# Filtering, Clustering and Hierarchy Construction: a New Solution for Ray-Tracing Complex Scenes

Frédéric Cazals,[1] George Drettakis,[1,2] Claude Puech [1]

[1] *i*MAGIS-IMAG, BP 53, F-38041 Grenoble, cedex 09 FRANCE. *i*MAGIS is a joint project of CNRS, INRIA, INPG and UJF. E-MAIL: Frederic.Cazals@imag.fr.
[2] Department of Software, Technical University of Catalunya, Diagonal 647, 08028 Barcelona, SPAIN.

**Abstract**

*Data structures that handle very complex scenes (hundreds of thousands of objects) have in the past either been laboriously built by hand, or have required the determination of unintuitive parameter values by the user. It is often the case that an incorrect choice of these parameters can result in greedy memory requirements or severely degraded performance. As a remedy to this problem we propose a new data structure which is fully automatic since it does not require the user to determine any input parameters. The structure is built by first filtering the input objects by size, subsequently applying a clustering step to objects of the same size and finally building a hierarchy of uniform grids ($\mathcal{HUG}$). We then show that this data structure can be efficiently constructed. The implementation of the $\mathcal{HUG}$ shows that the new structure is stable since it's memory requirements grow linearly with the size of the scene, and that it presents a satisfactory compromise between memory usage and computational efficiency. A detailed comparison with previous data structures is also presented in the results.*

## 1. Introduction

Dealing with very complex scenes is one of the major challenges for current computer graphics research. To facilitate manipulation of such environments it is necessary to develop spatial subdivision structures which allow the implementation of efficient searching algorithms for such applications as rendering (with ray-tracing or radiosity-style methods), collision detection in animation and also for interactive environments (virtual reality etc.).

One of the more challenging aspects of creating such structures is the treatment of scenes which contain large changes in scale, for example the model of a building, which at successive levels of scale contains the walls, the rooms, the doors, windows and furniture, and finally the small detail objects such as the knobs on a television or a phone. For such scenes it is often the case that the complexity of the geometry is to be found in the smaller scales (e.g., thousands of polygons for the models of household appliances).

The data structure we introduce in this paper represents a new approach, by first *filtering* the input objects by size, then *clustering* objects of the same size and finally constructing a hierarchy of uniform grids. Our structure is fully automatic since the user does not need to specify any parameters, and is thus capable of treating large scenes efficiently, without the need for time-consuming experimentation to determine unintuitive parameters.

### 1.1. Previous Structures for Highly Complex Scenes

From the very first presentation of the ray-tracing algorithm[1], it became clear that a spatial subdivision structure is required to cope with the millions of rays which are intersected with the objects in the environment. In the early algorithms hierarchies of bounding boxes were used[1, 2]. These methods have been extended to more generalised bounding volumes[3, 4, 5]. BSP trees[6] and regular subdivisions of space are also popular in dealing with the ray-tracing problems[7]. Octree structures have been used[8], as well as uniformly subdivided grids[9, 10].

In a few of the previously cited papers very large environments have been treated. Most notably Snyder and

Barr[10] handled scenes with millions of polygons, but user intervention was required to construct the hierarchical data structure. Large environments have also been treated by recursively subdividing uniform grids[11] and also for modelling[12].

### 1.2. Complexity analysis

The performance of spatial subdivision data structures is notoriously difficult to analyse. Some previous work has been done however[13, 14, 15]. In this section we present a more careful look at some of the spatial subdivision structures which in practice have been used with success[16, 17]. We are particularly interested in the following classes of spatial subdivision structures: *uniform grids*, *recursive grids* and *octrees* (i.e., object-octrees).


**Figure 1:** *(a) Uniform grid / uniform distribution* *(b) Uniform grid / non uniform distribution*

#### 1.2.1. Uniform grids

Uniform grids are built by subdividing the sides of the bounding box $\Delta$ of the scene, along the $x, y, z$-axes into $n_x$, $n_y$ and $n_z$ subdivisions. Each element of this subdivision is called a *voxel*. In each voxel a pointer toward the items intersecting it is stored.

Choosing $n_x = n_y = n_z = \sqrt[3]{n}$ yields $n$ voxels. This results in practically "optimal" performance if the objects of the scene $\mathcal{O}$ are scattered uniformly in $\Delta$. An example of such a situation is shown in Figure 1(a). Nevertheless, if some voxels contain too many objects the uniform grid data structure will fail, since when a ray traverses a highly populated voxel too many ray-objects intersection tests are performed (Figure 1(b)).

A typical remedy used in practice are higher subdivision factors $n_x, n_y, n_z$, but the time gained by avoiding ray-object intersections is eventually lost by the additional cost of voxel traversals. This approach is also limited by the rapidly increasing memory requirements. In addition arbitrary subdivision parameters are difficult to use since experimentation is necessary to find the best trade-off of "cost of the intersections" vs. "cost of traversal". Another alternative is a recursive data structure, discussed next.

#### 1.2.2. Recursive grids

Jevans[11] proposed a solution in which each voxel containing a number of pointers greater than MAXP (where MAXP stands for MAXimum number of Polygons), is recursively subdivided. In the tests presented[11], the same subdivision factors are used for every recursive subdivision (e.g. $n_x = n_y = n_z = 10$). As a consequence the memory requirements grow explosively.

Jansen[16, 17] proposes an adaptive subdivision criterion, which we will call the $\sqrt[3]{}$ criterion in what follows. This approach consists in setting $n_x = n_y = n_z = \sqrt[3]{n_i}$ for voxels that contain $n_i >$ MAXP items. In practice it seems that this choice works quite well. Nonetheless, several questions remain: is there an optimal termination condition MAXP ? If such a condition exists, does it avoid the problem of explosive memory growth ?

These questions are still unanswered in computer graphics, but have received some attention in the theory of search and sort bucket-like data structures[18]. We discuss a brief outline of this work in the following.

Suppose we want to store a set of real numbers $\{x_1 \ldots x_n\}$ that are known to belong to the real interval $\Delta = [a, b]$. The usual bucket-like data structures operate as follows: *(i)* subdivide $\Delta$ into $n$ intervals of equal length *(ii)* in all the buckets where the number of points is $> b$ (with $b$ a constant between 4 and 10 generally) subdivide and iterate recursively. Since the $\{x_i\}$ are assumed to be independent realisations of random variables, theorems assert that *(i)* for most random variables *"Two levels are as good as any"*[19] i.e at depth 2 most of the buckets contain less than $b$ points *(ii)* the number of buckets necessary to store $n$ is $3Mn$ where $M$ is the maximum of the probability density.

These theoretical results (mainly in one dimension) provide important intuition into the expected behaviour of recursive grids for very complex three-dimensional scenes in which large scale changes occur.

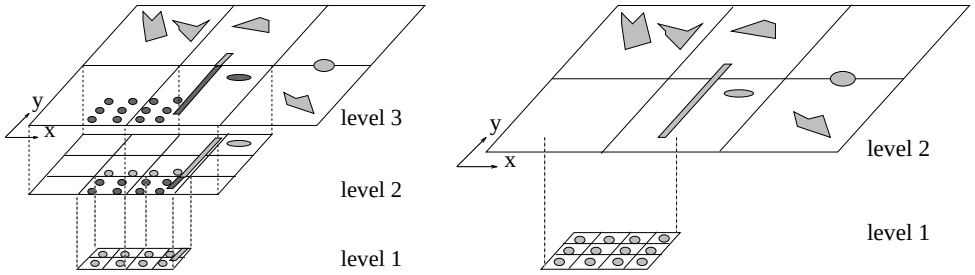



**Figure 2:** *(a) Recursive grid* *(b) Hierarchy of Uniform Grids*

A first conclusion is that the hierarchy should not be very deep and that it is possible to restrict the memory cost more efficiently than when using a constant subdivision factor. A major difference however is that points in one dimension are not shared between buckets. For the three dimensional case in which the contents of $\mathcal{O}$ are polygons, such overlap may occur. This has two important negative consequences:

- Suppose there are *N* polygons that are coplanar and the common voxel intersection of which is non-empty. If $N >$ MAXP, the recursion will not terminate.
- The number of references towards an object might be large: see the "long" item on Figure 2(a), that runs across 5 buckets.

### 1.2.3. Octrees

Octrees[8] can be viewed as a special case of recursive grids for which subdivision into eight sub-voxels is performed at every step. Their advantage is a natural adaptation to the geometric complexity of a scene and the fact that special optimisation can be performed for ray-traversal. Their main drawbacks are that a hierarchy of large depth may be created, a penalty of traversing the hierarchy is often incurred and that duplication of objects in lists is frequent.

## 1.3. Towards a better solution

Given the discussion above, we can now put our proposed solution in context. The steps of the new algorithm are summarised as follows:

1. Gather the objects into subsets of similar size.
2. Foreach group of objects of similar size, group the neighbours into *clusters.*
3. Construct a grid with the $\sqrt[3]{\ }$criterion for each cluster.
4. Construct a hierarchy of these grids.

The data structure we introduce can thus be seen as a recursive grid where unnecessary intermediate levels and multiple references towards items of different sizes are suppressed. An example of a $\mathcal{HUG}$ is shown in Figure 2(b) for the set $\mathcal{O}$ of Figure 2(a).

The filtering and clustering steps effect a *bottom-up* construction of the data structure, in the sense we that start with the objects and by grouping them, we construct a hierarchical data structure. This is in strong contrast with all other automatic methods which subdivide their structure adaptively in a *top-down* manner (there do however exist previous methods which construct hierarchies manually [20, 21]). As we shall see in Section 3 the use of such top-down methods can result in significant additional memory cost.

## 2. A New Structure: the Hierarchy of Uniform Grids

For the purposes of our construction we consider the *bounding boxes* of objects. In this manner our structure does not depend on the object type (e.g., polygons, bicubic surfaces etc.). In what follows we thus use the words

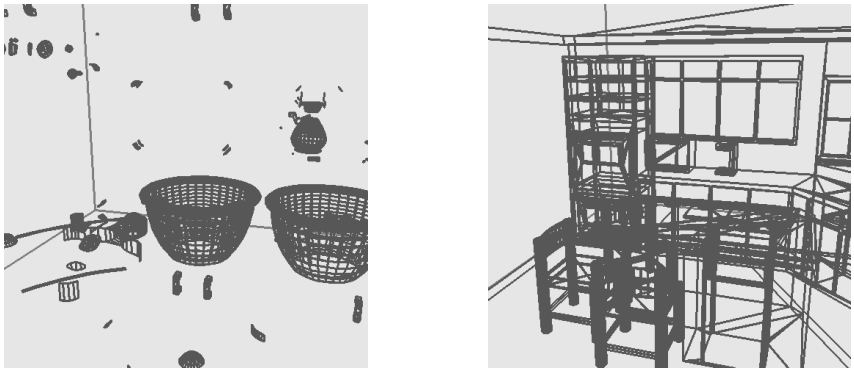
object and bounding box interchangeably. We define the following properties of objects: *(i)* the *length* of an object that will be its diameter *(ii)* the *projection* of an object along an axis that will be a line segment on that axis *(iii)* a distance function $d(s_1, s_2)$ between segments *(iv)* a distance function $d(o_1, o_2)$ between objects. For a pair of objects or segments $A, B$, and $a, b$ points of $A$ and $B$ respectively, the distance function is defined as $d(A, B) = \inf distance(a, b)$,$\forall a \in A$ and $\forall b \in B$. Finally, given the lengths $\mathcal{D}$ =$\{d_1, ..., d_n\}$ of all the objects from $\mathcal{O}$ we will note $d_{inf} = \inf_i length_i$ and $d_{sup} = \sup_i length_i$. Thus $d_{inf}$ is the length of the shortest item, and $d_{sup}$ is the length of the longest item.

### 2.1. Filtering the Objects by Size

**Definition 1 (Filter of lengths)** *Given a set $\mathcal{O}$ of objects, we call filter $\mathcal{F}$ a strictly increasing sequence of positive real numbers $\{f_1, ..., f_m\}$ such that $d_{inf} \in [f_1, f_2)$ and $d_{sup} \in [f_{m-1}, f_m)$. A $level$ of the filter $\mathcal{F}$ is an interval $[f_i, f_{i+1})$. Define also (i) $\bar{f}_k = (f_k + f_{k+1})/2$ the average length of level $k$ (ii) $L_k$ the set of all objects from $\mathcal{O}$ the lengths of which $\in [f_k, f_{k+1})$.*

The filter is used as follows: for $o_i \in \mathcal{O}$ we determine the position of $length(o_i)$ in $\mathcal{F}$ by a binary search. The filtering operation costs $O(n \log m)$ and yields at most $m-1$ non empty levels. If we consider the kernel-estimated density[22] $\widehat{f_{\mathcal{D}}}$ from the set $\mathcal{D}$ computing a good filter is a difficult problem. For typical computer graphics scenes this density contains a few peaked modes (because the complexity of the scene lies in the small objects that have similar sizes) and the filter can be determined by searching these modes on a histogram. In practice we subdivide the histogram into a given number of slices (e.g. three), such that each slice contains the same number of points of the histogram (see also Tsai and Chen[23]). This results in very satisfactory performance.

To give a precise idea of the filtering process, the following images show the result of filtering the scene *kitchen* which is used in Section 3 for the experimental study (see Figures 10(a) and 10(b)). Because of the intuition *"Two levels are as good as any"*, we computed a three level filter thus gathering objects in groups of three sizes.



(a) (b)

**Figure 3:** *Showing filter levels (a) Level 0 (small objects) (b) Level 2 (big objects)*

However, gathering the objects by homogeneous size is not sufficient: consider for example in Figure 10(a) the items corresponding to the bowl on the table and the coffee-pot on the counter: we cannot build a single uniform grid to store these two sets of objects without being faced with the problems mentioned in Section 1.2.1. We therefore wish to isolate two clusters of objects, one for the bowl and one for the coffee-pot.

### 2.2. Clustering Objects of the Same Size

#### 2.2.1. Projections and clusters

Given a set of objects $\mathcal{O}$ , finding a partition of $\mathcal{O}$ into clusters according to a given criterion is a well studied topic of statistics. Unfortunately, the best clustering methods that do not presuppose the final number of clusters[24, 25]

are $\Theta(n^2)$-space and $\Theta(n^2 \log n)$-time. For very large environments such a cost is unacceptably high. To see why, consider the algorithm step at which a new item is added to the previously computed clusters. This step requires testing this item against all the clusters in order to find the best fit.

As an alternative to this "grouping" method, we choose a "divisive" approach illustrated below. Consider for example Figure 4 where we would like to construct three clusters $A, B$ and $C$ from the set $A \cup B \cup C$. An appropriate necessary condition for two objects to belong to the same cluster is that their projections along the two $x, y$ axis themselves form a cluster. The idea is therefore to successively examine the projections of the items along $x, y$ and $z$ and then to split the initial input into sub-groups that cannot form a cluster. As will be seen, working with the projections enables us to compute clusters which are somewhat more restricted, but in a more efficient manner.

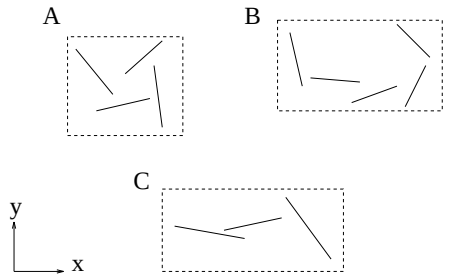



**Figure 4:** *A collection of clusters*

**Definition 2 (**$\delta$**-connectivity -** $\delta$**-cSet)** *Two objects* $o_i$ *and* $o_j$ *are called* $\delta$-connected *where* $\delta \in \mathbb{R}^{*+}$ *if their distance* $d(o_i, o_j) < \delta$. *If they belong to the same level* $k$ *of the filter* $\mathcal{F}$, *let* $\delta_k$ *be the* $\delta$ *associated to level* $k$. *We call* connectivity coefficient $\alpha$ *a strictly positive real number, and we set* $\delta_k = \alpha\ f_k$. *A set* $\mathcal{O}$ *is said to be* $\delta$*-connected or a* $\delta$*-cSet if* $\forall\ i \in 1..n$ *there exists a* $j \neq i$ *such that* $d(o_i, o_j) < \delta$.

Two items belong to the same cluster if the distance separating them is less than a small "percentage" of their relative length (which is nearly the same since the items belong to the same filter level). In practice $\alpha$ is set to a small number (e.g., $\alpha = .01$). The performance of the algorithm is not sensitive to the choice of $\alpha$. Varying its value between .00125 to .08 resulted in practically no change in the performance of the structure constructed for the test scenes of Section 3.

**Definition 3 (Property** $\pi_{xyz}$**, potential cluster and cluster)** *A set* $\mathcal{O}$ *is said to be* $\pi_x$ *or to have property* $\pi_x$ *if the objects' projections along axis* $x$ *form a* $\delta$*-cSet. The opposite will be noted* $\pi_{\bar{x}}$. *Thus a set* $\pi_{x\bar{y}\bar{z}}$ *will have the property* $\pi_x$, $\pi_{\bar{y}}$ *and* $\pi_z$. $\mathcal{O}$ *is said to be a potential cluster if it has property* $\pi$ *along one or two axis, and a cluster if it is* $\pi_{xyz}$. *By CPC we denote either a cluster or a potential cluster.*

In Figure 4, we see that neither $A \cup B$ nor $A \cup C$ are clusters since these two sets are respectively $\pi_{\bar{x}y}$ and $\pi_{x\bar{y}}$. It is important to understand that $\delta$*-cSet*s are formed independently for each of the one-dimensional projections allowing the correct isolation of clusters.

### 2.2.2. The Clustering Algorithm

The clustering algorithm consists of two parts: (a) *Isolate* that finds the subsets of *dataSet* that verify the necessary condition along *aDirection*, and (b) *Cluster3d* which computes the clusters using *Isolate*. The input of the algorithm is an array of objects *dataSet*, and more precisely all the items located between two indices $n_1$ and $n_2$ of this array. The information recorded for each cluster or potential cluster ( CPC ) is: *(i)* a pair of indices $index_1$ and $index_2$: all the items contained in the array *dataSet* between those indices belong to the same CPC *(ii)* a flag *dir* indicating the last direction where property $\pi$ has been checked (1 for $x$, 2 for $y$, 3 for $z$) *(iii)* a counter *stable* indicating how many directions have property $\pi$ (thus a CPC is a cluster if $stable = 3$).

The output is a stack *stackOfCPC* of CPC . Recall that the projection of object $o_i$ along a given direction is noted $s_i$ ($s$ for segment). The two endpoints of segment $s_i$ are noted $s_i.left$ and $s_i.right$.

Algorithm Isolate

**Variables**

*dataSet*: array[$n_1..n_2$] of objects
*aDir*: flag indicating which direction is processed (x, y or z)
*stackOfCPC*: stack of CPC
*pc*: a CPC

**Algorithm**

**funct** $Isolate(dataSet, n_1, n_2, aDir, stackOfCPC)$
$pc \leftarrow$ Pop $stackOfCPC$
Sort objects $dataSet[n_1, n_2]$
by increasing leftmost point along $aDir$
$pc.index_1 = n_1;\ pc.dir = aDir;$
**for** $i = n_1 + 1$ **to** $n_2$ **do**
**if** $d(s_i, s_{i-1}) < \delta_k$
**then if** $(s_{i-1}.right > s_i.right)$
**then** exchange($s_i$, $s_{i-1}$) **fi**
**else**
$pc.index_2 = i - 1;$
Push $pc$ on $stackOfCPC$;
$pc.index_1 = i;\ pc.dir = aDir$ **fi**
**od**
$pc.index_2 = i;$ Push $pc$ on $stackOfCPC;$


**Figure 5:** *Algorithm Isolate*

**Proposition 1** Algorithm $Isolate$ runs in $O(n\ \log\ n)$

PROOF

- The sort step ensures that when a new segment is inserted, the necessary condition has to be checked between the left point of this segment and segments located on its left. In Figure 5 the segments are therefore inserted according to their indices.
- The swap operation **if** $s_{i-1}.right > s_i.right$ is such that when $s_{i+1}$ will be inserted, the rightmost point of the previously inserted segments will be the right point of the segment on its left. For example (Figure 5) when 4 is inserted, the test is performed with 1 and not with 3.
- These two conditions guarantee $O(1)$ operations per insertion and the algorithm therefore runs in $O(n\ \log n)$ because of the sort step.

△

Given the algorithm $Isolate$, we can present the complete clustering algorithm in three dimensions: $Isolate$ is successively applied in the direction of the three axes until sets of items *stable* (i.e. not modified by successive calls to $Isolate$ along $x$, $y$ and $z$) are found. Since we know we will get at most $n$ clusters, we use an array of size $n$ as a "double stack" as follows: the bottom part of the array contains the stack of potential clusters that are computed by $Isolate$, and the top part contains the stack of final clusters. In other words, once a potential cluster is found to be $\pi_{xyz}$, it is popped from the bottom stack and pushed on the top stack. These two stacks grow in opposite directions, but we are sure they never overlap since we use an array of size $n$ and at any time we know there are at most $n$ clusters (potential and final).

Algorithm Cluster3d

**Environment**
*stackOfCPC*: array used as a "double" stack storing the potential clusters and the final clusters

**Algorithm**
$pc.index_1 = 1;\ pc.index_2 = n;\ pc.dir = 0;$
Push $pc$ on bottom stack
**while** bottom stack is not empty **do**
**begin**

$pc \leftarrow$ Pop a potential cluster on bottom stack;
$aDir = (pc.dir + 1)\%3;$
$Isolate(dataSet, pc.index_1, pc.index_2, aDir, stackOfCPC);$
**if** top cluster on bottom stack is stable
**then begin**

Pop this cluster;
Push it on the top stack
**end fi**
**end**

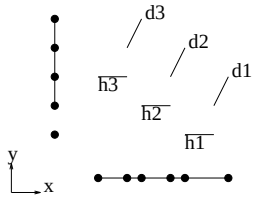



**Figure 6:** $S_0$

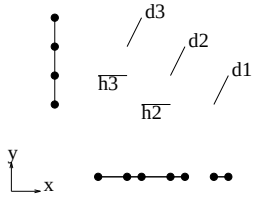



**Figure 7:** $S_1$

**Proposition 2** *The complexity of algorithm* $Cluster3d$ *is output sensitive and the best and worst case complexities are* $\Omega(n \log n)$ *and* $O(n^2 \log n)$*, respectively.*

PROOF
To see the lower bound, suppose $Isolate$ along the first direction, having processed $x$, finds that no two items can be put into the same cluster. $Cluster3d$ then stops and returns $n$ clusters of one item.

The upper bound is somewhat more involved. The worst case will occur for a set $\mathcal{S}$, such that each application of $Isolate$ along each direction results in the removal of one item from a cluster. This can occur at most $O(card(\mathcal{S}))$ times, and thus the cost is $\sum_{i=1}^{n-1} O(i \log i) = O(n^2 \log n)$. $\triangle$

As an example consider Figures 6 and 7 where the the set $S$ is composed of two kinds of segments, horizontal and diagonal. The circles show the projection of the endpoints along each axis.
- The set $\mathcal{S}_0 = \{h_1, h_2, h_3, d_1, d_2, d_3\}$ is $\pi_x$.
- *Isolate* along $y$ removes $h_1$. The resulting new set $\mathcal{S}_1 = \mathcal{S}_0$ - $h_1$ is $\pi_{\bar{x}y}$.
- *Isolate* along $x$ removes $d_1$ from $\mathcal{S}_1$ and $\mathcal{S}_2$ is $\pi_{x\bar{y}}$.
- As a consequence $h_2, d_2, h_3$ are successively removed.

For all our experiments, the ratio $\sum_{All\ the\ sorts\ steps}(N_i \log N_i)/n \log n$ was computed (with $N_i$ the cardinal of subsets treated by $Isolate$). For these test cases the value of the ratio belongs to the interval $[3., 5.]$, thus providing a strong indication that the running time is close to the lower bound complexity. Figure 8 shows some clusters computed from the level containing the smallest objects of the scene *kitchen*.

### 2.2.3. Gridding the clusters

Once $n$ objects of the same size have been gathered in a cluster, we have to construct a uniform grid to store the cluster. Taking $\sqrt[3]{n}$ subdivisions along each axis gives $n$ buckets but does not guarantee $O(1)$ objects per bucket. An alternative choice of subdivision parameter would be $n_x = \lambda(\cup_{i=1..n} P_x(o_i))/\overline{\lambda_{i=1..n}(P_x(o_i))}$, where $P_x$ is the projection of $o_i$ on the $x-$axis and $\lambda$ is the Lebesgue measure (in this case length). This is the measure of the overlappings along an axis, i.e., the length of the union of the projections divided by the average length of the projections. Unfortunately, for some sets $\mathcal{O}$ this can give $n_x = n_y = n_z = O(n)$ resulting in $O(n^3)$ buckets. Such memory cost would render the structure unusable in practice. The previously described $\sqrt[3]{\ }$ criterion is therefore used: the gridding of a cluster of $n$ objects contains exactly $n$ voxels. An exception is made for cluster containing the largest object where experiments have shown that $2n$ voxels lead to a substantial improvement. Determining optimal $n_x, n_y, n_z$ remains an open problem.

### 2.3. Creating the Hierarchy of Grids

Suppose that we have now filtered the items of $\mathcal{O}$ and the clusters of all the non-empty levels have been computed. In addition for each cluster the subdivision parameters have been determined. We also create a single cluster, called *World*, the bounding box of which contains the objects of the higher level of the filter (large objects) and which also covers all the bounding boxes of lower levels clusters. The construction of the $\mathcal{HUG}$ is given below, performed top-down for the filter levels. This must be not be confused with the top-down adaptive subdivision of recursive grids and octrees, which completely determines the form of those data structures. For the $\mathcal{HUG}$, the bottom-up filtering and clustering define the form of the structure.

#### 2.3.1. Algorithm

**Algorithm**

**proc** $CreateHUG$
    create the highest level cluster gridding and store its objects
    **for** all the others filter levels, in decreasing order **do**
        **for** each cluster of the level **do**
            create the cluster gridding and store its objects
            recursively insert this gridding in the hierarchy
        **od**
    **od**.

Let us consider the following example where the filter has 3 levels, levels 1, 2 and 3 respectively containing the clusters 1a and 1b, 2 and 3 (the grids are represented with continuous lines, and the items stored in each grid in dashed lines), shown in Figure 9.

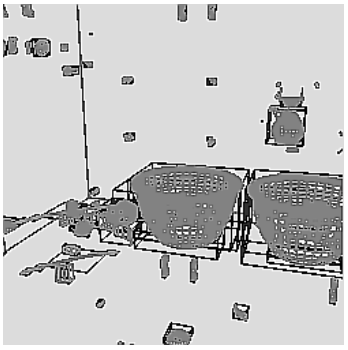


**Figure 8:** *Some level 0 clusters in kitchen*

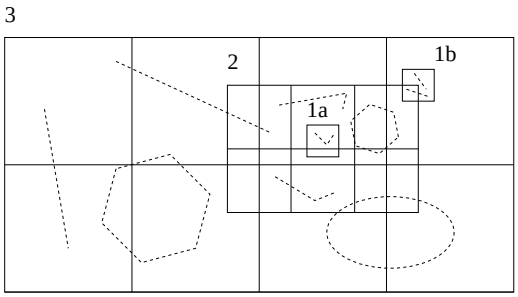



**Figure 9:** $\mathcal{HUG}$ *construction*

- We first create grid 3, which contains the whole scene.
- Next we create grid 2. A pointer towards grid 2 is inserted in each voxel it intersects.
- Next grid 1a. Since it intersects voxel (3,2) of grid 3, which contains a sub-grid, we try to insert grid 1a into the subgrid (grid 2). Since grid 1a is contained in grid 2, we store a pointer towards 1a in voxels (2,1) and (2,2) of grid 2 and return.
- Finally grid 1b. Since voxel (4,2) of grid 3 already contains a sub-grid, we try the recursive insertion. But grid 2 does not contain grid 1b and we set a pointer towards 1b in voxel (3,2) of grid 2 and voxel (4,2) of grid 3.

#### 2.3.2. Traversal of the structure for ray-casting

The traversal of the $\mathcal{HUG}$ is performed as follows. When a ray enters the grid *World*, the starting voxel is found in the same manner as for uniform grids [9]. First we intersect the ray with every object contained in this voxel, and we then recursively visit each subgrid referenced in this voxel. After returning from the recursion we visit the next voxel along the ray path. Note that if an intersection is found the ray-traversal will stop in the same manner as for uniform grids.

## 3. Implementation, Results and Comparative Study

We have fully implemented the filtering, clustering and hierarchical construction of the $\mathcal{HUG}$. In addition we have implemented recursive grids and simulated octrees in the same manner as Jevans and Wyvill[11]. The environment used was *OORT*, which is a public domain ray-tracer [26], using the uniform grid implementation of Amanatides and Woo [9] adapted to C++. It must be noted that once uniform grids were adapted to C++, the implementation of recursive grids and octrees was performed simply by creating a subclass. The complete filtering and clustering algorithms are also relatively simple to implement, with the entire source for these operations being less than 1500 lines of C++. Thus, given access to an existing ray-tracer already containing regular grids, implementing the $\mathcal{HUG}$ is a relatively simple endeavour.

Finally, since the same code base was used for the three kinds of hierarchies (recursive grids, octrees, $\mathcal{HUG}$), the performance comparisons are "fair".

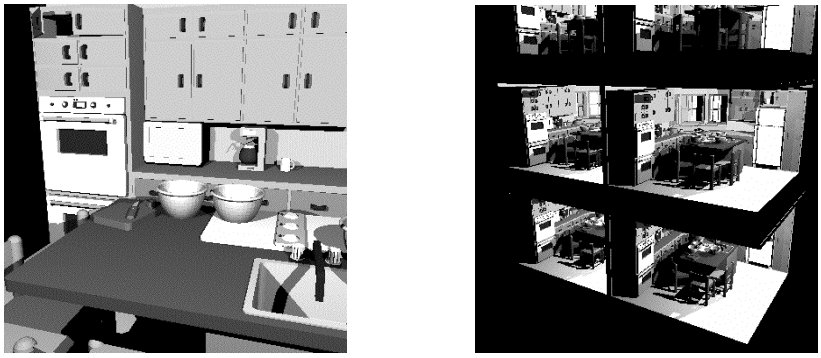


**Figure 10:** *(a) A close-up view of the kitchen scene (b) 6 kitchens*

### 3.1. Experiments

We used two sets of test scenes, constructed from a main model of kitchen:
(1) A kitchen with additional detail objects (bowls, teapots and stacks of plates) added to augment the complexity of the scene (the basic model is shown in Figure 10(a)).
(2) A set of rooms of increasing complexity (2, 4, 6 and 8 rooms) obtained by replicating the kitchen, in what amounts to a building model (see Figure 10(b)). Even though a replicated model is not particularly realistic, we believe that the essential geometric characteristics of a typical building model are well represented, since in each room a large amount of detail exists. Even though it may appear that rotating the kitchens with respect to each other may have generated a more interesting distribution of objects, the resulting clusters would not change significantly because of our use of minimum distance (see the beginning of Section 2) in the clustering of bounding boxes. The differences in scale are quite significant, since object lengths vary from .001m to 40m.

The data structures used for comparison with the $\mathcal{HUG}$ were the recursive grid and the octree with 3 different values for MAXP (50, 100 and 150) and the $\mathcal{HUG}$. We do not mention the uniform grid, the rendering time of which quickly becomes 10 times worse than those of the other data structures. Lastly, the experiments were run for a 200x200 ray-cast image (no reflections), and the parameters of interest shown above are the rendering time in seconds, the megabytes used by the data structures, and the ratio $\nu_1$ = number of voxels / number of items. The memory usage in megabytes was measured using the *sbrk* system call. The preprocessing times (construction of the structures), were of the same order for all the structures.

### 3.2. Analysis

The experimental results are shown in Figures 11-13. The horizontal axes display the number of polygons in the scenes used. We first examine the performance of octrees, then recursive grids and finally the $\mathcal{HUG}$.

*Octrees.* As one can see (3 upper curves of Figures 11(a)-(b)) the rendering times for octrees are worse than the other structures, while the number of voxels are much smaller than the other data structures. The total amount of

memory used is however larger. This apparent contradiction is due to severe memory fragmentation problems. These problems result from the straightforward implementation of octrees in our testbed. It is possible that the fragmentation could be avoided to a large extent by using a stack of static memory to allocate the lists. In addition the number of voxels is lower than for other data structures since the percentage of empty voxels is much smaller.

On the other hand, despite these considerations concerning memory management, the cause of inferior performance is due to the the depth of the hierarchy: from 10 levels for the "one kitchen" to about 15-20 for the "many kitchens."

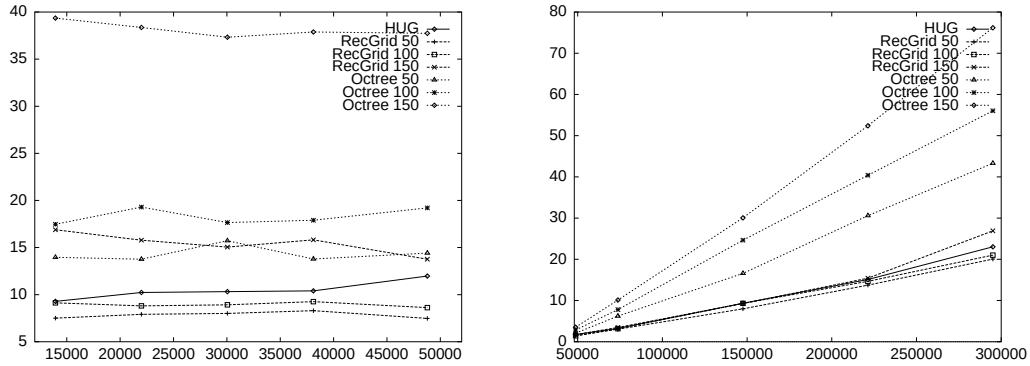



**Figure 11:** *Render time - (a) One kitchen Render time - (b) Many kitchens.*

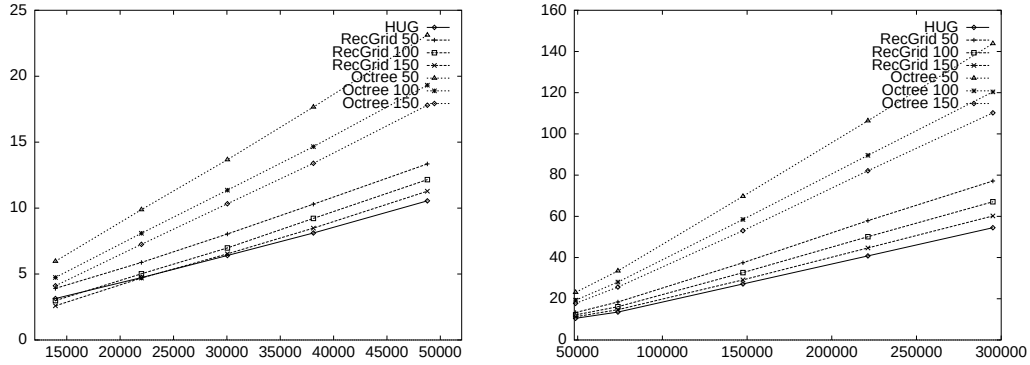



**Figure 12:** *MBytes - (a) One kitchen and (b) Many kitchens.*

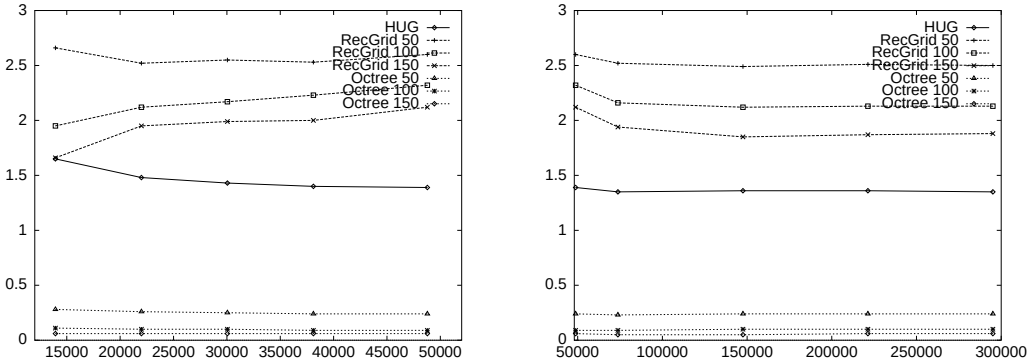



**Figure 13:** *(a)* $\nu_1$ *- One kitchen (b)* $\nu_1$ *- Many kitchens.*

*Recursive grids*. The intuition given by the one dimensional "bucket"-like data structures (section 1.2.2) appears to be verified for the recursive grids: the depth is low (in fact at most 5 in our case). This leads to moderately

high memory requirements but good computational performance. The value of MAXP affects both memory and running time: for the first set of scenes (Figure 11(a)), MAXP=50 (*recGrid 50*) leads to a rendering time 3.5 times faster than MAXP=150 (*recGrid 150*). For the second set of scenes, the choice of MAXP=50 which also gives the best rendering time, requires 30% more memory than MAXP=150.

$\mathcal{HUG}$. The $\mathcal{HUG}$ performs well on both sets of scenes, with running times that are between those of *recGrid 50* and *recGrid 150* , while always using less memory. As an example, for the largest scene of 300,000 polygons, the $\mathcal{HUG}$ uses 40 MBytes and *recGrid 50* uses 60 MBytes. It is interesting to note that on a workstation with 180 Mb of real memory, swapping began to occur for all test runs which are reported to use more than 50Mb. Thus for the largest of the scenes tested, the only practically usable structure was the $\mathcal{HUG}$. Both octrees and recursive grids vary significantly depending on the choice of the MAXP parameter, which often leads them to use significantly more memory and also to display degraded performance.

In summary, we can say the $\mathcal{HUG}$ displays stable behaviour both in terms of computational performance and memory, for both sets of scenes tested. Its memory requirements are the lowest of the 3 data structures studied (within the limitations of the comparative octree implementation), and its performance is similar to that of the recursive grids, depending on the choice of the termination condition. Thus the overall benefit of the new structure is the fact that automatic construction without the need for user-defined parameters provides consistent, satisfactory performance.

## 4. Conclusions and Future Work

We believe that the new approach we have presented is the first step in the development of data structures which have predictable behaviour for very complex scenes without the need for user-tunable parameters. We first discuss the limitations of the approach as presented and some ideas for future work, and we then summarise and conclude.

### 4.1. $\mathcal{HUG}$ restrictions and Future Work

Suppose the initial scene $\mathcal{O}$ is composed of two kinds of objects: some big objects that will give the structure of the biggest grid, and a great many "small" objects that are so far from one another that the clustering algorithm will return as many clusters (up to a constant factor) as small objects. In this case, each voxel of the big grid will contain many references towards small clusters: the cluster and its gridding have been devised for the items themselves, not for the sub-grids, and if this number is too high, the direct access to the grid structure is lost. A more general clustering algorithm should therefore be used, for instance computing the Voronoi diagram of the small objects, a cluster being in this case a connected set of Voronoi cells having approximately the same volume. More generally, whenever the density of objects and the density of sub-grids in a cluster are significantly different, two different data structures should be used. It must be noted however that a small number of uniformly distributed small objects will not affect the performance of the grid *World*.

In addition, finding an optimal (or at least a provably good) value for MAXP is an extremely interesting avenue of research. The question is however difficult, and will require analysis of scenes based on statistical properties. It may also be possible to use the $\mathcal{HUG}$, which provides automatic clustering of the input data-set, in animation and in "multi-precision" rendering.

### 4.2. Summary and Conclusion

In this paper we have introduced a novel approach to the construction of a spatial data structure for very complex scenes. We first group objects of the same size, and then create clusters of neighbouring objects in the same size group. An appropriately subdivided uniform grid is then placed around each cluster and a Hierarchy of Uniform Grids is constructed. We have shown that this construction can be performed efficiently.

The main advantages of the new structure are that: (a) it is fully automatic since it is constructed bottom-up by grouping objects and thus does not require the determination of any user-tunable parameters or termination conditions (b) it adapts well to very complex scenes without problems of memory usage other structures have when built for complex scenes and (c) it provides a good compromise between speed and memory usage since it uses less memory overall than all other structures (given the limitations of the comparative implementation) and performs satisfactorily in terms of execution speed.

**Acknowledgements**

The authors wish the thank the reviewers for their insightful comments and suggestions, Prof. Don Greenberg at the Cornell University Program for Computer Graphics for the model of the *kitchen* scene, and the Dynamic Graphics Project of the University of Toronto for use of the grid traversal code. We also wish to thank Andrew Woo of Alias Research for his many comments and bibliography help. George Drettakis is an ERCIM postdoctoral fellow and is currently financed by the Commission of the European Communities.

**References**

1. T. Whitted, "An improved illumination model for shaded display", *Communications of the ACM*, **23**(6), pp. 343–349 (1980).
2. S. M. Rubin and T. Whitted, "A 3-dimensional representation for fast rendering of complex scenes", *Computer Graphics (SIGGRAPH '80)*, **14**(3), pp. 110–116 (1980).
3. H. Weghorst, G. Hooper, and D. P. Greenberg, "Improved computational methods for ray tracing", *ACM Transactions on Graphics*, **3**(1), pp. 52–69 (1984).
4. T. L. Kay and J. T. Kajiya, "Ray tracing complex scenes", in *Computer Graphics (SIGGRAPH '86)* (D. C. Evans and R. J. Athay, eds.), vol. 20, pp. 269–278, (Aug. 1986).
5. J. Arvo and D. B. Kirk, "Fast ray tracing by ray classification", in *Computer Graphics (SIGGRAPH '87)* (M. C. Stone, ed.), vol. 21, pp. 55–64, (July 1987).
6. H. Fuchs, Z. M. Kedem, and B. F. Naylor, "On visible surface generation by a priori tree structures", in *Computer Graphics (SIGGRAPH '80)*, vol. 14, pp. 124–133, (July 1980).
7. A. Fujimoto, T. Tanaka, and K. Iwata, "Arts: Accelerated ray-tracing system", *IEEE C. G. & A.*, pp. 16–26 (1986).
8. A. S. Glassner, "Space subdivision for fast ray tracing", *IEEE C.G. & A.*, **4**(10), pp. 15–22 (1984).
9. J. Amanatides and A. Woo, "A fast voxel traversal algorithm for ray tracing", in *Eurographics '87*, pp. 3–10, North-Holland, (Aug. 1987).
10. J. M. Snyder and A. H. Barr, "Ray tracing complex models containing surface tessellations", in *Computer Graphics (SIGGRAPH '87)* (M. C. Stone, ed.), vol. 21, pp. 119–128, (July 1987).
11. D. Jevans and B. Wyvill, "Adaptive voxel subdivision for ray tracing", in *Proc. of Graphics Interface '89*, pp. 164–72, (June 1989).
12. M. Mantyla and M. Tamminen, "Localized set operations for solid modeling", in *Computer Graphics (SIGGRAPH '83)*, vol. 17, pp. 279–288, (July 1983).
13. J. Goldsmith and J. Salmon, "Automatic creation of object hierarchies for ray tracing", *IEEE C. G. & A.*, **7**(5), pp. 14–20 (1987).
14. K. R. Subramanian and D. S. Fussell, "Automatic termination criteria for ray tracing hierarchies", in *Proc. of Graphics Interface '91*, pp. 93–100, (June 1991).
15. J. Cleary and G. Wyvill, "Analysis of an algorithm for fast ray tracing using uniform space subdivision", *The Visual Computer*, **4**, pp. 65–83 (1988).
16. E. Jansen and W. de Leeuw, "Recursive ray traversal", *Ray Tracing News (available at nic.funet.fi:/pub/graphics/misc/RTNews)*, **5**(1), (1992).
17. E. Jansen, "Comparison of ray traversal methods", *Ray Tracing News (available at nic.funet.fi:/pub/graphics/misc/RTNews)*, **7**(2), (1994).
18. L. Devroye, *Lectures notes on bucket algorithms*. Birkhauser, (1986).
19. M. Tamminen, "Two levels are as good as any", *Journal of algorithms*, **6**, pp. 138–144 (1985).
20. A. S. Glassner, "Spacetime ray tracing for animation", *IEEE C. G. & A.*, **8**(2), pp. 60–70 (1988).
21. D. Kirk and J. Arvo, "The ray tracing kernel", in *Proc. of Ausgraph '88*, pp. 75–82, (1988).
22. B. Silverman, *Density Estimation*. Chapman and Hall, (1986).
23. D.-M. Tsai and Y.-H. Chen, "A fast histogram-clustering approach for multi-level thresholding", *Pattern Recognition Letters*, **13**, pp. 245–252 (1992).
24. A. Gordon, *Classification*. Chapman and Hall, (1981).
25. H. Day and H. Edelsbrunner, "Efficient agglomerative hierarchical clustering methods", *J. of Classification*, **1**, pp. 7–24 (1984).
26. N. Wilt, *OORT: The Object Oriented Ray-Tracer*. (available at gondwana.ecr.mu.oz.au:/pub), (1993).