Discrete Ray-Tracing of Huge Voxel Spaces

Nilo Stolte* and René Caubet

Institut de Recherche en Informatique de Toulouse
118, Route de Narbonne
31062 – Toulouse – France
stolte@irit.fr

Abstract

The quality of images produced by Discrete Ray-Tracing voxel spaces is highly dependent on 3d grid resolution. The huge amount of memory needed to store such grids often discards discrete Ray-Tracing as a practical visualization algorithm. The use of an octree can drastically change this when most of space is empty, as such is the case in most scenes.

Although the memory problem can be bypassed using the octree, the performance problem still remains. A known fact is that the performance of discrete traversal is optimal for quite low resolutions. This problem can be easily solved by dividing the task in two steps, working in two low resolutions instead of just one high resolution, thus taking advantage of optimal times in both steps. This is possible thanks to the octree property of representing the same scene in several different resolutions. This article presents a two step Discrete Ray-Tracing method using an octree and shows, by comparing it with the single step version, that a substantial gain in performance is achieved.

Keywords: Ray-Tracing, Discrete Ray-Tracing, Raster Ray-Tracing, Octree, Three-dimensional DDA, Voxel.

1. Introduction

Even though many efforts have been made to accelerate Ray-Tracing in many different ways it still has the reputation of being a high time consuming process, despite the fact of being widely accepted as a very powerful and simple tool for rendering realistic scenes.

This reputation is justified by the fact that 90% of the time is consumed in intersection calculations between rays and objects [25], worsening for complex scenes since for each valid intersection new rays are created that can potentially intersect other objects and create even more rays. Computational time grows exponentially with the complexity of the scene.

This is the reason why many acceleration methods always try to reduce the number of intersections between rays and objects. One way is using bounding volumes [19], grouping objects inside boxes or spheres, and ignoring the bounding volumes not crossed over by rays, thus eliminating (hopefully) most of the objects. A better approach for this method is organizing them hierarchically, since more objects can be eliminated with less calculation. Although a serious disadvantage is that for the best efficiency of these hierarchies they should not be created automatically. However there are methods for calculating tighter bounding volumes automatically [18], but properly grouping bounding volumes into hierarchies remains a difficult task and intelligent methods can be very time consuming.

Another way is decomposing the space into subspaces, maintaining a list of objects present in each subspace, and making rays to traverse only the subspaces where each ray passes through [7, 9, 4, 20,

*grants CNPq (Conselho Nacional de Desenvolvimento Científico e Tecnológico - Brasil)

23, 21, 26, 8]. This is the same principle as the previous method, but non-crossed subspaces and their included objects are discarded without any computation. This advantage diminishes when most of the objects are concentrated in just few subspaces for regular subdivisions. The octree [9] contours this problem, and has the the dual advantage of being simple and bounding objects into hierarchies automatically. Another asset of space subdivision is that objects are detected in the exact order they occur along a ray, thus eliminating regions not yet reached by the ray automatically.

Many other original methods have been suggested like Beam Tracing [12] and Ray-Tracing with Cones [2], where several rays can be treated at once inside a beam. Another good example is ray under-sampling [1], where less rays are shot and the illumination is hopefully most of the time interpolated. The disadvantage of these methods is that their performance are very often dependent on the kind of objects modeled and/or the complexity of the scene.

Methods that subdivide space require a preprocessing phase to rasterize the scene, called voxelization [15, 16, 17, 11, 6, 22], where objects' contours are assigned to voxels into a 3D grid or octants into an octree. In this process only parts of the geometry that pierce the voxels or octants are assigned. An advantage of the octree in this context is that certain subdivision methods require octree-like recursive space partitioning. One particularly interesting is proposed in [14] and [5], for implicit functions.

An efficient acceleration method based on space subdivision was proposed in [26]. It is called Discrete or Raster Ray Tracing, because it works entirely in 3D voxel space. The big advantage of this method is that no intersection calculation between rays and objects is needed, allowing a very fast Ray Tracing. The big disadvantage is that a high resolution 3D voxel grid is needed to achieve good quality images. This grid is then limited to the machine memory size. The grid resolution in [26] is 256^3 to a 80 Mb machine. The maximum resolution was 320^3 to a 128 Mb machine. This easily denotes the difficulty to run this method in a normal workstation. It is also evident that these resolutions are not enough for getting good quality images.

Considering this problem, we suggest in this paper an implementation of the Discrete Ray Tracing using an octree. Using the octree suggested here the utilization of the method will not be dependent on the machine memory, but on the number of occupied voxels. In this way simple scenes can be generated in machines limited in memory. On the other hand, it allows to use higher resolutions necessary to get good quality images, which is impossible currently by using 3D grids directly.

Nevertheless as stated in [26] most of method's processing cost falls in the discrete traversal algorithm, namely the three-dimensional DDA. In [26] a 26 connected DDA was created to reduce this time. This solution is not sufficient in high resolution 3D grids. It is a known fact that its efficiency is optimal between the fifth and the sixth levels of the octree [7, 23]). We suggest a two step process to benefit for its optimal performance in both steps. This allows us to visualize quite huge 3D grids in short time. On the other hand, the algorithm precision is very important for the correctness of this approach. The reason is that the transition between the two levels is calculated in floating point arithmetics. Our DDA is similar to the one shown in [7] but the mapping of floating point values in fixed point using integer variables, offers, at the same time, efficiency, accuracy and multiple precision flexibility.

Two step methods have been used before [24, 13], by using a regular 3D grid in the first step. We use only one octree for both steps. Therefore, our approach is more flexible, since the first and second step resolution can be dynamically adjusted without modifying the data neither moving it; more compact, since the empty space is not represented; and more efficient to skip empty volumes, since our three-dimensional DDA procedure loops while inside them, not needing to consult cells. Our skip procedure remembers Sung's *"SimpleSkipParent"* procedure [23] but in our case it's more efficient and two steps approach eliminates the need of *"ComplexSkipParent"* procedure, the main problem in Sung's algorithm.

We compare our two steps approach with the same program implemented with a single step. Results show that we achieved a five fold reduction in time.

2. Octree Traversal

The method's flow control is done by the octree traversal algorithm presented in Fig. 1. The octree uses spatial enumeration in fixed sized blocks, which we call cells. A cell, which represents an octant, is a simple array of eight elements, each one identifying a sub-octant and containing a pointer to a descendent cell or a null pointer if sub_octant is empty. The algorithm describes what we call the *"first step"* traversal since it considers only octree upper levels.

```
initialize 3DDDA; /* Algorithm Fig.4 */
/* Initialize XYZ and mask as follows (each square is a bit) */

XYZ  ← | 0 | x | x | ... | x | 0 | y | y | ... | y | 0 | z | z | ... | z |
mask ← | 0 | 1 | 0 | ... | 0 | 0 | 1 | 0 | ... | 0 | 0 | 1 | 0 | ... | 0 |

cell=octree root cell address;
push(cell);
i_c=0; /* Index of last changed coordinate */
end_condition=mask<<1;
XYZ_ant=XYZ;
/* Repeat while inside octree*/
while ((XYZ and end_condition)==0)
    begin                                                        1
        XYZ_ant=XYZ_ant xor XYZ;
        mask=mask<<1;
        /* Ascend octree until common father arrives*/
        while (XYZ_ant and mask)
            begin                                                2
                pop;                   /* Ascend one */
                mask=mask<<1;          /* octree level. */
            end
        mask=mask>>1;
        pop(cell);
        separate X,Y,Z from variable XYZ;
        bool=TRUE;
        while ( bool ) /* Descend octree */
            begin                                                3
                push(cell); /* push cell address */
                i=0;
                if ((Z and mask)≠0) i=i+4;   /*Calculate octant    */
                if ((Y and mask)≠0) i=i+2;   /*index.              */
                if ((X and mask)≠0) i=i+1;
                if ((mask and 1)==0)         /*Not Last level?     */
                  begin
                    if (cell[i]≠0)           /*Octant not empty?   */
                      begin
                        mask=mask>>1;        /* Descend one        */
                        cell=cell[i];        /* octree level.      */
                      end
                    else bool=FALSE;         /*Empty octant!       */
                  end
                else bool=FALSE;             /*Last level!         */
            end
        if (cell[i]≠0)                       /* Cell not empty?    */
          begin
            initialize second step ;         /* Algorithm Fig. 6.  */
            execute second step;             /* Similar to this    */
          end                                /* algorithm.         */
        execute 3DDDA;                       /* Algorithm Fig. 5.  */
    end
```

Figure 1: Octree traversal

Push and *pop* operators of Fig. 1 denote an external stack to keep the addresses of the parent cells, as done by many authors [7, 23, 8].

The first *"while"* identifies the main loop (number 1 in Fig.1) where a ray is traced into octree upper levels (first step). The second *"while"* loop (number 2 in Fig.1) ascends the octree until a common parent of the current and the precedent voxel is found. The third *"while"* loop (number 3 in Fig.1) descends the octree until an empty octant or the current voxel (a leaf in octree upper step) is found. In this last case the ray is traced into octree lower levels (second step). If the ray does not reach a non-empty voxels in the second step, it continues to be traced in the first step. A ray is traced by executing a three-dimensional DDA (algorithm Fig. 5). Second Step initialization algorithm is given in Fig. 6. Second step execution is similar to the algorithm in Fig. 1.

The algorithm's most important variable is *mask*. It has straight relationship with the variable *XYZ*, which contains the three coordinates of a first step voxel into the same register variable (diagram in Fig. 1). The width of these coordinates in bits is the number of levels of the octree in the first step. Initially the higher order bit corresponding to each coordinate in variable *mask* is set and all the other reset (diagram in Fig. 1). These set bits identify the current level of the octree and are exploited not only to filter the pertinent bits of *X*, *Y*, and *Z*, but also in a series of control tasks.

At the left side of the most significant bit corresponding to each coordinate there is a carry bit which is initially zero (diagram in Fig. 1). A carry bit will be set if any of the coordinates is out of the octree. In fact this is used as a final condition test (first *"while"* condition in algorithm).

A special strategy to ascend the octree, taking advantage of *mask* variable, was conceived in order to optimize it (loop number 2). Only one coordinate changes in each interaction, and the difference between this coordinate value and its precedent is always one. Due to the addition carry cascade effect, an exclusive or (*"xor"*) between *XYZ_ant* (*XYZ*'s precedent value) and *XYZ* will originate a uninterrupted sequence of ones identifying all modified bits. Since each coordinate bit corresponds to one octree level, all set bits in this sequence also indicate an octant boundary transition or, in other words, each one indicates that an ascension to the upper level must be done to find the neighbor octant.

The procedure to descend the octree (loop number 3) is more complex. For each cell octree level an octant index is calculated by grouping contiguously the three bits of *X*, *Y*, and *Z* to form an octal digit [8]. While the indexed octant is not empty (not a null pointer) the algorithm descends one octree level until the last level is reached.

```
list = NIL;
do
  begin if (tMaxX < tMaxY)
          begin
            if (tMaxX < tMaxZ)
              begin X = X + stepX;
                    if (X == justOutX)
                      return(NIL); /* outside grid */
                    tMaxX = tMaxX + tDeltaX;
              end
            else
              begin Z = Z + stepZ;
                    if (Z == justOutZ) return(NIL);
                    tMaxZ = tMaxZ + tDeltaZ;
              end
          end
        else
          begin
            if (tMaxY < tMaxZ)
              begin Y = Y + stepY;
                    if (Y == justOutY) return(NIL);
                    tMaxY = tMaxY + tDeltaY;
              end
            else
              begin Z = Z + stepZ;
                    if (Z == justOutZ) return(NIL);
                    tMaxZ = tMaxZ + tDeltaZ;
              end
          end
        list = ObjectList[X][Y][Z];
  end while(list == NIL);
return(list);
```

Figure 2: Three-dimensional DDA of Amanatides and Woo

3. Three-dimensional DDA

3.1. Introduction

Regular Space subdivision allows to use incremental techniques to identify ray-traversed voxels in their exact order of appearance in a very efficient way due to simplicity and fast integer operations. These techniques allow to implement Ray Tracing in a completely discrete way. For high resolution 3D grids, this solution is not sufficient. A better performance strategy becomes necessary. Our solution is dividing the task in two steps. This approach forces the use of algorithms of connection 6 [26] for the correct algorithm continuity in second step.

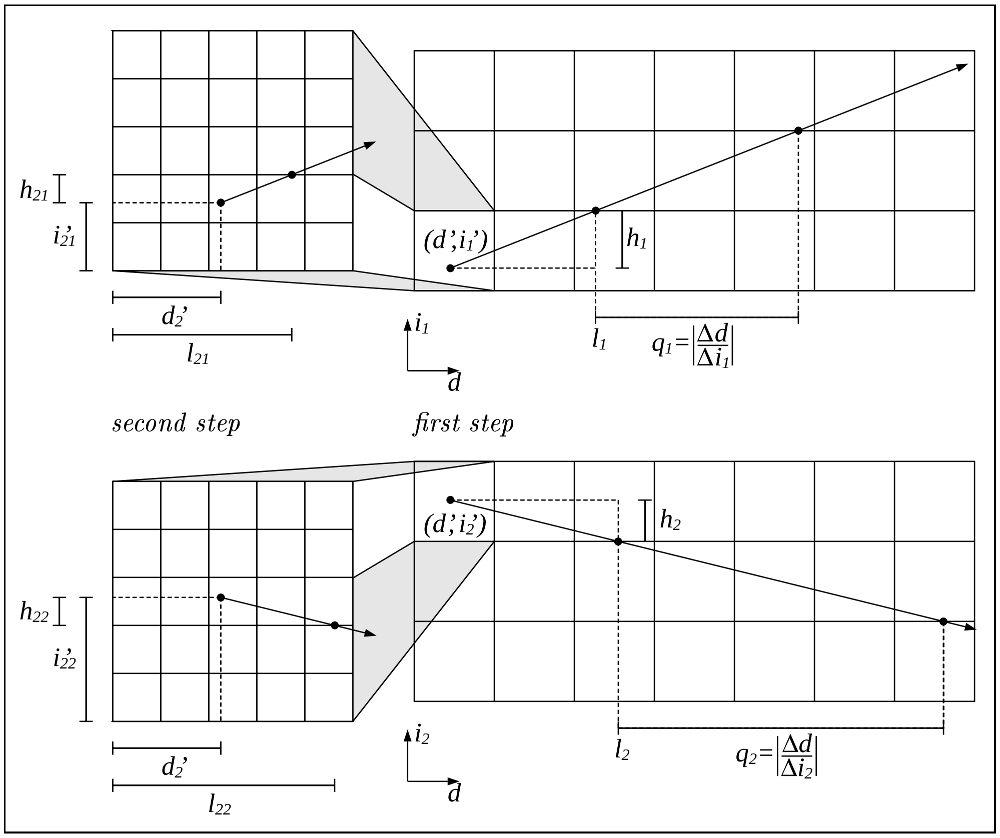

Figure 3: Principles of our Three-dimensional DDA

Fujimoto et al. [7] have suggested an incremental DDA for identifying ray-traversed voxels rapidly, called 3DDDA. The main advantage of their approach is using integer arithmetics. Our three-dimensional DDA is inspired on their implementation.

Another method was suggested by Amanatides and Woo ([3], Fig. 2), which is similar to the one shown in [20]. The big disadvantage of this method is using floating point calculation. Even though in some processors the floating point additions are as fast as integer ones, the algorithm precision is limited. Snyder [20] suggests that the algorithm could be implemented with integer arithmetic. Our experience [21], on the other hand, demonstrated that this solution is not very precise. It also generates many problems scaling t to an integer variable when $(t_{max} - t_{min}) < 1$.

For the two steps process, a 6 connected algorithm is required and also a very precise calculation is needed. If the initialization is done with double precision (64 bits) variables, the maximum mantissa precision is 53 bits [10]. However, the incremental calculus tends to propagate the errors very rapidly.

To prevent these errors, a higher precision than 53 bits is required. Using fixed point notation, it is possible to enhance the precision, since all the 64 bits can be used instead of only 53. The solution is straightforward, but this analysis is out of the scope of this article.

3.2. Principles and Initialization

Our three-dimensional DDA, illustrated in Fig.3, is based on the principles shown in [7]. The axis of greatest movement is indicated by identifier d and $|\Delta d|=max(|\Delta x|, |\Delta y|, |\Delta z|)$, where Δx, Δy and Δz are components of ray's direction vector. The other two axes are indicated by identifiers i_1 and i_2, where Δi_1 and Δi_2 are the other two components of ray's direction vector. A general algorithm's overview follows:

Execute step 1 and 2 until current voxel has exited empty octant:

1. if there is *not* an intersection between the ray and a voxel boundary perpendicular to i_1 or i_2 inside current voxel, select driving axis (d); otherwise, select incremental axis (i_1 or i_2) where intersection has occurred or the nearest to ray origin's if there are two intersections (ex.: in Fig.3, i_1 is selected between the two first intersections);
2. advance to next voxel by incrementing (or decrementing) current voxel selected axis coordinate only;

```
if ($\Delta i_1$<0) initialize $inc_1$ to -1;†
else        initialize $inc_1$ to 1;†
if ($\Delta i_2$<0) initialize $inc_2$ to -1;†
else        initialize $inc_2$ to 1;†
d=floor(d');      /* integer coordinates */
$i_1$=floor($i_1$);     /* are adjusted to the */
$i_2$=floor($i_2$);     /* voxel origin        */
put d, $i_1$, $i_2$ in XYZ;
signed_$q_1$=$\Delta d/\Delta i_1$; /* calculate inverse slopes */
signed_$q_2$=$\Delta d/\Delta i_2$;
$q_1$=|signed_$q_1$|;
$q_2$=|signed_$q_2$|;
/* for a negative drive displacement */
/* $l_1$, $l_2$ and d are negative          */
if ($\Delta d$<0) begin
                  $l_2=h_1 \cdot q_1$-d';
                  $l_1=h_2 \cdot q_2$-d';
                  d=-d;
                  inc_$q_1$=-$q_1$;
                  inc_$q_2$=-$q_2$;
                  initialize $inc_d$ to -1;†
              end
else          begin
                  $l_1=h_1 \cdot q_1$+d';
                  $l_2=h_2 \cdot q_2$+d';
                  inc_$q_1$=$q_1$;
                  inc_$q_2$=$q_2$;
                  initialize $inc_d$ to 1;†
              end
convert $l_1$, $l_2$, $q_1$ and $q_2$ to fixed point;
$d'_2$, $i'_{21}$, $i'_{22}$←d', $i'_1$, $i'_2$ (convert to 2nd step)
first_time=1;
if ($l_1<l_2$) limit=fix_integer_part($l_1$);
else        limit=fix_integer_part($l_2$);
†see text for details
```

Figure 4: Three-dimensional DDA Initialization

Identifiers l_1 and l_2 represent (as shown in Fig.3) coordinates in the driving axis where a voxel transition in each incremental axis takes place. If the displacement along the driving axis is negative (Δd<0) the values of d, l_1 and l_2 are forced to be negative to indirectly reverse the boolean value of the condition $l_1 < l_2$. This slight modification in our original algorithm eliminates pointers or arrays, allowing an implementation using only register variables.

The variables inc_d, inc_1 and inc_2 are respectively the increments for the coordinates d, i_1 and i_2 shifted to fit to the correspondent X, Y and Z in XYZ.

The first intersection points between the ray and each incremental axis are shown in Fig.3. The next intersection points can be obtained adding the inverse slopes (q_1 and q_2) as indicated in Fig.3. Identifiers h_1 and h_2 are heights to find the first intersection points (algorithm Fig.4).

3.3. Implementation

Our Three-dimensional DDA algorithm is shown in Fig.5. All variables in this algorithm are integers. Some are 64 bits integers representing real values in fixed point notation: q_1, q_2, l_1 and l_2.

```
XYZ_ant=XYZ and mask;             /* Filter last parent bits */
                          /* Loop while last parent bits unchanged */
while (TRUE)              /* that is, loop while in empty octant   */
    begin
        if ($d \neq limit$)        /* Incremental axis transition? */
                                  /* No, process driving axis      */
            begin                 /* Increment driving coordinate  */
                XYZ=XYZ+inc_drive;
                d=d+1;
                                  /* Last parent bits changed?     */
                if ((XYZ and mask) $\neq$ XYZ_ant)
                    begin         /* Yes, got out empty octant */
                        XYZ_ant=XYZ-inc_drive;
                        i_c=0;
                        break;                  /* Exit loop */
                    end
            end
        else
            /* Process incremental axis */
            begin
                            /* Choose the nearest incremental axis */
                if ($l_1 < l_2$)
                    begin     /* Increment $i_1$ axis coordinate */
                        XYZ=XYZ+$inc_1$;
                        $l_1$=$l_1$+$q_1$; /* Next intersection point */
                        if ($l_1 < l_2$) limit=fix_integer_part($l_1$);
                        else limit=fix_integer_part($l_2$);
                        if ((XYZ and mask)$\neq$XYZ_ant)
                            begin         /* out empty octant */
                                XYZ_ant=XYZ-$inc_1$;
                                i_c=1;
                                break;            /* Exit loop */
                            end
                    end
                else
                    begin     /* Increment $i_2$ axis coordinate */
                        XYZ=XYZ+$inc_2$;
                        $l_2$=$l_2$+$q_2$; /* Next intersection point */
                        if ($l_1 < l_2$) limit=fix_integer_part($l_1$);
                        else limit=fix_integer_part($l_2$);
                        if ((XYZ and mask)$\neq$XYZ_ant)
                            begin         /* out empty octant */
                                XYZ_ant=XYZ-$inc_2$;
                                i_c=2;
                                break;            /* Exit loop */
                            end
                    end
            end
    end
```

Figure 5: Our Three-dimensional DDA

The algorithm begins always with an octant to be skipped. This octant could be a non-empty first step voxel where no ray's intersection voxel is found in the second step. However, in most of the cases this octant is an empty octant and its size is proportional to the level where it is situated. In either case, the algorithm traverses this region, without consulting the octree, into a closed loop. The algorithm's efficiency resides in a very simple test to verify if traversal exited an empty region. Although it resembles Sung's [23] test in *"SimpleSkipParent"* procedure in logic, ours is integrated into the three-dimensional DDA and implemented with only two machine instructions (one logical *"and"* and one *"jump if not equal"*), which is considerably faster. The condition to continue the loop is that the current coordinate value, after incrementing it, has the same octant parent bits than its initial value before the loop. This is conveniently done by checking if octant parent low order bit has changed (algorithm Fig.5), which is essentially the same manner we know a ray arrived outside the octree (algorithm in Fig.1 and discussion in section 2).

Amanatides and Woo's algorithm takes 5 operations per coordinate (not counting the end condition neither the access to the 3D array), or 6 operations with our method to skip empty regions. Our algorithm takes 5 operations for the driving axis and 7 operations for the other two axes. However it can be reduced to 4 operations for the driving axis, avoiding incrementing XYZ, and substituting XYZ by d in the empty region test. To do that we must change the negative number notation for l_1, l_2, *limit* and d to one's complement.

Our algorithm is clearly optimized to process the driving axis, which is the axis where most processing is done. In addition, although Amanatides and Woo's algorithm is conceptually very simple it still depends on floating point arithmetic, which can affect efficiency, and limits the precision to 53 bits. Even if the precision problem could be solved with extended double precision (very rare in current machines), the efficiency would decrease, because extended double precision is usually slower than double precision. Our algorithm on the other hand, works in 64 bits precision, and is optimized to 64 bits machines.

3.4. Second Step Transition

The 3D grid representing the voxels is logically subdivided into two steps. The total resolution of this grid is therefore divided into two lower resolution steps. A 1024^3 grid, for example, can be represented in two steps as a 32^3 grid, where each voxel is in reality another 32^3 grid. This grid is represented by an octree with 10 levels, where the first 5 levels are in first step and the second 5 levels are in second step. When a ray is traced in the first level resolution and it arrives to an occupied voxel it must then be converted to the second level to be traced in the grid contained in this voxel.

The transition algorithm (Fig.6) determines ray's entry point (XYZ_2) in the second step grid and enables a second three-dimensional DDA (essentially equivalent to the first step one) tracing the ray along the second step sub-octree.

Thanks to our three-dimensional DDA concept the inverse slopes (q_1 and q_2) are conserved in the second step and don't need to be recalculated.

This conversion or transition is considerably easier and faster for the first voxel of a secondary ray (reflected, shadow or refracted ray). In this case the ray starting point is always inside the first voxel (algorithm's first part). Hence there is no need to determine ray entry point, but only to calculate l_{21} and l_{22} (Fig.3) and accommodate d_2', i_{21}', i_{22}' (Fig.3) in XYZ_2.

For a primary ray, or a secondary ray not in the first voxel, the slopes (inv_q_1 and inv_q_2), however, have to be determined (for entry point calculation) but just when needed and once per ray. For a primary ray that doesn't intersect any voxel in first step or a secondary ray that doesn't intersect any voxel in second step, for example, they are not calculated.

The entry point is an intersection with coordinates int_d, int_i_1, int_i_2, which always falls in a first step voxel boundary plane. This boundary plane can be perpendicular to the driving axis (*modf(int_d)=0*, algorithm's second part) or perpendicular to a passive axis (*modf(int_*i_1*)=0 or modf(int_*i_2*)=0*, algorithm's third part).

After initialization, the three-dimensional DDA works as in first step. An optimization that could be done is to continue the second step without initializing it if the precedent voxel in first step is a

```
if ((secondary ray) and (XYZ==XYZ_ant))
  begin    /* Secondary ray in 1st voxel */
    if ($\Delta d<0$) begin
                $l_{21}=h_{21}\cdot q_1-d'_2$;
                $l_{22}=h_{22}\cdot q_2-d'_2$;
             end
    else     begin
                $l_{21}=h_{21}\cdot q_1+d'_2$;
                $l_{22}=h_{22}\cdot q_2+d'_2$;
             end
    put $d'_2$, $i'_{21}$, $i'_{22}$ in $XYZ_2$;
  end
else /* all other rays */
  begin
    if (first_time) /* Calculate slopes */
      begin        /* just once.        */
        $inv\_q_1=\Delta i_1/\Delta d$;
        $inv\_q_2=\Delta i_2/\Delta d$;
        first_time=0;
      end
    if (i_c is driving axis)/* Intersection is */
      begin                 /*in driving axis. */
        if ($\Delta d<0$) /* Get driving axis coor- */
          begin    /* dinate of intersection */
            $int\_d=|d|+1$;
            $int\_d_2=$ max 2nd step value;
          end      /* point for 1st and 2nd  */
        else       /* step. When $\Delta d<0$,     */
          begin    /* intersection point is  */
            int_d=d; /* at higher voxel      */
            $int\_d_2=0$; /* boundary.           */
          end      /* Calculate the other 2  */
        $int\_\Delta d=int\_d-d'$; /* coordinates.     */
        $int\_i_1=int\_\Delta d\cdot inv\_q_1+i'_1$;
        $int\_i_2=int\_\Delta d\cdot inv\_q_2+i'_2$;
        $int\_i_{21} \leftarrow int\_i_1$ (convert 2nd step);
        $int\_i_{22} \leftarrow int\_i_2$ (convert 2nd step);
        /* Calculate limits for inc. axes.    */
        if ($\Delta i_1<0$)
          $l_{21}$=-$inv\_q_1\cdot modf(int\_i_{21})+int\_d_2$; †
        else
          $l_{21}=inv\_q_1\cdot (1-modf(int\_i_{21}))+int\_d_2$; †
        if ($\Delta i_2<0$)
          $l_{22}$=-$inv\_q_2\cdot modf(int\_i_{22})+int\_d_2$; †
        else
          $l_{22}=inv\_q_2\cdot (1-modf(int\_i_{22}))+int\_d_2$; †
      end
    else          /* Intersection is in */
      begin       /* incremental axis  */
        if (i_c is $i_1$ axis) j=1, k=2;
        else                  j=2, k=1;
        get $i_j$ from ZYZ;
        if ($\Delta i_j<0$)
          begin
            $int\_i_j=i_j+1$;
            $int\_i_{2j}=$ max 2nd level value;
          end
        else
          begin
            $int\_i_j=i_j$;
            $int\_i_{2j}=0$;
          end
        $int\_\Delta d=(int\_i_j-i'_j)\cdot signed\_q_j$;
        $int\_d=int\_\Delta d+d'$;
        $int\_d_2 \leftarrow int\_d$ (convert 2nd step);
        $l_{2j}=int\_d_2+inc\_q_j$;
        $int\_i_k=int\_\Delta d\cdot inv\_q_k+i'_k$;
        $int\_i_{2k}\leftarrow int\_i_k$ (convert 2nd step);
        if ($\Delta i_k<0$)
          $l_{2k}$=-$inv\_q_k\cdot modf(int\_i_{2k})+int\_d_2$; †
        else
          $l_{2k}=inv\_q_k\cdot (1-modf(int\_i_{2k}))+int\_d_2$; †
      end;
    $d_2=int\_d_2$;
    if ($\Delta d<0$) begin
                  $l_{22}$=-$l_{22}$;
                  $l_{21}$=-$l_{21}$;
                  $d_2$=-$int\_d_2$;
               end
    put $int\_d_2$, $int\_i_{21}$ and $int\_i_{22}$ into $XYZ_2$;
  end;
convert $l_{21}$, $l_{22}$ to fixed point;
if ($l_{21}<l_{22}$) $limit_2=fix\_integer\_part(l_{21})$;
else         $limit_2=fix\_integer\_part(l_{22})$;

†modf(n) → fractional part of n
```

Figure 6: Second step initialization algorithm

neighbor. Although fast, this Procedure must take in consideration error estimations to guarantee 53 bits precision. However it can be very interesting in cases where very few second step voxels are traversed in the precedent first step voxel.

4. Results

Our results for the test image 1 are summarized in Fig. 7. For test image 2, our results are summarized in Fig. 8. Images were calculated using a SGI Crimson workstation (R4000+R4010,100MHz,128Mb). The first columns in these tables indicate the screen resolution. Second columns indicate 3D grid resolution. Third columns give the time for single step ray tracing. Fourth columns indicate times for two step ray tracing with different number of levels of the octree in the first step and the second step. The numbers in parentheses are respectively the number of levels in the first step and in the second step. The "+" separating them indicates that their addition gives the total number of octree levels. All these computational times include the CPU time for the voxelization and the generation of a PostScript file of the image. The code was compiled with debugging option and for 32 bits (the calculation for the passive axes are doubled). Image 1 consumed 13.3 Mb for the voxels and 8.4 Mb for the octree in a 1024^3 resolution. Image 2 consumed 37.8 Mb for the voxels and 24 Mb for the octree in a 1024^3 resolution.

The best time in our two step process was always with 6 levels in the first step. This result was expected because it agrees with DDA performance given by other authors [7, 23]. The best time we have obtained in comparison with the single step process was with the largest 3D grid (2048^3) and largest image resolution, which, in our two step process, is represented by one fifth of the single step process time. This result agrees with the observation in [26] that most of the time of the discrete ray tracing was spent in three-dimensional DDA. We have bypassed this problem creating the two step process, thus profiting of its optimal performance. This was possible because of the the octree multiple resolution characteristic.

Res. 2D	Res. 3D	RT 1 step	RT 2 steps
1024^2	2048^3	25’06”	8’21” (4+7)
			5’38” (5+6)
			5’03” (6+5)
			5’36” (7+4)
1024^2	1024^3	12’52”	3’59” (4+6)
			3’03” (5+5)
			2’59” (6+4)
			3’40” (7+3)
512^2	1024^3	3’28”	1’14” (4+6)
			1’00” (5+5)
			0’59” (6+4)
			1’09” (7+3)
256^2	1024^3	1’06”	0’33” (4+6)
			0’29” (5+5)
			0’29” (6+4)
			0’32” (7+3)

Figure 7: Comparison results for image 1 (Fig.9)

Res. 2D	Res. 3D	RT 1 step	RT 2 steps
1024^2	1024^3	18’11”	5’27” (4+6)
			4’05” (5+5)
			3’44” (6+4)
			4’04” (7+3)
512^2	1024^3	7’10”	2’05” (4+6)
			1’44” (5+5)
			1’44” (6+4)
			1’46” (7+3)
256^2	1024^3	2’02”	1’14” (4+6)
			1’09” (5+5)
			1’07” (6+4)
			1’09” (7+3)

Figure 8: Comparison results for image 2 (Fig.10)

A curious fact however is that the advantage of the two step process decreases (although the computational times are always better than the single step ones) for lower screen resolutions. We believe that the origin of this behavior is the floating point calculation during the transition of the two steps. Therefore, our system is ideal for very high resolution 3D grids and large images. The resolution of the 3D grid is limited by the surface of the object and the memory size.

5. Conclusion

We presented the implementation of a Discrete Ray Tracing using an octree. A method to bypass completely the vertical traversal of the octree over empty regions was implemented (as previously shown). In empty regions, the DDA stays in a closed loop, thus reducing in about 50% the global time over the previous implementation. Our results with simple test scenes were similar to the ones obtained in [26]. Since the performance is very sensitive to the DDA processing and since the procedure of skipping empty regions still run the complete DDA, we have assumed that a multiple precision DDA (as briefly described previously) could lead us to even better results. Effectively this was verified for high resolution grids and high resolution images. The advantage of the method decays though for lower resolutions probably due to the floating point calculation for the transition between the two steps. Therefore the method is an efficient tool to visualize huge volumetric data realistically.

References

[1] T. Akimoto, Mase Kenji, and Suenaga Y. Pixel-Selected Ray Tracing. *IEEE - CGA*, 6(4):14–22, 1991.

[2] John Amanatides. Ray Tracing with cones. *Computer Graphics*, 18(3):129–135, 1984.

[3] John Amanatides and Andrew Woo. A Fast Voxel Traversal Algorithm for Ray Tracing. In *Eurographics'87*, pages 3–9, Amsterdam, August 1987. North Holand.

[4] James Arvo and David Kirk. Fast Ray Tracing by Ray Classification. *Computer Graphics*, 21(4):55–63, 1987.

[5] H. B. Bidasaria. Defining and Rendering of Textured Objects through The Use of Exponential Functions. *Graphical Models and Image Processing*, 54(2):97–102, March 1992.

[6] D. Cohen and A. Kaufman. 3D Scan-Conversion Algorithms for Linear and Quadratic Objects. *Volume Visualization*, pages 280–301, 1990.

[7] Akira Fujimoto, Takayaki Tanaka, and Kansei Iwata. ARTS: Accelerated Ray Tracing System. *IEEE - CGA*, 6(4):16–26, 1986.

[8] Irene Gargantini. Ray tracing an Octree: Numerical Evaluation of the First Intersection. *Computer Graphics forum*, 12(4):199–210, 1993.

[9] Andrew S. Glassner. Space Subdivision for Fast Ray Tracing. *IEEE - CGA*, 10(4):15–22, 1984.

[10] David Goldberg. What Every Computer Scientist Should Know About Floating-Point Arithmetic. *ACM Computing Surveys*, 23(1):5–48, March 1991.

[11] Ned Greene. Voxel Space Automata: Modeling with Stochastic Growth Processes in Voxel Space. *Computer Graphics*, 23(3):175–184, July 1989.

[12] Paul S. Heckbert and Pat Hanrahan. Beam Tracing Polygonal Objects. *Computer Graphics*, 18(3):119–127, 1984.

[13] David Jevans and Brian Wyvill. Adaptative Voxel Subdivision for Ray Tracing. In *Proceedings of Graphics Interface '89*, pages 164–172, Toronto, Ontario, June 1989. Canadian Information Processing Society.

[14] Devendra Kalra and Alan Barr. Guaranteed Ray Intersections with Implicit Surfaces. *Computer Graphics*, 23(3):297–306, July 1989.

[15] A. Kaufman. An Algorithm for 3D Scan-Conversion of Polygons. In *Eurographics'87*, pages 197–208, Amsterdam, August 1987. North Holand.

[16] Arie Kaufman. Efficient Algorithms for 3D Scan-Conversion of Parametric Curves, Surfaces, and Volumes. *Computer Graphics*, 21(4):171–179, July 1987.

[17] Arie Kaufman and Reuven Bakalash. Memory and Processing Architecture for 3D Voxel-Based Imagery. *IEEE - CGA*, November 1988.

[18] Timothy Kay and James Kajiya. Ray Tracing Complex Scenes. *Computer Graphics*, 20(4):269–278, August 1986.

[19] S. M. Rubin and T. Whitted. A 3-Dimensional Representation for Fast Rendering Complex Scenes. *Computer Graphics*, 14(3):110–116, 1980.

[20] John Snyder and Alan Barr. Ray Tracing Complex Models containing Surface Tesselations. *Computer Graphics*, 21(4):119–128, 1987.

[21] Nilo Stolte and René Caubet. Some More Enhancements to Ray Tracing. In *Compugraphics'92*, pages 53–60, Lisbon, December 1992. Harold P. Santo.

[22] Nilo Stolte and René Caubet. A fast scan-line method to convert convex polygons into voxels. In *Compugraphics'93*, pages 164–170, Alvor, December 1993. Harold P. Santo.

[23] Kelvin Sung. A DDA Traversal Algorithm for Ray Tracing. In *Eurographics'91*, pages 73–85, Amsterdam, June 1991. North Holand.

[24] Markku Tamminen, Olli Karonen, and Martti Mantyla. Ray-casting and block model conversion using a spatial index. *Computer-Aided Design*, 16(4):203–208, July 1984.

[25] Turner Whitted. An Improved Ilumination Model for Shaded Display. *Communications of the ACM*, 23(6):343–349, 1980.

[26] Roni Yagel, Daniel Cohen, and Arie Kaufman. Discrete Ray Tracing. *IEEE - CGA*, 12(5):19–28, 1992.

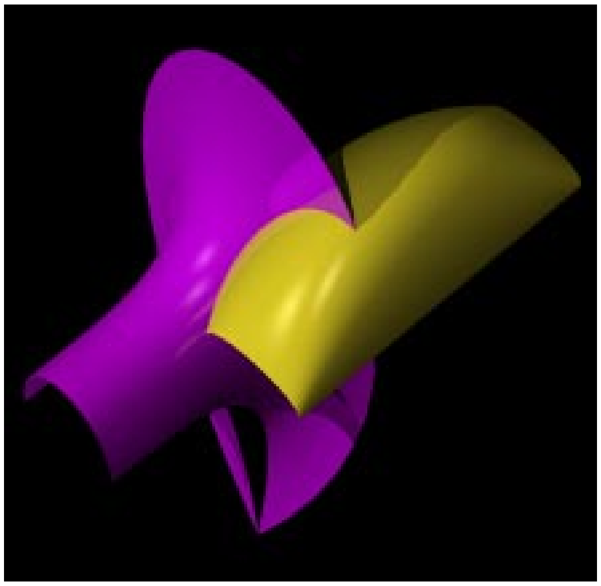
Figure 9: Image 1

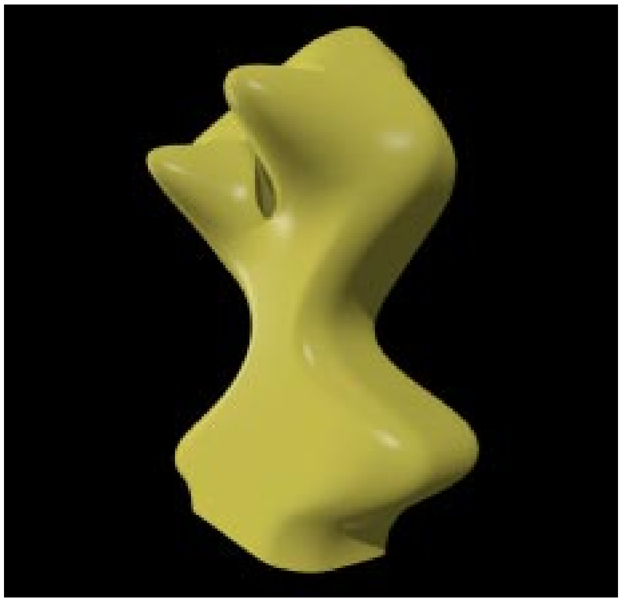
Figure 10: Image 2