Discrete Ray-Tracing High Resolution 3D Grids

Nilo STOLTE *
René CAUBET
Institut de Recherche en Informatique de Toulouse
118, Route de Narbonne
31062 – Toulouse – France
tel. (33) 61 55 67 65 / fax. 61 55 62 58
stolte@irit.fr

Abstract

This article suggests a new approach to discrete Ray-Tracing in which a two step three-dimensional DDA and an octree are used. A very important problem regarding this kind of Ray Tracing, also known as Raster Ray Tracing, is the amount of memory required to store the 3D raster grid which will contain the discretized scene to be visualized. Since the resolution of this grid is huge because the voxel is assumed to be approximately the size of a pixel on the screen, it is limited by the maximum amount of memory of todays' machines. Although using an octree and assuming that the majority of space will be empty, as such is the case in most scenes, an important memory saving is achieved. This memory saving helps using discrete Ray Tracing in normal workstations. It is shown that the special octree presented here doesn't slow down Ray Tracing significantly and even competes with normal discrete Ray-Tracing. It is also shown that in critical cases this octree will not occupy much more space than a normal 3D grid.

Another important problem is the bottleneck caused by the three-dimensional DDA in such huge resolution grids. This problem is solved by dividing these grids into two low resolution ones. The process can be divided in two steps where optimal times of three-dimensional DDA are achieved. This is shown through the comparative tests with the single step process.

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

1 Introduction

Ray-Tracing algorithm is now widely known not only by experts but also by the general public as a very powerful and simple tool for rendering realistic scenes. Despite that, it has the reputation of being a high time consuming process, although many efforts have been made to accelerate it in many different ways. This reputation is due to the fact that it has to determine which objects of the scene are intersected by a certain ray, task where

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

most of its time is consumed as observed in [Whitted, 1980]. In fact all acceleration methods are somehow related to the problem of ray/object intersection. Some use simple bounding boxes ([Rubin and Whitted, 1980]) to avoid most of the objects that are certainly not crossed over by the ray. Others use subdivision methods to decompose the universe in subspaces modifying the ray to travel these special structures ([Fujimoto *et al.*, 1986], [Glassner, 1984], [Arvo and Kirk, 1987],[Snyder and Barr, 1987], [Sung, 1991], [Stolte and Caubet, 1992], [Yagel *et al.*, 1992], [Gargantini, 1993]), to disregard subregions where ray does not pass through. Many other original ways have been suggested like the Beam Tracer, where several rays are treated at once inside a beam ([Heckbert and Hanraham, 1984]), but it works only with polygonal objects and good only for relatively simple scenes. Another is the ray under-sampling ([Akimoto *et al.*, 1991]), where less rays are shot and the illumination is hopefully most of the time interpolated, but it is not proper for very high quality images and also works well only in relatively simple scenes.

Among the methods that use 3D space subdivision, some subdivides space uniformly ([Fujimoto *et al.*, 1986],[Snyder and Barr, 1987], [Sung, 1991], [Stolte and Caubet, 1992], [Yagel *et al.*, 1992], [Gargantini, 1993]) or adaptatively ([Glassner, 1984]), or even other kinds of subdivision in more dimensions to exploit ray coherency ([Arvo and Kirk, 1987]).

The subdivision process, called voxelization ([Stolte and Caubet, 1993]), that subdivides space uniformly assigns the objects' outline geometry to a 3D voxel grid. In this process only the part of the geometry that pierces the voxel is assigned to it. This voxel grid is normally stored in the 3D array format, but it can also be stored in an octree ([Fujimoto *et al.*, 1986],[Sung, 1991], [Stolte and Caubet, 1992],[Gargantini, 1993]).

A special interesting acceleration method was suggested recently in [Yagel *et al.*, 1992]. It is called Discrete or Raster Ray Tracing, because it works entirely into the 3D voxel space, the discrete space. The big advantage of this method is that no intersection calculation between ray and objects is needed, allowing a very fast Ray Tracing. The big disadvantage is the high resolution 3D voxel grid necessary to achieve good quality images. This grid is then limited to the machine memory size. The grid resolution in [Yagel *et al.*, 1992] is $256 \times 256 \times 256$ to a 80 Mb machine. The maximum resolution was $320 \times 320 \times 320$ to a 128 Mb machine. This easily denotes the difficulty to run this method in a normal workstation. It is also clear 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. On this way simple scenes can be generated in machines poor in memory. On the other hand, it allows to use higher resolutions necessary to get good quality images, which is impossible nowadays by using directly 3D grids.

Nevertheless as stated in [Yagel *et al.*, 1992] most of method's processing cost falls in the discrete traversal algorithm, namely the Three-dimensional DDA. In [Yagel *et al.*, 1992] a non conventional DDA ,was created to reduce this time. This solution is not enough in high resolution 3D grids. It is a known fact that its efficiency falls down between the fifth and the sixth levels of the octree ([Fujimoto *et al.*, 1986],[Sung, 1991]). We suggest a two step process to profit of 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 [Fujimoto *et al.*, 1986] but the mapping of floating point values in fixed point using integer variables, offers, at the same time, efficiency, accuracy and multi-precision flexibility.

We compare our two steps approach with the same program implemented in only one step. Results show that an optimization of the order of one fifth of time is achieved.

2 The Octree

2.1 Introduction

The octree suggested here uses the spatial enumeration in fixed sized blocks, which we call cells, of eight elements each. This approach allows a very fast vertical traverse due to operations' simplicity. We show that memory requirements for this octree are also very interesting.

The octree is defined dividing space recursively in eight regions of equal dimensions in each level until each of these regions has the size of a voxel (Figure 1). Each element of each cell corresponds to a sub-space of the correspondent volume on a given level.

Figure 1 represents the octree implementation with five levels (left) and its visualization into space (right). Five cells are associated to the octree of Figure, where each one is located in successive levels of the octree. Each element on last level corresponds to a voxel into a 3D grid of $2^5 \times 2^5 \times 2^5$ resolution, which can be represented by its coordinates: (X,Y,Z). The eight voxels showed in Figure 1, on last level (k=4), pink-coloured, are: $V_0(30,30,30)$ -the only one not seen in 3D diagram-, $V_1(31,30,30)$, $V_2(30,31,30)$, $V_3(31,31,30)$, $V_4(30,30,31)$ -the only explicitly shown in Figure-, $V_5(31,30,31)$, $V_6(30,31,31)$, $V_7(31,31,31)$. They are shown in this order in diagram on the left. This order is the same adopted in [Fujimoto *et al.*, 1986]. Each cell is therefore an array of eight elements where the order of an element is related to its position inside the sub-volume that the cell represents.

We define the order of the voxels in cell for octrees with n levels. Each coordinate X, Y, Z of a voxel in an uniformly subdivided space with a resolution $2^n \times 2^n \times 2^n$ is a binary number that is defined as a summation of two powers (see below). The total number of levels of the octree is n. Be k the index of a bit in binary coordinate ($k = n-1$, the rightmost bit; $k = 0$, the leftmost bit). Therefore the index of an element of a cell in a certain level k is given by $i_k = x_k + 2 \cdot y_k + 4 \cdot z_k$, where x_k, y_k and z_k are the correspondent bits of the coordinates X, Y and Z indexed by k:

$$X = \sum_{k=0}^{n-1} x_k \cdot 2^k \qquad Y = \sum_{k=0}^{n-1} y_k \cdot 2^k \qquad Z = \sum_{k=0}^{n-1} z_k \cdot 2^k$$

$$0 \leq X \leq 2^n - 1 \qquad 0 \leq Y \leq 2^n - 1 \qquad 0 \leq Z \leq 2^n - 1$$

$$x_k, y_k, z_k \in \{0, 1\}$$

2.2 Vertical Traverse Performance

Because of the property of dividing space in eight portions recursively, the vertical traverse of the octree shown here, has a complexity of O($log_8 N$) for a full octree, where

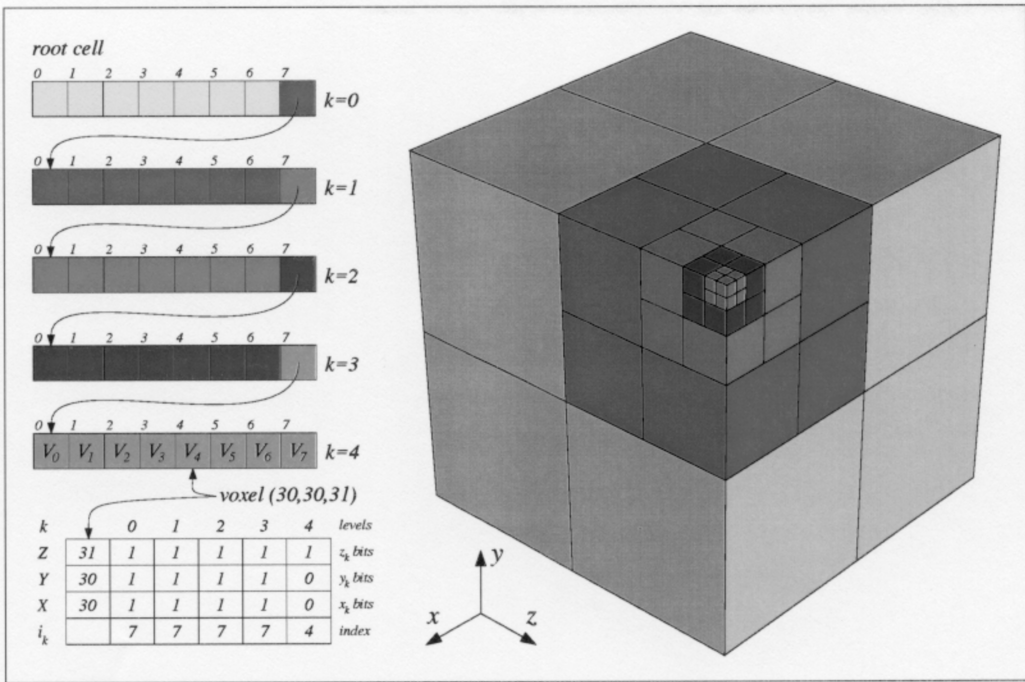

Figure 1: Block diagram showing the octree

N is the total number of cells.

Although the vertical traverse performance depends also on the processing in each of its levels. This processing is done by the algorithm that implements the lookup of a voxel. This algorithm is given in a notation similar to the C language (Figure 2). X, Y and Z are the coordinates of the voxel to be searched. The $<<$ and $>>$ operators are respectively the binary left and right shift. The array *cell* is a pointer but the notation of array is more convenient for clarity purposes. Hence variable i does not really exist. At algorithm exit variable *cell* will contain the address of the cell where the voxel is (a null address indicates that voxel does not exist) and the value of i is the voxel index in this cell (in the real implementation, the pointer *cell* has the voxel address).

In Figure 2 algorithm, variable $mask$ has initially the bit $n-1$ set (n is the number of octree levels, as seen before) and all the other reset. The pointer *cell* receives the address of the first octree cell, which is the father of the whole octree. The main processing is done into a loop shown here by a *while* command. The loop is totally controlled by variable $mask$ which is shifted right in each successive cell. When $mask$ arrives to one, the loop is interrupted as shown. Otherwise, the index of the element in cell is initialized and calculated. Making a logical "*and*" between $mask$ and each coordinate is an easy and fast way to ignore a zero bit, which indicates no contribution to the index. On the other hand, if it is one, its contribution is accumulated in the index.

Therefore, variable $mask$ has in reality two different functions in algorithm. These functions are the exit control of loop and filter of the bit to be considered. This is very important because with this scheme the proper bit and the loop control are both updated by just one shift operation. When a whole branch of the octree does not exist (when *cell*[i] is zero) the process is immediately interrupted. This extra test is necessary to the algorithm correctness and reduces useless processing when voxel does not exist.

```
mask = 1 << (n - 1)
cell = octree root cell
bool = TRUE
while (bool)
    { i = 0
      if  (Z and mask) ≠ 0)   i = i + 4
      if  (Y and mask) ≠ 0)   i = i + 2
      if  (X and mask) ≠ 0)   i = i + 1
      if ((mask and 1) == 0)
        { if (cell [i] ≠ 0)
            { mask = mask >> 1
              cell = cell [i]
            }
          else bool = FALSE
        }
      else bool = FALSE
    }
```

Figure 2: Octree vertical traverse

Then, in this algorithm, there are the following operations: 4 logical *ands*, 1.5 additions (because each addition has 50% probability of being executed), 2 comparisons and 1 shift. This makes only 8.5 integer arithmetic operations for each octree level. But it is even faster for empty regions.

It is easily shown that this implementation is faster than the linear octrees ([Gargantini, 1982]), but this demonstration is out of the scope of this article. This conclusion is particularly interesting since linear octree serves as a base to many octree implementations ([Glassner, 1984], [Sung, 1991], [Gargantini, 1993]).

2.3 Memory Requirements

The octree shown here is formed with cells of eight elements each. Each element can have eight son cells, right from the first cell. Hence, the memory allocation for the full octree obeys a geometric series with step 8:

$$8^0 + 8^1 + 8^2 + 8^3 ... + 8^{n-1} = \frac{1 - 8^n}{1 - 8}$$

For last level, there are 8^{n-1} cells that will correspond to 8^n leaf elements or voxels. Then for indexes, there will be in a full octree the following number of cells:

$$8^0 + 8^1 + 8^2 + 8^3 ... + 8^{n-2} = \frac{1 - 8^{n-1}}{1 - 8} = \frac{1}{7} \cdot 8^{n-1} - \frac{1}{7}$$

A 3D array of voxels of $2^n \times 2^n \times 2^n$, which has 8^n voxels, will contain 8^{n-1} cells. Therefore a full octree has practically $\frac{1}{7} \times 8^{n-1}$ more cells than the equivalent 3D array.

Although memory requirements for the full octree will be about 14% greater than a 3D grid, it is highly improbable that this situation would happen without being forced. Anyway, memory requirements for a 3D grid are still prohibitive for high resolutions. Hence the octree as proposed here seems to be much more appealing than the 3D grid in the Discrete Ray Tracing context, where grid resolution is high.

3 Three-dimensional DDA

Space Regular 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 enough. 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 ([Yagel *et al.*, 1992]) for the correct algorithm continuity in second step.

Fujimoto et al. ([Fujimoto *et al.*, 1986]) have suggested an incremental DDA for identifying ray-traversed voxels rapidly, called 3DDDA. The main advantage of their approach is using integer calculi. Our three-dimensional DDA is inspired of their implementation. Unfortunately, it is impossible to compare both algorithms since not enough details are given about their 3DDDA.

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

Figure 3: Three-dimensional DDA of Amanatides and Woo

Another method was suggested by Amanatides and Woo ([Amanatides and Woo, 1987], Figure 3), which is similar to the one shown in [Snyder and Barr, 1987]. The big disadvantage of the 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 ([Snyder and Barr, 1987]) suggests that the algorithm could be implemented with integer arithmetics. Our experience, on the other hand, ([Stolte and Caubet, 1992]) 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 pass process, we need not only a 6 connected algorithm but also a very precise calculation. If the initialization is done with double precision (64 bits) variables, the maximum mantissa precision is 53 bits. Our algorithm should work at least at this precision. Most of the processors work with 32 bits, which force us to use 2 integer

variables. Although the incremental calculi tend to propagate the errors very rapidly. To correct 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.

```
1  XYZ_ant = XYZ and mask
2  /* loops while inside empty space */
3  while (true)
4        { if (((XYZ >> shift_drive) and mask_drive) ≠ limit)
5            /* process driving axis */
6            { XYZ = XYZ + inc_drive
7              if ((XYZ and mask) ≠ XYZ_ant)
8                { XYZ_ant = XYZ - inc_drive
9                  break
10               }
11           }
12         else
13           /* process incremental axis */
14           { XYZ = XYZ + inc_inc
15             /* calculate next limit for this inc. axis */
16             limitf[i_inc] = limitf[i_inc] + limit_quotf[i_inc]
17             if (carry)
18               { limiti[i_inc] = limiti[i_inc] + inc_lim[i_inc] }
19             limiti[i_inc] = limiti[i_inc] + limit_quoti[i_inc]
20             /* decide which inc. axis is the nearest */
21             if (limiti[i1] > limiti[i2])
22               { limit = integer_part(limiti[0])
23                 i_inc = 0
24                 if ((XYZ and mask) ≠ XYZ_ant)
25                   { XYZ_ant = XYZ - inc_inc
26                     inc_inc = inc_inc1
27                     break
28                   }
29                 inc_inc = inc_inc1
30               }
31             else
32               { if (limiti[i1] < limiti[i2])
33                   { limit = integer_part(limiti[1])
34                     i_inc = 1
35                     if ((XYZ and mask) ≠ XYZ_ant)
36                       { XYZ_ant = XYZ - inc_inc
37                         inc_inc = inc_inc2
38                         break
39                       }
40                     inc_inc = inc_inc2
41                   }
42                 else
43                   { if (limitf[i1] > limitf[i2])
44                       { limit = integer_part(limiti[0])
45                         i_inc = 0
46                         if ((XYZ and mask) ≠ XYZ_ant)
47                           { XYZ_ant = XYZ - inc_inc
48                             inc_inc = inc_inc1
49                             break
50                           }
51                         inc_inc = inc_inc1
52                       }
53                     else
54                       { limit = integer_part(limiti[1])
55                         i_inc = 1
56                         if ((XYZ and mask) ≠ XYZ_ant)
57                           { XYZ_ant = XYZ - inc_inc
58                             inc_inc = inc_inc2
59                             break
60                           }
61                         inc_inc = inc_inc2
62                       }
63                   }
64               }
65           }
66       }
```

Figure 4: Our Three-dimensional DDA

Our algorithm (Figure 4) is based on the principles shown in [Fujimoto *et al.*, 1986]. The axis of greatest movement is called driving axis. The other two are called incremental axes. The basis of the algorithm is to increment unconditionally the driving axis coordinate while there is no change in the coordinates of the other two axes. Once such a change is detected, the correspondent incremental axis coordinate is incremented

instead. This clearly gives a connectivity 6 traversal.

The algorithm stays into a loop while it traverses an empty region (detected by octree nodes that do not have descendents). This is done to avoid descending the octree every time a new voxel is traversed (a more detailed description is given later).

During the initialization process (not shown in Figure 4), the first intersection points between line and the two passive axes are calculated. The coordinates of these points corresponding to the driving axis are called limits and are both stored in fixed point notation using integer arrays (*limiti* and *limitf*). One of these two limits will arrive first depending on the sense of displacement along the driving axis. The integer part of this limit is stored in the variable called *limit.*

For a 6 connected traverse, only one coordinate must be updated in each step. If the driving axis coordinate is not equal to *limit* (line 4), the chosen axis is the driving one, otherwise it is the corresponding passive one. In either case, the corresponding coordinate is incremented and a test to exit the loop is executed. However, each time a passive axis is processed, its new limit and the value of *limit* must be determined. Although these tasks are highly time consuming, their complexities are mainly due to 64 bits calculations using 32 bits variables. In 64 bits machines, they could be accomplished with only one arithmetic operation and one test.

The decision to keep the three coordinates (X, Y and Z) into just one variable (XYZ) was mainly to store them in just one register variable, and to accelerate certain tasks like ascending the octree (Figure 5), and tests for exiting loops. This decision force us to spend extra arithmetic operations to filter the coordinates.

The successive limit calculation is done incrementally. The increment is the quotient between the displacement along driving axis and the displacement along an increment axis, that is the inverse slope. It is also stored in integer arrays (*limit_quoti* and *limit_quotf*) using fixed point notation. In our 32 bits implementation, the incrementation is done in two steps, as indicated in algorithm (lines 16-19, *i_inc* indicates the increment axis as can be seen in lines 23, 34, 45 and 55). This separation (using two 32 bits variables instead of one 64 bits variable) is also responsible for the code duplication observed in lines 44-52 and 54-62.

Once the new limit is calculated, the nearest one of the two must be determined to be assigned to *limit.* The sense of displacement along driving axis can be forwards or backwards. Hence, the tests must be reversed accordingly. To avoid useless repetition of code and extra tests, this task can be better done indirectly through pointers (the real implementation) or through indexes (*i1* and *i2* in algorithm, lines 21, 32 and 43) assigned during initialization. Although this could be also done by reversing the signs of limits when the displacement is backwards, which allows an implementation entirely with registers.

It is evident that our 32 bits implementation can be slower than the algorithm of Amanatides and Woo (Figure 3), depending on machine. Although, if our method of skipping empty regions is adapted to their algorithm, a 64 bits version of our algorithm can be strongly competitive or even faster. In any case, our algorithm is 11 bits more precise which justifies its utilization in our two pass process.

The final condition test (lines 7, 24, 35, 46 and 56) allows us to skip empty spaces without descending or ascending the octree. Variable *mask* is shown in algorithm of Figure 2. It is used in our Three-dimensional DDA to identify the exact bit that changes the value only when the DDA leaves the empty region. If it has been changed the DDA's loop is finished and the octree's vertical traverse restarts as shown in algorithm of Figure

5.

Push and *pop* operators of Figure 5 denote an external stack to keep the addresses of the parents cells, as done by many authors ([Fujimoto *et al.*, 1986], [Sung, 1991] et [Gargantini, 1993]). The first part of main loop ascends the octree while no common father is found. The second part descends the octree (basically the same as Figure 2). The third part is the Three-dimensional DDA of Figure 4.

```
/* one mask for each coordinate */
mask = 1 << (n - 1)
mask = mask or (1 << ((n - 1) << 1))
mask = mask or (1 << (((n - 1) << 1) + (n - 1)))
/* three coordinates in one */
XYZ = (Z << ((log8 N) - 1))
XYZ = XYZ or (Y << (((log8 N) - 1) << 1))
XYZ = XYZ or (X << ((((log8 N) - 1) << 1) + ((log8 N) - 1)))
cell = root cell address of octree
push(cell)
i_c = 0 /* index of last changed coordinate*/
test_end = mask
XYZ_ant = XYZ
/* repeat while inside octree*/
while (true)
     { XYZ_ant = XYZ_ant xor XYZ
       if (XYZ_ant and test_end) break
       mask = mask >> 1
       /* ascend octree 'til common father arrives*/
       while (XYZ_ant and mask)
            { pop
              mask = mask << 1
            }
       mask = mask << 1
       pop(cell)
       X = XYZ and mask_X
       Y = XYZ and mask_Y
       Z = XYZ and mask_Z
       /* The lines different from fig. 2 are commented */
       bool = TRUE
       while (bool)
            { push(cell) /* push address cell */
              i = 0
              if ((Z and mask) ≠ 0) i = i + 4
              if ((Y and mask) ≠ 0) i = i + 2
              if ((X and mask) ≠ 0) i = i + 1
              if ((mask and 1) == 0)
                 { if (cell[i] ≠ 0)
                      { mask = mask >> 1
                        cell = cell[i]
                      }
                   else bool = FALSE
                 }
              else bool = FALSE
            }
       if (cell[i] ≠ 0)
          { /*second step*/ }
       /* 3DDDA, algorithm from fig. 4*/
     }
```

Figure 5: Octree traversal with empty space skip

4 Results

In table of Figure 6, our results for the test image 1 are summarized. For the test image 2, our results are summarized in Figure 7. Images were calculated using a Crimson SGI workstation (R4000+R4010,100MHz,128Mb). These tables indicate the screen resolution (first columns), 3D grid resolution (second columns), the time for the single step ray tracing (third columns), and times for the two step ray tracing with different number of levels of the octree in the first step and the second step (fourth columns). 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 the times include the CPU time for the voxelization and the generation of a PostScript file of the image.

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 6: Comparison results for image 1 (Figure 8)

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 7: Comparison results for image 2 (Figure 9)

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 ([Fujimoto *et al.*, 1986], [Sung, 1991]). The best time we have obtained in comparison with the single step process was with the largest 3D grid (2048 × 2048 × 2048) and largest screen 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 [Yagel *et al.*,

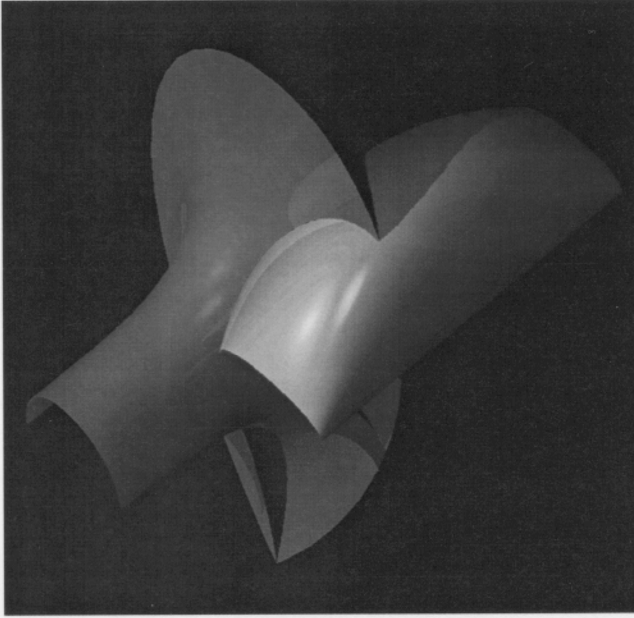

Figure 8: Image 1

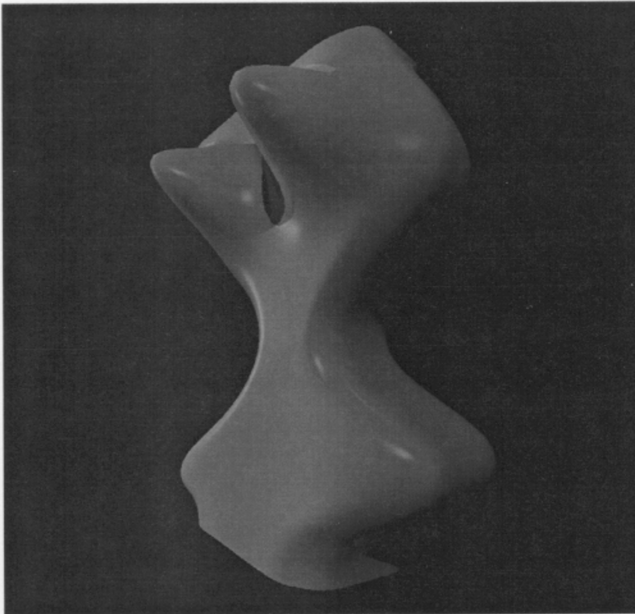

Figure 9: Image 2

1992] 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 only possible because of the the octree's multi-resolution characteristic.

A curious fact however is that the advantage of the two step process declines (although the times are always better than the single step) 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 avoid completely the vertical traversal of the octree over empty regions was implemented (as previously shown). While it was in empty regions, only the DDA was active in a closed loop. This reduced in about 50% the global time over the previous implementation. Our results with simple test scenes are similar to the ones obtained in [Yagel *et al.*, 1992]. Since the performance is very sensible to the DDA processing and since the skip empty regions procedure still run the complete DDA, we have assumed that a multi-precision DDA (as briefly described previously) could lead us to even better results. Effectively this was verified for high resolution grids and high resolution screen. The advantage of the method decays though for lower resolutions probably due to the floating point calculation for the transition between the two steps. Although the method is an efficient tool to visualize huge volumetric data realistically.

Free-form non symmetrical surfaces (like in Figure 9) have virtually a different normal vector for each voxel allowing no kind of compacting in memory space. The choice of B-Spline surfaces for our test scenes was to show that the technique is not limited to planar figures. On the other hand working with high resolution discrete ray tracing can simplify complicate tasks related to curved surfaces. Examples are the visualization of intersections between surfaces or between surfaces and cut planes. Although this visualization is possible with standard ray tracing, it is not simple to put in evidence the intersection (with a different colour for example) or to obtain the points of the intersection curve. These tasks are straightforward using high resolution discrete space. Our ray tracing can visualize them realistically in a short time as shown in our examples.

The octree and the DDA introduced here are shown to be very convenient to be used in the context of the Discrete Ray Tracing.

References

[Akimoto *et al.*, 1991] T. Akimoto, Mase Kenji, and Suenaga Y. Pixel-selected ray tracing. *IEEE - CGA*, 6(4):14–22, 1991.

[Amanatides and Woo, 1987] John Amanatides and Andrew Woo. A fast voxel traversal algorithm for ray tracing. In *Eurographics'87*, pages 3–9, Amsterdam, August 1987. North Holland.

[Arvo and Kirk, 1987] James Arvo and David Kirk. Fast ray tracing by ray classification. *Computer Graphics*, 21(4):55–63, 1987.

[Fujimoto *et al.*, 1986] Akira Fujimoto, Takayaki Tanaka, and Kansei Iwata. Arts: Accelerated ray tracing system. *IEEE - CGA*, 6(4):16–26, 1986.

[Gargantini, 1982] Irene Gargantini. Linear octtrees for fast processing of three- dimensional objects. *Computer Graphics and Image processing*, 20(4):365–374, 1982.

[Gargantini, 1993] Irene Gargantini. Ray tracing an octree: Numerical evaluation of the first intersection. *Computer Graphics forum*, 12(4):199–210, 1993.

[Glassner, 1984] Andrew S. Glassner. Space subdivision for fast ray tracing. *IEEE - CGA*, 10(4):15–22, 1984.

[Heckbert and Hanraham, 1984] Paul S. Heckbert and Pat Hanraham. Beam tracing poligonal objects. *Computer Graphics*, 18(3):119–127, 1984.

[Ibaroudene and Acharia, 1991] Djaffer Ibaroudene and Raj Acharia. Coordenate relationships between vertices of linear octrees and corners of the universe. *Computer & Graphics*, 15(3):375–381, 1991.

[Rubin and Whitted, 1980] Steven M. Rubin and Turner Whitted. A 3-dimensional representation for fast rendering complex scenes. *Computer Graphics*, 14(1/2/3):110–116, 1980.

[Snyder and Barr, 1987] John Snyder and Alan Barr. Ray tracing complex models containing surface tessalations. *Computer Graphics*, 21(4):119–128, 1987.

[Stolte and Caubet, 1992] Nilo Stolte and René Caubet. Some more enhancements to ray tracing. In *Compugraphics'92*, pages 53–60, Lisbon, December 1992. Harold P. Santo.

[Stolte and Caubet, 1993] 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.

[Sung, 1991] Kelvin Sung. A dda traversal algorithm for ray tracing. In *Eurographics'91*, pages 73–85, Amsterdam, June 1991. North Holand.

[Whitted, 1980] Turner Whitted. An improved ilumination model for shaded display. *Communications of the ACM*, 23(6):343–349, 1980.

[Yagel *et al.*, 1992] Roni Yagel, Daniel Cohen, and Arie Kaufman. Discrete ray tracing. *IEEE - CGA*, 12(5):19–28, 1992.