A DDA Octree Traversal Algorithm for Ray Tracing

Kelvin Sung
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801
Email: ksung@cs.uiuc.edu

Abstract

A spatial traversal algorithm for ray tracing that combines the memory efficiency of an octree and the traversal speed of a uniform voxel space is described. A new octree representation is proposed and an implementation of the algorithm based on that representation is presented. Performance of the implementation and other spatial structure traversal algorithms are examined.

1 Introduction

It is well known that the bottleneck operation of a ray tracing program is the search for the first object intersected by a given ray [29]. Two approaches to alleviating this problem are bounding box hierarchies [28, 18, 12] and modeling space subdivision [10, 7]. Bounding box hierarchies take advantage of the object coherence and groups nearby objects into hierarchical structures to allow easy elimination of objects that are far away from a given ray. The major advantage of bounding box hierarchies is that for any given ray there will be at most one intersection calculation per object [4]. One major drawback of the approach is the lack of a good algorithm for building the hierarchies [12].

There are currently two major approaches to partitioning the modeling space: adaptive and uniform spatial subdivision. Glassner [10] uses an octree encoding technique [20] to subdivide the modeling space. The advantage of this approach is that the algorithm adapts to the object distribution in the modeling space and subdivides only where necessary. When tracing a ray through an octree structure, only objects in the octants that are touched by the ray need be tested for intersection. In this way, only the objects that are close to a given ray are tested for intersection. Glassner traverses the octree structure by advancing rays across the current voxel boundary, and locates the next voxel by performing a *GetVoxel* search from the root of the octree. There are two major inefficiencies:

1. Ray advancements involve distance and intersection calculations between a ray and three sides of an octant. Since these operations are in the inner-most loop of a ray tracer, it is desirable to simplify them.

2. Each *GetVoxel* is a $log(N)$-time operation where N is the depth of the octree.

Kaplan [17] subdivides the modeling space using a bintree structure. The characteristics of his algorithm are similar to those of an octree algorithm. It should be pointed out that Tamminen's EXCELL method [27] was one of the first application of spatial subdivision technique to ray tracing. However, since the EXCELL method describes an adaptively subdivided modeling space with a full voxel space directory table, it is not as memory efficient as the other adaptive subdivision approaches.

Fujimoto et al. [7] uniformly subdivide the modeling space into equal size voxels, and implement a *3 Dimensional Digital Differential Analyzer (3DDDA)* to traverse the voxel space. The advantage of this approach is that each step of the 3DDDA is simple and thus this approach traverses the voxel structure efficiently. This was compared to Glassner's ray advancing octree traversal algorithm and a speedup of 10 or more was reported. Amanatides and Woo [1] discussed an alternative approach to advancing rays through a uniformly subdivided space (a similar algorithm was independently proposed by Cleary and Wyvill [5]). Instead of identifying a driving axis, they treat each ray as a vector in the voxel space and step through voxels touched by the ray by performing conceptual vector additions. This is achieved by first calculating the initial ray position in a voxel cell and then calculating the distance (along a given ray) from the initial position to the voxel cell boundaries. In each step, the index corresponding to the axis with the minimum distance to the voxel boundary is incremented.

Uniform spatial subdivision allows efficient ray traversal. However, the memory inefficiency hinders the adoption of this technique for ray tracing complex scenes. An adaptive structure (e.g. octree) is more memory efficient, but this efficiency is gained at the expense of fast traversal. In this paper, we propose an algorithm that uses a uniform spatial subdivision ray traversal technique on an octree structure. The resulting system provides an efficient octree traversal alternative and at the same time maintains memory efficiency.

2 Related Work

Fujimoto et al. also modified their 3DDDA to operate on octree structures. They treated each octree level as a uniformly divided space and used a 3DDDA to traverse the child octants of that level. If the voxel returned by the DDA operation was not a leaf of the octree, the algorithm performed a *vertical traversal* to descend the octree. Vertical traversal requires saving of the current traversal status and initialization calculations in order to determine the child voxel that the ray would enter. Fujimoto et al. concluded that the vertical traversal is too slow and that the modified 3DDDA is not a desirable alternative [7].

Jevans and Wyvill [16] adaptively subdivide uniform voxel space during ray tracing, and use the [5] algorithm to traverse each sub-voxel space. The descending and ascending from a sub-voxel space has similar spirit as Fujimoto's vertical traversal. However, due to their more efficient algorithm, and the fact that each sub-voxel space may contain more spatial cells than the 2^3 octants in an octree's case, the short comings of Fujimoto's vertical traversal do not seem to be a problem here. As pointed out by the authors, this algorithm does not traverse an octree structure efficiently.

A voxel walking scheme has been proposed by various researchers [14, 3, 9] to improve the log(n)-time *GetVoxel* search of Glassner's approach. The algorithm is basically a combination of Glassner's ray advancement and Fujimoto's vertical traversal. Instead of performing the *GetVoxel* search from the root of an octree, the voxel walking scheme saves the current ray position and performs the *GetVoxel* search by treating the current node as a root. However, as discussed by Arvo in the Ray Tracing News, the recursive nature of the algorithm and the fact that the average octree depth is *usually* less then n, may cause the implementation of the octree walking scheme to be slower than that of Glassner's original approach.

Devillers described an algorithm to gather voxels of a uniformly subdivided space that does not contain any objects into Macro-regions [6]. Since Macro-regions do not contain any interesting information, they can be ignored during ray tracing. The use of Macro-regions improves on the uniform spatial subdivision approach when there are large empty volumes in the modeling space. This has the same spirit as employing a fast traversal algorithm in an adaptively subdivided modeling space: traversing rays rapidly through large empty spaces and concentrating on regions with interesting information. As in all uniform spatial subdivision algorithms, the Macro-regions approach suffers from the memory inefficiency problem.

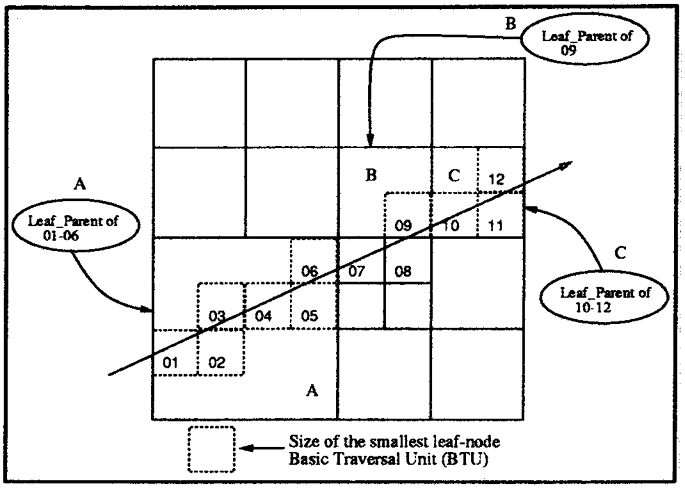

Figure 1: DDA on a Quadtree Structure.

3 Overview of the Algorithm

The idea of our algorithm is to use the smallest octree leaf as the base unit for the 3DDDA traversal. Conceptually, the modeling space is divided into a uniform voxel space. The modified 3DDDA operates in this *conceptual space* to locate the octree leaf needed for the ray-object intersection calculation. This differs from Fujimoto's [7] and Jevan's [16] approaches in that our algorithm does not require vertical traversal. It should be pointed out that our *smallest octree leaf* idea is similar to the *directory* of Tamminen's EXCELL method [27, 25, 26]. However, unlike the EXCELL method, which requires memory for the representation of each directory, our approach only stores the actual octree leaves.

Figure 1 uses a quadtree to illustrate the algorithm; it is straightforward to generalize this to an octree. In the case illustrated, the quadtree is defined by the solid lines and the dotted lines depict the conceptual uniform space in which the algorithm operates. *Basic Traversal Unit (BTU)* is the size of the smallest leaf node of the quadtree (or octree). Nodes 07 and 08 are examples of BTUs. A *leaf-parent* is a leaf node of the quadtree (or octree) which is the immediate parent of some BTU. For example, in Figure 1 node-A is a leaf of the quadtree and is also the leaf-parent of BTUs 01-06. The leaf-parent relationship also holds for node-B, BTU 09 and node-C, BTUs 10-12. Notice that nodes 07 and 08 are *both* leaf nodes *and* BTUs.

The algorithm uses a 3DDDA on a voxel space that is conceptually subdivided into equal size BTUs (as shown by the dotted lines in Figure 1). The conceptual space does not require any physical memory; only the actual octree structure must to be stored. For the quadtree example in Figure 1, there is memory associated with the area formed by the solid lines but no actual storage is necessary for the area formed by the dotted lines.

Figure 2 gives the basic algorithm. Applying the algorithm given in Figure 2 to the quadtree structure shown in Figure 1 (going through the outer-loop once) one obtains results as shown in Figure 3.

4 The LVSI Octree Representation

The success of the above algorithm is governed by the two operations in the loop (refer to Figure 2):

GetLeafParent This requires the ability to perform Basic Traversal Unit's (BTU) leaf-parent lookup efficiently.

```
  Octree_3DDDA_Traversal ( InputRay, OctreeStructure )
(1:) CurrentBTU = GetFirstBTU ( InputRay, OctreeStructure )
(2:) while ( CurrentBTU still in OctreeStructure )
(3:)     LeafParent  =  GetLeafParent ( CurrentBTU )

         /*
          * Process LeafParent normally
          */
(4:)     while ( CurrentBTU still in LeafParent )
(5:)         CurrentBTU = 3DDDA_Traversal
```

Figure 2: DDA Traversal of an Octree.

```
CurrentBTU  =  Node-01              /* First BTU  */
while ( CurrentBTU <= Node-12 )    /* While in OctreeStructure */
      LeafParent  =  Node-A         /* LeafParent of Node-01 */
       ... Process Node-A ...
      while ( CurrentBTU still in Node-A )
            CurrentBTU = 3DDDA_Traversal
            /*
             *    Must go through
             *    nodes 02, 03, 04, 05, 06
             */
```

Figure 3: Traversing the Figure 1 Quadtree.

While (in Leaf-Parent) This requires the ability to move through a given non-BTU octant rapidly.

In this section, we introduce and discuss the characteristics of the *Leveled Voxel Space Index (LVSI)* octree representation. In the next section, we will describe an implementation of the algorithm outlined in Figure 2 based on the LVSI representation and show that the above two operations can be supported efficiently.

Terminology

From here on, *voxel space* will refer to a uniformly subdivided modeling space that is made up of equal size *voxels* . An N-Level voxel space has 2^N voxels along each axis. The voxel index describes the position of a voxel in a given voxel space. This index ranges from 0 to $2^N - 1$ along each axis. For example, a level-2 voxel space has a total of 4^3 equal size voxels with indices ranging between 0 and 3 along each axis. *Octants* are the leaves of an octree. If a *Basic Traversal Unit (BTU)* is not an octant, it is called an ***abstract octant.*** An abstract octant always has a *leaf-parent* that is an octant of the octree.

Again, we use the quadtree as an example to explain the octree representation. The quadtree generalizes to an octree naturally. We first introduce an octant addressing scheme that maintains the correspondence between voxel space indices and octants. Then we discuss how this octree representation allows easy leaf-parent searching from an abstract octant.

4.1 LVSI Octant Encoding

A *Leveled Voxel Space Index (LVSI)* written, *L:(i, j, k),* identifies an L-th level octant by the voxel space index *(i, j, k)* of a level-L uniformly subdivided voxel space.

For example, the quadtree given in Figure 4 is represented by 10 LVSI octants[1] There are three first level octants with indices corresponding to the voxels of a Level-1 voxel space. These octants are: *1:(0,0), 1:(0,1), 1:(1,0)* . Similarly, there are three second level octants with indices corresponding to the voxels of a Level-2 voxel space. These octants are *2:(2,2), 2:(2,3), 2:(3,2)* . *3:(6,6), 3:(6,7), 3:(7,6), 3:(7,7)* are LVSI of the third level octants with indices that correspond to the voxels of a Level-3 voxel space. With this representation, the voxel space index alone does not uniquely identify any octant, the level information is a necessary part of the address.

[1] To avoid confusion, quadtree leaves are also referred to as octants.

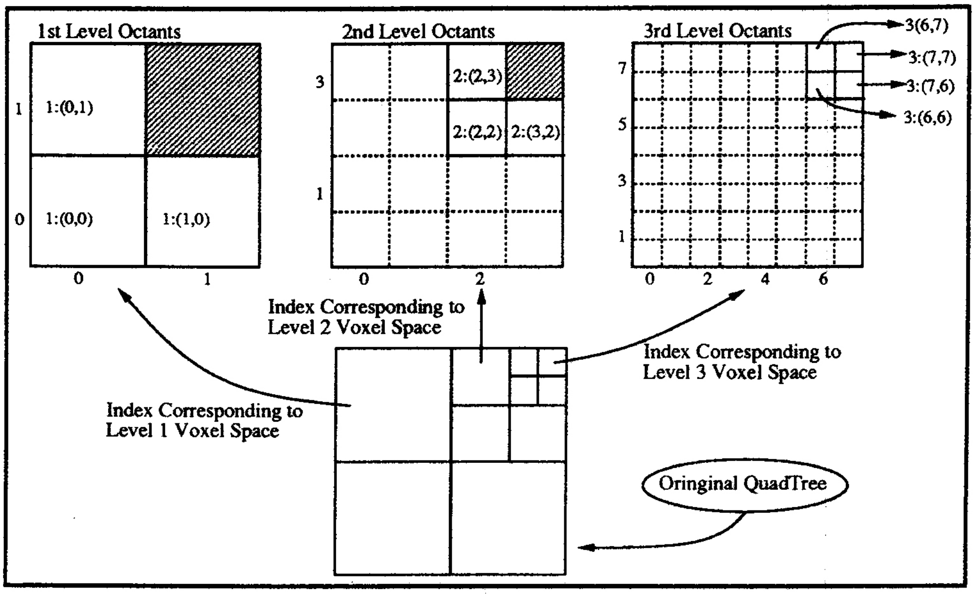

Figure 4: The LVSI Representation of a Quadtree.

4.2 Leaf-Parent Relationship

In the example of Figure 4, voxel 1:(1,1) is not an octant of the quadtree. It is the immediate parent of the three second level octants and the ancestor of the four third level octants. Notice the LVSI relationship between 1:(1,1) and second level octants is such that:

ParentLevel = ChildLevel - 1
ParentIndex = ChildIndex div 2 /* Integer Division */

Since each additional level of voxel space subdivision generates twice as many voxels on each axis and these additional voxels are formed by dividing along the medium point of each existing voxel, the above voxel-parent relationship is true in general. The parent of L:(i, j) always has an LVSI of (L-1):(i/2, j/2) [2]

Notice that the children voxel of L:(k,m) have indices:

(L+1):(2k, 2m)   (L+1):(2k+1, 2m)
(L+1):(2k, 2m+1)   (L+1):(2k+1, 2m+1)

This relationship is also true in general. For example, the children of (L+1):(2k, 2m) has indices:

(L+2):(2*2k, 2*2m)   (L+2):(2*(2k)+1, 2*2m)
(L+2):(2*2k, (2*(2m))+1)   (L+2):(2*(2k)+1, (2*(2m))+1)

5 Implementation

We have implemented the algorithm outlined in Figure 2 based on an LVSI octree representation using Amanatides and Woo's fast voxel traversing algorithm [1]. As in [1], our implementation can be applied to octrees with rectangular octants.

[2] Integer division can be implemented as a right shift.

```
/*
 * Return :  the leaf-parent of Level:(i,j,k)
 *
 * Notice that the input Level, i, j and k are changed
 * by this function. When the function returns,
 * Level:(i,j,k) is the LVSI of the leaf-parent.
 */
GetLeafParent ( Level, i, j, k )

   LeafParent = HashTableLookup ( Level, i, j, k )

   while ( LeafParent == NULL )
      Level = Level - 1
      i = i >> 1  /* This is division by 2 */
      j = j >> 1
      k = k >> 1
      LeafParent = HashTableLookup (Level, i, j, k)

   return (LeafParent)
```

Figure 5: The GetLeafParent Procedure.

```
SkipParentOctant ( Parent, d_level )

   while ( TRUE )

     if ( ( Parent_i  XOR ( BTU_i >> d_level ) or

          ( Parent_j  XOR ( BTU_i >> d_level ) or

          ( Parent_k  XOR ( BTU_k >> d_level ) )

            return

     else BTU = DDAStep()
```

Figure 6: The SimpleSkipParent Procedure

5.1 Octree Storage

Since an octree is uniquely represented by the LVSI encoded octants, we only need to store the leaf-octants. A hashing approach similar to the ones described by [8, 10, 30] can be employed to store the octants. In our implementation, for a given cell L:(i,j,k) the hash function is defined to be:[3]

$$(i \gg 1) \mid ((j \gg 1) \ll L) \mid ((k \gg 1) \ll 2L) \ \mathbf{mod}\ \mathit{tableSize}$$

Notice that the least significant bit of the input index is not used in the hash function calculation. As in [10], we store the subdivided child octants of the same parent in one of the eight slots in the same hash entry. The binary number formed by the least significant bits of the input index is used to identify one of these slots. It should be pointed out that in the example given in Figure 4, we only need to store the 10 leaf octants.

5.2 GetLeafParent

With the LVSI representation and the described octant storage scheme, the *GetLeafParent* operation can be simply stated as in Figure 5. Notice that this procedure is *Line* **(3:)** of Figure 2.

5.3 SkipLeafParent

According to the algorithm listed in Figure 2, after processing the leaf-parent, we need to advance across it (*Lines* **(4:)** *and* **(5:)**). From the discussion in Section 4.2 we see that given an LVSI of a BTU

$$BTU_{level} : (BTU_i, BTU_j, BTU_k),$$

and the LVSI of its leaf-parent

$$Parent_{level} : (Parent_i, Parent_j, Parent_k),$$

and defining

$$\delta_{level} = BTU_{level} - Parent_{level},$$

[3] "≪" and "≫" are left and right shift symbols, and "|" is the symbol for logical or.

```
ComplexSkipParent( $d_{level}$ ) {

    xStepToParentBoundary = CalculateStep( $d_{level}$, $BTU_i$ )
    yStepToParentBoundary = CalculateStep( $d_{level}$, $BTU_j$ )

    xDistToParentBoundary = xStepToParentBoundary * tDeltaX
    yDistToParentBoundary = yStepToParentBoundary * tDeltaY

    distX = tMaxX + xDistToParentBoundary
    distY = tMaxY + yDistToParentBoundary

    if (distX < distY)
        /*
         * Parent boundary will be touched by $BTU_i$
         */
        yStep = distX / tDeltaY
        xStep = xStepToParentBoundary
    else
        /*
         * Parent boundary will be touched by $BTU_j$
         */
        xStep = distY / tDeltaX
        yStep = yStepToParentBoundary

    ComplexDDAStep( xStep, yStep )
```

Figure 7: The ComplexSkipParent Procedure.

then, as long as[4]

$$((Parent_{axis} \oplus (BTU_{axis} \gg \delta_{level})) == 0)$$

the BTU_{axis} is within the $Parent_{axis}$'s octant limit, where *axis* is index i, j or k. Knowing this relationship, the *SkipParent* operation (*Lines* (4:) *and* (5:) of Figure 2) can be implemented as shown in Figure 6.

The procedure outlined in Figure 6 advances across the leaf-parent boundary by going through iterations of DDASteps. This becomes unacceptable when δ_{level} is relatively large. For example, a δ_{level} of 6 could potentially result in 2^6 DDAStep iterations. Clearly, when δ_{level} is large, we need a more intelligent way to skip across the leaf-parent.

Defining dTable as,

δ_{level}	0	1	2	3	4	5	6	7	8
	0	1	3	7	F	1F	3F	7F	FF

(e.g. $dTable[5] == 1F$, table content is in hexadecimal) then for any BTU, it is *MinStep* away from the minimum leaf-parent boundary and *MaxStep* away from the maximum leaf-parent boundary. Where

$$MinStep_{Axis} = BTU_{Axis} \,\&\, dTable[\delta_{level}]$$

and

$$MaxStep_{Axis} = dTable[\delta_{level}] - (BTU_{Axis} \,\&\, dTable[\delta_{level}]).$$

Recall that the fast voxel traversal algorithm of [1] records the distance to the current voxel boundary in $tMax_{axis}$ and the distance traveled by a change in voxel index in $tDelta_{axis}$. Given these relationships, a more intelligent SkipParent routine can be implemented as shown in Figure 7[5].

[4] $\oplus$ is exclusive-or.

[5] To conserve space, this routine is outlined for the 2D case, it is straightforward to generalize to 3D.

The *StepToParentBoundary* of Figure 7 is in BTU unit, while the *DistToParentBoundary* is the actual distance that would be traveled. Depending on the direction of the ray, the *CalculateStep()* routine in Figure 7 is either the $MinStep_{Axis}$ or the $MaxStep_{Axis}$. Instead of the simple DDAStep as listed in [1], the *ComplexDDAStep* of Figure 7 updates each voxel index by the input $step_{axis}$ and the $tMax_{axis}$ is updated by $step_{axis} \times tDelta_{axis}$.

ComplexSkipParent of Figure 7 involves more computation than the SimpleSkipParent of Figure 6. In our implementation, SimpleSkipParent is invoked to skip leaf-parents with δ_{level} of less than 3.

6 Results

For comparison purpose we have developed all four modeling space traversal algorithms:

DDA This is the fast voxel traversal algorithm described in [1]. The run time of all algorithms are compared against this, since it should be the fastest algorithm available.

GLAS This is the original octree traversal algorithm proposed by Glassner [10]. Our implementation deviates from his in that, instead of storing octants in a hash table, the octree structure is represented by a hierarchy of pointers. This is somewhat less memory efficient but should result in a faster system.

ARVO This is Arvo's linear octree walking algorithm [3]. We have implemented this by modifying the *GetVoxel* routine of our GLAS implementation to start the next octant search from the parent of the current octant. As suggested by Arvo [3], in our implementation, an explicit stack is maintained to avoid recursive calls and we "*longjump*" out of the recursion once an intersection is found.

DDAOCT This is the algorithm proposed in the paper.

All four algorithms are developed on the same global illumination ray tracer[6]. The system is written in C++, and has a set of well defined component interface [23]. As a result, the implementation of the four algorithms have identical interfaces to the base system and can be replaced transparently. In this way, we are able to isolate the changes and examine the difference in run time resulting from employing different traversal algorithms. We have also implemented the mail box idea as proposed by [1, 2] for all four approaches.

Timing

We have used the *sphereflake* (Sphere), the *tetrahedral pyramid* (Tetra) and the *fractal mountain* (Mount) environments as proposed by Haines [13] and a *radiosity scene* (Room) to test the above algorithms. All images (Figures 11 and 12) are rendered at 512x512 resolution with 4 sample rays per pixel. The timing presented below are in CPU-seconds, based on the IBM RS/6000 model 320 with 32 megabytes of main memory and the IBM RS/6000 model 540 with 64 megabytes of main memory. Due to memory limitation, all three octree based algorithms subdivided the Sphere and Tetra into level 8 octants, while the Room environment is rendered at octree level 4. We were able to subdivide the Mount environment into level 10 octants. The DDA algorithm was only able to subdivide the Sphere, Tetra, and Mount environments into 30x30x30 voxel space before we ran out of memory. As in octree algorithms, DDA rendered the Room environment in 16x16x16 voxel space.

Figure 8[7] shows the run-time object and spatial cell distribution statistics. In our implementation, GLAS, ARVO, and DDAOCT have identical octree subdivision routines and thus have the same statistics. From

[6] The original system and the GLAS algorithm was developed by Peter Shirley [22].

[7] The "*Cells*" in Figure 8 are referring to "*Spatial Cells*". They can either be octants of an octree or voxels of a uniformly subdivided voxel space.

Spatial Structure	Scene	Total Number of Spatial Cells	Total Number of Objects	Max Number of Objects in a Cell	Average Number of Objects in a Cell	Average Number of Cells an Object Spans	% of Objects Spanning 1 or 2 Cells
Voxel Subdivision	Sphere	27000	7388	1178	1.29	1.08	99.72%
	Tetra	27000	4098	125	31.92	2.02	86.12%
	Room	4094	710	17	4.53	10.27	9.01%
	Mount	27000	8198	5450	8.06	1.26	99.84%
Octree Subdivision	Sphere	2549	7388	45	6.23	1.81	91.09%
	Tetra	23500	4098	13	4.65	16.80	0.24%
	Room	2486	710	17	4.73	9.74	7.75%
	Mount	10991	8198	27	8.32	6.77	46.83%

Figure 8: Object and Spatial Cell Distributions in the Environments.

Figure 8, we see that uniform subdivision does not utilize the system resources efficiently for complex environments. For example, in the Sphere environment, the average number of objects in a voxel is 1.29 while there are voxels containing more than 1000 objects. Another observation is that the mail box approach may not work well in scenes where the majority of the objects do not span more than two spatial cells. Finally, we observe that the Mount environment contains the most number of objects; however, due to its object distribution, it could be subdivided into finer octants than that of Tetra and Sphere. It is also interesting to see that although the Tetra environment contains about half the number of objects of either Sphere or Mount, at level 8 subdivision, there are about twice as many octants (23500 octants) when comparing to the level 10 subdivision of the Mount environment (10991 octants).

With the above observations, let us examine the execution time. All four algorithms (Normal) and their corresponding mail box enhancements (Mail Box) are run against the four test environments.

Algorithm	Mode	Sphere	Tetra	Mount	Room
DDA	Normal	xxxxx	13428	xxxxx	12826
	Mail Box	xxxxx	14626	xxxxx	10451
GLAS	Normal	35632	6564	43095	15974
	Mail Box	34486	6637	45810	13009
ARVO	Normal	41050	7597	49018	16985
	Mail Box	40519	7549	51166	14583
DDAOCT	Normal	33290	5778	43368	13580
	Mail Box	32073	6194	46645	11793

DDA Uniform spatial subdivision works well when there are sufficient memory (Room). Notice that the timing statistics for the Sphere and Mount environments are absent. This is because, we have stopped the renderering process after a few CPU-hours with less than 10% of the image being generated. The extreme slow speed may be a result of processing the few voxels that contained a large number of objects (for both cases there are a few voxels containing more than 1000 objects). The slow rendering speed of the Tetra environment can be explained by the high average number of objects per voxel (refer to Figure 8, on the average, there are 31.92 objects in each voxels).

ARVO Surprisingly, this is always slower than GLAS. We believe the reason is that while ARVO improves the speed to *descend* an octree, it takes extra time to *ascend* one. For example, it takes longer for our ARVO implementation to traverse from a level 5 octant to a level 1 octant because it needs to ascend the octree. Apparently the complication in the recursive nature of the algorithm and the extra time needed to ascend an octree out weighted the improvements.

DDAOCT Somewhat surprisingly, this runs a little slower than GLAS for the Mount environment. We believe the reason behind this is that when subdivision level becomes too high (Mount was rendered at octant level 10) the complexity of the ComplexSkipParent routine eventually offsets the speed gained in the simplicity of the underlying DDA algorithm. We will discuss this issue further in the *Algorithm Sensitivity* section. At lower subdivision levels, DDAOCT is consistently faster than GLAS. However, no dramatic speedup can be observed (speedup observed ranges from 7.04% to 17.63%). One reason is that as the ray trace programs become more sophisticated, a smaller portion of the total rendering time is spent in the traversal of the spatial structure. Thus, solely improving the spatial structure traversal time cannot achieve drastic speedup in overall system execution time. From the source code profiling, we found that on the average the DDAOCT octree traversing speed is twice that of GLAS (between subdivision levels 4-8). However, since our GLAS implementation spends about 30% of the total rendering time in traversing the octree structure, doubling the speed can only achieve an average of 15% speedup.

Mail Box Significant speedup is observed for the Room environment only. This is somewhat expected from the object distribution statistics as shown in Figure 8. When the size of the objects in a scene is *small* in comparison to the size of the spatial cells, the probability of objects spanning multiple spatial cell boundaries also becomes small. The overhead involved in performing the pre-intersection checking and post-intersection information recording eventually offsets the time saved in avoiding the *small number* of multiple intersection calculations. It is interesting to note that, when subdivided by an octree structure, the Tetra environment produces a relatively large number of objects spanning multiple spatial cells (refer to Figure 8, on the average, an object in the Tetra environment spans 16.80 octants). Yet, the mail box enhancement still slows down the GLAS and DDAOCT rendering time. We believe this is because the non-reflecting, non-transparent primitives of the Tetra environment obstruct the spatial cells they spanned from the viewer. As a result, most of these spatial cells are never traversed during ray tracing.

Algorithm Sensitivities

Figure 9 shows the execution time of the four algorithms (without mail box enhancements) rendering the Room environment with different levels of subdivisions. At low spatial subdivision levels (level 2-3), all algorithms are slow because of the large number of objects contained in each spatial cell. However, the execution time does not always decrease as the level of subdivision increases. This is because after certain optimum level (level 4, in this case), further subdivision does not reduce the number of candidate primitives tested for intersections, and the traversal of the more finely subdivided structure would require more time. The sharp rise in execution time of DDA after level 4 subdivision reflects the exponential increase in initialization time. The somewhat sharp increase in execution time of ARVO reflects the fact that our implementation of the algorithm takes extra time to ascend the octree structure. We believe the faster rising in execution time of DDAOCT (sharper slope, when comparing to that of GLAS) is a result of the computational intensive ComplexSkipParent procedure.

From Figure 9, it seems that DDAOCT degrades rapidly after level 5 subdivision. However, we note that DDAOCT renders both the Sphere and Tetra environments faster than GLAS at level 8 subdivision. We believe what happens here is that DDAOCT degenerates faster than GLAS *only after* the optimum subdivision level. This can be verified by Figure 10. We see that both algorithms speed up considerably as the level of subdivision increases. Notice, unlike Figure 9, the speed up of DDAOCT over GLAS at lower subdivision levels (levels 7 and 8) cannot be observed in Figure 10. This is because at lower subdivision levels, the execution time of the Mount environment is dominated by the processing of the octants that contain large number of objects. Thus the speed-up gained in the octree traversal becomes relatively insignificant.

For the Mount environment, we believe level 10 is not yet the optimum subdivision level (this can be deduced by observing the continuing decrease in execution time of Figure 10). Level 10 octree subdivision corresponds to a possible maximum of 2^{10} (or 1024) octants on each axis. This means there can be a possible maximum of 1024^3 octants. Extreme high levels of octree subdivision should be avoided for the high storage requirement and the traversal time of the finely subdivided octree may become a significant portion of the total execution time. As suggested by Kirk and Arvo [19], it may be desirable to group nearby objects of a scene into different octrees and adopt a hybrid approach to overcome this

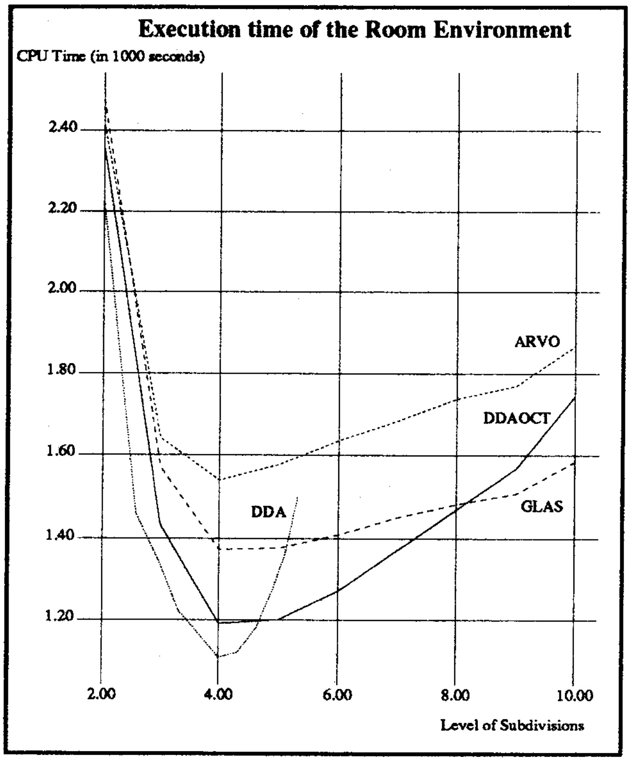

Figure 9: Algorithm Sensitivities (Room)

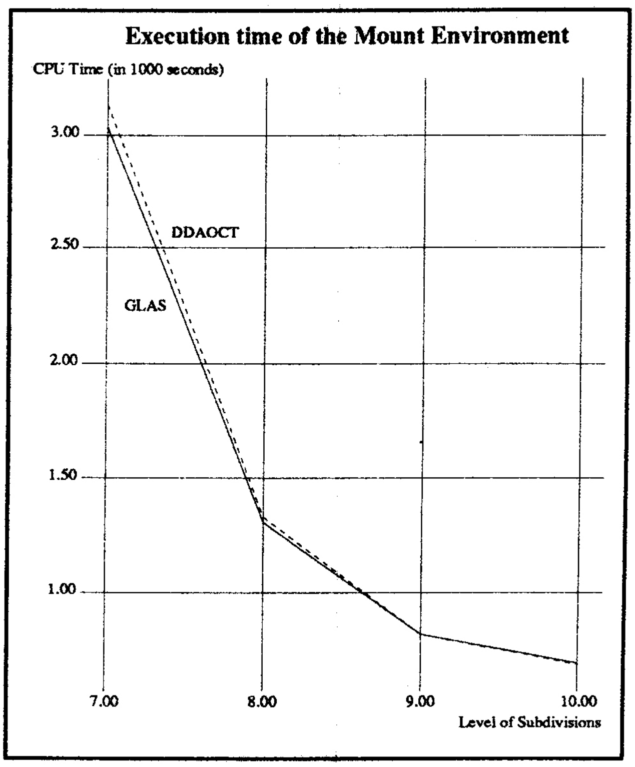

Figure 10: Algorithm Sensitivities (Mount)

shortcoming. For the Mount environment example, we can defined five bounding volumes: four for each of the spheres, and the fifth volume would bound an octree structure describing the fractal mountain itself.

All the spatial subdivision algorithms we have investigated in this paper are sensitive to optimum level of subdivision; which in turn, is highly dependent on the object distribution in a given environment. From our experience, it is worthwhile to examine the statistics of object distribution before starting a time consuming ray tracing process (imagine waiting for the DDA algorithm to trace the Sphere or Mount environment without the knowledge that some voxels contain more than 1000 objects). We propose the following as guidelines for ray tracing with spatial subdivision:

1. Keep the maximum number of objects in a spatial cell to less than 20 (if possible). From the DDA traversal of the Sphere and Mount environment experience, we know this is very important.
2. Keep the average number of objects in a spatial cell to about 10. From Figure 9, we realize that it may not be a good idea to minimize this number by extensive subdivision.
3. Use the mail box enhancement only if the average number of spatial cells spanned by objects are large (e.g. > 10). However, from the Tetra example, one should note that for some environments the mail box idea simply does not work well.
4. Use the DDA traversal algorithm if the environment is subdivided into relatively low level of spatial cells (e.g. < 5). DDAOCT should be used at subdivision levels between 5 and 9. GLAS seems to work the best at subdivision levels of 10 or more. However, one may consider adopting the hybrid approach as suggested by Kirk and Arvo [19] at very high subdivision levels (e.g. > 10).

We note that all the numerical data mentioned are from our experience. Alternative implementations of the algorithms and more extensive experiments with different environments are needed before this could be conclusive.

7 Conclusion

We have introduced an algorithm which combines the memory efficiency of an octree structure and the traversal speed of a uniformly subdivided voxel space. The LVSI octree representation is introduced to provide support for the implementation of the algorithm.

Detailed testings of different spatial structure traversal algorithms showed that the implementation of our proposed algorithm results in a faster octree based system when subdivision level is not too high (< 10). We believe very high level of subdivisions should be avoided by adopting the hybrid approach as proposed by Kirk and Arvo [19]. Some surprising results are revealed in the testing process. For example, mail box idea works well in some environments but may cause the rendering time of the other environments to increase. Another interesting observation is that our octree walking implementation dose not result in a faster system. It should be pointed out that, there are other approaches to implementing this algorithm (for example, see [14, 15]).

One important issue that has been left out in our discussion is the maximum number of allowed objects in an octant. This information together with the maximum allowed level of subdivision steer the construction of an octree structure. However, unlike the maximum allowed subdivision level, altering the maximum number of allowed objects in an octant does not produce a clear effect on the system execution time. We are still examining this issue.

Since DDAOCT is basically a DDA with two additional routines, the GetLeafParent and the SkipLeafParent, a spatial subdivision system should implement both algorithms and invoke the more efficient one during run time. If the subdivision level is small (e.g. $<= 5$), uniform spatial subdivision should be adopted. In this case, the GetLeafParent procedure should return the current voxel index, and the SkipLeafParent procedure becomes an NULL function. When the level of subdivision is large, octree structure should be adopted, and DDAOCT can be invoked to traverse the octree.

When subdivided by a uniform voxel space, some environments produce a large number of empty voxel cells. These empty voxels are similar to the zeros in a sparse matrix [24] because they do not contain any useful information. It is interesting to investigate the possibilities of adopting a sparse matrix storage and retrieval strategy for the storage of the non-empty voxels. In this way the memory inefficiency of the uniform spatial subdivision approach can be alleviated.

Finally, there has been work done in investigating ways to combine spatial subdivision and bounding volume hierarchies approaches [11, 21]. The algorithm presented in this paper does not interfere with the combination effort, it is merely an alternative way for traversing an adaptively subdivided structure.

8 Acknowledgments

Thanks to Peter Shirley for allowing me to try out my algorithm on his ray tracing system, general discussion and countless helpful suggestions; to Greg Rogers for fruitful discussions on the idea and guidance; to Professor Claude Puech, who's solid modeling class and patient explanations helped motivate this project; to Frederik Jansen for his detailed comments and suggestions that made this a much better paper; to William Brown for his critical comments on the initial testing images; to Andrew Woo for helpful discussions on the implementation of his algorithm; special thanks to Professor William Kubitz, who is the advisor for this project; and, last, but not least, thanks to IBM-TCS for providing the wonderful RS/6000 systems, without which none of these work would be possible.

References

[1] John Amanatides and Andrew Woo. A fast voxel traversal algorithm for ray tracing. In *Eurographics '87*, 1987.

[2] Bruno Arnaldi, Thierry Priol, and Kadi Bouatouch. A new space subdivision method for ray tracing csg modelled scenes. *Visual Computer*, 3:98–107, 1987.

[3] James Arvo. Linear-time voxel walking for octrees. *Ray Tracing News*, 1(2), March 1988. e-mail Edition, Available under anonymous ftp from drizzle.cs.uoregon.edu.

[4] James Arvo and David Kirk. A survey of ray tracing acceleration techniques. In Andrew S. Glassner, editor, *An Introduction to Ray Tracing*. Academic Press, San Diego, CA, 1989.

[5] John G. Cleary and Geoff Wyvill. Analysis of an algorithm for fast ray tracing using uniform space subdivision. *Visual Computer*, 4:65–83, 1988.

[6] Olivier Devillers. The macro-regions: an efficient space subdivision structure for ray tracing. In W. Hansmann, F. R. A. Hopgood, and W. Strasser, editors, *Eurographics '89*, pages 27–38. Springer-Verlag, North-Holland, 1989.

[7] Akira Fujimoto, Takayu Tanaka, and Kansei Iwata. Arts: Accelerated ray-tracing system. *IEEE Computer Graphics and Applications*, pages 16–26, April 1986.

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

[9] Andrew Glassner. Implementation notes for ray tracers. *Advanced Topics in Ray Tracing*, 1990. ACM Siggraph '90 Course 24 Notes.

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

[11] Andrew S. Glassner. Spacetime ray tracing for animation. *IEEE Computer Graphics and Applications*, 8(2):60–70, March 1988.

[12] Jeffery Goldsmith and John Salmon. Automatic creation of object hierarchies for ray tracing. *IEEE Computer Graphics and Applications*, pages 14–20, May 1987.

[13] Eric A. Haines. A proposal for standard graphics environments. *IEEE Computer Graphics and Applications*, 7(11):3–5, November 1987.

[14] Frederik W. Jansen. Data structures for ray tracing. In L. R. A. Kessener, F. J. Peters, and M. L. P. van Lierop, editors, *Data Sturctures for Raster Graphics*, pages 57–373. Springer-Verlag, Netherlands, 1986.

[15] Frederik W. Jansen. Re: Linear-time voxel walking for octrees. *Ray Tracing News*, 1(4), April 1988. e-mail Edition, Available under anonymous ftp from drizzle.cs.uoregon.edu.

[16] David Jevans and Brian Wyvill. Adaptive voxel subdivision for ray tracing. In *Graphics Interface '89*, pages 164–172, 1989.

[17] Michael R. Kaplan. Space-tracing, a constant time ray-tracer. *State of the Art in Image Synthesis*, July 1985. ACM Siggraph '85 Seminar Notes.

[18] Timothy L. Kay and James T. Kajiya. Ray tracing complex scenes. *Computer Graphics*, 20(4):269–278, August 1986. ACM Siggraph '86 Conference Proceedings.

[19] David Kirk and James Arvo. The ray tracing kernal. In *Proceedins of Ausgraph*, pages 75–82, July 1988.

[20] Donald Meagher. Geometric modeling using octree encoding. *Computer Graphics and Image Processing*, 19:129–147, 1982.

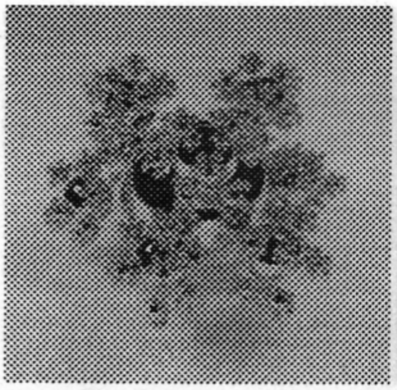
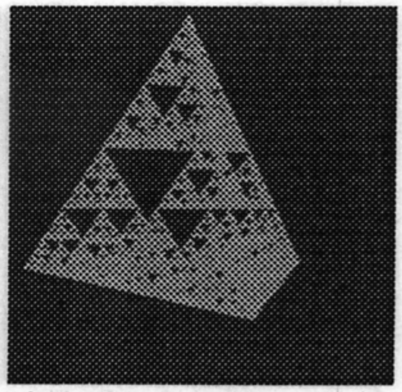
Figure 11: The Sphereflake (Sphere) and The Tetrahedral Pyramid (Tetra)

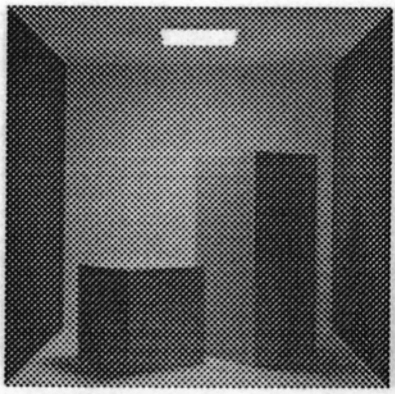
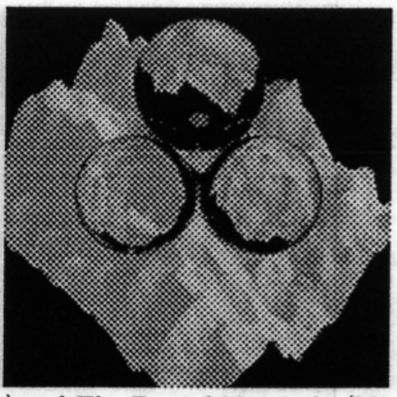
Figure 12: The Radiosity Scene (Room) and The Fractal Mountain (Mount)

[21] Isaac D. Scherson and Elisha Caspary. Data structures and the time complexity of ray tracing. *The Visual Computer*, 3(4):201–213, December 1987.

[22] Peter Shirley. *Physically Based Lighting Calculations for Computer Graphics.* PhD thesis, University of Illinois at Urbana-Champaign, November 1990.

[23] Peter Shirley, Kelvin Sung, and William Brown. A raytracing framework for global illumination systems. *Graphics Interface '91*, June 1991.

[24] Harold S. Stone. *High-Performance Computer Architecture.* Addison-Wesley, Reading, Massachusetts, 1987. pp. 266-268.

[25] Markku Tamminen. The extendible cell method for closest point problems. *BIT*, 22:27–41, 1982.

[26] 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.

[27] Markku Tamminen and R. Sulonen. The excell method for efficient geometric access to data. In *ACM Nineteenth Design Automation Conference*, pages 345–351, 1982.

[28] Hank Weghorst, Gary Hooper, and Donald P. Greenberg. Improved computational methods for ray tracing. *ACM Transaction on Graphics*, 3(1):52–69, January 1984.

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

[30] Brian Wyvill. 3d grid hashing function. In Andrew S. Glassner, editor, *Graphics Gem.* Academic Press, San Diego, CA, 1990.