I(s_c)$ is given by:

$$I(s_c) = \int_0^{\frac{\pi}{2}}\int_0^{\frac{\pi}{2}} s_c d\psi d\varphi. \tag{30}$$

From (Eq. 28) we get:

$$I(s_c) = d_c \int_0^{\frac{\pi}{2}} \int_0^{\frac{\pi}{2}} \frac{d\psi d\varphi}{cos\psi + sin\psi(sin\varphi + cos\varphi)}; \tag{31}$$

$I(s_e)$ is given by:

$$I(s_e) = \int_0^{\frac{\pi}{2}} \int_0^{\frac{\pi}{2}} s_e d\psi d\varphi = \frac{\pi^2}{4} d_e. \tag{32}$$

From (Eq. 31 and 32) it follows that the ratio is:

$$A = \frac{I(s_c)}{I(s_e)} = \frac{d_c}{d_e} \ \frac{4}{\pi^2} \ \int_0^{\frac{\pi}{2}} \int_0^{\frac{\pi}{2}} \frac{d\psi d\varphi}{cos\psi + sin\psi(sin\varphi + cos\varphi)}. \tag{33}$$

Evaluating this expression numerically, we get:

$$A = 0.7368737 \ \frac{d_c}{d_e}. \tag{34}$$

Note that d_c and d_e are unknown as they vary from one point in the scene to another, and from one scene to another. This means that in a sense A is a random variable. We can therefore apply a Monte Carlo simulation technique in order to obtain an approximation to $E(A)$, $\overline{A}$, in the following way: At each iteration i, a random integer number $m^{(i)}$ is selected, $m^{(i)}$ random points $p_j^{(i)}$, $j = 1, \ldots, m^{(i)}$ are chosen, and $d_c^{(i)}$, $d_e^{(i)}$ are calculated (Eq. 27). Then, $A^{(i)}$ (the i-th realization of A) is obtained using (Eq. 34). After n iterations are completed, where n is as large as can be, we compute:

$$\overline{A} = \frac{1}{n} \sum_{i=1}^{n} A^{(i)},$$

which is an unbiased estimator for $E(A)$. It turned out that $\overline{A} = 1.037$ which means that by using the City-Block metric the average skip size is just longer (3.7%) than with the Euclidean metric. This implies that using the simple City-Block metric does not entail any increase in computational load; it actually decreases it slightly.

7 Results

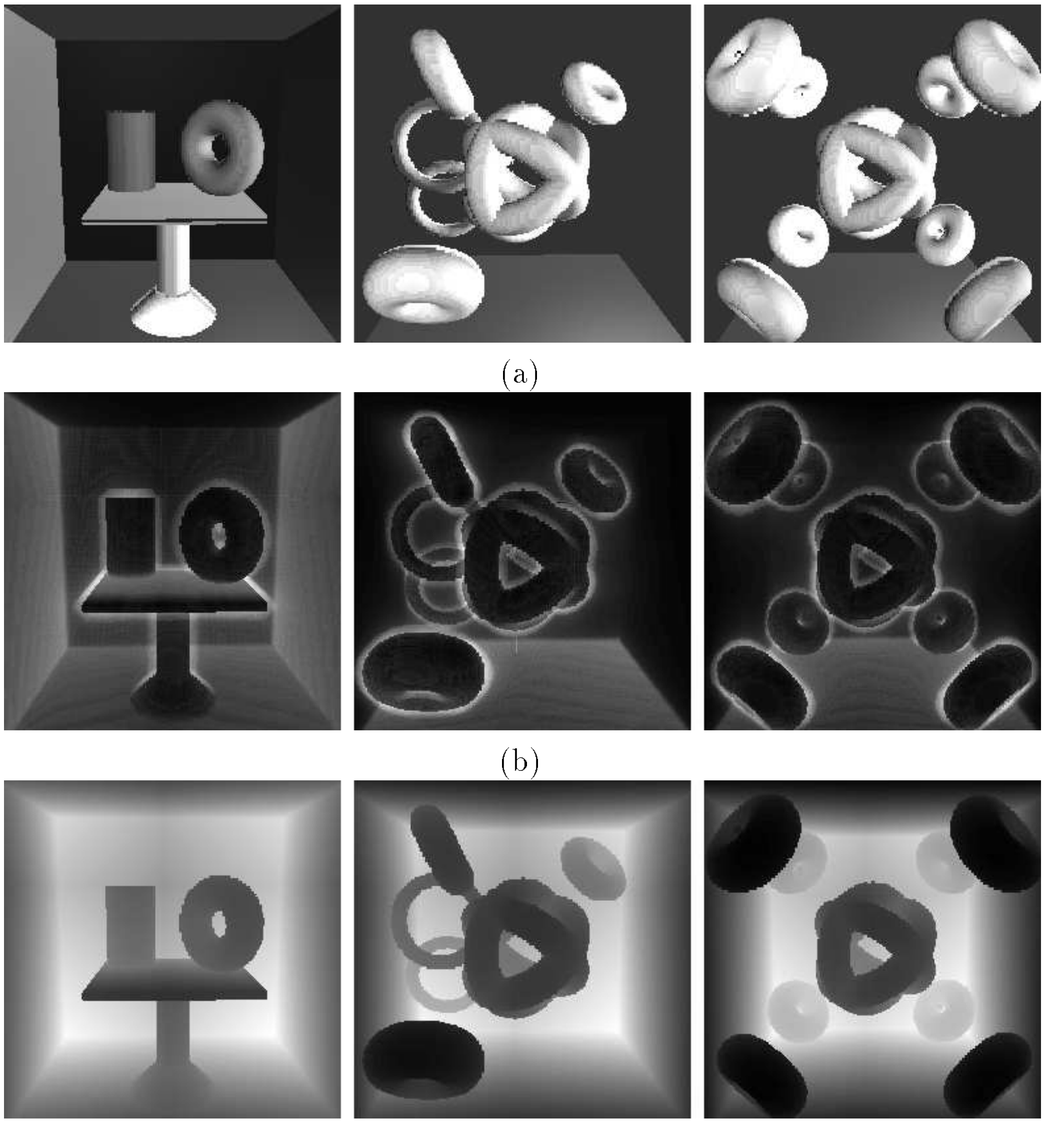
The new schemes have been tested on a discrete ray tracer where the grid cells (voxels) contain a color and a normal. This avoids the expensive ray-object intersection test, and when a high resolution grid is employed, most of the execution time is spent by the incremental traversal loop. Significant time is spent, however, on ray initialization and other calculations other than on the traversal itself, so the total execution time cannot be reduced dramaticly. Nevertheless, using the skipping schemes has yielded a speed-up of over 30% in the total time for ray tracing scenes such as those shown in Figure 12. A better criteria for the efficiency of the new traversal schemes can be the product of the average number of steps and the cost of each step. The average number of steps for the different schemes are presented in Table 1. First, we have implemented the 2D case of those schemes where it is possible to visualize the skipping process (see Figures 3 and 4), while the 3D implementation confirmed our 2D results. The 3D scenes were subdivided into 128^3 cells only due to memory limitations. Much better speed up is expected for higher resolution since the number of skips in the proximity clouds scheme is in the order of the logarithm of the number of steps taken by the incremental stepping scheme. To show that, observe the expected growth of the number of steps for a given scene when the resolution is doubled. Clearly, the number of incremental steps will be doubled, while only few more skips will be added to the proximity clouds traversal. In Figure 12 we presents some of the 3D scenes we have tested. The pixels of the images reflect the number of steps taken along the ray emanating from the pixel.

Note from Table 1 that the generation time of the proximity clouds together with the traversal time is only slightly more than the incremental traversal, which means that the preprocessing is justified when many views of the same scene are rendered.

A comparison of the relative cost of the skip step at the different schemes is more difficult. However, the flat pyramid traversal requires floating point arithmetic since a slight numeric inaccuracy might select a wrong cell and shift the logic of the algorithm. The proximity clouds scheme may use fixed-point arithmetic since in the vicinity of an object, when the algorithm switches to incremental stepping, it is possible to correct the accumulated error. The logarithmic scheme is more efficient and useful if the cost of a single shift operation is significantly cheaper than that of a multiplication (this holds for most current workstations).

8 Conclusions

The aim of this paper is to show that it is possible to use the space of the empty cells to accelerate their traversal. We presented the flat pyramid model and showed that it outperforms the classical acceleration techniques using octrees or similar schemes of adaptive space subdivision by avoiding the overhead of the vertical traversal. Then we introduced the proximity clouds technique and showed that it is even more efficient than the flat pyramid since it needs fewer skips to traverse the empty space. Moreover, the logarithmic scheme uses no multiplications and yields a significant speed-up relative to the other two schemes. Using a high resolution subdivision requires a huge


(a)

(b)

(c)

Figure 12: *(a) Shaded scenes (1-table, 8-tori, 11-tori) subdivided into* 128^3 *cells. (b) Traversal through proximity clouds. (c) Incremental traversal. In (b) and (c) each pixel reflects the number of steps needed for its pixel ray.*

Table 1: *Results for the scenes of Figure 12.*

Scene		**Incremental**	**Proximity clouds**
1-table	Distance trasform time	—	7.2
	Traversal time	29.1	23.1
	Number of steps	123.8	15.5
8-tori	Distance trasform time	—	7.0
	Traversal time	22.9	13.2
	Number of steps	94.1	9.1
11-tori	Distance trasform time	—	7.0
	Traversal time	22.0	14.3
	Number of steps	78.9	9.6
`average`	Distance trasform time	—	7.1
	Traversal time	24.7	16.9
	Number of steps	98.9	11.4

amount of memory; nevertheless, the new techniques in which large skips are performed with no additional cost relieve the burden of having to traverse the large number of empty cells.

Similar to regular or adaptive subdivision schemes these acceleration techniques require preprocessing. Generating a City-Block distance transform is relatively simple compared to the subdivision process itself or to the generation of an Euclidean distance transform. We have shown that using the simple City-Block metric rather than the Euclidean metric does not entail any increase in computational load, but it even decreases it slightly.

The traversal of the empty space with the assistance of the proximity clouds is efficient as long as the rays do not pass near and in parallel (or close to parallel) to an object (note the silhouettes in Figure 12(b)). It should be noted that when adaptive supersampling is used, many of the rays are cast towards the silhouettes. For those rays the skips are taken until they approach the object, but then, as mentioned above, there is a switch to the conventional incremental stepping. Note that the information stored in the empty cells is direction independent. If enough space is available, n values can be associated with n free-zone radial sections. A ray passing through a cell within a known section can use the appropriate value. Such a directional distance transform is more expensive to generate and it is not clear whether it is cost effective in general (i.e., in non-pathological cases). Note also that since logarithmic values use fewer bits for their representation, a number of these values can be stored within a single cell. We believe that the concept presented in this paper has the potential to provide a basis for future acceleration techniques that exploit the empty space.

Acknowledgments

We wish to thank Uri Shaft for helping to evaluate the performance of the City-Block and Euclidean metrics and for generating the scenes. We also wish to thank Yerucham Shapira and Roni Yagel for their careful review and useful suggestions.

References

Amanatides J, Woo A (1987) A fast voxel traversal algorithm for ray tracing. *Proc EUROGRAPHICS'87*, pp 3–9

Arvo J, Kirk D (1987) Fast ray tracing by ray classification. *Computer Graphics (Proc SIGGRAPH'87)* 21(4):55–64

Arvo J, Kirk D (1989) A survey of ray tracing acceleration techniques. In: Glassner AS (ed) *An Introduction to Ray Tracing.* Academic Press, London San Diego, pp 201–262

Avila R, Sobierajski L, Kaufman A (1992) Towards a comprehensive volume visualization system. *Proc Visualization'92*, pp 13–20

Borgefors G (1986) Distance transformations in digital images. *Comput Vision Graph Image Processing* 34(3):344–371

Cleary JG, Wyvill G (1988) Analysis of an algorithm for fast ray tracing using uniform space subdivision. *The Visual Computer* 4:65–83

Cohen D, Kaufman A (1991) 3D discrete lines: Voxelization algorithms and connectivity control. Technical report TR. 91.05.09, Computer Science, SUNY at Stony Brook, New York

Cohen D, Sheffer Z (1993) Proximity clouds - an acceleration technique for 3D grid traversal. Technical report FC 93-01, Math & Computer Science, Ben Gurion University, Beer-Sheva

Devillers O (1989) The macro-regions: an efficient space subdivision structure for ray tracing. *Proc EUROGRAPHICS'89*, pp 27–38

Fujimoto A, Tanaka T, Iwata K (1986) ARTS: Accelerated ray-tracing system. *IEEE Comput Graph Appl* 6(4):16–26

Glassner AS (1984) Space subdivision for fast ray tracing. *IEEE Comput Graph Appl* 4(10):15–22

Jansen FW (1986) Data structures for ray tracing. In: Kessener LRA, Peters FJ, van Lierop MLP (ed) *Data Structures for Raster Graphics.* Springer Verlag, Netherlands, pp 57–73

Jevans D, Wyvill B (1989) Adaptive voxel subdivision for ray tracing. *Proc Graphics Interface'89*, pp 164–172

Kay TL, Kajiya JT (1986) Ray tracing complex scenes. *Computer Graphics* 14(3):110–116

Kunii TL, Wyvill G (1985) A simple but systematic CSG system *Proc Graphics Interface'85*

Rubin SM, Whitted T (1980) A 3-dimensional representation for fast rendering of complex scenes. *Computer Graphics* 14(3):110–116

Samet H (1989) Implementing ray tracing with octrees and neighbor finding. *Comput and Graph* 13(4):445–460

Sung K (1991) A DDA octree traversal algorithm for ray tracing. *Proc EUROGRAPHICS'91*, pp 73–85

Yagel R, Cohen D, Kaufman A (1992) Discrete ray tracing. *IEEE Comput Graph Appl* 12(5):19–28

Zuiderveld KZ, Koning AHJ, Viergever MA (1992) Acceleration of ray casting using 3D distance transforms. *Visualization in Biomedical Computing*, pp 324–335