MKtree : Generation and Simulations

Marta Franquesa–Niubo Pere Brunet

January 23, 2002

Abstract

The problem to represent very complex systems has been studied by several authors, obtaining solutions based on different data structures. In this paper, a K dimensional tree (Multirresolution Kdtree, MKtree) is introduced. The MKtree represents a hierarchical subdivision of the scene objects that guarantees a minimum space overlap between node regions. MKtrees are useful for collision detection and for time–critical rendering in very large environments requiring external memory storage. Examples in ship design applications are described.

1 MKtrees

1.1 Introduction

In this paper, we introduce a new bounding volume hierarchy, the *Multirresolution Kdtree* (MKtree), to represent very complex systems. This kind of tree is generated taking advantage of two partitioning criteria: space partition and scene objects partition. Thereby, our MKtree can be considered as an hybrid between Kdtrees and R–trees. Our method precomputes and automatically stores different levels of detail of objects and groups of objects when constructing the tree.

Our MKtree incorporates the following features:

- The MKtree represents a hierarchical subdivision of the scene objects that guarantees a minimum space overlap between node regions.
- Levels of detail (LOD) with bounded tolerance are supported at every hierarchical level of the tree.
- Bounding approximations of objects and groups of objects are used
- The method has been conceived to manage memory efficiently by developping external memory based algorithms and it is therefore useful for collision detection in large environments and for time critical–rendering, for instance.

The particular structure of the MKtrees is specially well suited for collision and proximity detection, using external memory in very large virtual environments. MKtrees are also useful for frustum–based collision detection: on–line collision detection during navigation through the virtual model.

1.2 Related Definitions

Before introducing the MKtrees, let us start with some related definitions:

Definition 1.2.1 *The environment is defined as a collections of 3D objects that are usually polyhedra, even though they can be a set of more general models. Thus,*

$$S = \{o_1, .., o_N\}$$

Definition 1.2.2 *We name $R(S)$ the axis–aligned bounding box, AABB, of all the set S. In general $R(S_n)$ is the AABB of a subset of objects of S: S_n. Where $S_n \subset S$.*

Definition 1.2.3 *$xOverlap(R(S_1), R(S_2))$ is the length of the intersection of the one–dimensional x projections of $R(S_1)$ and $R(S_2)$, say $xProj(R(S_1))$ and $xProj(R(S_2))$. In other words,*

$$xOverlap(R(S_1), R(S_2)) = length(xProj(R(S_1)) \cap xProj(R(S_2)))$$

The functions yOverlap and zOverlap are defined in a similar way.

Assumption: There exists a boolean function $InPage(S_n)$ that returns true if the complete geometry of S_n fits in one disk block (and can be retrieved to the core memory using a single disk paging operation) and false otherwise.

1.3 Description

Following a similar nomenclature of [KHM+98, He99] a $MKtree$ is a tree, $MKT(S)$, that specifies a bounding volume hierarchy on S. S is a set of objects. Each node, n, of $MKT(S)$ corresponds to a subset, $S_n \subset S$, with the root being associated with the full set S. Each internal node has two children. Thus, the MKtree is a binary tree. The union of the two subsets associated to the children of n is equal to S_n. Each node is also associated to the AABB box of S_n: $R(S_n)$.

A basic issue that is directly related to the performance of a MKtree is the selection of the *splitting rules* when building the hierarchy. The main goal is that during tree construction we would like to assign subsets S_{n_1} and S_{n_2} of objects to each child of a node, n, in such a way to minimize the probability that their $R(S_{n_1})$ and $R(S_{n_2})$ intersect. Let us assume that we have a procedure, *xDivideList*, that divides the collection of objects belonging to S_n in two subsets, S_{n_1} and S_{n_2}, minimizing the $xOverlap(R(S_{n_1}), R(S_{n_2}))$. Let us assume, also, that we have similar procedures for the y and z dimensions, *yDivideList* and *zDivideList*.

Now we can define a node, n, of a MKtree corresponding to a subset S_n:

- If $InPage(S_n)$, then n is a leaf node and stores the *geometry* of objects in S_n
- If *no* $InPage(S_n)$, then n is an internal node with two pointers to n_1 and n_2, respectively. S_{n_1} and S_{n_2} are obtained by using *wDivideList* (where w can be x, y or z) with w being such that $wOverlap(R(S_{n_1}), R(S_{n_2}))$ is equal to:

$$\begin{gathered}Min(xOverlap(R(S_{n_1}), R(S_{n_2})),\\ yOverlap(R(S_{n_1}), R(S_{n_2})),\\ zOverlap(R(S_{n_1}), R(S_{n_2})))\end{gathered}$$

Note: The overlap_band is the value of $wOverlap(R(S_{n_1}), R(S_{n_2}))$

In this way the tree construction procedure automatically generates a subdivision and a hierarchy of all objects of S with a minimum overlap. Groups of objects are automatically generated (like in [WLML99]) by using, here, a minimum–overlap criterium. As it will be seen in the result sections (sec. 3.1.1 and 3.2.1)), when dealing with virtual ship environments, in many cases *overlap_band* is *null*. In these cases, our building tree method is at the same time an space and object partitioning method. Thereby, our MKtree can be considered as an intermediate model between *Kdtrees* [DECM98, Sam90] and *Rtrees* [SRF87, BKSS90, Gut84] in 3D.

1.4 Specification

The bounding approximations of objects, b, are described in this section. It is desirable that these class of shapes accomplish some properties:

1. The bounding approximation of an object o_i must fulfill $o_i \subset b(o_i, k)$, (where k is the level of detail).
2. The bounding approximation $b(o_i, k)$ must approximate objects depending on a desired resolution LOD, k.

3. The volume distance ($vDist$) [ABA01] between the geometry of an object, o_i, and its bounding approximation associated to one specified level of detail k, $b(o_i, k)$, has to satisfy to be less than a known geometric threshold ε_k. In other words,

$$\forall p \in o_i \;\; \exists \; p\prime \in b(o_i, k) \mid dist(p, p\prime) < \varepsilon_k \;\; and,$$
$$\forall p\prime \in b(o_i, k) \;\; \exists p \in o_i \mid dist(p\prime, p) < \varepsilon_k$$

We have chosen as bounding volumes surrounding polyhedra with small number of faces [And99, AAB99]. Each object is modeled by an offset polyhedron generated by topology simplification [ABA01]. In fact, each object, o_i, and each group of objects, S_n, have a set of bounding approximations, one for each level of detail, k. Ranging k from 1 to M. The approximation error, ε_k, that restricts the volume distance between bounding approximation of an object and its geometry, $vDist(b(o_i, k), o_i)$, is decreasing while k is increasing. In other words: $b(S, k+1)$ is more accurate than $b(S, k)$. $Faces(b(S, k))$ is the number of faces of the bounding approximation of S at k level of resolution. On the other hand, the set of bounding approximations of objects $(o_i, ..., o_j)$ that belong to a subset S_n, will be represented by $B(S_n, k)$. Thus,

$$B(S_n, k) = \{b(o_i, k), ..., b(o_j, k)\}$$

Figure 1 shows a simple example in 2D of $b(S_n, k)$ and $B(S_n, k)$. Observe that $b(S_n, k)$ corresponds to the bounding polyhedron of a pipe (red color) and $B(S_n, k)$ corresponds to the set of bounding approximations of the set of the pipe elements (green color). Figure 2 presents an other example in 3D where the $b(S_n, k)$ is the bounding approximation of a whole set of pipes (in a ship environment).

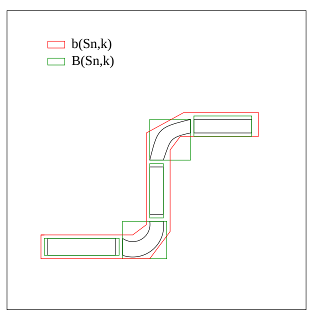

Figure 1: Example of $b(S_n, k)$ and $B(S_n, k)$ in 2D

Coming back to the specification of the MKtree, its data structure must be designed to efficiently return the approximations $b(o_i, k)$ and $B(S_n, k)$. Moreover, in very large models, the number of disk block accesses must be minimized during the retrieval of this information. This must be guaranteed by a suitable splitting rules and page structure of the MKtree. As a consequence, the basic queries to the MKtree will be the following:

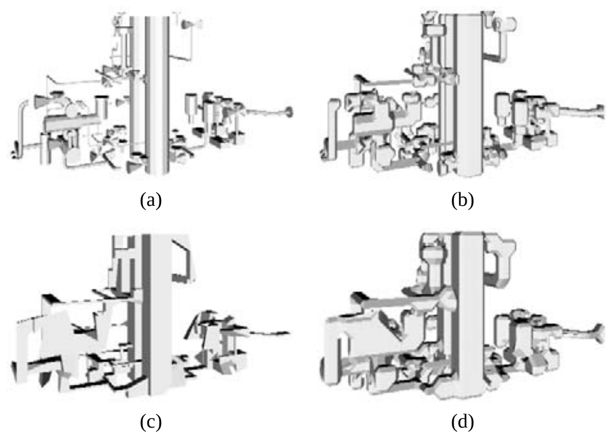

Figure 2: Example of bounding polyhedra: a) original model (S) with 222 objects and 3989 faces. b) $Faces(b(S,4)) = 1277$. c) $Faces(b(S,2)) = 122$. d) $Faces(b(S,3)) = 351$

- Given a node n and a resolution k return $b(S_n, k)$
- Given a node n and a resolution k return $B(S_n, k)$
- Given a node n and a resolution k return the number of disk accesses to obtain the whole $b(S_n, k)$
- Given a node n and a resolution k return the number of disk accesses to obtain the whole $B(S_n, k)$

1.5 Implementation

We can think on different possible implementations for the MKtrees. In our present implementation we are using the following structure (in figure 3 can be seen an example of a simple MKT(S) data structure):

- For an arbitrary node n we store the AABB Box $R(S_n)$ and two integer and two boolean arrays of dimension M (being M the number of allowed levels of detail):
  - The boolean elements, $BFits[k]$, $k = 1..M$, are set to true iff InPage($B(S_n, k)$) is true
  - The boolean elements, $bFits[k], k = 1..M$, are set to true iff InPage($b(S_n, k)$) is true
  - The integer elements, $BPage[k], k = 1..M$, store a pointer to the disk page location of the geometry of $B(S_n, k)$ if $BFits[k] = true$. Otherwise, if $BFits[k] =$

$false$, they store the number of disk page retrievals that are required to obtain the full geometry of $B(S_n, k)$

- In a similar way, the integer elements, $bPage[k], k = 1..M$, store a pointer to the disk page location of the geometry of $b(S_n, k)$ if $bFits[k] = true$. Otherwise, if $bFits[k] = false$, they store the number of disk page retrievals that are required to obtain the full geometry of $b(S_n, k)$

For any node n, we can require its $b(S_n, k)$ (or its $B(S_n, k)$). We can be in two different cases:

- The information $b(S_n, k)$ fit into a memory page. In this case the pointer to the page (from the node n) is returned.
- $b(S_n, k)$ doesn't fit into one page (see for instance $b(S_1, 2)$ in fig 3). In this situation, the information has being distributed between the childs of node n and to obtain the whole $b(S_n, k)$ we only have to traverse the subtree of n. This is always possible due to that the MKtree is generated making sure that objects fit into a memory page at leaf–nodes and the $b(S_n, k)$ approximations use less memory than the set of objects of S_n. Thus, $MemorySize(b(S_n, k)) \subset MemorySize(S_n)$ always.

However, in most cases the number of pages required for storing $b(S_n, k)$ is less than the number of pages required for $B(S_n, k)$. Observe, for instance, figure 2: $B(S, 1)$ has 1332 faces, as $B(S, 1)$ includes the AABB box of every object in the node. We can see that $Faces(b(S, k)) < Faces(B(S, 1))$ for any k from 1 to 4. In these cases, instead of storing $B(S_n, k)$ we can simply store a pointer to the corresponding $b(S_n, k)$. This is done for $k = 1..M - 1$ and not for $k = M$, since we want to preserve the particular identity of the objects at the maximum resolution level (see for instance $b(S_4, 3)$ and $b(S_3, 4)$ in fig. 3). Anyway, we can explicitly differentiate $B(S_n, k)$ from $b(S_n, k)$ for any resolution $k < M$ (figure 4) if we want to preserve the particular identity of the objects at this resolution level.

Finally, a possible improvement to reduce the total number of disk pages of the model could be the following: whenever the geometric information corresponding to an arbitrary $b(S_n, k)$ occupies less than one page, then we can decide not to store it, and point to the page associated to $b(S_n, j)$ having more resolution ($j > k$) and such that $InPage(b(S_n, j))$ is true. Anyway, we have not adopted this option because it increases the complexity of the bounding approximations and it can slow down the algorithms that use the MKtrees to compute interference or collision detection, for instance.

On the other hand, in our implementation, the representations associated to the two first levels of detail have been chosen to give maximum speed to future interference tests:

- For $k = 1$, we use the corresponding $AABB$: $b(S_n, 1) = R(S_n)$.
- For $k = 2$, we use the cuberille representation of S_n (figure 4). For fast collision testing, $b(S_n, 2)$ is represented as a bit map, each bit corresponding to a voxel in a voxelixation (see [JW89, MF99]) of $R(S_n)$.
- For $k > 2$, the bounding representations from [ABA01], figure 2, are used.

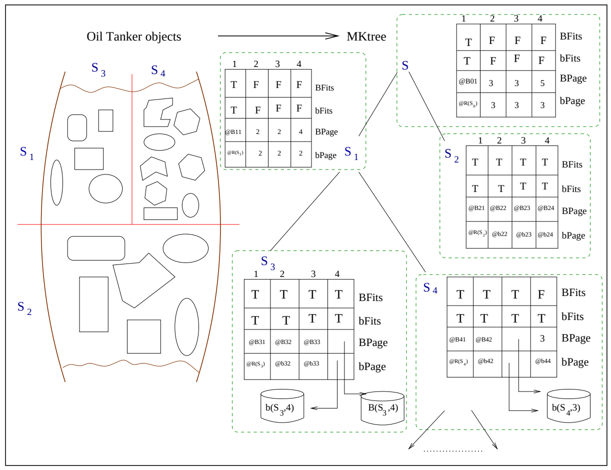

Figure 3: Example of a simple MKT(S) data structure

Figure 4 shows an example in 2D of three LODs (corresponding to three different levels of detail) of a MKtree node n. Observe that the bounding volumes of objects that belong to S_n are more accurate for high values of k, and that collisions will be better detected, if it is the case.

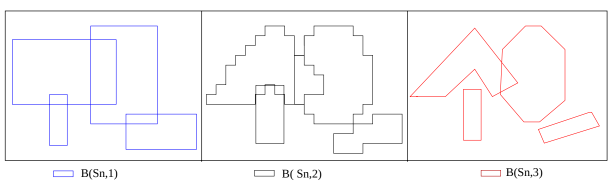

Figure 4: Example in 2D of bounding volumes of node n in 3 different LODs (k=1,2,3)

2 Input Data Description

We have generated several MKtrees from different input data. All the data are a subset of oil tankers. In table 1 the features of the input data are presented. Figures 5, 6,

Input Data				
identifier	objects	polygons	vertices	dimensions (mm)
Oil_tanker_1	222	3989	16432	3920 x 1935 x 1308
Oil_tanker_2	3900	106703	449605	17294 x 12924 x 27351
Oil_tanker_3	4096	132557	546795	22087 x 7965 x 10301

Table 1: Description of the input data

7 show external and detailed views of the Oil_tanker_1, Oil_Tanker_2 and Oil_Tanker_3, respectively.

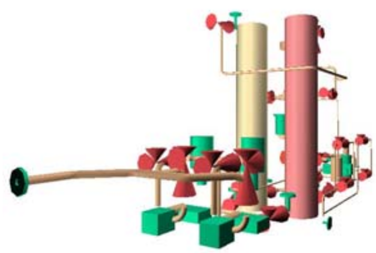

Figure 5: Oil_Tanker_1: View corresponding to an equipment element

3 MKtree Generation

To generate the MKtrees we have designed two heuristic methods that state and solve the problem looking for the best solution by evaluating the intermediate results that are obtained in the direction to the final result, in the form of a guided processes. Therefore, we have implemented two algorithms based on the two approaches.

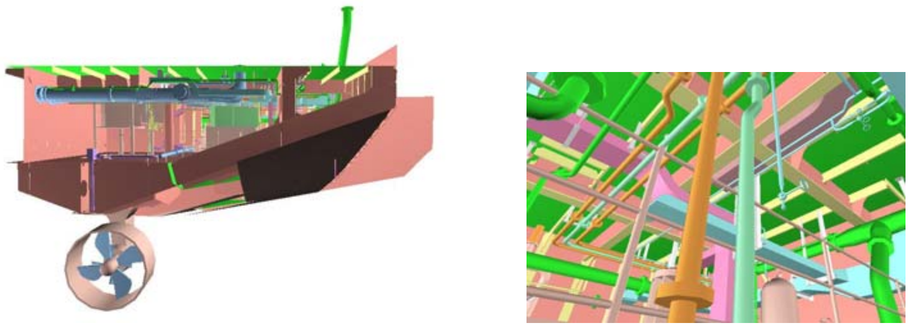

Figure 6: Oil_Tanker_2: Outside and inside view

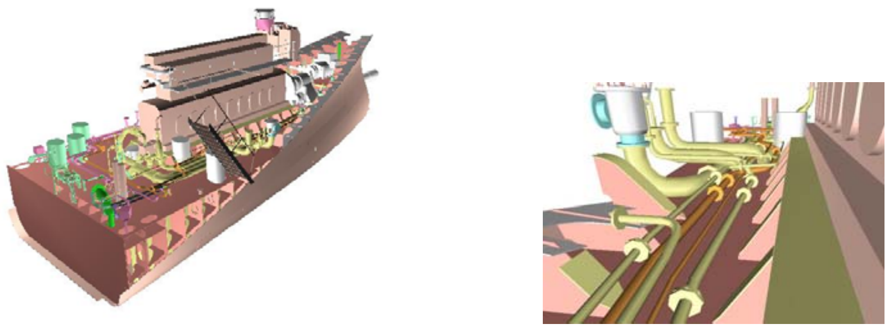

Figure 7: Oil_Tanker_3: General and detailed view

3.1 MKtree Generation: Minimum Overlap Algorithm (MOA)

The MOA algorithm computes the MKtree making use of the *wDivideList* procedure. For each intermediate node, n, of the tree the minimum $wOverlap(R(S_{n_1}), R(S_{n_2}))$ is computed to fine the best dimension and value to distribute the objects belonging to S_n in two subsets S_{n_1} and S_{n_2} [FNB01].

The first procedure, *generate_tree*, has two input arguments. The block size or page capacity, $MemPage$, and the limit (rate) of the sublist of objects that will be examined, R. This second parameter helps the tree to be more or less balanced depending on its value. The sublist of objects to be examined is defined by the following rank of object indices (see the algorithm MOA1)

$$R * N..(1.0 - R) * N$$

Where N is the number of total objects in the actual list of objects.

algorithm MOA1

```
procedure generate_tree(in List_of_objects, in R, in MemPage)
  first = List_of_objects.first_element
  last = List_of_objects.last_element
  wDivideList(first, last)
endprocedure


procedure wDivideList(in first, in last)
  N = last − first + 1
  if
    no InPage(b(S_n, k)) →
                      firstTreat = first + R * N
                      lastTreat = last − R * N
                      for dim in [x, y, z] do
                          SortObjList(first, last, dim, min)
                          ComputeMinOverlap(first, last, firstTreat, lastTreat, dim, minOv)
                          SortObjList(first, last, dim, max)
                          ComputeMinOverlap(first, last, firstTreat, lastTreat, dim, minOv)
                      endfor
                      dimension = Select_minimum(minOv)
                      Icut = Number_of_Selected_objects(dimension, minOv)
                      newfirst = first; newlast = Icut
                      wDivideList(newfirst, newlast)
                      newfirst = Icut + 1; newlast = last
                      wDivideList(newfirst, newlast)
    InPage(b(S_n, k)) → return
  endif
endprocedure
```

Where N is the number of objects in the actual list of objects.

The procedure $wDivideList$ works as follows: When the total size of the geometry of S_n is bigger than the block size ($MemPage$ argument), the actual list of objects is divided in two subsets and a recursive call is done for each one. To compute the best dimension and location to cut the actual list of objects belonging to S_n, six sorts of the actual list of objects are performed. The three first sorts are based on the $Xmin$, $Ymin$ and $Zmin$ increasing order, respectively. The other three sorts are based on the $Xmax$, $Ymax$ and $Zmax$ increasing order, respectively. Then, for each resulting sorted list the minimum overlap is computed (by using the $ComputeMinOverlap$ procedure) and stored in $minOv$ together with the index location where it has been produced. In this way, we only have to select the minimum of the six overlaps. Once the minimum overlap is selected, the actual list is sorted again based on the associated dimension. After, the actual list of objects is partitioned in two sublists, at the corresponding location associated to this dimension (see algorithm MOA 2). Finally, two recursive calls are performed, one for each of the two resulting sublists. And so on.

The $ComputeMinOverlap$ is at most the key to determine the overlap space between the objects of the future sublists by one specified dimension and by one determined sorted list. This procedure will be called six times for each intermediate node. One call for each possible dimension and order criterium (by maximum or by minimum values of the coordinate dimension). The procedure is presented in what follows.

algorithm MOA2

procedure $ComputeMinOverlap$(**in** $first$, **in** $last$, **in** $fTreat$, **in** R, **in** dim, **inout** $minOv$)
  $max = SearchMax(first, fTreat, dim)$
  $min = SearchMin(fTreat + 1, last, dim)$
  $minOv.overlap[dim] = list.O[max].max.xyz[dim] - list.O[min].min.xyz[dim]$
  $minOv.Icut[dim] = fTreat$
  for n **in** $[fTreat + 1 \,..\, lTreat]$ **do**
    $max = SearchMax(first, n, dim)$
    $min = SearchMin(n + 1, last, dim)$
    $overlap = list.O[max].max.xyz[dim] - list.O[min].min.xyz[dim]$
    if
      $overlap < minOv.overlap[dim] \rightarrow minOv.overlap = overlap$
                                                       $minOv.Icut[dim] = n$
    endif
  endfor
endprocedure

The procedures $SearchMax(first, n, dim)$ and $SearchMin(n + 1, last, dim)$ search the maximum value of the coordinate dim of all the objects of the list that are included from $first$ to n and the minimum value of the coordinate dim that are included from $n+1$ to $last$, respectively. Then, the subtraction operation of those values gives the one–dimensional overlap, $wOverlap$ (introduced in sec. 1.3). If it is less than the overlap found up to now it is stored in the $minOv$ structure together with the n value that indicates the place of the list where it has to be cut.

3.1.1 Results of the Minimum Overlap Algorithm

We have executed the Minimum Overlap Algorithm with the input data described in section 2. The main results obtained are presented in several tables. The input values of the arguments used in each execution are detailed in the caption region of each table and their meaning is:

Oil_Tanker_1 Minimum Overlap Algorithm Results				
tree level	N. leaves	ovMin	ovMinRel	ovMax
0	0	18.895020	0.014446	18.895020
1	0	0.000000	0.000000	34.998993
2	0	15.000000	0.021244	156.999023
3	2	0.000000	0.000000	59.250992
4	4	0.000000	0.000000	136.105011
5	11	0.000000	0.000000	122.001007
6	7	0.000000	0.000000	53.001984
7	3	30.000000	0.040000	61.999023
8	6			

Table 2: *Minimum Overlap Algorithm applied to Oil_Tanker_1 with* $R = 0.2$ *and* $MemPage = 10$

- R: Rate that limits the sublist of objects to be examined.
- $MemPage$: Number of objects allowed in one page or block (size of one block).

The information contained in each table is:

- *tree level*: corresponding level of the MKtree. Beginning by the level 0 associated to the root node.
- *N. leaves*: Number of leaves that have been found at the corresponding tree level.
- *ovMin*: Minimum overlap calculated at the corresponding tree level
- *ovRel*: Minimum relative overlap: ovRel$=\frac{ovMin}{size(b(S_n,k))}$
- *ovMax*: Maximum overlap encountered in the actual tree level

In tables 2, 3 and 4 the results obtained when applying the algorithm to the Oil_Tanker_1 with the input values: $R = 0.2$ and 10, 20 and 30 objects that can be in a block at once, values of $MemPage$, respectively, are presented. Tables 5, 6 and 7 show the results obtained changing the value of the first argument to 0.3 and keeping on the values of the second argument, respectively.

In tables 8, 9 and 10, the results obtained when applying the MOA algorithm to the Oil_Tanker_2 are presented. In all the cases the input value corresponding to the rate is 0.2 ($R = 0.2$), while the $MemPage$ has been: 100, 200 and 500 objects, respectively. In tables 11, 12 and 13, the results of applying the MOA method to the Oil_Tanker_2 are

Oil_Tanker_1 Minimum Overlap Algorithm Results				
tree level	N. leaves	ovMin	ovMinRel	ovMax
0	0	18.895020	0.014446	18.895020
1	0	0.000000	0.000000	34.998993
2	1	15.000000	0.011299	156.999023
3	2	0.000000	0.000000	59.250992
4	6	100.000000	0.141772	136.105011
5	2	35.359985	0.030784	122.001007
6	3	0.000000	0.000000	0.000000
7	2			

Table 3: *Minimum Overlap Algorithm applied to Oil_Tanker_1 with* $R = 0.2$ *and* $MemPage = 20$

Oil_Tanker_1 Minimum Overlap Algorithm Results				
tree level	N. leaves	ovMin	ovMinRel	ovMax
0	0	18.895020	0.014446	18.895020
1	0	0.000000	0.000000	34.998993
2	2	20.000000	0.011527	156.999023
3	1	14.999298	0.013380	59.250992
4	4	100.000000	0.141772	136.105011
5	2	35.359985	0.030784	122.001007
6	4			

Table 4: *Minimum Overlap Algorithm applied to Oil_Tanker_1 with* $R = 0.2$ *and* $MemPage = 30$

Oil_Tanker_1 Minimum Overlap Algorithm Results				
tree level	N. leaves	ovMin	ovMinRel	ovMax
0	0	156.999023	0.040051	156.999023
1	0	80.000000	0.027875	129.894989
2	0	0.000000	0.000000	150.000000
3	1	0.000000	0.000000	94.248596
4	5	0.000000	0.000000	181.000000
5	13	0.000000	0.000000	100.000000
6	9	70.113007	0.063451	70.113007
7	1	0.000000	0.000000	0.000000
8	2			

Table 5: *Minimum Overlap Algorithm applied to Oil_Tanker_1 with* $R = 0.3$ *and* $MemPage = 10$

Oil_Tanker_1 Minimum Overlap Algorithm Results				
tree level	N. leaves	ovMin	ovMinRel	ovMax
0	0	18.895020	0.014446	18.895020
1	0	0.000000	0.000000	34.998993
2	1	15.000000	0.011299	156.999023
3	2	0.000000	0.000000	59.250992
4	6	100.000000	0.141772	136.105011
5	2	35.359985	0.030784	122.001007
6	3	0.000000	0.000000	0.000000
7	2			

Table 6: *Minimum Overlap Algorithm applied to Oil_Tanker_1 with* $R = 0.3$ *and* $MemPage = 20$

Oil_Tanker_1 Minimum Overlap Algorithm Results				
tree level	N. leaves	ovMin	ovMinRel	ovMax
0	0	156.999023	0.040051	156.999023
1	0	80.000000	0.027875	129.894989
2	0	0.000000	0.000000	150.000000
3	6	14.999298	0.013380	94.248596
4	3	122.001007	0.189369	122.001007
5	2			

Table 7: *Minimum Overlap Algorithm applied to Oil_Tanker_1 with* $R = 0.3$ *and* $MemPage = 30$

exposed. The value of the arguments are $R = 0.3$ for all of the cases, and, 100, 200 and 500 as values of the $MemPage$ argument, respectively.

The tables 14, 15 and 16 correspond to the results obtained when computing the MKtree of the Oil_Tanker_3 by using the MOA algorithm. The value of the rate argument is 0.2 ($R = 0.2$) for all of them and the value of the $MemPage$ argument is 100, 200 and 500, respectively. In tables 17, 18 and 19, the results obtained when applying the MOA algorithm to the Oil_Tanker_3, with 100, 200 and 500 as values of the $MemPage$ argument, respectively and, with $R = 0.3$ for all of the cases, are exposed.

Observing the results, presented in the tables, we can see how the values of the input arguments affect to the resulting MKtrees. When we change the value of the block size, keeping on the rate, we obeserve:

- As $MemPage$ increases, the deep of the MKtree decreases in all the cases. See, for instance, tables 8 and 10. In the first table the value of the $MemPage$ is set to 100 and in the second one, this value is set to 500. While the deep of the first MKtree is 14 (0 .. 13), the deep of the second is 7 (0 .. 6).
- When the $MemPage$ is low (i.e. 100 for the Oil_Tanker_2, see table 8) the values of the $ovMin$ and $ovMinRel$ are low also, and those values have the tendency to be zero. Compare, for instance, the table 14 where $ovMin$ is zero in six levels, and the table 16 where there is one $ovMin$ equal to zero, only.

On the other hand, the effects of fixing the value of the $MemPage$ argument and changing the R values to the resulting MKtrees can be summarized in:

- For high values of the R the computed MKtree has less levels than for low values of $rate$ argument. See tables 14 and 17, for instance. Thus, for high values of R

Oil_Tanker_2 Minimum Overlap Algorithm Results				
tree level	N. leaves	ovMin	ovMinRel	ovMax
0	0	1431.099609	0.110736	1431.099609
1	0	250.000000	0.063354	2021.650391
2	0	0.000000	0.000000	2304.980469
3	1	0.000000	0.000000	2398.529785
4	7	0.000000	0.000000	2500.000000
5	3	0.000000	0.000000	2534.000000
6	12	0.000000	0.000000	2615.029785
7	11	0.000000	0.000000	2699.029785
8	11	0.000000	0.000000	2853.000000
9	11	330.500000	0.058100	2845.529785
10	4	739.799805	0.086693	2892.589844
11	2	943.899902	0.234670	1829.939941
12	3	2213.599854	0.291267	2213.599854
13	2			

Table 8: *Minimum Overlap Algorithm applied to Oil_Tanker_2 with* $R = 0.2$ *and* $MemPage = 100$

argument, the computed MKtrees are more balanced than for low values of this parameter.

- For higher levels of the *rate* the MKtrees have a higher values of $ovMin$ and $ovRel$ than for lower values of this parameter.
- As a consecuence, we can see that depending on the context problem we can decide to get MKtrees with low values of $ovMin$ or MKtrees with high values of $ovMin$. In the second case, we will get more balanced MKtrees than in the first, but the page faults when computing collisions, for intance, will be higher. Although this is true, on the other hand, for higher values of the *rate* argument the generated MKtrees will be more balance (as said before) and not so deep as for lower values of *rate*. It means, that high values give higher $ovMin$ between spaces belonging to different grey nodes, but at the same time, the number of grey nodes is less. If the number of the grey nodes is low it implies less queries to retrieval pages from disk.

Figures 8 and 9 graphicaly represent the MKtrees of the Oil_Tanker_2 generated with values of arguments $R = 0.3$ and $MemPage = 500$, for the first figure, and $R = 0.2$ and $MemPage = 500$ for the second one. The grey nodes of the MKtree have been draw as boxes and the leaves nodes as ellipses. Inside each ellipse there apears the number

Oil_Tanker_2 Minimum Overlap Algorithm Results				
tree level	N. leaves	ovMin	ovMinRel	ovMax
0	0	1431.099609	0.110736	1431.099609
1	0	250.000000	0.063354	2021.650391
2	1	372.190430	0.222077	2304.980469
3	2	305.810059	0.018059	2398.529785
4	2	0.000000	0.000000	2500.000000
5	7	0.000000	0.000000	2534.000000
6	5	0.000000	0.000000	2615.029785
7	8	0.000000	0.000000	2699.029785
8	2	815.000000	0.099975	2853.000000
9	2	533.600098	0.049869	2845.529785
10	3	739.799805	0.086693	739.799805
11	2			

Table 9: *Minimum Overlap Algorithm applied to Oil_Tanker_2 with* $R = 0.2$ *and* $MemPage = 200$

Oil_Tanker_2 Minimum Overlap Algorithm Results				
tree level	N. leaves	ovMin	ovMinRel	ovMax
0	0	1431.099609	0.110736	1431.099609
1	0	250.000000	0.063354	2021.650391
2	1	372.190430	0.222077	2304.980469
3	5	2398.529785	0.230440	2398.529785
4	0	500.000000	0.080000	2500.000000
5	2	849.500000	0.135920	2534.000000
6	3	1068.490234	0.170958	1068.490234
7	2			

Table 10: *Minimum Overlap Algorithm applied to Oil_Tanker_2 with* $R = 0.2$ *and* $MemPage = 500$

Oil_Tanker_2 Minimum Overlap Algorithm Results				
tree level	N. leaves	ovMin	ovMinRel	ovMax
0	0	1747.000000	0.135180	1747.000000
1	0	405.000000	0.095026	2398.529785
2	0	159.000000	0.392593	2497.000000
3	0	0.000000	0.000000	2534.000000
4	4	0.000000	0.000000	2699.029785
5	11	0.000000	0.000000	2601.700195
6	15	0.000000	0.000000	2970.020020
7	18	0.000000	0.000000	2916.000000
8	6	1217.560059	0.259478	3120.589844
9	4			

Table 11: *Minimum Overlap Algorithm applied to Oil_Tanker_2 with* $R = 0.3$ *and* $MemPage = 100$

Oil_Tanker_2 Minimum Overlap Algorithm Results				
tree level	N. leaves	ovMin	ovMinRel	ovMax
0	0	1747.000000	0.135180	1747.000000
1	0	405.000000	0.095026	2398.529785
2	0	159.000000	0.392593	2497.000000
3	1	0.000000	0.000000	2534.000000
4	7	146.000000	0.010440	2699.029785
5	10	312.000000	0.317557	2601.700195
6	5	957.000000	0.054403	2970.020020
7	5	1207.560059	0.257347	1207.560059
8	2			

Table 12: *Minimum Overlap Algorithm applied to Oil_Tanker_2 with* $R = 0.3$ *and* $MemPage = 200$

Oil_Tanker_2 Minimum Overlap Algorithm Results				
tree level	N. leaves	ovMin	ovMinRel	ovMax
0	0	1747.000000	0.135180	1747.000000
1	0	405.000000	0.095026	2398.529785
2	1	649.500000	0.152393	2497.000000
3	3	791.599609	0.185734	2534.000000
4	4	721.140137	0.047132	2699.029785
5	4			

Table 13: *Minimum Overlap Algorithm applied to Oil_Tanker_2 with* $R = 0.3$ *and* $MemPage = 500$

Oil_Tanker_3 Minimum Overlap Algorithm Results				
tree level	N. leaves	ovMin	ovMinRel	ovMax
0	0	1455.949951	0.182819	1455.949951
1	0	117.823997	0.080128	1591.000000
2	0	40.850098	0.346704	1393.000000
3	1	0.000000	0.000000	1749.649902
4	5	0.000000	0.000000	1869.319824
5	8	0.000000	0.000000	2012.000000
6	11	0.000000	0.000000	1847.000000
7	9	100.000000	0.052301	2035.669922
8	9	0.000000	0.000000	907.500977
9	13	0.000000	0.000000	709.082031
10	6	103.000000	0.014548	1016.700012
11	5	180.000000	0.024000	1022.005981
12	5	376.000000	0.044282	376.000000
13	2			

Table 14: *Minimum Overlap Algorithm applied to Oil_Tanker_3 with* $R = 0.2$ *and* $MemPage = 100$

Oil_Tanker_3 Minimum Overlap Algorithm Results				
tree level	N. leaves	ovMin	ovMinRel	ovMax
0	0	1455.949951	0.182819	1455.949951
1	0	117.823997	0.080128	1591.000000
2	1	150.000000	0.102010	1393.000000
3	2	0.390015	0.000040	1749.649902
4	2	0.000000	0.000000	1869.319824
5	6	0.000000	0.000000	2012.000000
6	7	50.000000	0.003246	1847.000000
7	7	119.050049	0.068145	2035.669922
8	5	0.000000	0.000000	0.000000
9	0	399.800049	0.094829	570.000000
10	3	1016.700012	0.211887	1016.700012
11	2			

Table 15: *Minimum Overlap Algorithm applied to Oil_Tanker_3 with* $R = 0.2$ *and* $MemPage = 200$

Oil_Tanker_3 Minimum Overlap Algorithm Results				
tree level	N. leaves	ovMin	ovMinRel	ovMax
0	0	1455.949951	0.182819	1455.949951
1	0	117.823997	0.080128	1591.000000
2	1	150.000000	0.102010	1393.000000
3	2	0.390015	0.000040	1749.649902
4	7	1869.319824	0.377259	1869.319824
5	1	2012.000000	0.252320	2012.000000
6	1	1847.000000	0.231628	1847.000000
7	1	2035.669922	0.419408	2035.669922
8	1	0.000000	0.000000	0.000000
9	2			

Table 16: *Minimum Overlap Algorithm applied to Oil_Tanker_3 with* $R = 0.2$ *and* $MemPage = 500$

Oil_Tanker_3 Minimum Overlap Algorithm Results				
tree level	N. leaves	ovMin	ovMinRel	ovMax
0	0	1950.030029	0.244859	1950.030029
1	0	474.599609	0.021489	1187.209961
2	0	232.199219	0.014256	1162.169922
3	0	55.900391	0.003538	785.270020
4	2	0.000000	0.000000	1353.479980
5	13	0.000000	0.000000	1500.000000
6	21	0.000000	0.000000	1900.900391
7	11	0.000000	0.000000	1017.528992
8	12	254.170044	0.103556	260.000000
9	3	124.619995	0.079096	124.619995
10	2			

Table 17: *Minimum Overlap Algorithm applied to Oil_Tanker_3 with* $R = 0.3$ *and* $MemPage = 100$

Oil_Tanker_3 Minimum Overlap Algorithm Results				
tree level	N. leaves	ovMin	ovMinRel	ovMax
0	0	1950.030029	0.244859	1950.030029
1	0	474.599609	0.021489	1187.209961
2	0	232.199219	0.014256	1162.169922
3	1	55.900391	0.003538	785.270020
4	6	0.000000	0.000000	1353.479980
5	11	1.799805	0.000147	1500.000000
6	8	505.000000	0.048827	670.030029
7	3	434.000000	0.037034	434.000000
8	2			

Table 18: *Minimum Overlap Algorithm applied to Oil_Tanker_3 with* $R = 0.3$ *and* $MemPage = 200$

Oil_Tanker_3 Minimum Overlap Algorithm Results				
tree level	N. leaves	ovMin	ovMinRel	ovMax
0	0	1950.030029	0.244859	1950.030029
1	0	474.599609	0.021489	1187.209961
2	1	369.169922	0.112708	1162.169922
3	2	81.000000	0.004229	785.270020
4	7	1353.479980	0.197387	1353.479980
5	1	1235.979980	0.229652	1235.979980
6	2			

Table 19: *Minimum Overlap Algorithm applied to Oil_Tanker_3 with* $R = 0.3$ *and* $MemPage = 500$

of objects that the associated leaf contains (this number is always less that the value of $MemPage$). The information associated inside each box, grey node, is:

- nO: Number of objects that belong to S_n (for a grey node n).
- dim: Cut dimension selected for the subset S_n to obtain S_{n_1} and S_{n_2}
- ovS: $wOverlap$ selected for the current node (corresponding to the minimum *overlap_band* introduced in section 1.3)
- ovR: Minimum relative overlap, where ovR$=\frac{ovS}{size(b(S_n,k))}$

Thus, for example the root node, say S_n, of the MKtree of figure 8:

- $nO = 3900$: Contains 3900 objects
- $dim = Y$: The cut dimension corresponds to the Y axis, it means that $wOverlap(R(S_{n_1}), R(S_{n_2})) = yOverlap(R(S_{n_1}), R(S_{n_2}))$ (see sec. 1.3 for details)
- ovS: The minimum *overlap_band* computed has been $1747.0mm$
- ovR: The ovS relative to the $ySize(R(S_n))$ is 0.135180

Looking at the MKtree of figures 8 and 9, we observe how the values of the *rate* argument affect to the resulting MKtrees. The first MKtree, corresponding to $R = 0.3$, is more balanced than the second one, where $R = 0.2$.

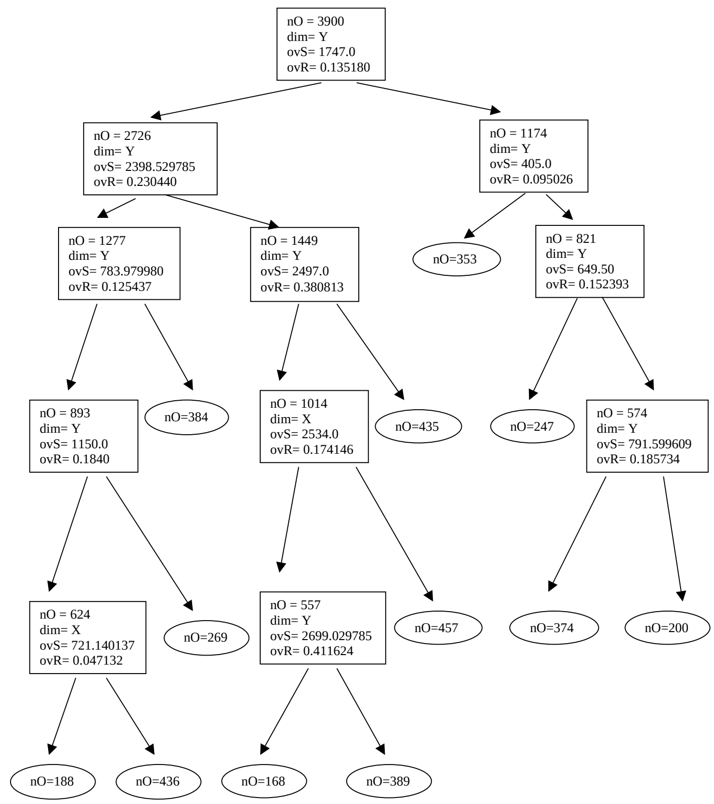

Figure 8: Example of MKtree corresponding to the Oil_tanker_2 computed by using the Minimum Overlap Algorithm with $R = 0.3$ and $MemPage = 48000bytes$ $(500obj)$

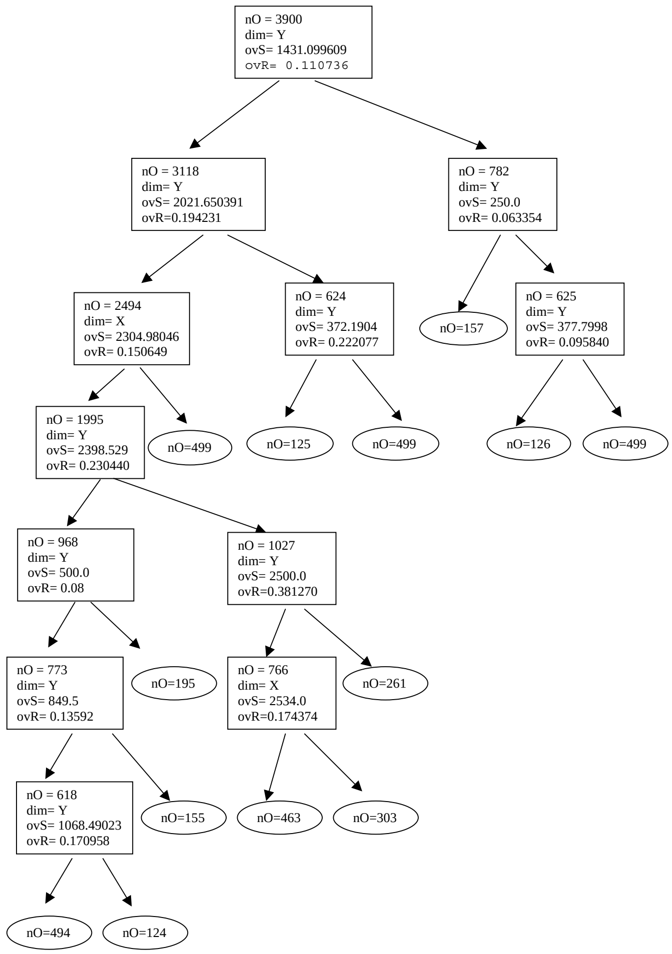

Figure 9: Example of MKtree corresponding to the Oil_tanker_2 computed by using the MOA algorithm with $R = 0.2$ and $MemPage = 48000bytes$ $(500obj)$

Figures 10, 11 and 12, are images of MKtrees of the three oil tankers, presented in several colors corresponding to the MKtree spaces (nodes), up to level 1 or 2, of the tree and with different values of LOD, depending on the image. The values of the input arguments to the MOA algorithm of every image are exposed in associated caption region of each image.

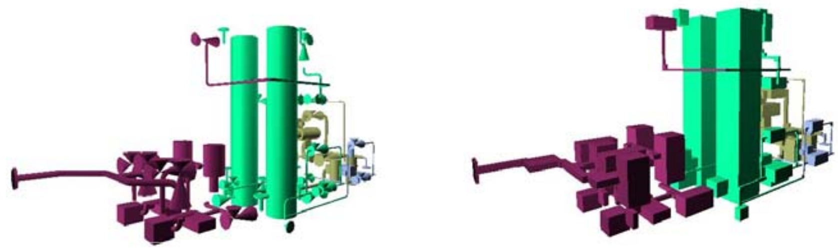

Figure 10: MKtree up to level 2 of the Oil_Tanker_1, computed by using the MOA algorithm with $R = 0.3$ and $MemPage = 50$. Left $LOD = max$. Right $LOD = 1$.

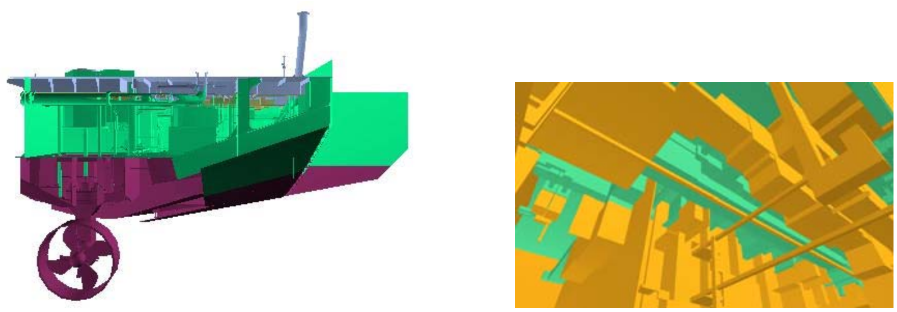

Figure 11: MKtree up to level 2 and $LOD = max$ (left) and to level 1 and $LOD = 1$ (right) of the Oil_Tanker_2, computed by using the MOA algorithm with $R = 0.3$ and $MemPage = 48000 bytes$ $(500 objs)$.

3.2 MKtree Generation: Look Ahead Algorithm (LAA)

The LAA algorithm computes the MKtree by using the $wDivideList$ procedure (see algorithm LAA1) which makes use of a *LookAhead* function (see algorithm LAA2). This function selects the best dimension and location to cut S_n in two subsets, S_{n_1} and S_{n_2}, looking for the minimum accumulated overlap between all the possible subtrees that begin from n. In other words, the *LookAhead* function explores the possible subtrees that results from each overlap previously computed (by the $ComputeMinOverlap$ procedure – see algorithm LAA1) and selects the best cut depending on the accumulated overlap found

Figure 12: MKtree up to level 2 with $LOD = max$ (left) and to level 1 with $LOD = 1$ (right) of the Oil_Tanker_3, computed by using the MOA algorithm with $R = 0.3$ and $MemPage = 48000 bytes$ ($500 objs$).

from the $actual\_level$ (corresponding to the grey node n) until to a pre–specified $look\_level$ of the simulated or trial MKtree. Where $Look\_level \geq actual\_level$. The trial generation that permits to compute the simulated MKtrees is performed by the $wDivideListTrial$ procedure (see algorithm LAA3), which is called six times by the $LookAhead$ function.

The first procedure, $generate\_tree$, has three input arguments (see algorithm LAA1). The block size or the capacity of the page, $MemPage$, the rate value that limits the sublist of objects to be examined, R, and, finally, the $LookAhead\_level$ to compute the trial "subtrees". The last one, determines the deep of the simulated subtrees.

Thereby, for each intermediate node, n, of the tree, six values of $wOverlap(R(S_{n_1}), R(S_{n_2}))$ are computed. One for each dimension (x, y, and z) and sorting criterium. The three first sorts are based on the $Xmin$, $Ymin$, and $Zmin$ (coordinates of objects) increasing order, respectively. The other three sorts are based on the $Xmax$, $Ymax$ and $Zmax$ (coordinates of objects) increasing order, respectively.

Then, for each resulting sorted list, the minimum overlap is computed (by using the $ComputeMinOverlap$ procedure) and stored in $minOv$ structure, together with the index location where it has been produced (see algorithm LAA1).

To select the best dimension and location to cut S_n in two subsets, S_{n_1} and S_{n_2}, a call to $LookAhead$ function is performed (see algorithm LAA1). The $LookAhead$ function generates, in a trial way, six simulated subtrees corresponding to the six overlaps previously computed and stored in $minOv$. Thus, the best cut, is selected taking into account the accumulated overlaps found in each simulated subtree.

In fact, the trial subtrees are generated, in a simulated way, from the $actual\_level$ of the MKtrees to an pre–specified $look\_level$. Where:

$$look\_level = actual\_level \ + \ LookAhead\_level$$

To compute the $LookAhead\_trees$ (or simulated subtrees) we have implemented a $wDivideListTrial$ recursive procedure (see algorithm LAA3) that is a trial of the $wDivideList$.

Thus, the $wDivideListTrial$ computes all possible subtrees and the $LookAhead$ function chooses the optimal one. The algorithm is presented is what follows.

algorithm LAA1

```
procedure generate_tree(in List_of_objects, in R, in MemPage, in LookAhead_level)
  first = List_of_objects.first_element
  last = List_of_objects.last_element
  actual_level = 0
  wDivideList(first, last, actual_level, LookAhead_level)
endprocedure


procedure wDivideList(in first, in last, in actual_level, in LookAhead_level)
  act_level = actual_level
  if
    no InPage(b(S_n, k)) →
                     firstTreat = first + R * number_objects
                     lastTreat = last − R * number_objects
                     for  dim in  [x, y, z]  do
                         SortObjList(first, last, dim, min)
                         ComputeMinOverlap(first, last, firstTreat, lastTreat, dim, minOv)
                         SortObjList(first, last, dim, max)
                         ComputeMinOverlap(first, last, firstTreat, lastTreat, dim, minOv)
                     endfor
                     which = LookAhead(first, last, minOv, actual_level, LookAhead_level)
                     Icut = minOv.I[which]
                     act_level = act_level + 1
                     newfirst = first; newlast = Icut;
                     wDivideList(newfirst, newlast, act_level, LookAhead_level)
                     newfirst = Icut + 1; newlast = last
                     wDivideList(newfirst, newlast, act_level, LookAhead_level)
    InPage(b(S_n, k)) → return
  endif
endprocedure
```

The $wDivideList$ procedure computes the cut dimension and value to divide the initial list of objects in two sublists. To select those values (from $minOv$ structure) it calls to the $LookAhead$ function indicating the $LookAhead\_level$. This last argument limits the MKtree level up to the $LookAhead$ function has to explore. The output of the $LookAhead$ function is an integer that indicates the value of the index of the $minOv$ structure that has to be chosen.

algorithm LAA2

function $LookAhead$(**in** $first$, **in** $last$, **in** $minOv$, **in** $actual\_level$, **in** $LookAhead\_level$) **return integer**
  $FloatSix[1..6] = minOv.V[1..6]$
  $look\_level = LookAhead\_level + actual\_level$
  for i **in** $[1..6]$ **do**
    $first\_now = first$
    $last\_now = minOv.I[i]$
    $child\_over = minOv.V[i]$
    $wDivideListTrial(first\_now, last\_now, actual\_level, look\_level)$
    $FloatSix[i] = child\_ove + FloatSix[i]$
    $first\_now = minOv.I[i] + 1$
    $last\_now = last$
    $child\_over = minOv.V[i]$
    $wDivideListTrial(first\_now, last\_now, actual\_level, look\_level)$
    $FloatSix[i] = child\_ove + FloatSix[i]$
  endfor
  return $minimum(FloatSix)$
endfunction

The $LookAhead$ function returns an integer value (1 .. 6) that indicates for which axes (dimension) and where (in terms of number of objects) have the subset S_n to be cut. All the possible *subtrees* that will be generated, depending on each value of the *overlap_band* for S_n, are taken into account. In other words, the function explores the possible subtrees that results from each overlap computed (and stored in $minOv$) and selects the best cut depending on the accumulated overlap found from the $actual\_level$ to the $look\_level$ of the trial MKtree (where $look\_level = actual\_level + LookAhead\_level$). The trial generation that permits to compute the optimal tree is performed by the $wDivideListTrial$ procedure, which is called six times by the $LookAhead$ function. One for each value contained in $minOv$. The $wDivideListTrial$ is presented in what follows.

algorithm LAA3

procedure $wDivideListTrial$(**in** $first$, **in** $last$, **in** $actual\_level$, **in** $look\_level$)
  if
    no $InPage(b(S_n, k))$ **and**
      $actual\_level <= look\_level \rightarrow$
        $firstTreat = first + R * number_objects$
        $lastTreat = last - R * number\_objects$
        for dim **in** $[x, y, z]$ **do**
          $SortObjList(first, last, dim, min)$
          $ComputeMinOverlap(first, last, firstTreat, lastTreat, oversix)$
          $SortObjList(first, last, dim, max)$
          $ComputeMinOverlap(first, last, firstTreat, lastTreat, oversix)$
        endfor

$which = minimum(oversix)$
$child\_ove = oversix.V[which] + child\_ove$
$actual\_level = actual\_level + 1$
$newfirst = first;\ newlast = oversix.I[which]$
$wDivideListTrial(newfirst, newlast, actual\_level, look\_level)$
$newfirst = oversix.I[which] + 1;\ newlast = last$
$wDivideListTrial(newfirst, newlast, actual\_level, look\_level)$

$InPage(b(S_n, k))$ **or** $(actual\_level > look\_level) \rightarrow return$
endif
endprocedure

The *wDivideListTrial* computes a simulated subtree of an MKtree from the input *actual_level* up to the *look_level* = *actual_level* + *LookAhead_level*. Thus, the simulated subtree will have *look_level* levels. In this way, the value of the *LookAhead_level*, argument of the *generate_tree* procedure, limits the deep of the MKtree to be explored and, as a consecuence, if its value is high the final MKtree generated will be more optimal than if its value is low. In fact, when its value is equal to zero the MKtree generated by using the LAA method will be similar than the MKtree generated by using the MOA method.

3.2.1 Results of the Look Ahead Algorithm

The tables 20, 21 and 22, show the results obtained when applying the *LookAhead* algorithm to the Oil_Tanker_1. The values of the arguments are: $R = 0.2$ (for all of the three cases), 10, 20 and 30, respectively, as values of the *MemPage* and the *LookAhead_level* has been set at the maximum level in all the cases. Tables 23, 24 and 25 present the results when changing the R argument to 0.3 with respect the previous three tables, and mantain the values of the two other arguments.

Tables 26, 27 and 28, summarize the results obtained by applying the *LookAhead* algorithm to the Oil_Tanker_2. The values of the arguments are: $R = 0.2$, for every case, 100, 200 and 500 as a values of *MemPage*, respectively, and the maximum value for the *LookAhead_level* have been selected for all of the executions. Applying the algorithm to the same Oil_Tanker, the results obtained, when setting the *rate* value to 0.3 and keeping on the values of the other two arguments (with respect to the three first cases), are presented in tables 29, 30 and 31.

In tables 32, 33, 34, 35, 36 and 37 the results obtained when applying the LAA algorithm to the Oil_Tanker_3 are presented. The values of the arguments that have been used for each execution are described in the caption of each table.

In most of the tables presented in this section the value of the *LookAhead_level* has been set to the maximum possible in order to obtain the more optimal MKtree. As

Oil_Tanker_1 Look Ahead Algorithm Results				
tree level	N. leaves	ovMin	ovMinRel	ovMax
0	0	286.132996	0.147872	286.132996
1	0	9.001297	0.008230	18.895020
2	0	0.000000	0.000000	343.108978
3	2	0.000000	0.000000	0.000000
4	7	0.000000	0.000000	0.000000
5	7	0.000000	0.000000	25.002014
6	4	0.000000	0.000000	40.000000
7	2	0.000000	0.000000	35.359985
8	2	0.000000	0.000000	0.000000
9	2	30.000000	0.040000	34.999596
10	4			

Table 20: *Look Ahead Algorithm applied to Oil_Tanker_1 with* $R = 0.2$*,* $MemPage = 10$ *and LA=maximum*

Oil_Tanker_1 Look Ahead Algorithm Results				
tree level	N. leaves	ovMin	ovMinRel	ovMax
0	0	18.895020	0.014446	18.895020
1	0	57.132996	0.501629	352.175964
2	1	0.000000	0.000000	34.998993
3	2	0.000000	0.000000	140.112976
4	4	0.000000	0.000000	150.000000
5	6	35.001598	0.049736	35.359985
6	3	0.000000	0.000000	0.000000
7	2			

Table 21: *Look Ahead Algorithm applied to Oil_Tanker_1 with* $R = 0.2$*,* $MemPage = 20$ *and LA=maximum*

Oil_Tanker_1 Look Ahead Algorithm Results				
tree level	N. leaves	ovMin	ovMinRel	ovMax
0	0	18.895020	0.014446	18.895020
1	0	15.000000	0.011299	352.175964
2	1	0.000000	0.000000	34.998993
3	4	20.000000	0.011527	140.112976
4	3	150.000000	0.124622	150.000000
5	1	0.000000	0.000000	0.000000
6	2			

Table 22: *Look Ahead Algorithm applied to Oil_Tanker_1 with* $R = 0.2$*,* $MemPage = 30$ *and LA=maximum*

Oil_Tanker_1 Look Ahead Algorithm Results				
tree level	N. leaves	ovMin	ovMinRel	ovMax
0	0	156.999023	0.040051	156.999023
1	0	91.895996	0.075448	265.000000
2	0	8.895020	0.024708	495.359497
3	0	0.000000	0.000000	80.000000
4	8	0.000000	0.000000	142.500000
5	11	0.000000	0.000000	34.998993
6	7	0.000000	0.000000	9.248993
7	6			

Table 23: *Look Ahead Algorithm applied to Oil_Tanker_1 with* $R = 0.3$*,* $MemPage = 10$ *and LA=maximum*

Oil_Tanker_1 Look Ahead Algorithm Results				
tree level	N. leaves	ovMin	ovMinRel	ovMax
0	0	156.999023	0.040051	156.999023
1	0	80.000000	0.027875	91.895996
2	0	10.000000	0.008706	20.000000
3	3	0.000000	0.000000	175.112976
4	7	0.000000	0.000000	39.248993
5	5	122.001007	0.180943	122.001007
6	2			

Table 24: *Look Ahead Algorithm applied to Oil_Tanker_1 with* $R = 0.3$*,* $MemPage = 20$ *and LA=maximum*

Oil_Tanker_1 Look Ahead Algorithm Results				
tree level	N. leaves	ovMin	ovMinRel	ovMax
0	0	156.999023	0.040051	156.999023
1	0	80.000000	0.027875	91.895996
2	1	15.000000	0.013863	20.000000
3	3	14.999298	0.013380	175.112976
4	5	39.248993	0.034642	39.248993
5	2			

Table 25: *Look Ahead Algorithm applied to Oil_Tanker_1 with* $R = 0.3$*,* $MemPage = 30$ *and LA=maximum*

Oil_Tanker_2 LookAhead Results				
tree level	N. leaves	ovMin	ovMinRel	ovMax
0	0	2978.149902	0.108887	2978.149902
1	0	677.000000	0.061668	1243.299805
2	0	182.000000	0.020975	1809.129883
3	1	0.000000	0.000000	2398.529785
4	8	0.000000	0.000000	976.560059
5	4	0.000000	0.000000	2371.000000
6	6	0.000000	0.000000	1163.000000
7	11	0.000000	0.000000	1881.939941
8	11	0.000000	0.000000	1690.890015
9	12	207.000000	0.015181	732.000000
10	2	944.209961	0.206336	2177.959961
11	2	0.000000	0.000000	839.700195
12	4			

Table 26: *Look Ahead algorithm applied to Oil_Tanker_2 with* $R = 0.2$, $MemPage = 100$ *and LA_level=maximum*

Oil_Tanker_2 LookAhead Results				
tree level	N. leaves	ovMin	ovMinRel	ovMax
0	0	2978.149902	0.108887	2978.149902
1	0	677.000000	0.061668	1243.299805
2	1	264.500000	0.067059	1809.129883
3	2	364.500000	0.092466	1969.740234
4	2	43.497986	0.002589	2375.509766
5	6	223.000000	0.156053	2392.000000
6	8	310.000000	0.036267	444.120117
7	5	0.000000	0.000000	966.799805
8	4	10.500000	0.001757	622.009766
9	4			

Table 27: *Look Ahead algorithm applied to Oil_Tanker_2 with* $R = 0.2$, $MemPage = 200$ *and LA_level=maximum*

Oil_Tanker_2 LookAhead Results				
tree level	N. leaves	ovMin	ovMinRel	ovMax
0	0	1431.099609	0.110736	1431.099609
1	0	250.000000	0.063354	2904.490234
2	1	377.799805	0.095840	1824.700195
3	5	1969.740234	0.128746	1969.740234
4	0	1226.060059	0.139830	2375.509766
5	3	478.689941	0.097437	478.689941
6	1	732.000000	0.068411	732.000000
7	2			

Table 28: *Look Ahead algorithm applied to Oil_Tanker_2 with* $R = 0.2$*,* $MemPage = 500$ *and LA_level=maximum*

Oil_Tanker_2 LookAhead Results				
tree level	N. leaves	ovMin	ovMinRel	ovMax
0	0	1747.000000	0.135180	1747.000000
1	0	405.000000	0.095026	3988.099609
2	0	280.000000	0.691358	2375.509766
3	0	0.000000	0.000000	2123.619873
4	5	0.000000	0.000000	2334.000000
5	8	0.000000	0.000000	2355.029785
6	17	0.000000	0.000000	2491.294922
7	17	0.000000	0.000000	301.850098
8	8	118.979980	0.013620	1030.000000
9	4			

Table 29: *Look Ahead algorithm applied to Oil_Tanker_2 with* $R = 0.3$*,* $MemPage = 100$ *and LA_level=maximum*

Oil_Tanker_2 LookAhead Results				
tree level	N. leaves	ovMin	ovMinRel	ovMax
0	0	1747.000000	0.135180	1747.000000
1	0	405.000000	0.095026	3383.900391
2	0	159.000000	0.392593	2398.529785
3	1	0.000000	0.000000	2500.000000
4	7	0.000000	0.000000	2534.000000
5	9	0.000000	0.000000	3892.629883
6	8	1090.310059	0.139212	1175.000000
7	4			

Table 30: *Look Ahead algorithm applied to Oil_Tanker_2 with* $R = 0.3$*,* $MemPage = 200$ *and LA_level=maximum*

Oil_Tanker_2 LookAhead Results				
tree level	N. leaves	ovMin	ovMinRel	ovMax
0	0	2398.529785	0.185594	2398.529785
1	0	783.979980	0.125437	1434.000000
2	1	329.500000	0.083439	1954.500000
3	2	371.629883	0.021473	2503.000000
4	7	2534.000000	0.174146	2534.000000
5	2			

Table 31: *Look Ahead algorithm applied to Oil_Tanker_2 with* $R = 0.3$*,* $MemPage = 500$ *and LA_level=maximum*

Oil_Tanker_3 LookAhead Results				
tree level	N. leaves	ovMin	ovMinRel	ovMax
0	0	2511.000000	0.113692	2511.000000
1	0	665.799805	0.099299	1509.589966
2	0	106.399399	0.069812	1949.010010
3	1	0.000000	0.000000	907.320313
4	3	0.000000	0.000000	693.099609
5	12	0.000000	0.000000	547.451050
6	6	0.000000	0.000000	959.701172
7	17	0.000000	0.000000	536.017029
8	16	0.000000	0.000000	536.017029
9	9	80.020020	0.014584	322.817078
10	6			

Table 32: *Look Ahead algorithm applied to Oil_Tanker_3 with* $R = 0.2$*,* $MemPage = 100$ *and LA_level=maximum*

Oil_Tanker_3 LookAhead Results				
tree level	N. leaves	ovMin	ovMinRel	ovMax
0	0	2493.399902	0.242075	2493.399902
1	0	383.709961	0.079616	1310.000000
2	1	98.476303	0.074350	2181.449951
3	1	80.899414	0.004224	1950.030029
4	5	0.000000	0.000000	437.679932
5	6	0.000000	0.000000	443.100586
6	3	26.199219	0.001836	745.449951
7	7	0.000000	0.000000	794.158997
8	5	636.000000	0.044522	636.000000
9	1	937.969971	0.214478	937.969971
10	1	253.119995	0.081317	253.119995
11	1	302.599609	0.030925	302.599609
12	2			

Table 33: *Look Ahead algorithm applied to Oil_Tanker_3 with* $R = 0.2$*,* $MemPage = 200$ *and LA_level=maximum*

Oil_Tanker_3 LookAhead Results				
tree level	N. leaves	ovMin	ovMinRel	ovMax
0	0	1950.030029	0.244859	1950.030029
1	0	164.873993	0.050335	1219.020020
2	1	195.199219	0.009949	1193.040039
3	4	369.020020	0.073567	621.599609
4	2	450.000000	0.028121	834.579956
5	3	823.599609	0.042707	823.599609
6	1	0.000000	0.000000	0.000000
7	1	570.000000	0.114458	570.000000
8	2			

Table 34: *Look Ahead algorithm applied to Oil_Tanker_3 with* $R = 0.2$, $MemPage = 500$ *and LA_level=maximum*

Oil_Tanker_3 LookAhead Results				
tree level	N. leaves	ovMin	ovMinRel	ovMax
0	0	2549.350098	0.320113	2549.350098
1	0	248.419922	0.052203	1503.699951
2	0	147.410156	0.565158	2530.000000
3	0	0.000000	0.000000	2627.910156
4	4	0.000000	0.000000	2178.699951
5	8	0.000000	0.000000	1360.099609
6	22	0.000000	0.000000	702.780029
7	14	0.000000	0.000000	760.695007
8	10	97.489990	0.020982	144.000000
9	4			

Table 35: *Look Ahead algorithm applied to Oil_Tanker_3 with* $R = 0.3$, $MemPage = 100$ *and LA_level=maximum*

Oil_Tanker_3 LookAhead Results				
tree level	N. leaves	ovMin	ovMinRel	ovMax
0	0	1950.030029	0.244859	1950.030029
1	0	1187.209961	0.118955	1366.487305
2	0	241.341003	0.176098	1162.169922
3	1	0.000000	0.000000	195.199219
4	7	0.000000	0.000000	1133.299805
5	9	187.900391	0.014241	792.800049
6	7	62.200195	0.004354	1488.000000
7	5	401.050049	0.128841	401.050049
8	2			

Table 36: *Look Ahead algorithm applied to Oil_Tanker_3 with* $R = 0.3$, $MemPage = 200$ *and LA_level=maximum*

Oil_Tanker_3 LookAhead Results				
tree level	N. leaves	ovMin	ovMinRel	ovMax
0	0	1950.030029	0.244859	1950.030029
1	0	474.599609	0.021489	1187.209961
2	1	369.169922	0.112708	1162.169922
3	2	81.000000	0.004229	767.398987
4	7	1412.000000	0.431614	1412.000000
5	1	648.335999	0.092407	648.335999
6	2			

Table 37: *Look Ahead algorithm applied to Oil_Tanker_3 with* $R = 0.3$, $MemPage = 500$ *and LA_level=maximum*

Oil_Tanker_2 LookAhead Results				
tree level	N. leaves	ovMin	ovMinRel	ovMax
0	0	1747.000000	0.135180	1747.000000
1	0	405.000000	0.095026	2398.529785
2	1	649.500000	0.152393	2497.000000
3	3	791.599609	0.185734	2534.000000
4	4	721.140137	0.047132	2699.029785
5	4			

Table 38: *Look Ahead algorithm applied to Oil_Tanker_2 with* $R = 0.3$*,* $MemPage = 500$ *and LA_level=2*

the value of the $LookAhead\_level$ argument descreases the final MKtree generated is less optimal in terms of accumulated overlap in the whole tree. For instance, the accumulated overlap associated to the MKtree corresponding to the table 38, where the $LookAhead\_level = 2$, is 16376.779297, with 11 grey nodes and with an accumulated relative overlap per node equal to 1488.798118. While the MKtree associated to the table 31, where $LookAhead\_level = 5$, has an accumulated overlap of 14688.180664, with 11 grey nodes and the accumulated relative overlap per node is 1335.289151. Therefore, the MKtree resulting of the LAA method is more or less optimum depending on higher or lower values of the input argument $LookAhead\_level$.

The effects of the values of the other two parameters, R and $MemPage$, to the computed MKtrees, are similar than in the case of the MOA algorithm. See section 3.1.1 for details.

In figure 13 the MKtree of Oil_Tanker_2 generated with the values of input parameters: $MemPage = 48000 bytes$, $R = 0.2$ and $LookAhead\_level = 7$ (maximum), is presented. The information associated to each node of the tree has the same meaning than in the case of MOA figures (see. section 3.1.1).

In figure 14 the MKtree of Oil_Tanker_2 generated with the values of input parameters: $MemPage = 48000 bytes$, $R = 0.3$ and $LookAhead\_level = 5$ (maximum), is presented. The information associated to each node of the tree has the same meaning than in the case of MOA figures (see. section 3.1.1).

Figures 15, 16 and 17 are images of the MKtrees obtained by using the LAA algorithm of the three oil tankers, presented in several colors corresponding to the MKtree spaces (nodes) up to level 1 or 2 of the tree and differents values of LOD, depending on the

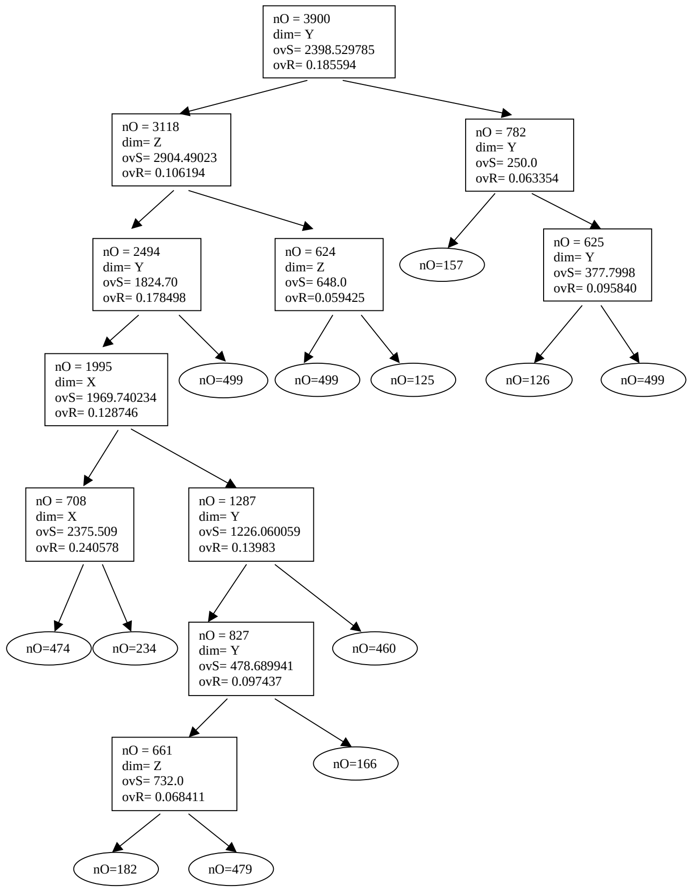

Figure 13: Example of MKtree corresponding to the Oil_tanker_2 generated by using the Look Ahead Algorithm. With $R = 0.2$, $MemPage = 500$ and $LookAhead\_level = 7$ (maximum)

image. The values of the input arguments to the LAA algorithm, for each MKtree, are exposed in each associated caption region.

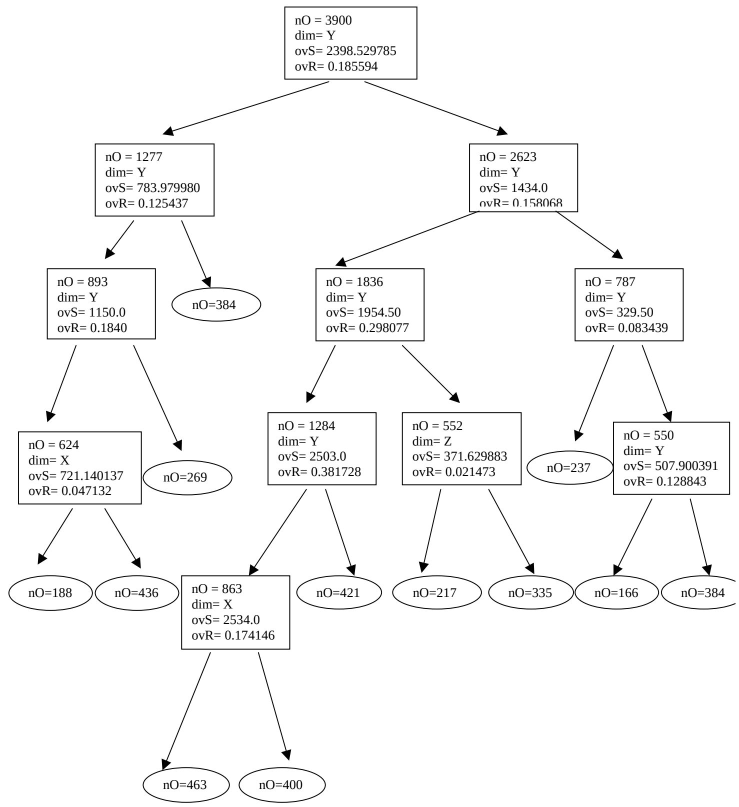

Figure 14: Example of MKtree corresponding to the Oil_tanker_2 generated by using the Look Ahead Algorithm. With $R = 0.3$, $MemPage = 500$ and $LookAhead\_level = 5$ (maximum)

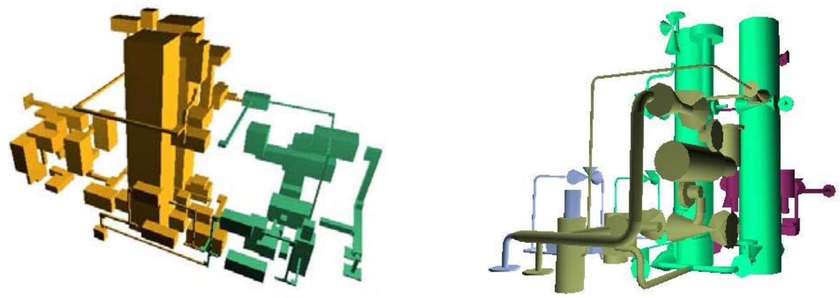

Figure 15: Images of MKtrees of the Oil_Tanker_1. Left image: MKtree up to level 1 and $LOD = 1$. Right image: MKtree up to level 2 and $LOD = max$. Input values of the LAA algorithm: $R = 0.3$, $MemPage = 50objs$ and $LookAhead\_level = max$.

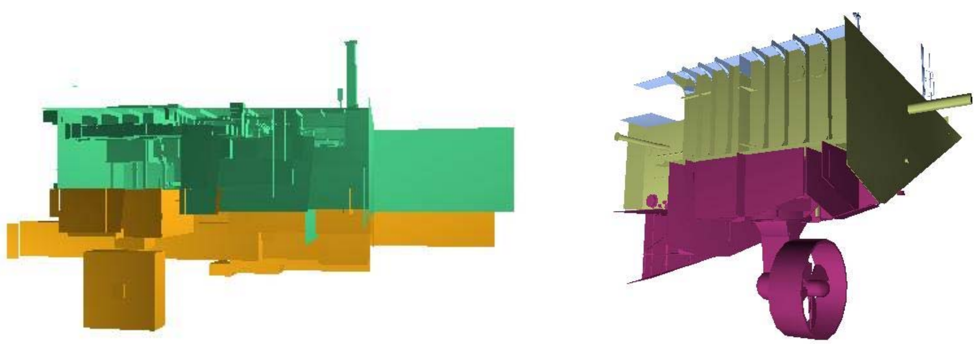

Figure 16: Images of MKtrees of the Oil_Tanker_2. Left image: MKtree up to level 1 and $LOD = 1$. Right image: MKtree up to level 2 and $LOD = max$. Input values of the arguments the LAA algorithm: $R = 0.3$, $MemPage = 48000bytes$ ($500objs$) and $LookAhead\_level = max$.

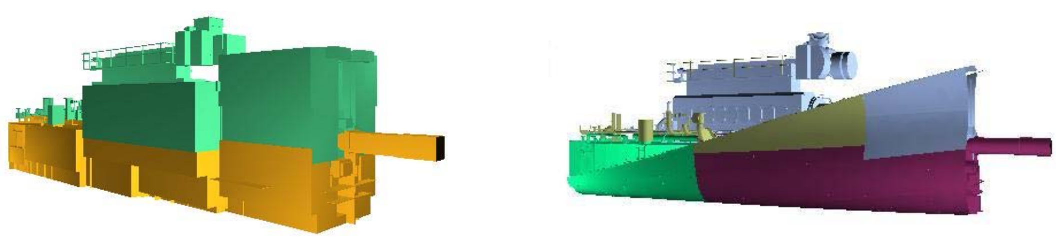

Figure 17: Images of MKtrees of the Oil_Tanker_3, computed by using the LAA algorithm with $R = 0.3$, $MemPage = 48000$ ($500objs$) and $LookAhead\_level = max$. Left image: MKtree up to level 1 and $LOD = 1$. Right image: MKtree up to level 2 and $LOD = max$.

Oil_Tanker_1 MOA accumulated overlap				
R	MemPage	ovAccumulated	N_g	ovAverage
0.01	10	2302.828125	62	37.142387
0.01	20	2017.762939	35	57.650372
0.01	30	1305.764893	25	52.230595
0.2	10	1148.223022	32	35.881969
0.2	20	753.609253	15	50.240616
0.2	30	738.609253	12	61.550770
0.3	10	1558.700439	30	51.956680
0.3	20	753.609253	16	61.343933
0.3	30	783.142944	10	78.314293

Table 39: *Minimum Overlap Algorithm applied to Oil_Tanker_1. Results of the total accumulated overlap in each MKtree*

4 Algorithms evaluation

In the way to compare the *Minimum Overlap Algorithm* (MOA) and the *Look Ahead Algorithm* (LAA) we have computed the total accumulated overlap, *ovAccumulated*, of each MKtree generated by using the two methods. The *ovAccumulated* for an MKtree has been computed as:

$$\sum_{i=1}^{N_g} ovMin_i$$

Where N_g is the number of grey nodes in the MKtree.

In fact, to evaluate the MKtrees obtained by using the two algorithms, MOA and LAA, we have used the *average overlap*, *ovAverage*. This data is the result of dividing the total accumulated overlap for an MKtree by the number of grey nodes that it has, N_g. Thus,

$$ovAverage = ovAccumulated/N_g$$

The results obtained are presented in tables 39, 40 and 41 for the MOA method and in tables 42, 43 and 44 for the LAA method. The *ovAccumulated* and the *ovAverage* are expressed in millimeters in all the tables.

Looking at the tables, in general, we can see that in the most of cases the MKtrees obtained by the LAA method are less expensive, in terms of *ovAverage*, than the MKtrees computed by the MOA algorithm. Observe that, in tables 39 and 42, the number of grey nodes of the MKtree when the input parameters are: $R = 0.01$ and $MemPage = 10$ is less when using the LAA than when using the MOA (49 vs 62), and the *ovAverage* is also less

Oil_Tanker_2 MOA accumulated overlap				
R	MemPage	ovAccumulated	N_g	ovAverage
0.01	100	300182.218750	353	850.374573
0.01	200	168028.343750	218	770.772217
0.01	500	108011.234375	108	1000.104004
0.2	100	47179.414063	66	714.839600
0.2	200	34059.988281	33	1032.120850
0.2	500	16608.240234	12	1384.020020
0.3	100	43289.328125	57	759.461914
0.3	200	32216.587891	29	1110.916870
0.3	500	16376.779297	11	1488.798096

Table 40: *Minimum Overlap Algorithm applied to Oil_Tanker_2. Results of the total accumulated overlap in each MKtree*

Oil_Tanker_3 MOA accumulated overlap				
R	MemPage	ovAccumulated	N_g	ovAverage
0.01	100	235352.046875	363	648.352722
0.01	200	152447.703125	242	629.949158
0.01	500	75662.484375	109	694.151245
0.2	100	35165.062500	73	481.713196
0.2	200	23332.447266	34	686.248474
0.2	500	16408.642578	15	1093.909546
0.3	100	28212.699219	63	447.820618
0.3	200	18183.812500	30	606.127075
0.3	500	10562.667969	12	880.222351

Table 41: *Minimum Overlap Algorithm applied to Oil_Tanker_3. Results of the total accumulated overlap in each MKtree*

Oil_Tanker_1 LAA accumulated overlap					
look level	R	MemPage	ovAccumulated	N_g	ovAverage
24	0.01	10	1423.674316	49	29.054578
17	0.01	20	1050.001587	29	36.206951
14	0.01	30	672.504822	19	35.394991
10	0.2	10	882.499939	29	30.431032
7	0.2	20	882.926575	17	51.936859
6	0.2	30	731.182983	10	73.118301
7	0.3	10	1524.173828	31	49.166897
6	0.3	20	834.283264	16	52.142704
5	0.3	30	662.149292	10	66.214928

Table 42: *Look Ahead Algorithm applied to Oil_Tanker_1. Results of the total accumulated overlap in each MKtree*

Oil_Tanker_2 LAA accumulated overlap					
look level	R	MemPage	ovAccumulated	N_g	ovAverage
2 (188)	0.01	100	290495.156250	359	809.178711
2 (155)	0.01	200	164230.750000	216	760.327576
2 (95)	0.01	500	100999.585938	96	1052.078979
12	0.2	100	35009.968750	60	583.499451
9	0.2	200	24902.388672	31	803.302856
7	0.2	500	14218.088867	11	1292.553533
9	0.3	100	36231.816406	58	624.686462
7	0.3	200	26248.199219	28	937.435669
5	0.3	500	14688.180664	11	1335.289151

Table 43: *Look Ahead Algorithm applied to Oil_Tanker_2. Results of the total accumulated overlap in each MKtree*

Oil_Tanker_3 LAA overlap accumulated					
look level	R	MemPage	ovAccumulated	N_g	ovAverage
2 (176)	0.01	100	125628.828125	214	587.050599
2 (151)	0.01	200	109005.312500	162	672.872299
2 (106)	0.01	500	84843.609375	107	792.930929
10	0.2	100	21815.908203	69	316.172577
12	0.2	200	17254.753906	32	539.211060
8	0.2	500	8602.962891	13	661.766357
9	0.3	100	23687.410156	61	388.318207
8	0.3	200	15367.146484	30	512.238220
6	0.3	500	9968.633789	12	830.719482

Table 44: *Look Ahead Algorithm applied to Oil_Tanker_3. Results of the total accumulated overlap in each MKtree*

(29.054578 vs 37.142387). And, for the Oil_tanker_1, in general, the *ovAverage* column of the second table (42) has lower values for the *ovAverage* than the corresponding column of the first table (39).

Observing the tables 40 and 43 for the Oil_tanker_2, and 41 and 44 for the Oil_tanker_3, we see that the N_g and the *ovAverage* are less when using the LAA method than when using the MOA method.

Those results confirms that the LAA is more efficient, in terms of memory occupancy and in the number of page retrievals from disk to core memory (even though more expensive in terms of time computing), than the MOA method. This is due to that the first one looks for the best MKtree that it can computes from the minimum overlap, in a trial way.

The obtained value of *ovAverage* can give us, also, an idea of the *hybridness degree* of the resulting MKtree. When its value is low it implies that the MKtree is near to be an Kdtree. When its value is high, it means that the resulting MKtree is near to be an Rtree. Thus, after looking the results applying the two algorithms to the three oil_tankers, we can conclude that the LAA method produces better MKtrees (that are nearest to Kdtrees) than MOA method. In other words, the overlapping spaces between different grey node regions are less when using the LAA than when using the MOA method. Then, the number of blocks in disk to store the whole MKtree is less, also. Even more, when the level of the trial process is high, *look_ahead_level*, the results obtained by the LAA are much better than the ones obtained by the MOA.

On the other hand, as mentioned before, every resulting overlap computed by our algorithms and presented in the present chapter is expressed in millimeters. Thereby, the accumulated overlap in average, $ovAverage$, is expressed in mm too. Observing the $ovAverage$ rank values contained in tables 39, 40 and 41 for the MOA method and in tables 42, 43 and 44 for the LAA method, and taking into account the dimensions of the input data (see table 1), we can conclude that the results obtained by our methods are acceptable and satisfactory. Look, for instance, at table 44 where the $ovAverage$ minimum value is $316.172577mm$ and the maximum is $830.719482mm$.

5 Conclusions

In this paper we have introduced the MKtrees to represent complex systems and two algorithms to generate them, *Minimum Overlap* and *LookAhead*. Those algorithms compute automatically an MKtree that represents a hierarchical subdivision and grouping of the scene objects guaranteeing a minimum space overlap. The algorithms minimize the amount of disk accesses.

The methods to generate MKtrees, MOA and LAA, have been exposed and explained in detail. The results of applying them to the input data described, corresponding to three oil_tankers, are presented. The different results obtained depending on the values of the input parameters to the algorithms are exposed and analyzed. Finally, the evaluation and comparation of the efficiency of the two methods have been exposed, too.

Other bounding volume hierarchies that have been proposed are not based on external memory representations and require considerable storage on memory (k-dops [KHM+98], OBBtrees [GLM96, BCG+96, GASF94], spherical shells [KPLM98], sphere trees [Hub96]).

6 Acknowledgements

This work has been partially supported by the CICYT project TIC–98–0586–C03–01.

References

[AAB99] C. Andújar, D. Ayala, and P. Brunet. Validity-preserving simplification of very complex polyhedral solids. In *Proc. 5TH Eurographics workshop on virtual environments, EGVE99*, pages 1–10, 1999.

[ABA01] C. Andújar, P. Brunet, and D. Ayala. Robust topology simplification. *ACM Transactions on Graphics*, 2001. Accepted for publication.

[And99] C. Andújar. *Octree-based Simplification of Polyhedral Solids.* PhD thesis, Dept. LSI, Universitat Politècnica de Catalunya, Barcelona, Spain, 1999.

[BCG+96] G. Barequet, B. Chazelle, L.J. Guibas, J.S.B. Mitchell, and A. Tal. *BOXTREE*: A hierarchical representation for surfaces in 3*D*. In *EUROGRAPHICS Conf. Proc.*, volume 15, pages 387–396. Blackwell Publishers, August 1996.

[BKSS90] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The r*-tree: An efficient and robust access method for points and rectangles. In Hector Garcia-Molina and H. V. Jagadish, editors, *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, May 23-25, 1990*, pages 322–331. ACM Press, 1990.

[DECM98] A. Duch, V. Estivill-Castro, and C. Martínez. Randomized k-dimensional binary search trees. In K.-Y. Chwa and O.H. Ibarra, editors, *Proc. of the 9th Int. Symp. on Algorithms and Computation (ISAAC)*, volume 1533 of *Lecture Notes in Computer Science*, pages 199–208. Springer-Verlag, 1998.

[FNB01] M. Franquesa-Niubo and P. Brunet. Collision detection using multirresolution *Kdtrees. Computer Graphics Forum. Submitted for publication*, 2001.

[GASF94] A. Garcia-Alonso, N. Serrano, and J. Flaquer. Solving the collision detection problem. *IEEE Computer Graphics and Applications*, pages 36–43, May 1994.

[GLM96] S. Gottschalk, M.C. Lin, and D. Manocha. *OBBtree*: A hierarchical structure for rapid interference detection. In *ACM SIGGRAPH Conf. Proc.*, pages 171–180, August 1996.

[Gut84] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *In Proc. ACM SIGMOD Int. Conference on Management of Data*, pages 47–57, June 1984. Boston, MA, USA.

[He99] T. He. Fast collision detection using *QuOSPO* trees. *Symp. on Interactive 3D Graphics, Atlanta, ACM*, pages 55–62, 1999.

[Hub96] Philip M. Hubbard. Aproximating polyhedra with spheres for time–critical collision detection. *ACM Transactions on Graphics*, 15(3):179–210, July 1996.

[JW89] D. Jevans and B. Wyvill. Adaptive voxel subdivision for ray tracing. In *Proc. Graphics Interface*, pages 164–172, 1989.

[KHM+98] J. T. Klosowski, M. Held, J. S.B. Mitchel, H. Sowizral, and K. Zikan. Eficient collision detection using bounding volume hierarchies of k–dops. *IEEE Transactions on Visualization and Computer Graphics*, 4(1):21–36, january-march 1998.

[KPLM98] S. Krishnan, A. Pattekar, M. Lin, and D. Manocha. Spherical shell: A higher order bounding volume for fast proximity queries. In *In Proc. of Third International Workshop on Algorithmic Foundations of Robotics*, 1998.

[MF99] G. Mueller and D. W. Fellner. Hybrid scene structuring with application to ray tracing. In *Proc. of the Intl. Conference on Visual Computing*, pages 19–26, 1999.

[Sam90] H. Samet. *The Design and Analysis of Spatial Data Structures.* Addison-Wesley, 1990. ISBN 0-201-50255-0.

[SRF87] T. Sellis, N. Roussopoulus, and C. Faloutsos. The R^+–tree: A dynamic index for multidimensional objects. In *Brighton 13th.*, pages 507–518. VLDB Conf., 1987.

[WLML99] A. Wilson, E. Larsen, D. Manocha, and M.C. Lin. Partitioning and handling massive models for interactive collision detection. *Computer Graphics Forum*, 18(3):319–329, September 1999.