An Efficient Parametric Algorithm for Octree Traversal

J. Revelles[†], **C. Ureña**[†], **M. Lastra**[‡]

[†] Dpt. Lenguajes y Sistemas Informáticos, E.T.S. Ingeniería Informática,
University of Granada, Spain,
e-mail: [jrevelle,almagro]@ugr.es,
URL: http://giig.ugr.es

[‡] Dpt. de Informática, E.U.P. Linares,
University of Jaén, Spain,
e-mail: mlastral@ujaen.es

ABSTRACT

An octree is a well known hierarchical spatial structure which is widely used in Computer Graphics algorithms. One of the most frequent operations is the computation of the octree voxels intersected by a straight line. This has a number of applications, such as ray-object intersection tests speed-up and visualisation of hierarchical density models by ray-casting. Several methods have been proposed to achieve this goal, which differ in the order in which intersected voxels are visited. In this paper we introduce a new top-down parametric method. The main difference with previously proposed methods is related to descent movements, that is, the selection of a child sub-voxel from the current one. This selection, as the algorithm, is based on the parameter of the ray and comprises simple comparisons. The resulting algorithm is easy to implement, and efficient when compared to other related top-down and bottom-up algorithms for octrees. Finally, a comparison with Kelvin's method for binary trees is presented.

Keywords: Octree, Binary tree, Ray Tracing, Acceleration Techniques

1 Introduction

By *Octree Traversing* we mean the process of finding the subset of voxels in an octree pierced by a directed line. As stated in the abstract, this has several applications in Computer Graphics. Maybe the best known one is the speeding-up of intersection tests of a line with a set of objects. In this case, the octree is used as a spatial index. Each voxel holds a pointer to the subset of objects that it intersects or contains. Then, the set of objects hit by the ray is necessarily included in the set of objects intersecting voxels pierced by the ray. By traversing the octree it is possible to restrict the ray-object intersection test to this set of objects. This is only an improvement when the traversal process is much faster than the test on all objects. Octrees are also used in solid modelling. In this case, the density of one or several materials is considered constant for each voxel of the octree. Generally, the internal structure of this density distribution can be visualised by using a rendering algorithm which displays it as a transparent material. For these kind of algorithms, the accumulated opacity along a ray is computed by obtaining the traversed voxels. Several algorithms for octree traversing have been proposed [Agate91, Cohen93, Endl94, Fujim86, Garga93, Glass84, Jevan89, Samet89, Sung91]. They can be classified into two groups, according to the order in which pierced voxels are obtained:

Bottom-Up Methods: Traversing starts at the first terminal node intersected by the ray. A process called *neighbour finding* is used to obtain the next terminal node from the current one [Glass84, Samet89, Samet90].

Top-Down Methods: These methods start from the root voxel (that is, from the one covering all others). Then a recursive procedure is used. From the current node, its direct descendants hit by the ray are obtained, and the process is (recursively) repeated for each of them, until terminal voxels are reached [Agate91, Cohen93, Endl94, Janse85, Garga93].

In this paper we introduce a new top-down algorithm. This new algorithm is based on the parametric representation of the ray. This representation allows us to map real values (parameter values) to ray points. The algorithm computes the parameter values at which the ray intersects the three planes that divide each voxel. These values are computed incrementally, by using additions and divisions by two. In essence, this algorithm is related to other parametric top-down algorithms, such as the one introduced by Jansen [Janse85], Arvo [Arvo88], and Agate [Agate91]. Afterward, Sung and Shirley [Sung92] proposed a version on a BSP tree of Jansen's work. Authors of these papers pointed out that the cost of traversing an octree and a BSP tree are similar, (these results have been formally presented by Reinhard [Reinh96]). Both parametric methods (for octrees and BSPs) are based on the same principles as our work. However, here we propose a new method for selecting the first sub-voxel of a voxel, by using comparisons on previously obtained parameter values. This allows us to avoid the *neighbour finding* process, thus saving memory and computing time. With respect to the movements between neighbour voxels with the same parent voxel, we use an algorithm which is also based on the ray parameter, as the algorithm described by Amanatides [Amana87].

Sung [Sung91] presented an octree traversal algorithm which has the same theoretical basis as the 3D-DDA traversal algorithm [Amana87] for uniform spatial subdivision, however, for unbalanced octrees, efficiency decreases. This is due to the fact that Sung's algorithm uses the smallest octree leaf as traversal base unit.

Moreover, we must refer a significant work presented by J. Spackman and P.J. Willis [Spack91]. The authors presented a very efficient algorithm to traverse an octree that only employed operations with integer arithmetic throughout the traversal. That algorithm is outlined into the paper and it is very robust. In essence, both algorithms must be equivalent because they use the same principles for each movement. We think the proposed method in this paper is easy to understand and very didactic for teaching.

This paper is organised as follows: section 2 describes a simplified version of the algorithm for quadtrees. The extension for octrees is detailed in section 3. A comparison in terms of efficiency with other methods, such as the bottom-up method introduced by Samet [Samet89], two extensions of it proposed by Endl [Endl94], and a recursive top-down method [Garga93], is also included. We use the same software system for all the algorithms.

Results show that the new algorithm is more efficient than these other methods except on the algorithm presented by Spackman and Willis. Furthermore, its simplicity makes it straightforward to implement, as can be observed in the source code listings in section 4.

2 The Algorithm for 2D case

We define a ray r as a pair (p, d), where $p = (p_x, p_y)$ is the origin, and $d = (d_x, d_y)$ is the unit length direction vector. For each real value $t \geq 0$ exists a point $(x_r(t), y_r(t))$ on the ray, where x_r and y_r are two scalar functions defined as follows:

$$\begin{aligned} x_r(t) &= p_x + td_x \\ y_r(t) &= p_y + td_y \end{aligned} \tag{1}$$

A node o in a quadtree is the set of points inside an axis-aligned rectangle (whose four edges have equal length). Formally, o is the set of points (x, y) such that $x_0(o) \leq x < x_1(o)$ and $y_0(o) \leq y < y_1(o)$, where x_0, x_1, y_0 and y_1 are four scalar valued functions which define the position of each node o (we also define $s(o)$ as the length of each edge of o)

From the above definitions, we deduce that an intersection between a ray r and a node o occurs if at least one real value t exists such that:

$$\begin{array}{rl} & x_0(o) \leq x_r(t) < x_1(o) \\ \wedge & y_0(o) \leq y_r(t) < y_1(o) \end{array} \tag{2}$$

The algorithm we propose is called a *parametric* algorithm because all computations use values of t such that $(x_r(t), y_r(t))$ is a point on a node boundary. For a node o and ray r, $t_{x0}(o,r), t_{y0}(o,r)$, and $t_{x1}(o,r), t_{y1}(o,r)$ are defined as the ray parameter values for which the ray intersects with the boundary of the node. Formally, these values obey the following equalities:

$$\left.\begin{aligned} x_r(t_{xi}(o,r)) &= x_i(o) \\ y_r(t_{yi}(o,r)) &= y_i(o) \end{aligned}\right| \quad i \in \{0,1\} \tag{3}$$

By using the inverse functions of x_r, and y_r the above parameter values can be defined explicitly

$$\left.\begin{aligned} t_{xi}(o,r) &= (x_i(o) - p_x)\,/\,d_x \\ t_{yi}(o,r) &= (y_i(o) - p_y)\,/\,d_y \end{aligned}\right| \quad i \in \{0,1\} \tag{4}$$

being p_x, and p_y the origin of the half line, and d_x and d_y the unit length direction vector. These four values are computed for each node when traversing the quadtree. Initially they are obtained for the root node, and then an incremental computation is performed for each child node. In order to detail this incremental computation, we first define the following quantities

$$\begin{aligned} \Delta t_x(o,r) = t_{x1}(o,r) - t_{x0}(o,r) \\ \Delta t_y(o,r) = t_{y1}(o,r) - t_{y0}(o,r) \end{aligned} \tag{5}$$

Substituting (4) into (5), we get $\Delta t_x(o,r) = s(o)/d_x$, for each node o. In the case that o is non-terminal, for all children o_i of o we have $\Delta t_x(o_i,r) = \Delta t_x(o,r)/2$. Similar relations hold for Δt_y. Thus, the values Δt_x and Δt_y can be incrementally computed for a child node just by halving the value for

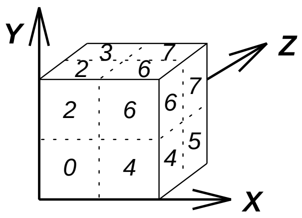

Figure 1: Labeled octree (the hidden node has label 1).

the parent node. If we substitute the following recurrence relations

$$\begin{array}{rcl} x_0(o_i) & = & x_0(o) + s(o_i)\Delta x_i \\ y_0(o_i) & = & y_0(o) + s(o_i)\Delta y_i \end{array} \qquad (6)$$

into (4) we obtain

$$\begin{array}{rcl} t_{x0}(o_i, r) & = & (\ x_0(o) + s(o_i)\Delta x_i - p_x\ )\ /\ d_x \\ & = & (\ x_0(o) - p_x\ )\ /\ d_x\ +\ (s(o_i)/d_x)\Delta x_i \\ & = & t_{x0}(o, r)\ +\ \Delta t_x(o_i, r)\,\Delta x_i \end{array}$$

In this case, Δx_i and Δy_i are components of the following two vectors:

$$\begin{array}{c} \Delta x = \{0, 0, 1, 1\} \\ \Delta y = \{0, 1, 0, 1\} \end{array} \qquad (7)$$

The last result also holds for t_{y0}. Thus, we have shown how these values can be incrementally computed for all child nodes of the current node. The computation of these values for the root node q is carried out by using (4).

Knowing the definitions of a node and a ray, we easily deduce that an intersection between a ray r and a node o occurs if at least one real value t exists such that:

$$\begin{array}{rcl} & x_0(o)\ \leq\ x_r(t)\ <\ x_1(o) \\ \wedge & y_0(o)\ \leq\ y_r(t)\ <\ y_1(o) \end{array} \qquad (8)$$

Where a intersection occurs, an interval o values of t satisfies the above inequalities. This interval is closed at the left and open on the right for half lines with a positive or zero valued direction vector.

By taking all these results into account, we can now rewrite the condition 8 by using the parameters of the ray. For instance, taking condition $x_r(t) < x_1(o)$ from (8), we can substitute $x_1(o)$ by $x_r(t_{x1}(o, r))$, then, as x_r is an increasing function, we obtain $t < t_{x1}(o, r)$. By using the other inequalities in (8) the same way, we can state that an intersection between o and r occurs if and only if exists $t \geq 0$ such that

$$\begin{array}{rl} & t_{x0}(o, r)\ \leq\ t\ <\ t_{x1}(o, r) \\ \wedge & t_{y0}(o, r)\ \leq\ t\ <\ t_{y1}(o, r) \end{array} \qquad (9)$$

This equation can be further simplified by defining t_{min} and t_{max} for a node o and a ray r as

$$\begin{array}{rcl} t_{min}(o, r) & = & \max(t_{x0}(o, r), t_{y0}(o, r)) \\ t_{max}(o, r) & = & \min(t_{x1}(o, r), t_{y1}(o, r)) \end{array}$$

If a t exists obeying (9), then $t_{min} \leq t < t_{max}$. The inverse implication also holds, thus equation (9) is equivalent to

$$t_{min}(o, r)\ <\ t_{max}(o, r) \qquad (10)$$

When above condition is true, all values of t in the interval $[t_{min}, t_{max})$ are mapped to points $(x_r(t), y_r(t))$ which belong to the node. If the condition is false, no intersection occurs. It is now possible to outline the proposed parametric algorithm used to traverse a quadtree. First, we check condition (10) for the root node. If this condition is not satisfied then the ray does not intersect with the octree. But where it is, the four parameters (t_{x0}, t_{y0}) and (t_{x1}, t_{y1}) need to be computed for the root node by using (4). The main recursive procedure is subsequently executed accepting a node as input parameter, and its corresponding four parameter values. In cases where the node is terminal, this node is added to the resulting pierced nodes list. If it is non-terminal, those child nodes which are pierced by the ray are checked using (10) for each of them. A recursive call to the procedure is carried out for each of them.

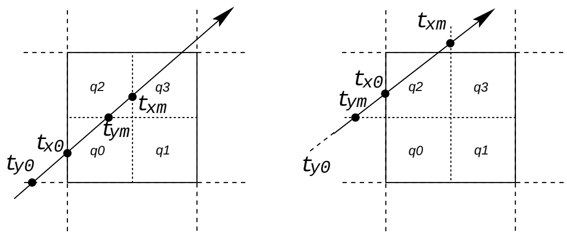

Figure 2: Sub-nodes crossed when $t_{x0} > t_{y0}$ (2D case).

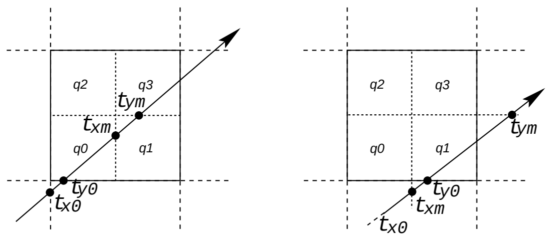

Figure 3: Sub-nodes crossed when $t_{y0} > t_{x0}$ (2D case).

Note that, for any non-terminal node o, $t_{x0}(o_0, r) = t_{x0}(o, r)$, and $t_{x1}(o_0) = t_{x0}(o_1)$. Other child nodes behave in a similar way. Thus, computation of the entry and exit parameters for each child node of o is redundant, because some of them can be taken directly from the parent node, and the others are shared by several child nodes. In fact, there are just six different parameters for a child node. These are the four parameters of the parent plus the following two values:

$$t_{xm}(o,r) = (t_{x0}(o,r) + t_{x1}(o,r))/2$$
$$t_{ym}(o,r) = (t_{y0}(o,r) + t_{y1}(o,r))/2 \quad (11)$$

Here $t_{xm}(o,r)$ is the value of the ray parameter for which the ray crosses the horizontal line which divides the node in two equal halves. A similar equation holds for t_{ym}. This kind of coherence can be used to improve the algorithm further by using a sequential algorithm. The selection of pierced sub-nodes of a node is carried out in two steps:

1. Select the first sub-node hit by the ray
2. For each pierced node, select the next one, until the current parent node is exited.

Assuming a node q is crossed by a ray r, and the ray direction components are non negative, then the next implications hold:

if $t_{min}(q,r) < t_{max}(q,r)$ **then**
  if $t_{ym}(q) < t_{xm}(q)$ the ray crosses q_2, and
    if $t_{x0}(q) < t_{ym}(q)$ the ray crosses q_0.
    if $t_{xm}(q) < t_{y1}(q)$ the ray crosses q_3.
  else if $t_{xm}(q) < t_{ym}(q)$ the ray crosses q_1, and
    if $t_{y0}(q) < t_{xm}(q)$ the ray crosses q_0.
    if $t_{ym}(q) < t_{x1}(q)$ the ray crosses q_3.

These assumptions may be easily proved (see figures 2 and 3). When t_{xm} and t_{ym} are computed, only three comparisons are necessary to determine the sub-nodes which are crossed.

3 Extending the algorithm to octrees

To extend the above algorithm to traverse an octree, we have into account the two steps previously detailed, including the third dimension when necessary. With respect to the first step, we introduce a new method which allows the use of a nine parameter set to compute which voxel is pierced first. This computation avoids *neighbour finding* to obtain the first voxel, as it is the case for the algorithm introduced by Glassner[Glass84] and improved in other papers[Arvo88, Samet89, Endl94]. This yields a simpler method. The following sections describe these two steps in detail. The second step is carried out by using a parametric DDA algorithm, as described by Amanatides[Amana87], but restricted to eight voxels (which can be viewed as a $2 \times 2 \times 2$ uniform grid space). The parameter values are computed by successive additions[Amana87]. In our case, the values are incrementally computed from those of the parent by using three additions and three shifts.

A recursive top-down parametric algorithm called HERO is described in the paper by Agate et al.[Agate91]. The main difference with the proposed algorithm is in the computation of the sequence of visited sub-voxels from the current voxel (see section 3.2).

3.1 Obtaining the First Crossed Node

To find the first sub-voxel at which the ray enters the current voxel, first we obtain the entry face of the current voxel. This step is made by computing $\max(t_{x0}(o), t_{y0}(o), t_{z0}(o))$. In table 2 we show the entry plane selected for each case. Once the entry plane has been determined, four sub-nodes are candidates. To determine the first sub-node crossed, we examine $t_{xm}(o)$, $t_{ym}(o)$, and $t_{zm}(o)$. In table 1 the necessary comparisons are shown. The results of evaluating this condition (a bit) is copied to one of the bits which form the index of the first sub-node crossed. When a condition is true, the bit associated is set to 1, otherwise it is set to 0. For each entry plane there are two conditions, so we have four possibilities, one for each sub-node touched by that plane. In this way, any node could be selected with the exception of node 7, because the ray direction vector components are assumed to be positive. The whole process can be implemented using the OR operator to combine the necessary bits. Several illustrations are shown below to explain in greater detail the first step for a quad-tree case. If the comparison of t_{x0} and t_{y0} results $t_{x0} > t_{y0}$ then the first edge crossed is *edge 1*. For this case, the first sub-node crossed can be q_2 or q_0 (Figure 2). However, when the above comparison results $t_{y0} > t_{x0}$, then the first edge crossed is *edge 2*. In this case, the sub-node that can be crossed is either $q0$ or $q1$ (Figure 3).

Entry Plane	Conditions to examine	Bit affected
XY	$t_{xm}(o) < t_{z0}(o)$	0
	$t_{ym}(o) < t_{z0}(o)$	1
XZ	$t_{xm}(o) < t_{y0}(o)$	0
	$t_{zm}(o) < t_{y0}(o)$	2
YZ	$t_{ym}(o) < t_{x0}(o)$	1
	$t_{zm}(o) < t_{x0}(o)$	2

Table 1: Comparisons to obtain the first node intersected.

Maximum	Entry plane
t_{x0}	YZ
t_{y0}	XZ
t_{z0}	XY

Table 2: First plane intersected.

3.2 Obtaining the Next Node

Once the first sub-node of a node has been found, the sequence of the rest of the traversed sub-nodes must be obtained[Amana87]. Since the location of the sub-nodes is fixed, we can design an automaton whose states correspond to the voxels and whose transitions are associated to the movements between neighbouring sub-nodes visited sequentially by the ray. In figure 5 all possible transitions are shown. For a given ray, the sequence of traversed nodes or states visited will be established so that the followed path is one of the possible ones in the before mentioned figure. In order to obtain the exact path followed by the ray it will be necessary to compute the face where the ray leaves each of the visited nodes, which will make it possible to obtain the next node to be visited. Let o_j be the current visited node. The ray can leave this node through the following faces: YZ, XZ or XY and this is calculated by getting the minimum of $t_{x1}(o_j), t_{y1}(o_j), t_{z1}(o_j)$. In figure 4 a 2D example of computation of the exit edge for node o_0 is shown. You can see how the ordering of t_{x1} and t_{y1} determine the exit edge.

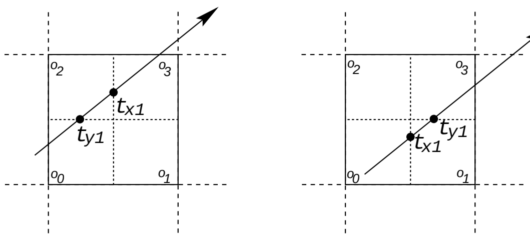

Figure 4: Exit Edge.

Table 3 shows the next visited node given the current node and the exit plane. The entries labelled with "End" mean that the ray leaves node o. For example, if the first node is

Current sub-node (state)	Exit plane YZ	Exit plane XZ	Exit plane XY
0	4	2	1
1	5	3	End
2	6	End	3
3	7	End	End
4	End	6	5
5	End	7	End
6	End	End	7
7	End	End	End

Table 3: State transitions.

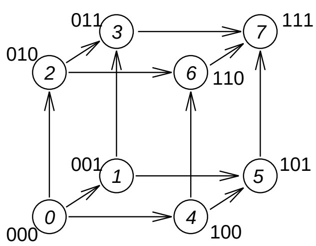

Figure 5: Sub-nodes that can be reached from an initial node.

0, the nodes that can be visited are 4, 2 and 1, depending on the exit plane. From node number 6, node number 7 is the only one that can be reached because movements along the X or Y axes would mean that the ray leaves the parent node. The longest path includes four nodes and an example is the one including the sub-nodes $(0, 1, 3, 7)$ which involve three transitions. In each one we choose the minimum of those three values. Therefore two comparisons are made in each transition which, when added to the four operations needed to obtain the first sub-node, results in ten comparisons in the most unfavourable case.

In the HERO algorithm it is necessary to sort a list of potentially visited sub-voxels and then explicitly check each of them for intersection. However our algorithm is simpler because this is not necessary, as visited sub-voxels are obtained sequentially. Some computations are unnecessary because non-visited sub-voxels are not considered at all. Thus this yields a faster and simpler algorithm.

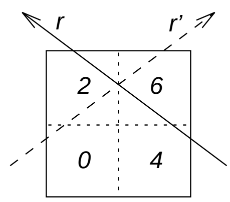

Figure 6: Reflection transformation for the generalisation.

3.3 Generalising for Rays Parallel to One Main Axis

In the case that d_e is zero for any e, the direction vector d of the ray is parallel to an octree face or even to one of the main axes. In this case, some of the entry and exit parameters are not defined, because their definition involves a division by d_e. In our algorithm, we have tried to keep the program simple, avoiding any special case handling. So we have chosen to allow the parameters to take infinite values. This way the intersections between a ray and a plane may occur at an infinite distance from the origin of the ray. Note that in this case the interval of parameter values at which the ray is in a voxel may be equal to the whole real line. Any parameter t_{ei} with $e \in \{x, y, z\}$ and $i \in \{0, \ldots, 7\}$ can take real values and also the values $-\infty$ and $+\infty$. All operations involving parameter values should take this into account. We set $x/0 = +\infty$ for all $x > 0$ and $x/0 = -\infty$ for all $x < 0$. We also have $-\infty < x < +\infty$ for all real values x. With these definitions it is possible to obtain the initial entry and exit parameters for the root voxel, and also to compare parameter values. The computation of t_{em} is also affected by infinite values. Assuming that $d_x = 0$, ray r enters any voxel o if and only if $t_{x0}(o, r) = -\infty$ and $t_{x1}(o, r) = +\infty$. In this case, the value t_{xm} cannot be computed because $t_{x0} + t_{x1}$ is undefined. However, in these conditions it can be shown that:

$$t_{xm}(o, r) = \begin{cases} +\infty & \text{when } p_x < \dfrac{x_0(o) + x_1(o)}{2} \\ -\infty & \text{otherwise} \end{cases} \tag{12}$$

Similar relations can be derived for the other axes, thus we have a method to obtain t_{em} in these cases. This ensures that the comparisons yield the correct parameter intervals where the ray intersects the voxels (including the case when those intervals are equal to the whole real line), and the visited voxels are also correct. Some hardware architectures include infinite values in the set of allowed real values, and are produced after a division by zero has taken place.

3.4 Generalising for Rays with Negative Directions

As has been pointed out at the beginning, this method only works properly if the direction vector of the ray has no negative components. To solve this problem, eight different versions of the method, one for each sub-voxel, could be written and implemented. However, this solution has been avoided because of the redundant code that would be necessary to produce. An example where the ray direction is only negative for the X axis is now considered. In this case the ray can be reflected with respect to the middle plane of the octree to turn the negative direction component positive. Starting from the reflected ray, the visited nodes must be relabelled and this way the previous scheme is valid. One ray with origin p and direction d is transformed to other ray p' and d' by using the following relations:

$$\begin{array}{rcl} d'_x & = & -d_x \\ p'_x & = & s(q) - p_x \end{array} \tag{13}$$

The only difference will be in the labelling of the sub-nodes of a node. In figure 6 it can be observed when the original ray r traverses sub-nodes $4, 6, 2$ of the octree and that the transformed ray traverses sub-nodes $0, 2, 6$ which would produce wrong movements in the octree. To avoid this, a function f is needed to transform labels, so that when node i is computed as the next node, in fact $f(i)$ is accessed. In the example, where $d_x < 0$, the image by f of the sub-nodes would be: $(4, 5, 6, 7, 0, 1, 2, 3)$. Function f changes the value of the most significant bit of the binary representation of a node label. In general (any component of the direction can be negative) it is necessary to change label bits when the direction component corresponding to that bit is negative. This is done by using a XOR operation with the original label and an integer value lower than 8 (3 bits). Formally, $f(i) = i \oplus a$, where $a = 4s_x + 2s_y + s_z$, and where s_e is 1 if $d_e < 0$, and 0 in other case (for each axis e).

4 Parametric Algorithm Pseudocode

A pseudocode of the algorithm will now be shown, where recursion is used for ease of understanding, although this may be avoided by implementing an iterative algorithm with a stack. We consider an octree as a data structure that contains a pointer to the root node whose type is *node*. It also contains its dimensions called *(xmin, xmax, ymin, ymax, zmin, zmax)*, and its size called *sizei*, where $i \in \{X, Y, Z\}$. Each non-terminal node has eight pointers pointing to its eight children. Function `first_node` implements tables 1 and 2, such as described in section 3.1. Function `new_node` implements table 3. This function accepts three `float` values, and three `int` values as parameters. The function returns the i-th integer where the i-th float value is the minimum of the three float values. Function `proc_terminal` is used to perform any computation required when a terminal node is reached by the ray. Its parameter is a pointer to that node.

```
unsigned char a ;

void ray_parameter ( octree *oct, ray r )
{  a= 0 ;
   if (r.dx<0.0)
   {  r.ox = oct->sizeX-r.ox ;
      r.dx = -r.dx ;
      a |= 4 ;
   }
   if (r.dy<0.0)
   {  r.oy = oct->sizeY-r.oy ;
      r.dy = -r.dy ;
      a |= 2 ;
   }
   if (r.dz<0.0)
   {  r.oz = oct->sizeZ-r.oz ;
      r.dz = -r.dz ;
      a |= 1 ;
   }
   tx0 = (oct->xmin - r.ox)/r.dx ;
   tx1 = (oct->xmax - r.ox)/r.dx ;
   ty0 = (oct->ymin - r.oy)/r.dy ;
   ty1 = (oct->ymax - r.oy)/r.dy ;
   tz0 = (oct->zmin - r.oz)/r.dz ;
   tz1 = (oct->zmax - r.oz)/r.dz ;

   if (Max (tx0,ty0,tz0) < Min (tx1,ty1,tz1))
       proc_subtree (tx0,ty0,tz0, tx1,ty1,tz1, oct->root) ;
}

void proc_subtree (  real tx0, real ty0, real tz0,
                     real tx1, real ty1, real tz1,
                     node *n )
{  real txm, tym, tzm ;
   int  currNode ;

   if (tx1<0.0 || ty1<0.0 || tz1<0.0)
      return ;

   if (n->type == TERMINAL)
   {  proc_terminal (n) ;
      return ;
   }

   txm = 0.5*(tx0+tx1) ;
   tym = 0.5*(ty0+ty1) ;
   tzm = 0.5*(tz0+tz1) ;

   currNode= first_node ( tx0, ty0, tz0, txm, tym, tzm ) ;
   do
   { switch (currNode)
     { 0 : proc_subtree ( tx0, ty0, tz0, txm, tym, tzm, n->son[a] ) ;
           currNode = new_node ( txm, 4, tym, 2, tzm, 1 ) ;
           break ;
       1 : proc_subtree ( tx0, ty0, tzm, txm, tym, tz1, n->son[1^a] ) ;
           currNode = new_node ( txm, 5, tym, 3, tz1, 8 ) ;
           break ;
       2 : proc_subtree ( tx0, tym, tz0, txm, ty1, tzm, n->son[2^a] ) ;
           currNode = new_node ( txm, 6, ty1, 8, tzm, 3 ) ;
           break ;
       3 : proc_subtree ( tx0, tym, tzm, txm, ty1, tz1, n->son[3^a] ) ;
           currNode = new_node ( txm, 7, ty1, 8, tz1, 8 ) ;
           break ;
       4 : proc_subtree ( txm, ty0, tz0, tx1, tym, tzm, n->son[4^a] ) ;
           currNode = new_node ( tx1, 8, tym, 6, tzm, 5 ) ;
           break ;
       5 : proc_subtree ( txm, ty0, tzm, tx1, tym, tz1, n->son[5^a] ) ;
           currNode = new_node ( tx1, 8, tym, 7, tz1, 8 ) ;
           break ;
       6 : proc_subtree ( txm, tym, tz0, tx1, ty1, tzm, n->son[6^a] ) ;
           currNode = new_node ( tx1, 8, ty1, 8, tzm, 7 ) ;
           break ;
       7 : proc_subtree ( txm, tym, tzm, tx1, ty1, tz1, n->son[7^a] ) ;
           currNode = 8 ;
           break ;
     }
   } while (currNode<8) ;
  }
```

5 Results

The proposed method was implemented in a rendering system written in C++, which already included several acceleration techniques based on octrees, and a wide range of algorithms for octree traversal. Time comparisons were made for these algorithms.

The comparisons were carried out on a Silicon Graphics *Indigo 2* with a MIPS R4000 processor and 64MB of RAM. Three different scenes were used: the first is a sphere flake with 1890 objects (see figure 10), the second is a structure, composed of spheres and cylinders, with 4320 objects (see figure 11), and the third a *patio* with 4350 objects (see figure 12). Space subdivision has been applied to these scenes using octrees with maximum depth levels 5, 6, 7 and 8. The traversal algorithms

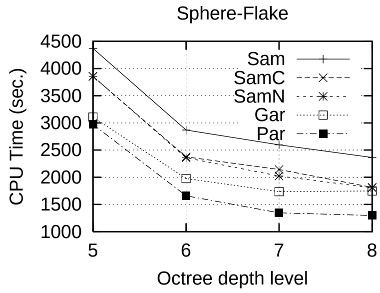

Figure 7: Timing graph for Sphere-Flake.

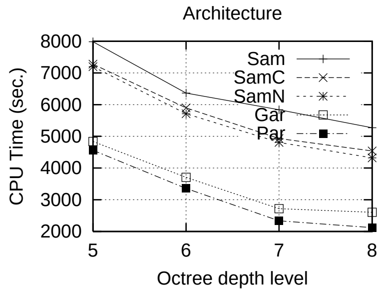

Figure 8: Timing graph for Architecture.

used to compare the performances with our method were: *Samet[Samet89] (Sam)*, *SametCorner[Endl94] (SamC)* and *SametNet[Endl94] (SamN)* as bottom-up algorithms, and *Gargantini[Garga93] (Gar)* as top-down algorithm. The time comparisons are plotted in figures 7, 8, and 9. *SametNet* method requires much memory because pointers to neighbour voxels must be kept within each voxel (they are required for the neighbour finding process). So, a higher rendering time is obtained after increasing the octree depth level due to swap time. In summary, we can see that the Gargantini method presents better results than the other bottom-up methods shown. In Gargantini's paper, we may see a comparison with Samet's method. The improvement of the first one is shown in terms of execution time and a run-time analysis.

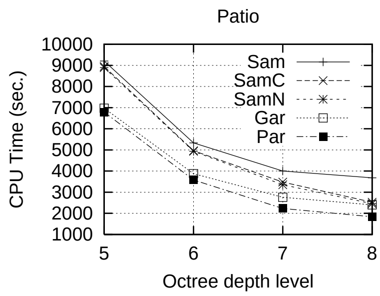

Figure 9: Timing graph for Patio.

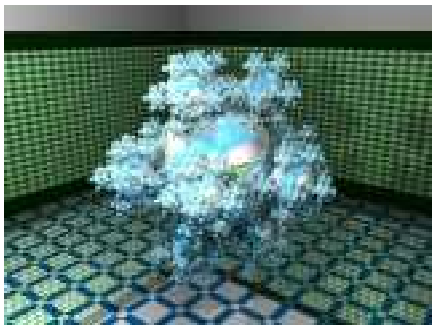

Figure 10: Sphere-Flake.

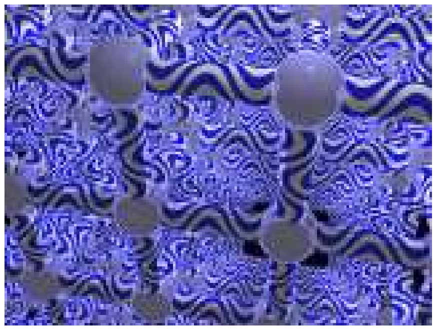

Figure 11: Architecture.

5.1 Comparison with the Binary Tree Structure

To study the efficiency of our method [Sung92], a comparison with the Kelvin Sung method has been done. We have obtained two images from the same scene. The rendering system was the same in both cases. One of these images was rendered using an octree structure, and the other using a binary tree. The scene is shown in Figure 14. This scene has one object per leaf node. The number of spheres is 2^{3n}. We ran three tests setting n to 3, 4, and 5. The total number of spheres was 512, 4096, and 32768. The cpu time employed when using the octrees was 10.4%, 12.7% and 6.8% lower than the time for the binary trees.

These results shows that octrees are more efficient than bi-

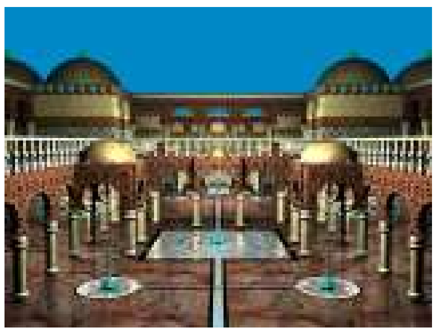

Figure 12: Patio.

nary trees when the tree is full or nearly full, due to the number of necessary descents to reach one terminal node. In an octree, the number of recursive calls to the main function is lower, meanwhile the rest of computations are basically equivalent (as it was established by Reinhard et al. [Reinh96]).

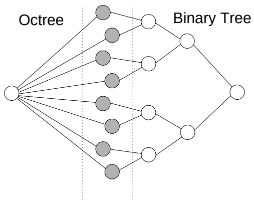

Figure 13: Bintree and Octree equivalent structures (depth level for octree is 1, and 3 for bintree).

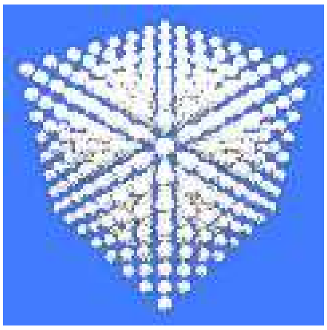

Figure 14: 3D array of spheres.

6 Conclusions

The recursive top-down algorithm which has been presented improves the performance of existing ones, as the time graphs show. It minimizes the number of operations because they are carried out in a top-down recursive style, and using at each stage the results of the previous stage. Besides efficiency, it is also desirable for any method to be easy to read, comprehend and implement into a rendering system. This method is very easy to understand and to incorporate into such systems because of its simplicity, when compared to other referenced algorithms.

7 Acknowledgements

This work has been supported by a grant coded as TIC95-0614-C03-02, and TIC98-0973-C03-01 from the Committee for Science and Technology of the Spanish Government (CICYT).

REFERENCES

[Agate91] M. Agate, R.L. Grimsdale, P.F. Lister. *The HERO Algorithm for Ray Tracing Octrees. Advances in Computer Graphics Hardware IV* R.L. Grimsdale, W. Strasser (eds) Springer-Verlag, New York, 1991.

[Amana87] J. Amanatides, A. Woo. *A Fast Voxel Traversal Algorithm for Ray Tracing. Eurographics'87. Proceedings of the European Computer Graphics Conference and Exhibition, August 1987*, pp 3-10.

[Arvo88] J. Arvo. *Linear-time voxel walking for octrees. Ray Tracing News 1(12), March 1988.* Available under anonymous ftp from drizzle.cs.uoregon.edu

[Cohen93] D. Cohen, A. Shaked. *Photo-Realistic Imaging of Digital Terrains. Computer Graphics Forum* Vol. 12(3), pp 363-376, 1993.

[Coqui85] S. Coquillart. *An improvement of the ray-tracing algorithm. Eurographics'85: Proceedings of the European Computer Graphics Conference and Exhibition, Ed C. E. Vandoni*, pp. 77-88, 1985.

[Endl94] R. Endl, M. Sommer. *Classification of Ray-Generators in Uniform Subdivisions an Octrees for Ray Tracing. Computer Graphics Forum* Vol. 13(1), 1994.

[Fujim86] A. Fujimoto, K. Iwata. *ARTS: Accelerated Ray Tracing System. IEEE Computer Graphics & Applications* Vol. 6(4), pp. 16-26. 1986.

[Garga93] I. Gargantini, H.H. Atkinson. *Ray Tracing an Octree: Numerical Evaluation of the First Intersection. Computer Graphics Forum* Vol. 12(4), 1993.

[Glass84] A. S. Glassner. *Space Subdivision for Fast Ray Tracing. IEEE Computer Graphics & Applications* Vol. 4(10), pp. 15-22 October 1984.

[Janse85] F.W. Jansen. *Data Structures for Ray-tracing. Data Structures for Raster Graphics* L. Kessner, F. Peters, M. van Lierop (eds). Springer-Verlag, 1985, pp. 57-73.

[Jevan89] D. Jevans, B. Wyvill. *Adaptive Voxel Subdivision for Ray Tracing. Graphics Interface'89*, pp. 164-172. June 1989.

[Reinh96] E. Reinhard, Arjan J.F. Kok, F.W. Jansen. *Cost Prediction in Ray Tracing. Rendering Techniques 96*, Springer-Verlag, 1996, pp. 42-51.

[Samet89] H. Samet. *Implementing Ray Tracing with Octrees and Neighbor Finding". Computer & Graphics* Vol. 13(4), pp. 445-460, 1989.

[Samet90] H. Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS.* Addison-Wesley 1990.

[Spack91] J. Spackman, P.J. Willis. *The Smart Navigation of a Ray Through an Oct-Tree: Computer & Graphics* Vol. 15(2), pp.185-194, 1991.

[Sung91] K. Sung. *A DDA Octree Traversal Algorithm for Ray Tracing. Eurographics'91. Proceedings of the European Computer Graphics Conference and Exhibition*, F.H Post and W. Barth (eds), pp. 73-85, North Holland, 1991.

[Sung92] K. Sung, P. Shirley. *Ray Tracing with the BSP tree. Graphics Gems III.* pp. 271-274, 1992.